THESIS

DETECTING NON-SECURE MEMORY DEALLOCATION WITH CBMC

Submitted by

Mohit K. Singh

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Fall 2021

Master's Committee:

Advisor: Vinayak Prabhu
Co-Advisor: Indrajit Ray

Sudipto Ghosh
Indrakshi Ray
Steve Simske

Copyright by Mohit K. Singh 2021
All Rights Reserved

ABSTRACT

DETECTING NON-SECURE MEMORY DEALLOCATION WITH CBMC

Scrubbing sensitive data before releasing memory is a widely recommended but often ignored programming practice for developing secure software. Consequently, sensitive data such as cryptographic keys, passwords, and personal data, can remain in memory indefinitely, thereby increasing the risk of exposure to hackers who can retrieve the data using memory dumps or exploit vulnerabilities such as Heartbleed and Etherleak. We propose an approach for detecting a specific memory safety bug called Improper Clearing of Heap Memory Before Release, referred to as Common Weakness Enumeration 244. The CWE-244 bug in a program allows the leakage of confidential information when a variable is not wiped before heap memory is freed. Our approach uses the CBMC model checker to detect this weakness and is based on instrumenting the program using (1) global variable declarations that track and monitor the state of the program variables relevant for CWE-244, and (2) assertions that help CBMC to detect unscrubbed memory. We develop a tool, SecMD-Checker, implementing our instrumentation based algorithm, and we provide experimental validation on the Juliet Test Suite that the tool is able to detect all the CWE-244 instances present in the test suite. The proposed approach has the potential to work with other model checkers and can be extended for detecting other weaknesses that require variable tracking and monitoring, such as CWE-226, CWE-319, and CWE-1239.

ACKNOWLEDGEMENTS

I am deeply grateful to Dr. Vinayak Prabhu for his support as my research advisor and for ensuring that we followed through with this thesis to completion. I highly appreciate his invaluable advice and assistance at every research stage, from fine-tuning the research to giving prompt feedback on my reports.

I would also like to express my sincere gratitude to Dr. Indrajit Ray, Mr. Stefano Righi, and American Megatrends International for initiating, commissioning, and supporting this project.

I would also like to extend my sincere thanks to my committee members, Dr. Sudipto Ghosh, Dr. Indrakshi Ray, and Dr. Steve Simske, for sharing their knowledge, insightful comments, and suggestions on this thesis.

Finally, I would like to thank everyone who supported me to bring the best out of me and helped me by sharing their expertise to complete the thesis.

TABLE OF CONTENTS

ABSTRACT . . . ii
ACKNOWLEDGEMENTS . . . iii
LIST OF TABLES . . . vi
LIST OF FIGURES . . . vii

Chapter 1 Introduction . . . 1
1.1 Secure Memory Deallocation . . . 3
1.2 Analysis Tools . . . 6
1.3 Our Contributions . . . 8
1.4 Related Work . . . 9
1.5 Thesis Organisation . . . 11

Chapter 2 Preliminaries . . . 12
2.1 Common Weakness Enumeration . . . 12
2.2 Shallow and Deep Copy . . . 13
2.3 Model Checker . . . 13
2.3.1 Software Model Checkers . . . 14
2.4 CBMC . . . 16

Chapter 3 SecMD-Checker: Detecting Non-secure Memory Deallocation . . . 18
3.1 Wiping Confidential Information . . . 18
3.2 Overview and Scope of our Approach . . . 19
3.2.1 Assumptions . . . 21
3.2.2 Limitations . . . 23
3.3 Instrumentation Algorithm . . . 24
3.3.1 Subroutines of the algorithm . . . 32
3.4 Realloc Handling . . . 43
3.5 Example: Instrumenting multi-function program . . . 44
3.6 Dynamic tracking SecMD-Checker . . . 52
3.6.1 May-Alias: False Witness to a Positive . . . 54
3.7 Bad Compiler Optimization . . . 55
3.8 Recommended practices . . . 56
3.8.1 Checking Data after Null Character . . . 56
3.8.2 Storing Hard coded or unprotected Key . . . 57
3.8.3 Wiping non-volatile sensitive data on Restart . . . 58

Chapter 4 Experiments . . . 59
4.1 Environment . . . 59
4.2 Juliet Test Suite . . . 59
4.3 Result for detecting other memory related CWEs . . . 61
4.3.1 CBMC Properties . . . 62

4.4 SecMD-Checker Test Results 63

Chapter 5 Conclusions and Future Directions 64

Bibliography 65

LIST OF TABLES

4.1 Environment. 59
4.2 Relevant Common Weakness Enumerations. 60
4.3 Test Results: Detection of memory related CWEs 62
4.4 Test Results: Detection of CWE-244 63

LIST OF FIGURES

1.1 NVD Vulnerability Growth Statistics. 1

2.1 ShallowCopy 13
2.2 DeepCopy 14
2.3 Software Model Checker. 15
2.4 CBMC Architecture. 17

3.1 Wiping Confidential Information before freeing 19
3.2 Overview of our Approach. 20
3.3 Generation of Graph Nodes. 37
3.4 Generation of Graph Edges. 38
3.5 Code Formatting: Adding braces to one-line conditional statements. 39
3.6 Code Formatting: Broken braces and splitting statements by newline. 40
3.7 Example: Generated graph vertices. 46
3.8 Example: Generated graph edges. 47
3.9 Checking Data after Null Character. 57

4.1 CBMC Test Result on Juliet Test Suite. 61

Chapter 1

Introduction

Software is omnipresent; it drives all the appliances that we use throughout the day like television, microwaves, telephone, automobile computer system, and many more. Software has become an integral part of our lives; thus, making it vulnerability or bug free is more pivotal now than before. All software execution paths should be predictable and consistent in a system as per the design and requirement, with no surprises to the users. Software testing, which aims to improve quality by identifying and removing the product's errors and checking if the product is not doing anything unintended is, consequently, extremely important. However, even after the widespread adoption of software testing, a drastic increase in the number of reported vulnerabilities can be seen every year (depicted in Figure 1.1). The bar chart is generated by the National Vulnerability Database (NVD) search engine. These vulnerabilities can cause car model throttle control failures [1], malicious take over of automobiles [2, 3], untrusted modifications in bootloaders [4] and several others.

