VULNERABILITY DETECTION USING STATIC TAINT ANALYSIS

A Thesis in
Computer Science and Engineering
by
Nirupama Talele

© 2017 Nirupama Talele

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

December 2017
The thesis of Nirupama Talele was reviewed and approved* by the following:

Trent R. Jaeger  
Professor of Computer Science and Engineering  
Thesis Advisor

Gang Tan  
Associate Professor of Computer Science and Engineering

Mahmut Kandemir  
Professor-in-Charge of Computer Science and Engineering Graduate Program  
Professor of Computer Science and Engineering

*Signatures are on file in the Graduate School.
Abstract

Increase in internet of things has consequently made ubiquitous the embedded devices that deal with security critical data. The embedded devices generally have proprietary firmware with limited access to its code and documentation. In this work we look at the binaries lifted from these firmware, use a third party tool to decompile them and then analyze them for security flaws. We introduce a Context sensitive, Flow sensitive and selective field sensitive Static Taint Analysis Tool CF-STAT. CF-STAT provides mechanism to semi automatically detect vulnerability in a given program. The vulnerabilities like authentication bypass, hard coded backdoor and user input based undesirable program launching can be detected using the data flow analysis feature provided by CF-STAT. We semi-automatically analyze over 50 binaries using CF-STAT to determine the presence or absence of malice in these programs.
# Table of Contents

List of Figures ........................................ vi
List of Tables ......................................... vii
Acknowledgments ...................................... viii

## Chapter 1
**Introduction** ........................................... 1

## Chapter 2
**Background** ............................................. 4
  2.1 Control Flow Analysis ................................. 4
  2.2 Data Flow Analysis ................................... 5
  2.3 Taint Analysis ......................................... 6
  2.4 LLVM Overview ....................................... 6

## Chapter 3
**Taint Analysis Tool** .................................... 7
  3.1 *CF-STAT* Design ..................................... 7
    3.1.1 Bottom-up pass .................................. 7
    3.1.2 Top-down pass ................................... 8
  3.2 DSA points-to analysis ............................... 8
  3.3 Flow-Aware DSA Analysis ............................. 8
    3.3.1 Algorithm ....................................... 9
    3.3.1.1 Data Structures and Functions used .......... 9
    3.3.2 Example ......................................... 12
  3.4 Context-Sensitive Analysis .......................... 13
    3.4.1 Caller-Callee Map ................................ 13
  3.5 Selective, Field-Sensitive Analysis .................. 14

## Chapter 4
**Analysis and Features** ............................... 15
  4.1 Binary Lifting ....................................... 15
4.2 Taint Assignment and Propagation .......................................................... 16
4.3 Queries on Taint .................................................................................. 17
4.4 Inter-module Analysis ........................................................................ 17
4.5 Implementation Design ....................................................................... 19

Chapter 5

Evaluation ................................................................................................. 21
5.1 Types of Vulnerabilities ...................................................................... 21
  5.1.1 Bypass Identification .................................................................... 21
  5.1.2 Backdoor Identification ................................................................. 22
  5.1.3 Launcher Identification .................................................................. 23
5.2 Analysis Approach ............................................................................... 23
5.3 Case Studies ....................................................................................... 25
  5.3.1 sshd .............................................................................................. 25
  5.3.2 ls.coreutil .................................................................................... 28
  5.3.3 Agetty .......................................................................................... 29
5.4 Other Analysis .................................................................................... 30

Chapter 6

Conclusion and Future Work ..................................................................... 32

Bibliography .............................................................................................. 33
List of Figures

3.1 Data Dependence graph for main function lines 3 to 9 and corresponding RDS updates . . 11
3.2 Example to understand flow Sensitive algorithm . . . . . . . . . . . . . . . . . . . . . . . . 12

5.1 Sshd code containing backdoor vulnerability . . . . . . . . . . . . . . . . . . . . . . . . . . 27
5.2 ls code containing user input based malicious operation . . . . . . . . . . . . . . . . . . . . 29
List of Tables

3.1 Tainted Nodes for with and without Flow Aware Analysis .......................... 12
5.1 Programs Analyzed with $CF$-$STAT$ ................................................. 31
Acknowledgments

I would first like to express my sincerest gratitude towards my thesis advisor Dr. Trent Jaeger for providing me with great opportunities to work on some of the very interesting projects over my years with him. He was always there to guide me through whenever I hit a wall. He always believed and supported me through the entire journey while encouraging me when I needed it. I would also like to express my gratitude towards Dr. Gang Tan, for the invaluable guidance provided to me. I have been blessed with great team members and lab mates who have been a source of continuous inspiration and support throughout my journey. I would like to thank Divya Muthukumaran, Hayawardh Vijayakumar, Frank Capobianca, Xinyang Ge, Yuqiong Sun, Giuseppe Petracca and Suman Saha who have all helped a lot with their inputs, contributions and encouragements.

My deepest gratitude towards my parents and my family who believed in me and encouraged me to follow the path while being always supportive of my decisions. My husband Vinayak Jagtap, has always been my great source of inspiration and support. I cannot express enough gratitude to express my thanks for his immense patience and understanding through the whole process.

Finally I would like to acknowledge Defense Advanced Research Projects Agency (DARPA) for the research opportunity and funds. The research reported here was supported in part by the DARPA under agreement number N66001-13-2-4040. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of any of the above organizations or any person connected with them.
Chapter 1

Introduction

With the advent of internet of things, there exists more than one consumer level connected device per person with a continued growth that is exponential with time. A lot of technologies like embedded systems, sensors, actuators and network connectivity converge to enable the internet of things. As a consequence the use of embedded devices that deal with sensitive and privacy related data is on rise and are becoming omnipresent. Most of these devices run proprietary software and firmware and thus the source code or a proper documentation is not always available for them. Since these software are widely used and deal with security critical data, it is essential that these be analyzed for possible vulnerabilities. The vulnerabilities can be of the type that the attacker can leverage to either gain access to the sensitive information (e.g. wireless routers in home environment, if compromised can leak user’s sensitive information to the attackers) or to induce unintended behavior (e.g. Smart meters, RFID doors etc, if compromised can be made to function as per attacker’s commands). Thus security is critical for these devices, and it is imperative that the software used in these devices not have any flaw that can be exploited by the attacker.

There has been a rise in the cases of attacks on these devices, on routers alone the CVE (common vulnerabilities and exposure) database records about 800 vulnerabilities total and about 100 in 2017 alone [1, 2]. There are also various other surveys and reports suggesting how insecure home routers are [3, 4]. The vulnerabilities listed in them range from authentication bypass [5], backdoors, information leak, denial of service to random code execution. Most of them are caused by flaws in the software/firmware on the devices while some are due to protocol and weak passwords, etc. Apart from routers, various other smart devices with more complex software and critical functionality also continue to grow. Attacks exploiting vulnerabilities in these devices may imply huge repercussions in digital as well as physical world.

Software verification for these devices is non-trivial and has challenges at multiple stages from gaining access to the software to analyzing the binary for exploitable flaws. To analyze the binaries we first need to lift them to an intermediate representation with semantic constructs that can be analyzed using static analysis techniques. The challenge here is to have accurate lifting of the binaries. We use Hex Rays Decompiler [6]
with some manual effort to ensure maximum coverage. Our tool \textit{CF-STAT} then works on the assumption that the lifting is complete enough for our analysis. There are many challenges when we analyze the lifted binary, such as incomplete type interpretation, stripped symbols, pointer operations to access structure fields and sometimes but very rarely missing segments of the code. Considering we have some heuristics to address the above issues, the next challenge is to formulate the analysis to identify the vulnerability, assuming the access to the source code and a proper documentation is not always guaranteed, we need to rely on some general heuristics to identify the analysis process.

Recently a lot of research has been done in reverse engineering of binaries and analyzing them for anomaly detection. The research progressed in developing binary analysis frameworks like BAP, IDA pro and angr \cite{7, 8, 9}, out of which BAP and angr are open source and have seen growth in usage in the community. Each of these tools have their own intermediate representation and analysis on top of that. Some components of angr are also used in an analysis for automated authentication bypass detection \cite{10}. Angr analyzes the binary using static as well as symbolic (concolic) analysis. Given the difficulty of obtaining access to the binaries from the firmware and analyzing it, it is also challenging to generate an environment to perform dynamic analysis. There has also been work towards emulating and optimizing the firmware binaries for dynamic analysis \cite{11}.