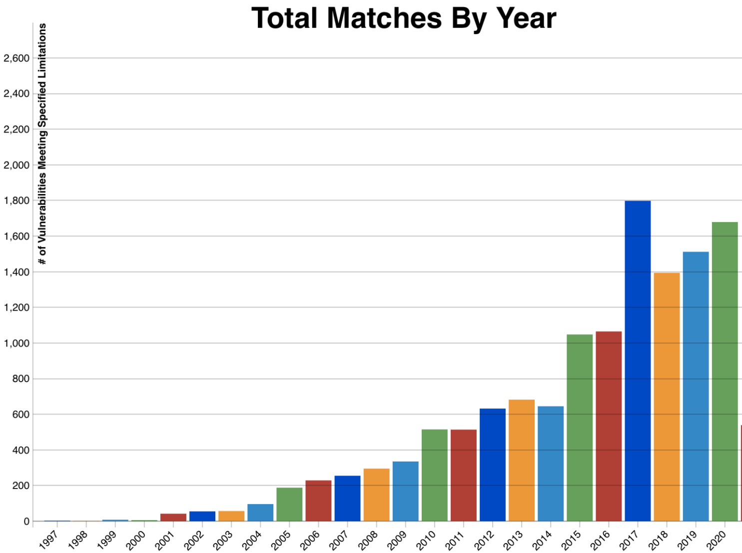

Figure 1.1: NVD Vulnerability Growth Statistics.

Vulnerability in the software can potentially be exploited to cause security breaches in itself and any dependent modules. The exploitation of software vulnerabilities could lead to loss of confidentiality, integrity, and availability. Furthermore, fixing the bugs adds an overhead to the system since it requires all dependable functionalities to be turned off during patching. Therefore rigorously detecting the bugs in an early development phase is essential to software. Otherwise, it can substantially impact security, including system instability, information leakage, and system failure. For example, a complex system like a computer contains three primary software layers: firmware, operating system (OS), and application layer. Each layer has its own importance, as firmware works as a software interface between the operating system and hardware. So if a vulnerability is present in the firmware, it can cause a security breach to the OS because the OS is dependent on firmware for interaction with hardware. Fixing a vulnerability in firmware will require the OS to stop its functionality. Therefore, to decrease the overhead to the system, every running software should be thoroughly tested before the final production phase.

Many applications are written in C because it is a middle-level language providing the features of both high-level and low-level languages. Low-level languages allow the program to access the system resources directly, making it easy for a programmer to create software that interacts with hardware efficiently and effectively. This is the reason why most drivers and kernels are written in low-level languages. High-level languages are mainly used for better comprehensibility and faster software development. Even though C is widely used, its benefits come with a trade-off: it is a memory unsafe language. It allows users to use arbitrary pointer arithmetic with pointers implemented as direct memory addresses with no provision for bounds checking. Memory unsafe languages are susceptible to errors such as buffer-overflow, buffer-overread, invalid memory access, memory leak etc. These errors increase vulnerabilities in systems, consequently exposing them to undetectable disclosure of confidential data, inconsistent and unpredictable program behavior, run-time errors, and system crashes. In this era, a significant number of such

errors are still found in software that are rigorously tested before the production release. Testing methods are further explained in section 1.2. Different types of errors and weaknesses in software and hardware have been categorized into a community-developed list known as Common Weakness Enumeration (CWE); We will further discuss CWE in section 2.1.

1.1 Secure Memory Deallocation

Thorough cleaning of sensitive data such as cryptographic keys of Heap Memory before release is widely accepted practice for developing secure software [5, 6]. Otherwise, it could lead to CWE-244: Improper Clearing of Heap Memory Before Freeing ('Heap Inspection') [7]. For example, all versions of Dell RSA BSAFE Crypto-C Micro Edition before 4.1.4 were affected by the CWE-244. This led to CVE-2019-3733 vulnerability [8] that could potentially allow a malicious remote user to extract the sensitive information leaving data at risk of exposure. Aim of this thesis is to focus on the model checking approach for detecting CWE-244. Reasons for CWE-244 occurrence:

- Confidential information (such as a password or encrypted key) not wiped from memory before freeing.
- Usage of the "realloc()" function. The "realloc()" function is used to extend or shrink the allocated memory block. It copies the data from the old memory block and pastes it into the new memory block. Making the old memory block inaccessible to the program. Therefore, "realloc()" usage makes program susceptible to heap inspection as the original memory remains intact even after memory reallocation.

Due to CWE-244, an attacker can retrieve confidential information by taking a memory dump or utilizing other methods.