In this work we focus on static analysis to detect the vulnerabilities like authentication bypass, backdoor, information leak and random code launch. We introduce Context sensitive, Flow sensitive and selective field sensitive Static Taint Analysis Tool \textit{CF-STAT} based on LLVM \cite{12}. The primary idea behind using LLVM is to extend the analysis for binaries as well as source code in various languages and platforms. We target the subset of the vulnerabilities that can be detected using static taint analysis and apply various heuristics on it to analyze more efficiently and precisely. The insight is that any input read by the program can potentially be an attacker input and thus by tracking the flow of these input sources we can determine if the attacker can reach the sensitive objects or operations in the program without being mediated. This also involves identifying mediators and verifying whether the mediators are correctly verifying and following the policy specified for the program. It is very challenging to completely automate this kind of detection, since it is practically very difficult to determine the intent of the application logic in source code and even more so in a binary. Thus we use a semi-automated approach to aid the analysis to make some decisions at intermediate stages and to infer the presence or absence of a vulnerability in a binary. Our insight on the mediator identification involves how the data from various sources like user input and the policy are used in the program to determine the credibility of an access. The instructions in the code that uses all these data together to make decisions are selected as candidate mediators, with some heuristics and manual intervention. We then determine valid candidates out of these for further analysis. We address the efficiency by using the concept of program slices on the control flow graphs of the program to analyze only the relevant subset of the program. Similarly we also selectively make the analysis field sensitive for the relevant fields
in the data flow graph. The taint sources and sinks are custom identified for each program based on the rule to be applied to that program.

We show how the tool *CF-STAT* enables us to identify vulnerabilities in software using taint analysis. We analyzed a little over 50 programs provided as test set for DARPA VET program [13] using the *CF-STAT* and performed a semi-automated analysis to determine the presence or absence of the vulnerability by proving whether a certain rule is violated or not. The analysis times vary from a few seconds to a few hours based on the program size, the programs like sshd was one of the larger programs we analyzed which takes about 4 hours for the analysis. We have analyzed other programs like lighttpd, passwd, RFID door, kadmin, coreutils, etc. We show how inter-module analysis helps in ensuring the analysis scope and covers the sensitive data going in and out of the program.

In this thesis we will review the background on static analysis and explore the environment we use in Section 2. Section 3 will provide a complete description of the tool structure, the algorithm used and the precision improvements implemented. We then dive into the additional features and heuristics implemented in the tool for binary analysis in Section 4. In Section 5, we discuss the various types of vulnerabilities that can be targeted by *CF-STAT*, the approach on detecting each and provide some case studies showing the use of *CF-STAT* for identifying vulnerabilities. Finally we briefly discuss the future work and conclusion for this work in Section 6.
Chapter 2

Background

In this chapter we will briefly review the basic concepts on which we base our analysis tool and approach. We will first take an overview of static analysis and its components. Then we will dig into the basics of taint analysis and the framework we use for CF-STAT. This will give us background to understand the taint analysis tool we introduce in this thesis for vulnerability detection.

There are various techniques developed over years for security analysis and one of them is static analysis [14, 15]. In static analysis we analyze the source code of the program to find various properties like data flow, typing errors and code optimization options without any need for concrete execution. For data flow properties, static analysis is in general less accurate than dynamic analysis, which can sometimes impose high run-time overhead or can be impractical in certain cases. The purpose of static code analysis is to find vulnerabilities in the code that could enable the attackers to exploit the code in order to get some sensitive information or induce undesired behavior. Static analysis on a software can extract various relevant information from the program, like the data flow graph, the control flow graph, the path constraints and various other code properties. We will look at some of these basic techniques that we use in detail in the following sections. We use static taint analysis in our tool CF-STAT to determine presence of various kind of vulnerabilities related to information flow in a program.

2.1 Control Flow Analysis

To perform most of the static analysis techniques on the program, the knowledge of the control flow for the program is required. The control flow analysis in a program tracks the order in which the basic blocks are executed in a program. The basics block is a set of sequential instruction with single entry and single exit point. The branching and call like instructions in the program define how the flow of the control is governed in the program, we call these as control instructions. The control flow is expressed in a control flow graph (CFG) which represents all possible execution paths in the program. A control flow graph for a program is
generated by identifying the possible target basic blocks at any given control transfer instruction and adding the control flow edges accordingly. The intra-procedural CFG construction is relatively straightforward and is done by identifying the basic blocks and the control flow structures. The direct calls and jumps have a predefined jump targets, but we need additional analysis to determine the targets for indirect calls and jumps. There are various approaches that can be used to compute a precise CFG for the program by resolving the targets for dynamic dispatch in a program. The CFG generation is often an over-approximation of valid targets for the indirect control transfers, these can be coarse grained or fine grained based on how much over-approximation is allowed. The coarse grained CFG construction generally just considers all the valid targets in the program as the targets for each indirect transfer [16, 17]. This adds many additional edges to the CFG which will never be taken, to make the target computation more fine grained, many researchers use the signature based approach for determining the targets by matching the function signatures of the called function to the possible target functions [18, 19]. However this approach is not very suitable in the case where we deal with a decompiled binary and not all type information is restored for the analysis, we use a taint based approach as proposed in [20] which tracks the taint from pointer variables assigned with an address to the indirect transfer statement to determine the targets, this method proposes an efficient target computation without the need to employ expensive pointer analysis. We use this approach to compute control flow graph for our taint analysis in \textit{CF-STAT}.

2.2 Data Flow Analysis

The data flow graph represents how information flows in the program, how it is copied and modified over its path. The data flow graph (DFG) is generated by following the control flow graph of the program and interpreting the flow behavior in each instruction as we traverse the program. The flow relations are generated based on the interpretation rule for each instruction where in most cases the data from the right hand side operands flows to the left hand side variables. In a data flow graph the nodes consist of the variables in the program and the directed edges define the flow of data. There can be different representation and different information that is propagated while performing a data flow analysis [21] based on the goal of the analysis. In our analysis we especially use the reaching definition analysis which checks whether a particular definition for a variable reaches a given program point or not. We generate the graph based on the reachability of data from one node to another. For instance in an instruction like $a = b + c$ the data from b and c flows to a, thus we add the data flow edges accordingly for each instruction.
2.3 Taint Analysis

The technique used for taint analysis can be either dynamic or static, depending on whether the data is tracked dynamically using instrumented code as it flows through the program or by statically analyzing the code for information flow. Taint analysis is basically a method where we assign certain taint labels to data of interest, in most cases it is the user input data or untrusted data, which is then propagated through the program to understand the behavior of the tainted data through the program run. Static time analysis does not have actual execution path information so we generally perform a conservative analysis which provides us with an over-approximated taint on the program, the over-approximation can be contained using various approaches to improve the precision of the data flow graph computation. In this work we use context sensitive and field sensitive analysis to reduce the over-approximation in the taint, we also introduce selective field sensitive analysis which can specify set of relevant structure fields to track the information flow through them more precisely. A wide range of information flow based security problems can be addressed using taint analysis, as shown for software attack prevention [22, 23] and information flow control and data leak problems [24, 25].

2.4 LLVM Overview

LLVM [12] is a very popular compiler infrastructure with modular and reusable components. It is designed so that it can support multiple languages which are then compiled into LLVM Intermediate Representation (IR). It provided various passes that can optimize and analyze the programs statically, while providing an easy to use interface to write your passes. Our tool CF-STAT is based on LLVM IR and have catered some algorithms specifically for dealing with LLVM IR idiosyncrasies. The LLVM IR is represented in an SSA form for the scalar variables and thus providing a leverage in performing use-def like analysis. We extend this further by adding the SSA like behavior even on the stack variables for obtaining more precision. There are various front ends independently developed for LLVM for various languages like C/C++, Ruby, Python, Java, PHP, etc.
Chapter 3

Taint Analysis Tool

In this paper we present Context sensitive, Flow sensitive and selective field sensitive Static Taint Analysis Tool CF-STAT. We use LLVM as our platform to build CF-STAT on account of its wide range of language support thus extending our analysis tool support for all the languages supported by LLVM like Ruby, Python, Haskell, Java, D, PHP, Pure, Lua. There are various independent projects that provide the front end to compile these languages in the LLVM IR representation and if not available there are guidelines provided to build one if needed. For this project we will primarily focus on C/C++ for all examples and evaluations that we discuss. In this chapter we will discuss the design for the tool, features and the challenges that we deal with for making the tool more precise while defining the trade-off between the precision and performance.