Security practices like zeroing out memory before freeing, also known as scrubbing or secure memory deallocation, are often overlooked by the programmer when it comes

to protecting sensitive data. These practices are frequently not followed in applications such as web browsers and web servers that currently handle most of the world's sensitive data, such as passwords and confidential documents. Consequently, sensitive data could be found throughout the user and kernel memory and stay there for an indefinite period [9]. Sensitive data buildups are formed because data can remain in the memory for an indefinite time. Therefore, sensitive data buildups increase the risk of exposure, the chance of revealing information, and attack severity in case of system compromise. Generally, the natural lifecycle of data ends when new data overwrites the old data during the subsequent allocation of the same memory location. However, waiting for the data to be overwritten produces a data lifetime of 10 to 100 times the minimum lifetime, which is from the first write to the last read. A lifetime of sensitive data could be reduced within 1.35 times the minimum possible data lifetime by zeroing the data at deallocation [10].

Sensitive information can be exposed by running a program on a system with access privileges to read the system memory. Password managers like 1Password7 are used to store and retrieve sensitive information. However, this was breached [11] by running programs on the system that allowed the attacker to read the system memory. Consequently, program revealed the stored passwords because 1Password7 never scrubbed the master key, individual passwords, or secret key before freeing the respective memory.

Leaks could also happen by exploiting other vulnerabilities. For example, Heartbleed is a serious vulnerability present in OpenSSL cryptographic software library. This library is used on a wide scale on the web for providing secure communication. SSL protocol includes a heartbeat option, which allows one computer to know that the computer at the other end is still online by sending a short message and getting a response back. It was found that a cleverly formed heartbeat message could be sent to the computer, allowing the attacker to exploit the system by buffer overread vulnerability, where more data is read than should be allowed. Therefore, this attack can reveal the sensitive information buildups from the server's memory (RAM) [12]. This vulnerability occurred because of

the missing bound checks. A similar buffer overread problem has been encountered in curl [13].

Program errors could lead to accidental data leaks. For example, Etherleak [14] occurred in ethernet Network Interface Card (NIC) device drivers. It occurred due to incorrect implementations of RFC requirements and poor programming practices. Therefore, this attack can expose the portions of kernel memory. Etherleak could be violated quickly by sending an ICMP echo message to the vulnerable machine, which returns the portions of kernel memory in the padding of the response messages.

Unintended feature interactions could also result in an accidental leak through core dumps or logging. In one study, 2.5 million crash reports were examined [15]. This revealed that memory dumps from the crash reports contained an extensive amount of private data, including session Ids, clear passwords, and other personal information. For instance, Mozilla could leak bits of uninitialized memory when rendering certain types of truncated images [16]. Leaking of private kernel information from previous executions to the userspace processes [17]. Solaris leaked passwords that were left in memory by dumping core files to a directory accessible via anonymous FTP [18].

Sensitive data remnants also make the system susceptible to Cold Boot attacks [19] where an attacker has physical access to a machine and can lay their hands on the encryption keys. It can stay in memory for an indefinite time if the information leaks from the disk by paging or any other technique, extraordinarily expanding the risk of exposure. Even information that has been overwritten on a disk can be recovered [20].

Non-secure memory deallocation in software that stores encryption keys on Read-Only Memory (ROM) has a more severe impact. The purpose of using flash memory, ROM, EPROM, or EEPROM is to make sure that anything stored in them cannot be changed, i.e., making it immutable. Hence, specific system design such as Root-of-trust requires passwords/keys to be stored in ROM to be utilized during the software authentication procedures. Therefore, storing sensitive data in ROM ensures that the malicious

entity cannot alter the software. However, for authentication, we need to bring that key in the RAM, and if we carelessly free the memory without zeroing them out, keys can be exposed by Heap Inspection. In such cases, companies could require physical replacement of the chip holding the software or be reprogrammed through a particular upgrading procedure [21]. Physical replacement or reprogramming the chips is expensive and a hectic task as it requires all the component functionalities and dependent services to be turned off during the procedure. Therefore, this makes CWE-244 a serious weakness in a system.

As explained, data could be exposed in numerous ways. Additionally, we have to consider that the system could be compromised and could have memory leaks. Therefore, rigorously detecting the data exposure weakness in early development is crucial as the weakness could lead to a vulnerability. Hence, it is incumbent to detect sensitive data exposure in a system at the early development phase.

1.2 Analysis Tools

Software Testing has become an integral part of the software development process. It is performed to check if the software product is free of defects when delivered to the customer. It helps to prevent bugs, improve performance, and reduce development costs. Software testing involves multiple types of tests [22] such as integration, unit, functional, acceptance, performance, stress, regression, security, and many more. These tests can be done in a manual or automated manner. A manual approach is an expensive option and not always practical, as it is a very time-consuming process. Therefore, tests are generically automated, and their testing reports serve as information regarding the quality of the software product.

Security testing is done to reveal weaknesses, vulnerabilities, threats, and risks in software. A few of the famous security tools are Metasploit [23], Nessus [24], and Snort [25]. There are several techniques for bug detection such as static analysis, dynamic

analysis [26], model checking [27] or a combination of these techniques. Static Analysis and Model Checking are used to detect bugs without running the program. Static analysis determines run-time properties by inspecting code structures, whereas model checking explores the relevant computational states. Both techniques have their own tradeoffs [28].