3.1 CF-STAT Design

We compute the data flow graph for CF-STAT in two passes of the input file. The bottom up pass which computes the data flow graphs for each individual function and a top down pass which connects the function graphs based on the call graph. The input file we use is the llvm bitcode file which is generated by frontend tools like clang for C/C++ programs converting them to llvm IR representation which CF-STAT then parses to compute the required data and control flow graphs for analysis.

3.1.1 Bottom-up pass

The bottom up starts with processing the leaf nodes first in the call graph going in reverse topological order. CF-STAT generates the data and control flow graph for these functions and then goes up processing all the functions until it reaches the root/main function. CF-STAT also summarizes the data flow graph for each function representing the dependence on the input values or values loaded in the function. Any side effect caused by these values on objects out of scope of the given function or on the return value is included in the summary, this summarization method is described in Sharir and Pnueli [26]. We do this to avoid
reprocessing huge irrelevant data when we connect the graphs in a context sensitive approach. We will look at the details added to this pass for each precision feature in the following sections.

### 3.1.2 Top-down pass

The top down pass then starts with the main functions and in topological order progressively maps the called function’s data flow graph with the caller function. *CF-STAT* connects the formal and actual parameters from the callee and caller respectively following the call graph in a context sensitive approach. We discuss the operations performed in this pass for each individual precision feature in the following sections. We obtain a data flow graph for the entire module after this phase, which we can then use for asking various taint based queries.

### 3.2 DSA points-to analysis

Alias analysis identifies the variables in the code that points to and may be used to access a given memory location. Points to analysis is a subset of alias analysis which determines the set memory location that a given pointer may point to. DSA points to analysis [27] is a unification based algorithm on Steensgaard was released with earlier versions of LLVM and was removed due to patent issues. There are some variations of the project that were adapted for later versions and we use the adapted version for LLVM 3.7. The thesis describes how DSA is more precise and performs better than the available alias analysis implementations for LLVM [28].

### 3.3 Flow-Aware DSA Analysis

LLVM converts the code into an Single Static Assignment(SSA) based representation. That is, any variable in the Intermediate Representation(IR) will have only one definition. This representation helps in maintaining type safety, low level operations and flexibility in representation of various high level languages. Also the IR provides partial flow sensitivity by representing all the register variables in SSA format. Although the flow sensitivity is not maintained when we have data flow operations on the memory, (loads and stores from and to the memory). In the context of taint analysis we need the flow sensitivity to determine that a particular memory location was tainted only after a particular assignment. The DSA analysis is field and context sensitive but flow insensitive. Thus when an tainted update is made to a memory location at a later point the read from an alias in the preceding blocks will also be falsely tainted. To address this issue the following algorithm suggests a method of making these data updates to memory location to be flow aware. The assumption is that all the memory operations happen only using the store and load instructions in this
context. The algorithm in this document will focus on explaining how the appropriate stores and loads are mapped to avoid such spurious flows.

3.3.1 Algorithm

The algorithm is based on the gen-kill approach for live variable analysis algorithm. The high level parsing of the program using call paths and processing function control flow graph (CFG) is eliminated in this discussion. The Algorithm 1 represents the processing at each basic block level, where we update a global set of recent definitions (RDS) for relevant memory locations. The definitions are ordered as per their occurrence and are associated with the respective store instructions. Thus $d_n$ will be the latest definition encountered, $d_{(n−1)}$ will be the definition before that and so on as discussed in [29]. At each write (store instruction) in the memory a new data flow graph node is created to keep a record of that update. Thus at every use we can track to the immediately dominating definition node for it and add the flow edge accordingly.

3.3.1.1 Data Structures and Functions used

- **LiveStores** is a map which maintains the relation between all the pointers at any given point of time and the recent store operations nodes at that memory location that were not overwritten. map $<\text{Pointer, StoretoPointer}>$

- **RecentDefStack** keeps track of the new data flow node generated at each store operation in the order that it was generated. The data flow node is then used to add the edge when a load reachable from this store is identified. This stack is updated for each basic block keeping record of new definitions.

- **UpdateRDS** function generates the new memory definition node for the store and keeps map of the store, pointer and the node in the RecentDefStack

- **UpdateLiveStore** Finds and updates all the new definitions for the memory location at that point, uses all the alias information to find updates made to the memory location using any alias pointer as well.

- **GetRecentDefs** Retrieves all the recent definitions for the memory location pointed to by the load pointer to add appropriate edges.

The affected Basic blocks are again added to the list of blocks to be processed, thus whenever a new definition is discovered, the corresponding uses will get processed. Every match is done with all the aliases, when looking for the uses, instead of just checking for the uses of the pointer symbol, we will also look for the uses of any of the alias of that pointer to add the affected blocks. While making the update to the
Algorithm 1 Process Recent Memory Definitions in Blocks

Input: BB Basic Block to process, Data flow graph to be updated and the use def chains to be updated
Output: The set of basic blocks Affected_BB those have the uses which are defined in block BB

1: function ProcessBlock(BB, DataFlowG, RecentDefStack)
2: for all Pred = Predecessor(BB) do
3:     RecentDefStack ∪ = MergeDefs(Pred);
4: end for
5: for all Instructions I in BB do if IsStore(I)
6:     newDef = Update_RDS(I, RecentDefStack)
7:     Update_LiveStore(I, LiveStore, DSA);
8:     //Add all blocks affected by def to reprocess
9:     Uses = GetAllUses(I);
10: for all Use u inUses do
11:     B = GetBlock(U);
12:     Affected_BB = Affected_BB ∪ B;
13: end for
14: if IsLoad(I)
15:     RecentDef = GetRecentDefs(RecentDefStack, Livestores, I);
16: end if
17: Process_Inst(I, DataFlowG, RecentDefStack, LiveStores);
18: return Affected_BB;
19: end function

LiveStores we will be replacing any definition to the memory location, unless it is a merge of two branches where each update definition is preserved.

While computing the recent definition and Live stores along with just the pointer symbol, we also check for the updates made to any of the aliasing pointer to that memory location, using the DSA points to information. We check the pointer with all live loads to see if a relevant update to the pointer or an alias of pointer is made. When a match is found for any of the aliasing pointer, we retrieve the data flow store nodes created for that store and add them as the live stores for the given load instruction. An edge is then added accordingly to represent the flow from all the recent updates in all paths from the memory location to appropriate load pointer for further propagation.

If there are n statements in the given function, our iteration is then of the order O(n). Let m be the number of memory locations/ pointers in the code. Then at each pointer check we compare the a given pointer with all m other pointers. Memory usage for keeping track of the loads an stores is in the order of number of pointers in the given function as i.e. m.

The following Algorithm 2 explains how each instruction is being processed while doing a flow sensitive data flow analysis. Most of the flow sensitivity in LLVM can be harnessed from its SSA representation. Thus we add the edges from the operand on the right hand side to the SSA variable of that instruction. We handle
the stores and calls differently and have handled the stores in the Process Block algorithm. Also when we do a load we need to identify which recent definition at the location do we need to add the edge to which is also discussed in detail in the process block algorithm. The details of the store instruction are described in the algorithm for block handling, thus once we have the recent definition information for the required variables we process all the other instructions and add appropriate edge in the Process Instruction module.

We also describe how we add the node for a structure field and an array or other container type. For each field node, we keep track of the base pointer and the string of offsets to the field.