Static program analysis examines the text of a program statically without attempting to execute it. Static analysis constructs an abstract representation of the program behaviors and examines its states. Generated representation is an approximation of the parsed program. Therefore, examination of these created states can expose the hidden vulnerabilities. Static analyzers consider the run time properties/errors at compilation time automatically. They do not require any code instrumentation or supervision. It is much faster than manual auditing and saves time for the auditor to learn the weaknesses and vulnerabilities. A few of the benefits are faster execution speed, less level of environment knowledge required before the testing, ability to cover all the code paths, and easy addition of properties.

Model Checking [29, 30] computes the run-time states of the program without actually running the program, which is further utilized to check whether a specific property/specification holds for the program. If the program is small or has a finite number of states, the model checker can do an exhaustive analysis on the states. However, if the program does not have a finite number of states, then approximations are needed. Otherwise, it could lead to a state explosion problem. With the approximations, we can provide assurance for error/bug detection within the defined bound for the checked property. Furthermore, if a particular property is violated, then the model checker provides an execution path (trace) that leads to the specific violation, also referred to as a counterexample. Model-checking requires a deep understanding of the system because it requires a carefully crafted environment model for checking the properties. A difference between model checker and static analysis is that model checking executes only those paths that are explicitly triggered by the generated model. Model checker performs better when it comes to checking

for more affluent set properties, finding the corner cases, and exploring the system's involute behaviors. One of the bounded model checkers is the C-Bounded model checker (CBMC) [31] created by Daniel Kroening at Carnegie Mellon University. Capabilities of CBMC have been showcased on Common Weakness Enumerations (CWEs) and Amazon Web Services [32, 33]. Recent advances have made it possible to use both the techniques, static analysis and model checking on large systems. Although both approaches have significant differences, they are well suited in detecting errors/bugs in the programs [34].

1.3 Our Contributions

In this work, we propose an instrumentation based approach for detecting non-secure memory deallocation and develop a tool – SecMD-Checker – implementing the proposed algorithm. Our tool takes two inputs: (1) compilable C programs and (2) the name of a pointer to a location holding sensitive data. The tool then checks if this original memory location, and other memory locations that have been copied from it, are always zeroed out before they are freed.

Our algorithm has three main phases: (1) first, it performs a coarse flow-insensitive interprocedural static analysis on the program to construct a set of pointer variables that could point to memory locations containing sensitive data; (2) then, it instruments the program by inserting required variable tracking, monitoring, and assertion logic, utilizing the set of variables inferred in the previous phase. (3) finally, the tool invokes the C Bounded Model Checker (CBMC) to check for assertion violations in the instrumented program – an assertion violation, in this case, is a probable instance of non-secure memory deallocation.

Our tool implementation can handle the original tracked variable's shallow copies (references) and also deep copies (copy by value). To handle shallow copies, we track the memory address pointed by the variable name, not the variable name. Moreover, we treat

each deep copy as an individual variable and do instrumentation for both the input and deep copies. While we use CBMC in our work, our algorithm is also applicable to other model checkers and static analyzers, with appropriate minor changes to the tool-specific assertion logic code.

For experimental validation of our technique, we utilized the Juliet Test Suite [35], a collection of test cases in C/C++ classified under different CWEs maintained by NSA. We ran SecMD-Checker on the 72 CWE-244 test cases present in the Juliet Test Suite. Our tool correctly pointed out the functions which had non-secure memory deallocation instances in all the cases. Moreover, SecMD-Checker did not raise any false alarms in functions that did not have non-secure memory deallocation.

1.4 Related Work

Computer scientists have done their research on how the Static Analysis can be used for finding Common Vulnerabilities and Exposures (CVEs) in stripped firmware images [36]. They were able to find vulnerabilities in real-world firmware with high accuracy efficiently. By utilizing static analysis, researchers can also analyze the embedded-device firmware by modelling and tracking multi-binary interactions. In their approach, they propagated taint information between binaries to detect non-secure interactions and identify vulnerabilities [37]. Studies have been conducted focusing only on individual vulnerabilities such as Mirai Exploitable Vulnerability [38], and Intrusion detection [39].

Recent research suggests that model checking is viable for finding and eradicating security bugs quickly [40]. Evolution in model checking has shown that model checkers can be used on the whole Linux Distribution for security violations and can become an integral part of the software development process [41], where researchers make use of MOPS [42]. Kronos is a model checking tool [43] that is available for real-time systems which includes real-time communication protocols [44, 45], timed asynchronous circuits [46], and hybrid systems [45, 47]. SMV model checker [48] has been used to

analyze a specification of a software system for aircraft collision avoidance [49] and network vulnerabilities [50]. SPIN [51] and DiVinE [52] could be used on multi-threaded software [53, 54]. DiVinE could also be used efficiently on large scale systems [55, 56].

CBMC is a bounded model checker for C, C++, and Java programs. It can verify the absence of violated assertions under a given loop unwinding bound, array bounds (buffer overflows), pointer safety, and arithmetic exceptions. It can also provide a counterexample if the assertion is violated. Research has been conducted on CBMC in an industrial setting where they proved initial boot code in data centers at AmazonWeb Services is memory safe [32]. CBMC could be effectively and efficiently be used for detecting CWEs [33]. Our work applies to any sound model checker.

Research has shown that limiting data lifetime can reduce the risk of sensitive data exposure when compromises occur. Additionally, this problem should be addressed across all levels in the systems [57]. Zeroing the memory before deallocation or within a specific time period can reduce the data lifetime within 1.35 times the minimum possible data lifetime; [10] proposes a way of ensuring this by making changes in the compiler, libraries, and kernel. However, It is not always practical, and we cannot just believe these changes will be present in kernels, compilers, and libraries in the application execution environment. This approach can be practical only if all existing systems consist of compiler, kernel and library changes, which is not the case for all the systems. Therefore, it behooves us to ensure that individual software is not increasing the lifetime of data unnecessarily.

Aliasing occurs when the memory location can be accessed by more than one name. There exist two types of aliases `must-alias` and `may-alias`. `must-alias` are distinct names for the same memory on every executable path, whereas `may-alias` are distinct names for the same memory that may exist on some executable path. Alias analysis is a research area for seeking out all the aliases in the program. Many techniques are present for alias analysis, such as Interprocedural Pointer Alias Analysis [58], Type-Based Alias Analysis [59], and Context-Sensitive Pointer Analysis [60].

1.5 Thesis Organisation

In Chapter 2 we give a brief background about the software model checker and terminologies that will be required to understand our work. In Chapter 3, we describe a software weakness (CWE-244: Heap Inspection) that should be addressed while writing software that inherently cannot be detected by the model checker. This Chapter also describes our instrumentation approach for enabling the model checker for detecting CWE-244 efficiently. In Chapter 4, we provide experimental validation of our proposed solution by evaluating our tool SecMD-Checker on CWE-244 examples from the Juliet Test Suite. Finally, Chapter 5 concludes the thesis and provides the future direction.

Chapter 2

Preliminaries

In this chapter, first, we discuss the Common Weakness Enumeration (CWE) in brief. Next, we discuss Shallow and Deep copy. Next, we will provide an overview of a model checker.

2.1 Common Weakness Enumeration

Weaknesses are faults, bugs, flaws, or other errors in software or hardware implementation. If left unaddressed, they could result in systems, networks, or hardware being vulnerable to attacks. Weaknesses, attacks, and environmental conditions are combined to create exploitable vulnerabilities in software systems which are known as Common Vulnerabilities and Exposures (CVE). CWE is a community-developed category list for software and hardware weakness types. It was created with a purpose in mind that was to educate software and hardware architects, designers, programmers, and acquirers on how to eliminate the known types of security weaknesses before products are delivered. CWE works as a common language, measuring matrix and baseline for weakness identification, mitigation, and prevention efforts. CWE has over 600 categories, including classes for race conditions, buffer overflows, non-secure random numbers, and hardcoded passwords which have gone through many refinements. High-quality tools and services are being developed to find security weaknesses that benefit from the CWE listing as it serves as a measurement matrix. The MITRE Corporation currently maintains CWE. A detailed CWE list is available at MITRE website [61].

2.2 Shallow and Deep Copy

Shallow Copy stores the reference of the object to the original memory address. It only clones the reference, not the actual object. Therefore, any changes done on the shallow copy reflect in the original object. A shallow copy is quicker because it only copies a reference of the object. If a shallow copy is created of an object 2.1 then the user can free the allocated memory using any reference object. After freeing the memory, none of the reference pointers can be used to access the original data.

Figure 2.1: Shallow Copy.

Deep Copy truly clones the data from the original memory address. A deep copy of an object does not share the same reference; thus, any changes done to either object do not affect the other. Deep Copy is slower because it copies the actual underlying object value. If a deep copy is created of an object 2.1 then the user must free the allocated memory for all the addresses separately. Otherwise, it could lead to a memory leak.

2.3 Model Checker

Model-checking is one of the successful approaches for formal verification, which can verify if a system satisfies the desired property. Users define properties to ensure that our system always remains within a set of finite behaviors, i.e., the undesirable state never

Figure 2.2: Deep Copy.

happens. In model checking, system design or requirements are referred to as models, and specifications that need to be satisfied by the system are called properties. Properties can be defined as the scenario or the action that should be performed mandatorily at any point of time during the system run. Model checker outputs success if the system satisfies the property and generates a counterexample otherwise. A counterexample explains why the systems do not satisfy the property or why the property is violated. If our system is a program/code, then a counterexample will be a trace of program execution, leading to a scenario where the specified property does not hold. By examining the counterexample, the user can find out the source of error in the model and can fix/correct it. Model-checking has shown great promise in detecting security bugs or vulnerability [32, 62–64]. They are being used in many domains varying from car automation, IoT, textile industry, and many more.

2.3.1 Software Model Checkers

Software model checkers can be easily explained if we divide them into modules, i.e., 1) parsing and program transformation, 2) instrumentation and 3) verification, as shown in Figure 2.3. In the first step, the program is parsed and transformed into a model which the verifier can recognize. After the model has been generated, the model checker instruments the code according to the property the user wants the system to

follow. Global variables, assertion logic and pattern matching transitions are defined in the code during the instrumentation phase. This instrumentation defines a finite state in the form of a program, where the state changes when a particular pattern is matched. Pattern matching can trigger the assertion logic during failure and move the system to the error state, resulting in a property violation. Finally, instrumented code is passed to the verifier for checking if the defined properties are violated in the system. The error trace is produced if the property is violated.

Some model checkers enable users with a mechanism to define a property that could save the efforts required by the user to do the instrumentation. Blast enables the user to define a property utilizing a Spec tool [65]. If such a mechanism is absent, then the property can be directly defined by changing the source code. However, this approach needs deep knowledge on how model checkers internally work at the code level. Otherwise, users need to instrument the code by treating the model checker as a black box.