**Algorithm 2 Process Instruction**

**Input:** $I$ Instruction in a basic block to Process  
**Output:** Updates the data dependence graph for the given instruction  

1. function $\text{Process\_Inst}(I, DataFlowG, RecentDef\_Stack, LiveStores)$ if IsCall($I$)  
2. $\text{PointstoMap} = \text{CallSiteStackUpdate}(I)$  
3. IsStore($I$)  
4. $\text{pointerVal} = \text{getPointerVal}(I)$;  
5. $\text{storeVal} = \text{getStoreVal}(I)$;  
6. $\text{AddNode}(DataFlowG, \text{pointerVal})$;  
7. $\text{AddNode}(DataFlowG, \text{storeVal})$;  
8. $\text{AddEdge}(DataFlowG, \text{storeVal}, \text{pointerVal})$;  
9. IsLoad($I$)  
10. $\text{loadOp} = \text{GetLoadOp}(I)$;  
11. $\text{RecentDef} = \text{GetRecentDef}(\text{RecentDef\_Stack}, \text{LiveStores}, \text{loadOp})$;  
12. $\text{AddEdge}(DataFlowG, \text{RecentDef\_loadOp})$;  
13. IsGetElementPointer($I$)  
14. $\text{basePtr} = \text{getBasePointer}(I)$;  
15. $\text{offset} = \text{getOffsetString}(I)$;  
16. $\text{AddNode}(DataFlowG, \text{basePtr} + \text{offset})$;  
17. $\text{ptrOp} = \text{getPointerOp}(I)$;  
18. $\text{AddEdge}(DataFlowG, \text{basePtr} + \text{offset}, \text{ptrOp})$;

Figure 3.1: Data Dependence graph for main function lines 3 to 9 and corresponding RDS updates
3.3.2 Example

Consider a simple example as represented in the figure 3.2.

Figure 3.1 represents how the data dependence graph is created for the small example. The example helps us understand how do we handle multiple definitions and fields in the analysis. Consider the small snippet from line 3 to 9. We process each instruction at a time using the above defined algorithms and get the following Data dependence graph. Note that we do not create nodes for just declarations, we add the node if a definition or use exists for it. Thus we create a definition at line 6 for b, in the IR representation this is a converted to a store instruction, thus when we hit isStore we add the appropriate info in the RDS table. Then when we encounter a use for it we add the node in the graph and add the data flow edge to the lhs operand. In the next line we get another definition for b and we update that in the RDS table as b_8 and at line 9 we add edge b_8 to (srv+offset_pass_ref).

Table 3.1: Tainted Nodes for with and without Flow Aware Analysis

<table>
<thead>
<tr>
<th>Programs</th>
<th>DSA Flow Aware</th>
<th>DSA Flow Insensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>lighttpd</td>
<td>23008</td>
<td>22501</td>
</tr>
<tr>
<td>wpa supplicant</td>
<td>53627</td>
<td>52585</td>
</tr>
<tr>
<td>sshd</td>
<td>45253</td>
<td>45059</td>
</tr>
</tbody>
</table>

We performed the flow aware analysis on the programs provided as a part of the DARPA program and
found that there is a lot of precision gain for some programs where a structure pointer ends up tainting a lot of data even if one of the fields is assigned a tainted value, making all the previous reads from the structure also tainted. Table 3.1 shows the results for three programs representing the total number of nodes generated and nodes tainted for each program with and without the above method.

3.4 Context-Sensitive Analysis

We follow the classic Sharir and Pnueli [26] Algorithm for context sensitive analysis using the call strings approach. While we follow the order as per the algorithm there are more complexities while mapping the formal and actual parameters introduced by the flow aware and field sensitivity implementation in our algorithm. We will discuss some of the details for mapping the calls in the program.

3.4.1 Caller-Callee Map

Based on earlier discussion, the caller callee map algorithm talks about parsing of each function on the call path and processing at the instruction level. Where for every call site instruction encountered the stack of points to nodes at that instruction are updated with the function parameters map as well as all the points to relations before the call site.

The proposed algorithm tries to incorporate the changes along with the live variable analysis. There are the following things to be taken care of when using the DSA analysis for computing the data flow graph.

1. The input for computing the context sensitive graph for the relevant section of the code, we use the input as sets of call paths from the source function to the sink function. These paths are processed instead of call graph for the context sensitive analysis.

2. The initial stage then consist of identifying various contexts that are required for a given function. If we have different call strings leading to function and the relevant memory location is being updated then a new context is generated for that function.

3. At every call site if a new context is to be generated for the function, then relevant points to information map is generated specific for that call site based on the call string and the points to updates until that instruction in the function.

4. This points to information at each call site is then used for finding the latest updates to the memory location along with the recent definition map and appropriate edges are added.

Algorithm 3 represents the generation of the points to stack for each call site where a new context is required for the function. The points to information for the actual parameters in the call site is updated in
the stack appropriately. Also the maps from the actual parameters at the callsite to the formal parameters in added in the stack. This points to info will help in adding edged when changes are made to the memory location inside the called function. If a new context is not required then the existing points to info is retrieved for that function.

Algorithm 3 Caller Callee map

| Input:  | I Call instruction in a basic block to Process |
| Output: | The relevant points to stack for the given call site. |

1: function CallSiteStackUpdate(I)
2:   ActivePointers = GetPreceedingPtrSymbols(I); if IsnewContext(I)
3:   params = getActualParams(I);
4:   PointstoMap[I, func] + =
5:     getDSApointstoInfo(params, pointstoGraph, ActivePointers);
6:   PointstoMap[I, func] + =
7:     getformalParamInfo(params, formalParams[func]);

3.5 Selective, Field-Sensitive Analysis

There are programs that use huge structures to maintain critical information as well as to store the user provided input, in these cases if one field of the structure is tainted it will taint the entire structure in a field insensitive analysis. Although a field sensitive analysis for huge program with such huge structure will be very performance and time intensive. We introduced selective field sensitive analysis with the intention to select only the relevant fields and track them.

When we work with lifted binary, it does not always maintain the type of the structure and represents the field accesses using pointer arithmetic. We use structure recovery using the offset information to retrieve the structure for the object, although if done for whole of the program will increase the analysis time. We thus select only those fields and structure that are actually being assigned any tainted data at any point in the code. This process can be iterative provided the tainted fields will in turn taint other fields, which will have to be added in the relevant set for analysis. We iteratively perform the addition until no new field is added before generating the data flow graph. To add the field sensitivity we keep track of the field index for any store or load operations while we generate the RDS and add the edges to the load instructions if the appropriate field for the structure is matched. If a field read or write happens and the field is not identified as a relevant field no special node is created to track the field separately and the whole structure is treated as one. In programs with huge structures with only few tainted fields, we have observed that introducing the field sensitive analysis reduces the extraneous taint by almost 80%.
Chapter 4

Analysis and Features

The *CF-STAT* is developed primarily with the intention of analyzing binaries lifted from proprietary firmwares and vetting them of vulnerabilities. The scope of vulnerabilities that can be semi automatically identified using *CF-STAT* are authentication bypass, backdoors that use hard-coded values and launching of undesired programs based on adversary input. The vulnerabilities like buffer overflow and memory corruption are out of the scope of this taint analysis tool *CF-STAT*. In this chapter we discuss the semi automatic analysis process followed by *CF-STAT* for isolating and identifying the suspicious parts of the code.

4.1 Binary Lifting

We start with the binary of the program to be analyzed which can be extracted from the firmware and provided as input for analysis. There are various binary lifting tools which attempt to lift the binary in and Intermediate representation for ease of analysis most of these tools have their own IR representations. We started with using one such sophisticated tool BAP [7] which lifted the binary to BAP IL and provided a plugin that can convert the BAP IL to LLVM IR. The representation was similar to assembly type of instructions and required some complimentary analysis to augment the information needed for the taint propagation in LLVM. We implemented some helper modules that will extract additional information using BAP modules to aid in our analysis. Unfortunately the plugin for BAP IL to LLVM IR was not supported for further versions and hence was obsolete to use.