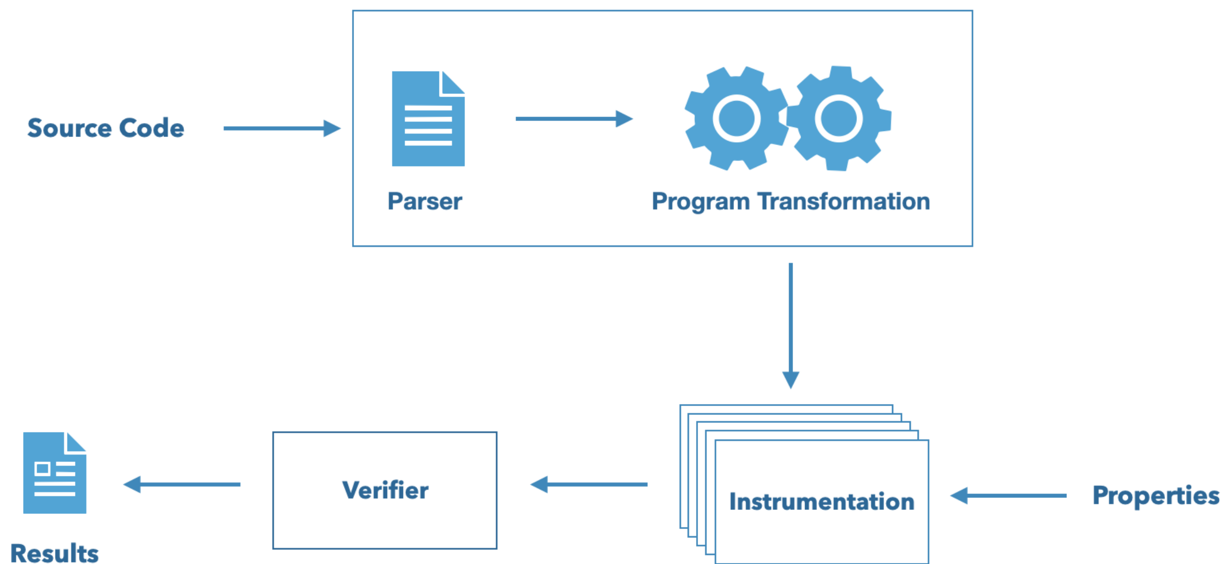

Figure 2.3: Software Model Checker.

Internal functioning of CBMC and property definition can be understood by [63]. In our research, we have used CBMC [31] for running the experiments. We created two environ-

ments for running CBMC on the CWEs present in the Juliet Test Suite [35] provided by the MITRE. Multiple weaknesses were researched that were not detected by the CBMC and can leave the system vulnerable and exploitable. We proposed a pragmatic solution for checking such weaknesses, and our approach aligns with instrumenting the code before invoking the CBMC, where CBMC is treated as a black box. We describe the detected weaknesses in section 3.1 and our approach for validating such properties in section 3.2.

2.4 CBMC

CBMC stands for C - Bounded Model Checker. It can be used on C and C++ programs developed by Daniel Kroening and has been maintained for more than ten years [31]. It can verify the assertions under a given loop unwinding bound, array bounds (buffer overflows), pointer safety, arithmetic exceptions and can provide a counterexample during assertion violations. Currently, it supports C89, C99, most of C11 and most compiler extensions provided by GCC and Visual Studio.

Figure 2.4 depicts the steps involved in CBMC internal processing. The steps involved are as follows:

- C program and command options are passed as input to the CBMC. Parse tree is then generated corresponding to the passed input.
- Language Type Checking stage generates a symbol table, i.e., mapping identifiers to symbols utilizing the parse tree.
- CBMC converts all the symbols from the symbol table with function type to construct a corresponding GOTO program.
- CBMC adds checks for null-pointer dereferences, arithmetic overflow, and other undefined behavior according to the passed command options, including the code coverage assertions.

- CBMC transforms the GOTO program by unwinding the verification loops until a fixed bound and then translates the GOTO program into Static Single Assignment (SSA) Form.
- SSA form is then converted into a CNF formula.
- CNF formula is passed to SAT solver to check its satisfiability.
- If not satisfied, the SAT solver generates a counterexample that needs processing to convert it into a human-readable format.

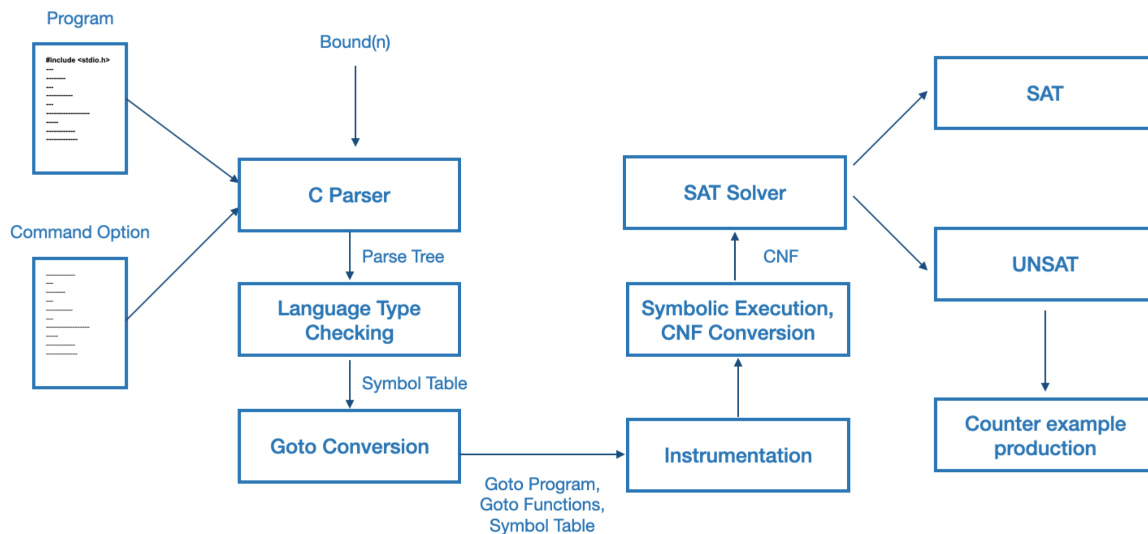

Figure 2.4: CBMC Architecture.