We then worked with IDA Hex ray to decompile the code, IDA represents the decompiled code in a C like language. Due to some type uncertainties IDA cannot decompile it in a semantically correct decompilation that can be used to compile directly. We use scripts to identify such issues and convert the C like code to a more compilable C like code. The script can remove all the semantic errors in small programs and make it completely compilable although there could be still some errors that would need manual fixing for larger programs with complex structures. Many unresolved types are assigned a default type in this process,
provided that the taint flow relations does not change even if the type is not identified exactly. For instance an unidentified type like DWORD in the IDA decompilation can be changed to an char or int and the taint flow relations will still be maintained. The scripts also take care of some undefined headers and missing definitions by introducing some default ones. Once the code is made error free, we then compile it to LLVM IR using the front end tool Clang. The bitcode file thus generated is then used as an input for our LLVM analysis passes.

### 4.2 Taint Assignment and Propagation

The adversary can control a program by providing some external inputs that can affect either the security sensitive data or the flow of the program, can also be used to trigger some backdoor. The interfaces that can be used by the adversary to provide such inputs need to be identified and mapped as tainted sources. *CF-STAT* identifies the standard input commands and maps them as taint sources. The standard inputs identified are the command prompt inputs and the inputs read from the standard commands like get, read, scanf, recv, getchar, fgets, etc. In a general setting all of these inputs will be considered as adversary accessible and be marked as the taint sources to identify if any of variables tainted with these inputs do not perform anything suspicious. Each of the taint source input is assigned a unique label to identify the source of the taint, which allows the analyst to understand the reachability of input coming from various sources and define rules to identify any anomaly. *CF-STAT* uses some set of rules for the given program and analyzes to determine if the program satisfies the rule. The rule is in terms of the type of vulnerability that we want to find in the program. The relevant taint sources can then be filtered manually using the the rule to direct the analysis in the right direction.

The taint labels are propagated over the flow aware data flow graph as computed in the previous section. When multiple taint labels reach a node, the union of the taint labels is assigned to the new node. The problem of taint explosion is contained due to flow aware handling of the memory locations, thus when a tainted value is assigned in a memory location it will be propagated only to the reads after the assignment. We also introduce a declassify functionality which allows us to specify certain functions as mediator functions, the tainted data passing through these mediator functions will be declassified as mediated before reaching a security sensitive sink. This features allows the analyst to find the tainted inputs which can reach the sink without passing through any mediator. Such unfiltered inputs can potentially be exploited by the adversary to gain access to the security sensitive section or objects in the code.
4.3 Queries on Taint

*CF-STAT* produces a sub-graph with the tainted nodes which can be used to check various properties of the tainted data. To identify any suspicious behavior in the code we need to know how the different user input data interacts with each other. To be able to answer such questions on the tainted data we implement a query mechanism in *CF-STAT*. The tool can answer the queries like "display the nodes where data from \(\langle taint\_label1 \rangle\) and \(\langle taint\_label2 \rangle\) interacts" or the conditional statements where multiple tainted data affects the branching condition. *CF-STAT* provides support for displaying union and intersection of various labels. The union operation will give the nodes with any of the taint labels from the query. The intersection operation will provide only the nodes that are tainted by all of the taint labels.

4.4 Inter-module Analysis

There are various forms in which two different modules can exchange data. While performing an inter module data flow analysis we need to model these paths to understand how the information is flowing. We identify the following data communication options and discuss the approach to handle those.

1. **Shared global data:** The shared global data among the two modules need to be identified, if the global data is tainted the taint label and the data is specified as taint source for the module \(< Global\ Variable\ Taint\ Label >\). In LLVM IR we can look for global data that is visible to other modules and the global data that is imported from other modules. We look for taint information on these shared global objects for further propagation.

2. **Inter module Communication:** We assume that this is the standard form used for inter module communication and elaborate further on this.
   a. **Parameter passing:** The data can be shared as a parameter passed between the two modules, given the parameter is tainted in the first module it is identified as taint source for the second module for further investigation. The specification of taint source \(< Function\ Variable\ Taint\ Label >\), where the *Function* is the called in the module whose parameter is tainted and the variable represents the parameter.
   b. **Return values:** The specification of a return value taint may differ since the function can be called from multiple locations in the executable and thus specifying it in terms of the called function. Again we have two cases where just the return value is tainted \(< Return\ Called\ Func\ Taint\ Label >\), the taint label is simply added to the return value of the call statement to the *Called Func*. Second case where one of the parameter is an output...
< Param Called Func Param_location Taint_Label >, in this case we assign the taint label to the actual parameter of the called function given the parameter location.

We perform the taint flow analysis iteratively on each module and retain the taint label information intra as well as inter module data as we proceed to the next phase of iteration. We perform the taint source identification for each module, giving us the basic set of sources to start the analysis. We specifically deal with the case where one of the modules is an executable and uses the library modules. Thus we start our analysis with the executable and recursively process the libraries that it uses.

Algorithm 4 shows how we can automate the inter module analysis for analyzing taint flow information among them.

**Algorithm 4 Inter Module Taint Flow Analysis**

**Input:** modules all dependent modules, TaintSource taint source information for each, Threshold Analysis Depth

**Output:** The taint flow information for all modules

```plaintext
1: function IntermoduleAnalysis(modules, TaintSource, Threshold)
2:   for all mod in modules do
3:     RunTaintAnalysis(mod, TaintSource);
4:     called_modules = RecordModulesCalledIn(mod);
5:     getTaintInfo(mod);
6:   end for
7:   iter = 0; while noNewTaintSource and iter ≤ Threshold
8:     iter + +;
9:   for all calledMod in called_modules do
10:      TaintSource = GetTaintSources(calledMod, ParamTaint, GlobalTaint, ReturnTaint);
11:      RunTaintAnalysis(calledMod, TaintSource);
12:      getTaintInfo(calledMod);
13:    end for
14: end function
15: function getTaintInfo(mod)
16:   ParamTaint = GetParameterTaintInfo(mod);
17:   GlobalTaint = GetGlobalTaintInfo(mod);
18:   ReturnTaint = GetReturnTaintInfo(mod);
19: end function=0
```

We run the basic taint propagation if there are individual taint sources for any of the modules, else only the modules that called from the executable will be analyzed. The functions which are called in each module will also be recorded and used for truncating the source code input for the modules.
4.5 Implementation Design

This section will describe the breakdown of the implementation specific tasks relating to the intermodule analysis for data flow. The following describe the breakdown of tasks to identify forward flow, i.e. data going out of the module.

1 Collect all the data that is externally accessible. (a). Globals that can be externally accessible, (b). Calls to external functions. These will be treated as sinks for initial taint propagation, to identify outgoing tainted data.

   i Pass 1 - Collecting Globals: For Data flow analysis, collect all the globals with external visibility, for function pointer, collect all the globals of type function pointer or one of the contained type is function pointer (recursive iteration over global types to identify contained types).
   Input: Module to be analyzed, type of analysis (data, function pointer)
   Output: Collection of all the relevant global data that can be externally accessed.

   ii Pass 2 - Collecting Parameters: Determine all the function calls which calls external function.
   Data going out of the module → Collect all the parameters sent to the external functions.
   Data with external side effects → (a) The return values of the functions, (b) Reference values sent as parameters. (to compute this information we need to maintain it for all the functions in a given module, since it will not be known until later that that function was being called at an external site and that the return value or any parameters are influenced)

2 Pass 3 - Source Mapping Along with original taint sources, add the taint sources identified in other modules if available. (a) Parameters: Add taint label (with source module prefix) for the parameters of the externally called functions. (b) Add taint labels (with source module prefix) for the globals which are extern or externally visible and were tainted in the other module.

3 Pass 4 - Taint Propagation Compute a regular data flow analysis with all the taint sources if available (If data from other module is not available, use Zombie taint for externally accessible and extern globals). This pass will also refine the data collected in Pass 1 and 2 to represent only the data which is actually tainted and can flow out of the module.

The flow from module whose functions are called in the current module, can be analyzed only after one round of analysis of the current and the loaded library module. This will provide info on whether the taint is returning to, or had a side effect on the calling module.

1. Check if all the extern globals collected in pass 1 were tainted by the library module taint, if so reflect the incoming taint flow on the current module. (add source and rerun, if not done the zombie run, else update the zombie taint label with the received label)
2. In case of return value and reference parameters of the external calls being tainted by the library module, add these sources and re run the analysis with the received labels.

The modules are iteratively loaded and analyzed until no new external taint information is generated for any of the dependent modules. We perform an elf red to get initial set of dependent modules and the function list to identify the modules with function having external taint. We found iterative inter-module analysis to be more precise than the analysis of the code which was statically linked together before the analysis, thus make the analysis more efficient.
Chapter 5

Evaluation

We discuss the complete process of the analysis using \textit{CF-STAT} in this section. Then we will apply those analysis techniques on some examples and discuss the results. The tool \textit{CF-STAT} was primarily developed for the DARPA VET \cite{13} program which entails vetting commodity IT software and firmware of vulnerabilities. As a part of the program we evaluated various firmware binaries which involved ARM binaries extracted from the Gumstix firmware. Each of the binaries comes with a possible rule set which provides some guidelines on what kind of vulnerability could possibly be exploited from the given binary. This rule set is provided as an input to the analysis. The rule can be generated by analyzing the whole system to determine what atoms are exploitable, with probable set of inputs. These specifications help in determining the taint source inputs for these programs. In some cases the rule can be very broad and thus we need to test with the standard inputs identified and applying all analysis approach to find if any vulnerability that is in scope of \textit{CF-STAT} is present in the code. As a default case if no rules are available all the test approaches are applied to the binary to determine what vulnerability will potentially be present.

In this section we first look at various types of vulnerabilities that we address and the analysis approach to identify them. We later look at specific examples and explain the how \textit{CF-STAT} is used to identify the vulnerabilities in those programs. Finally we resent some general statistics for \textit{CF-STAT} usage and analysis.

5.1 Types of Vulnerabilities

5.1.1 Bypass Identification

There are various ways an attacker can bypass the authentication in the program. The code that authenticates the user inputs are referred to as mediators in this paper. To be able to determine if malicious user input can effect the sink without going through a mediator we first need to identify the mediators. The intuition is that a mediator will verify the authentication inputs, (for example user name and password) and verify them
using the specified policy, the policy can be configuration rules that specify the authentication behavior and it can also be the credentials database (for example a shadow file). Thus to identify potential mediators we find the statements that work on data that is tainted by the two types of sources mentioned above. Once we have a set of potential mediators we manually filter out the cases which are definitely not performing any kind of check or update operations on the data of concern and work with only the potential mediators. We make use of the taint label updates to specify that a particular user input has been filtered using one of the potential mediators. Thus the taint labels without this update are the ones that never passed through any kind of mediator before reaching the sinks, raising a flag of potential threat from the unfiltered input reaching the sinks.

The attacker can stage a bypass attack by taking advantage of various other flaws, like the mediator itself not functioning properly or has a trigger controlled by the attacker, or the verification output of the mediator is either overwritten or disregarded while providing access to the resources. We will look at some of these cases in detail with the examples later in this section.

5.1.2 Backdoor Identification

The attacker can gain access to the secured resources either by bypassing an authentication due to a flaw in the program or could also be due to deliberately added bypass functionality by the programmer. The type of vulnerability due to deliberately added pieces in the code that provide access to the attackers, are called backdoor vulnerability. To be able to identify these kinds of vulnerabilities in addition to checking if the mediator has filtered the inputs before reaching the sinks, we also need to check whether any trigger was set, before or after the mediator. These can be purely code based maneuverings (for instance a time bomb type backdoor where access could be provided after certain number of login attempt) or could be based on some hard-coded values in the code (where the user provided credentials are checked with some hard coded values or overwritten by them). There also could be many cases that can be missed in a source code, certain back doors may only manifest in the binary when the code is compiled. Since we are working with a decompiled binary, these cases will be seen by CF-STAT. Following are some categories of back doors that we will be looking for using CF-STAT: (i) special credentials. (ii) hidden/unintentional functionality and (iii) manipulation of security critical parameters [30], where in the first category certain user name and/or password can be defined for a special access purposes and are bypassed by the main authentication mechanism, can be exploited if known by the adversary. The unintentional or hidden functionality is where special access code is embedded inside the program to be able to bypass the authentication using certain special values, which could be directly the input coming from user or based on certain property of the input, for instance checking only the length of a password and last few hash characters. Manipulation of security critical parameters involves changing the parameters that are critical in terms of giving access to certain resources, for instance the parameter that is used to store the output of an authentication can be tampered
with to give access regardless of whether the authentication passed or failed.

### 5.1.3 Launcher Identification

A program can launch another process using commands from within the code. The attacker may also be able to launch its own malicious code by sending an unfiltered or a hard-coded value to such commands. The sinks in these kind of attacks are the commands like `system` and the `exec` family. To detect such attacks we check whether the user input influences these commands in any way and if the input is not filtered or is modified or overwritten to make the program launch the attacker process. If a hard-coded value is used for these commands, the user input does not taint the sinks in this case, we also check the reachability of the sinks to see if that is indeed reachable and can be executed by the attacker regardless of the inputs provided.

### 5.2 Analysis Approach

The following are the analysis steps we take while analyzing a given application for vulnerabilities.

1. **Source Identification:** First, for any vulnerability identification in a program we first need to identify the possible taint sources in the program. These may or may not be accessible by the attacker, initially we assume all these inputs are part of potential taint sources for the analysis with CF-STAT. The source identification is based on the application we want to analyze and the potential vulnerability type we need to infer, for instance some application requires network input to be identified as tainted source, while some may need even the configuration file used also identified as the taint source. CF-STAT also considers all the standard known input commands and collects them as potential taint sources, the analyst may choose to select from them and mark others not as taint sources.

2. **Control Flow Scope Identification:** We also identify the possible source of the program, the main function in most of cases and the sink function for cases where known. This helps us reduce the scope of analysis for larger programs. We compute the sub graph between source function and the sink function from the call graph of the program. And process only those functions that are part of this sub graph for taint propagation. This optimization is possible only because we analyze with predefined sets of tainted sources and propagate them through the program to find how it reaches the sinks.

3. **Identification and Filtering of Candidate Mediators:** We use the query mechanism discussed earlier to identify the sets of candidate mediators for the given program. The query used to identify a mediator again will be tailored to the application and the type of mediator we intend to identify. For input filtering based mediators we would look for functions that take in the input, policy and produce a string [31], for authentication based mediators, the output constraint need not be string, but should be used in a branch that dominates the sink operation but the sink operation does not post dominate the branch statement, i.e. the sink should not be reachable regardless the authentication check. Even after the property check for
mediators, the query may provide results that may not be relevant and thus need to be manually filtered out. We rerun the analysis with thus selected candidate mediators for further analysis.

4. **Sink Taint Verification:** We run the analysis with the above identified inputs, for taint sources, the start and end functions if available and the candidate mediators. *CF-STAT* then outputs the processed functions, the total number of nodes, the tainted nodes based on each label, the branch instructions that use tainted data in its predicates. If the sink is known we check the taint labels on the sinks to determine the data reaching the sink from each taint source and if all the taint sources were filtered. if the sink has a taint label that has not processed by any mediator, then it indicates that the input can reach the sink without any filtering. When such cases are identified, we flag them and the analyst can then examine the anomaly to confirm the vulnerability.

5. **Domination Check:** The *CF-STAT* also produces the branch instructions that use a tainted data in its predicate, although for the analysis we do not propagate the taint label further for these implicit flows due to taint explosion. The branches thus identified are then checked for dominance with regards to sink. For instance if a command like `system` is a sink we verify whether any of the tainted branch instructions dominates the sink operation. If so then we can determine the choice made based on user input is responsible for triggering the sink operation. If a dominance relationship is found between these branch instructions and sinks, we provide the information to the analyst for further examination and to determine if that is a part of vulnerability.

6. **Reachability Check:** Similar to the dominance check we also need to verify if the backdoor code, or the bypass code is if at all reachable in the call graph. To confirm if the code is part of dead code, we further analyze for possible accessibility and if none found then the vulnerability is determined to be not exploitable. Some of the paths can be lost due to incomplete decompilation and needs to be manually forced to interpret certain parts of the code. The reachability check is also important to detect an unintentional functionality when the sink execution is not dependent on any user input, but still executes when the program is run. This could also be based on some hard coded value in the program.