By default, CBMC comes with a MiniSat solver. CBMC can verify memory safety checks for array bounds, safe use of pointers, exceptions, variants of undefined behavior, and user-specified assertions [66].

Chapter 3

SecMD-Checker: Detecting Non-secure Memory Deallocation

In this chapter, we discuss the probable causes that lead to CWE-244 in section 3.1. We discuss our approach for C programs to detect if sensitive memory locations are wiped before freeing by invoking CBMC. We have created a tool SecMD-Checker that implements our approach for detecting the non-secure memory deallocation (CWE-244). It is described in detail in section 3.2.

3.1 Wiping Confidential Information

Generally, a programmer thinks their job ends after freeing the allocated memory by calling the function "free()" without considering the consequences of data exposure. The function "free()" deallocates the allocated heap memory. It takes an argument as a pointer to the memory that the programmer wishes to deallocate. It marks the memory as free so that the processor can assign it to other processes for their usage and reducing the memory wastage, as depicted in fig 3.1. However, the data in heap memory exists in the RAM until the system restarts, or completes its natural lifetime, i.e., another program overwrites the memory. If the confidential information is not wiped before freeing, then the sensitive information will live in the memory after freeing and will be inaccessible to the user. Moreover, in such cases, an attacker can use tools such as Dumpit and Volatility to extract the data from the RAM, leading to leakage of the information. This memory leakage is also described by Common Weakness Enumeration 244 (CWE-244) Improper Clearing of Heap Memory Before Release [7] which is well known in academia.

Another reason for CWE-244 is the usage of the "realloc()" function. The "realloc()" function is utilized to extend or shrink the allocated memory block size. It may allocate the

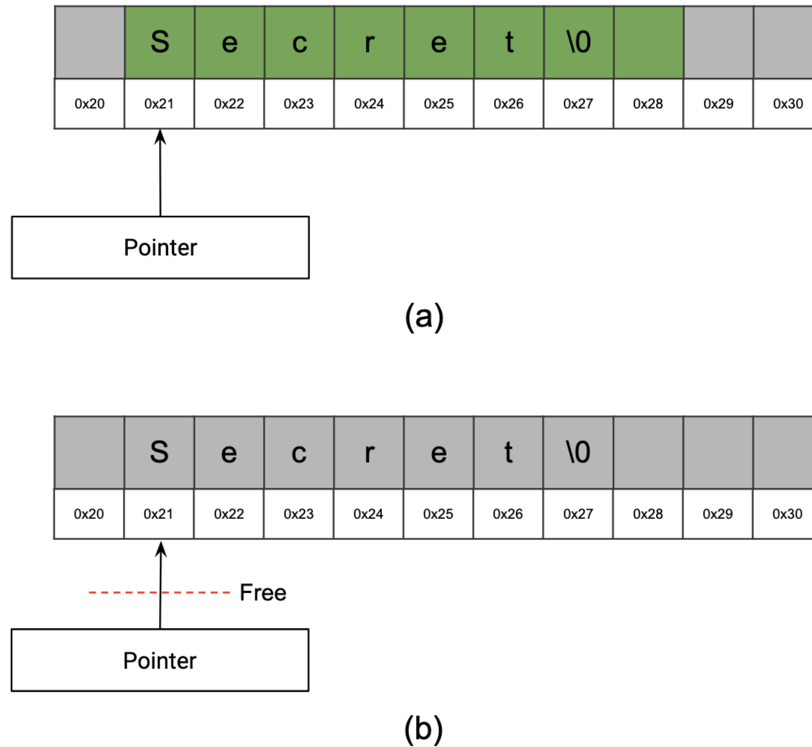

Figure 3.1: Wiping Confidential Information before freeing.

memory to the new address because space after the end of the old memory block may be in use [67]. This results in copying data from the old memory block to the new one - making the old memory block with the data intact inaccessible to the program. Therefore, using "realloc()" makes code susceptible to heap inspection.

We need to make sure such weakness is not present in the code, leading to sensitive data exposure. Hence, we proposed a solution that can be used to detect non-secure memory deallocation.

3.2 Overview and Scope of our Approach

We implemented our SecMD-Checker in Python 3.7 version. It takes two arguments as an input 1) a compilable C program 2) name of the variable holding sensitive data[1].