7. **Hard Coded Strings Check:** To check for special credential and hidden functionality kind of backdoors, we need to verify if a static value was used to determine the access for the user. To be able to verify this we identify all the constant values used with the data that was tainted by the user input. If a hard coded value is overwritten to the input variable before authentication or if the hard coded value is used to compare with the input value for some check then these operations are flagged and the strings are provided to the analyst to check for any suspicious values used in the mediators.

We will look at multiple examples that we analyzed using *CF-STAT* and will elaborate on the above approach as we deal with each case.
5.3 Case Studies

5.3.1 sshd

We analyze a modified sshd binary, which was provided as part of the DARPA VET [13] project. It is a server-side program for remote login using SSH protocol. The goal is to identify if a vulnerability is present in the binary that violates a specific rule. A rule can be broad like 'Check if the authentication can be bypassed' and 'If the user input can be used to launch a process' or can be specific like 'Check if file < file_name > is used for reading configuration' In this paper we will term the programs we test as articles. The rule for the sshd article that we analyze here is to check for a backdoor that can bypass the authentication sequence. Following our analysis approach described above, we deduce either the presence or absence of the vulnerability. LLVM compiled compatible bitcode file for the provided program is used as an input for the analysis. For sshd binary, we faced a few challenges in the process of lifting the entire binary in a compilable C code. As mentioned previously we use the Hex-Rays decompiler [6] to lift the binary in a C like code and then transform it to a compilable C code. In the process of decompilation, there are cases when the decompiler is not able to classify if the contents of the memory is code or data and thus fails to translate it to the C representation. In these cases we need to force them to detect that piece as a function and include it in the code. Once we identify and force these sections to the C code, we are then ready to make it compilable and generate a bitcode representation for our analysis.

1. Source Identification: The source taint identification as mentioned is tailored to the type of program we want to analyze. For sshd, and based on the rule provided we look for the inputs locations that would take the username, password and the policy file as an input. In sshd the analyst knows that the username and password will come from a network input, for instance a socket call. We also look for functions like getpwnam to check what variables are associated with the password data and record them as potential taint sources. We use Hex-Rays to review the code for these functions and locate the sources for the analysis. Given the functionality we also know that the password file /etc/passwd and /etc/shadow will be accessed for the credentials verification. These sources become the taint sources for policy in our analysis. The identified taint sources for this analysis are: Function: sub_11F48, Username: v5, Password: v3, Shadow File: v6)

The function name and the variable names will not have any intuitive notion given that it was lifted from the binary, thus the function names are generated based on the start location of the function in memory and variables are just name sequentially as v1, v2, etc.

2. Control Flow Scope Identification: For sshd we just have the starting point (i.e, the main function which is sub_D950 in this case) and there is no definite sink function yet, thus in this case, we select a random leaf function as the end point. We can also skip the specification for the first run and let it analyze the whole program, once we have better data on mediators and sinks, we can update these inputs accordingly for avoid reprocessing irrelevant code.
3. Identification and Filtering of Candidate Mediators: We run the analysis on sshd with the above inputs. To determine the candidate mediators for sshd, where the user name will be used to retrieve the password from the shadow file and then the user provided input will be verified with the credentials. Thus we are looking for the code sections which operate on data tainted with all three labels, i.e. username, password and shadowfile. We specify the appropriate query to the analysis asking for an intersection of all these labels. The analysis provides all the statements and the corresponding functions that work with the tainted data as per query. For sshd we get the following functions as possible candidate mediators: (sub_11F8, sub_51D10, sub_12004, sub_1A3EC, sub_1A4AC, sub_21AC8, sub_213BC, sub_25338) Out of these candidate mediators, we then again filter the functions that are not performing any kind of check, for instance there are functions which pass these values to other functions for verification and use the results, thus the statements are tainted with all the labels for them. There are various heuristics for filtering the candidate mediators, one is look for operations like compare, which indicate some sort of value checks. There are cases where these checks are performed in a library function, which could be a black box if we cannot apply the inter-module analysis, for those we check the if the intersection data is being used to make any choice, i.e. look for branching statements. These help us identify the code performing the checks. In sshd we identify function sub_51D10 as a candidate primary mediator, since it uses the crypt function for passwords and then performs a branching operations. The functions sub_12004 and sub_11F48 also use the tainted data in branch instructions, but we observe that they are part of the call path reaching the primary mediator where the actual check is performed.

4. Sink Taint Verification: Running the analysis with the above identified inputs we get the following results for sshd: The analysis generates a data flow graph for the program with a total of 123,189 nodes. The tainted nodes for each label is: username: 189 nodes and 11 branch instructions, password: 149 nodes and 10 branch instructions, and for policy file: 141 nodes and 8 branch instructions. The intersection query generates 125 instructions across 8 functions tainted by all three taint labels. We also compute the external calls that send out tainted data for inter-module analysis. For sshd we see 2 external tainted calls for strcmp function, one of which is tainted by all three labels and is used to compare the authentication result. The second call is tainted only by the label password and is used to compare it with a string. The approach steps 5 to 7 are performed based on the rule for each article.

For sshd we can consider the result of the mediator as a sink and verify the taint on the sink, the rule is to check for backdoor that allows to bypass authentication, thus first we need to see if the taint has reached the authentication result without the mediator check first. We don’t see this as the case in sshd. Second, we check for use of hard coded string with the tainted data. We get the following result for those: Strings used with username:

```
@.str.50 = private unnamed_addr constant [1 x i8] zeroinitializer, align 1
@.str.122 = ... [53 x i8] c"Packet integrity error (%d bytes remaining)
at %s:%d\00", align 1
```
The strings used with password are the similar to that of username. The constant string used with only the password tainted data is observed as suspicious and the corresponding code is verified for confirming the anomaly. Figure 5.1 shows the C code for the detected problem in the analysis. We see that the password is compared with a predefined constant string and the authentication result is being assigned based on this comparison. This leads us to conclude the existence of the backdoor vulnerability in sshd. The sshd being a huge program the analysis took about 4 hours to run the analysis that processes about 800 functions in the program.
5.3.2 ls.coreutil

The program ls is part of the coreutils provided with a Linux package, and is used for listing files. It takes various arguments to specify things like what directory and format to display. The goal in ls is to determine if the user input triggers any unintentional malicious behavior like creating or writing any files in the program. We again follow the same analysis approach steps.

1. **Source Identification:** The taint source determination is pretty straightforward in ls program given there is no network interface and the specification mentions input from the command line. We identify the command line inputs and their assignments as taint sources for the analysis of ls.

2. **Control Flow Scope Identification:** The start is the main function which is identified as `sub_9FE4`. We identify functions that contain operations of the family fopen and fwrite or puts etc. as sink functions. Since this is a small program and we have multiple functions that can be sink, we still choose a leaf function as a control flow sink to make sure we cover the entire program. We eliminate step 3 in this analysis since there is no authentication involved and thus no mediator identification is required.

3. **Sink Taint Verification:** Running the analysis with the identified inputs we get a data flow graph of 43691 total nodes. The command prompt input taints about 703 nodes and the tainted data is used as predicate in 46 branch instructions. There are 5 tainted external calls in the program that use functions like `strcmp` and `strlen` on the tainted data. On a preliminary analysis we do not see any taint reaching the sink functions i.e. the file open and file write functions, but considering the fact that we don’t perform any implicit taint propagation we need to study the implications of the tainted branch instructions as directed in step 5.

4. **Domination Check:** In domination check we verify if there are any tainted branches that dominate the sink functions, thus if any of the file open or write operation is performed as result of choice made based on a user input then we can infer that the user input indeed triggers some malicious behavior given if ls does not support that functionality. We observe that the following tainted branch instruction in function `sub_9FE4` has a taint label from the command prompt assignment.

```
br il %cmp99, label %if.then.101, label %if.end.103, !dbg !3769,
!TAINT_CmdAssign !3725, !CmdAssign !3725
```

And that it dominates the block with sink operation:

```
if.then.101: ; preds = %land.lhs.true.96
%call1102 = call i32 (...) @fopen64(), !dbg !3770
store i32 %call1102, i32* @dword_2CF1C, align 4, !dbg !3771
br label %if.end.103, !dbg !3772
```

Then in a subsequent function, it checks if the file was created, if so then the program dumps some data in the file. Figure 5.2 shows the concise C code for the vulnerability. This process could potentially be used for dumping any sensitive data in the file. Thus we establish that the program is indeed doing some
unintended operations. The analysis for *ls* takes about 2 hours and processes about 196 functions in the program.

### 5.3.3 Agetty

Agetty is a Unix program that manages physical or virtual terminals. It prompts for username and runs the login program to authenticate the user, when it detects a connection. We verify two rules in agetty, *rule 1:* Can agetty launch any process outside of the default login program with the default login options and *rule 2:* Agetty should not allow an attacker to launch processes outside of the default. Essentially both the rules mention that agetty should not be able to launch any unintentional script, this may or may not be triggered by the user.

1. **Source Identification:** The taint sources for agetty are from the command prompt when the program is launched. Based on the rule we want to check if the default command prompt options lead us to open non default login program. Thus the we start with the taint for the command line arguments namely the argc and argv arguments to check if any of the user input reaches the launch and can modify it. In the binary lifted code, we need to select the returned values from the function optarg, which deals with interpreting the command line inputs and assigns the options and inputs into a structure. In our analysis we find that and select that as the taint source.

2. **Control Flow Scope Identification:** The control flow scope for this program should also start at the main function and given that there are more than one sink functions we select a leaf function for the analysis to have maximum coverage. We identify the main function as `sub_9CD8`. The sink functions for this program and the rules provided would be functions with commands that can launch another process, like exec family, system, fork, etc. There are more than one instance of these sink functions, so we select the leaf function `sub_C42C` as the control sink for the analysis.

3. **Identification and Filtering of Candidate Mediators:** The mediator for these rules would mean that input provided by the user is filtered or checked in some way before it reaches the launch command. For this we do not have a policy input to fire a query on but we check for heuristics to identify instructions...
that take in string and produce a string \[31\] where the input string is tainted with the user input taint label. We found about 13 instances of calls to functions like strcmp, strspn and strlen, etc which use the tainted strings but on further examination found that none of them are performing any valid filtering on the input.

4. Sink Taint Verification: We run the analysis for agetty with the command line inputs as sources and the other identified inputs as discussed above, and generate a data flow graph with 7002 total nodes, where the argc taints about 56 nodes and the argv options taints about 309 nodes. There are about 29 tainted branch instructions for the input taint. We check the taint associated with all the sink functions and identify which ones are tainted by the user input. We find that the execv command directly uses the user provided string and executes it. This is executed with the \(-l\) option for specifying the login program, thus establishing the first rules where any other login program can be executed by the program agetty.

\[
%\text{call1134} = \text{call i32 } \text{@execv(i8* } %\text{700, i8** } %\text{argv) } #3, !\text{dbg } !\text{2978, !TAINt_CmdOption } !\text{917}
\]

5. Domination Check: Then we look for all the sinks which are dominated by a tainted branch instructions. We see that the command \text{system} is being executed with a hard coded string \text{(}/bin/sh -c /tmp/agetty; /bin/rm -f /tmp/agetty\text{)} and is dominated by a tainted branch instruction. Following are the branch instruction tainted that dominate the sink:

\[
br \text{i1 } %\text{tobool858, label } %\text{if.end.866, label } %\text{land.lhs.true.859, !dbg } !\text{2565, !TAINt_argc } !\text{1041, !TAINt_argv } !\text{1041, !TAINt_CmdOption } !\text{1041}
\]

\[
\text{land.lhs.true.859: } ; \text{ preds } = %\text{if.end.846}
\]

\[
br \text{i1 } %\text{tobool863, label } %\text{if.end.866, label } %\text{if.then.864, !dbg } !\text{2572, !TAINt_argv } !\text{1041, !argv } !\text{1041}
\]

The dominated sink function call:

\[
\text{if.then.864: } ; \text{ preds } = %\text{land.lhs.true.859}
%\text{call865} = \text{call i32 } \text{@system(i8* getelementptr inbounds ([49 x i8], [49 x i8]* @.str.37, i32 0, i32 0)), !dbg } !\text{2573}
\]

On the corresponding source code examination we identify the condition to trigger the system call is \text{argc} \text{> 2} and \text{argv[2]} = “tty1”, thus the script can be launched by starting agetty using a command similar to the following: \text{/sbin/agetty --nocleartty2}. The attacker can perform any operation by adding it to the script and once the script is run the program also removes the traces of the script from the system. Thus confirming the presence of unintentional script launch vulnerability in agetty.

5.4 Other Analysis

We used \text{CF-STAT} to perform experiments on about 50 and more programs provided as a part of the DARPA VET program. We provide summary statistics on some of those in this section and discuss some interesting
cases encountered while analyzing some of these programs. There were also C++ programs in the set which added another set of challenges when trying to lift them to C- like language and making it compilable. The scoping and inheritance constructs added complexity in trying to get a correct representation of the decompilation. We address some of the issues by translating them to simpler types, but overall it has been still an ongoing work to be able to lift the C++ binaries while ensuring its correctness. We analyzed programs from a very small application to huge ones with analysis time ranging from few seconds to couple if hours. The sshd article discussed above which took about 4 hours of analysis run time is one of the larger programs that we analyzed. There are smaller programs like RFID door which took about 6 seconds for analysis.

<table>
<thead>
<tr>
<th>Program</th>
<th>Vulnerability Type</th>
<th>Malice Found</th>
<th>Total Node in Data Flow Graph</th>
<th>User input Tainted Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>sshd</td>
<td>Authentication Bypass</td>
<td>Yes</td>
<td>80824</td>
<td>201 username 126 policy 113 password</td>
</tr>
<tr>
<td>RFID door</td>
<td>Launch</td>
<td>Yes</td>
<td>5881</td>
<td>322 Badge ID</td>
</tr>
</tbody>
</table>

Table 5.1 provides overview of some more of the programs that we analyzed with \textit{CF-STAT} which covers identification of vulnerabilities like authentication bypass, backdoor, launch and information leak. The program sizes vary from one of the big ones we analyzed (sshd) and the smaller ones (RFID door) which is also a C++ code. We could fix the lifting issues in this binary with minimal effort due to its size. The analysis that we perform is semi-automated and hence there is considerable amount of human effort involved in analyzing the binaries. The decompilation is simplified using automated scripts to fix some common error, thus reducing the time required in generating the input format for the analysis. The source sink identification time is dictated by the familiarity of the program and the known knowledge if its input interfaces. If a proper documentation is not available for the program this process can take more time to identify relevant input sources from all the available default input statements. The efforts towards automating and aiding the analysis set up process will be an ongoing work and part of future work.
Chapter 6

Conclusion and Future Work

In this thesis we introduced the tool *CF-STAT* and reviewed the semi-automated process of analyzing binaries to determine the presence or absence of vulnerabilities. We refined *CF-STAT* with various precision feature to make it easier to identify the anomalous code with as few false positives as possible. The major challenge was that the types not being completely inferred in a code lifted from binary and thus the implementation of pointer analysis, field and flow sensitivity was challenging. With all the heuristic refinements to address that *CF-STAT* was successfully able to generate appropriate tainting information to aid the analyst in identifying the behavior of the user input with respect to security sensitive sinks. We analyzed more than 50 programs with and without malice, The decisions are well guided in the presence of a malice but in cases of absence of malice the analyst needs to be very careful of the corner cases while directing the analysis. *CF-STAT* helps in establishing certain rules that will help analyst make the decision about the vulnerability.

*CF-STAT* currently heavily relies on the manual setup of the analysis and one direction of future work will be towards reducing this human effort and automating most of the setup process for the analysis. We have been identifying all the default input sources and the possible sink functions like system calls or write operations, although the selection from these is still based on the rule and analyst’s interpretation. On identification of the anomalous case, it is again manual effort to confirm the malice, there is a lot of scope in automating this and being able to answer certain rules in an automated way as part of future work. This would also benefit by some element of dynamic or symbolic analysis to prove or disprove the rules. There are various directions the tool can be extended with the goal of automating the entire process with high precision.
Bibliography