[1] For multiple variables, we can rerun the program

Currently, it only supports single file processing at a time. After we pass the input to the program, SecMD-Checker does a flow insensitive interprocedural analysis on the C program for fetching information about sensitive variables such as deep copies, shallow copies (aliases), data allocated on heap memory. Next, SecMD-Checker instruments the program for detection of non-secure memory deallocation. Instrumentation includes 1) declaring variables that track and monitor program variables containing sensitive data without changing the coding logic, 2) embedding assertion logic (that we define) before every "free()" statement that deallocates memory containing sensitive data, 3) inserting logic for tracking copies of sensitive data into the program. This instrumented file has sufficient information to know when and how to validate the non-secure memory deallocation. CBMC is then invoked to analyze the instrumented C program file. if non-secure memory deallocation is present in the program, an alert is raised, and a failure trace could be obtained for CWE-244. Otherwise, CBMC indicates successful validation of the CWE-244 test.

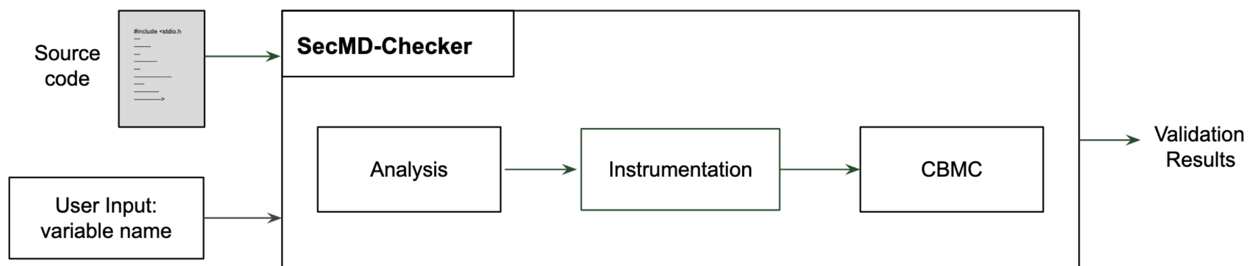

Figure 3.2: Instrumenting File to include wiping confidential information

Figure 3.2 shows the high level design of SecMD-Checker. SecMD-Checker requires a programmer to input the variable name which will be containing the sensitive information. Let us say the variable is X. The reason for requesting the user for the variable is to reduce the false positives. Suppose all the variables present in the code are checked for the CWE-244. In that case, the variable pointing at non-sensitive data will also be validated for the CWE-244. This validation is not required because non-sensitive data is

not mandatory to be wiped before freeing. This will generate false positives whenever the non-sensitive data is freed. In order to avoid these false positives, we request input from the programmer.

We defined the `__Memory_Wiping_Check` function containing the assertion logic (understood by CBMC) to validate the non-secure memory deallocation. Calls to this `__Memory_Wiping_Check` are embedded in the instrumented program before every "free()" of any sensitive variable. Adding a function call to __Memory_Wiping_Check just before all the calls to the "free()" statement informs CBMC to check whether the variable under consideration is pointing to one of the addresses that are being tracked. If that is the case, CBMC checks whether the memory is wiped by the assertions defined in the respective function.

3.2.1 Assumptions

- Suppose the data pointed by the input variable X is copied wholly or partially in another memory. In that case, the variable pointing at the memory will be considered as a deep copy. Therefore, it will be tracked as well. Deciding if the partial data should be counted as sensitive data is tangential to our research because it requires a deep analysis of the data itself. For example, a user's home address is sensitive data, but when looking at it partially, such as the State by itself could be considered as non-sensitive information. However, adding the street name to it could make it sensitive. Therefore, we consider any partial data as sensitive data that is copied from memory pointed by a sensitive variable.

  For example

```
1  void Dummy()
2  {
3      char *X = ( char* ) malloc( 100 * sizeof( char ) );
4      char *Y = ( char* ) malloc( 100 * sizeof( char ) );
5      char *Z = ( char* ) malloc( 100 * sizeof( char ) );
```

```
6       memcpy(Y, X, 1 * sizeof( char ) );
7       memcpy(Z, X, 100 * sizeof( char ) );
8   }
```

In the above example, X holds the sensitive information, and X is being partially copied to Y (line 6) and completely deep copied to Z (line 7). However, variables Y and Z will both be considered as a deep copy of X.

- Suppose a variable is suspected of holding sensitive data. In that case, it will be checked for non-secure memory deallocation at every free statement, independent of how many times we allocate and free the memory. As mentioned above, deep analysis is required to state/decide if data could be sensitive or not. So if a variable tends to hold sensitive data at some instance, then it is practical to consider that any data ever held by the variable could be sensitive.

  Example

```
1   void function1()
2   {
3       char *X = ( char* ) malloc( 100 * sizeof( char ) );
4       X = nonSensitiveData(); // returns non sensitive data
5       free(X);
6       X = ( char* ) malloc( 100 * sizeof( char ) );
7       X = sensitiveData(); // returns sensitive data
8       zero(X); // zero out the memory
9       free(X);
10  }
```

  In the above example, variable X will be checked for non-secure memory deallocation two times because we cannot state whether data stored in the variable is sensitive or not. Therefore, we check for non-secure memory deallocation just before line 5 and line 9. As of line 5, X was not zeroed out before "free()" even though it contained the non-sensitive data, an alert (false positive) will be generated.

- We treat deep copy relations as non-commutative, i.e., if the deep copy Y is created from variable X, then Y is a deep copy of X, but X is not considered as a deep copy of Y. We treat the user-provided variable name as the root of the chain, and the copies created out of it as its respective copies. Therefore, the variables that are used/deep copied to create the input variable are not monitored/tracked.

  Example

```
1   void Foo(int aliasX)
2   {
3       char *X = ( char* ) malloc( 100 * sizeof( char ) );
4       X = sensitiveData(); // returns sensitive data
5       char *Y = ( char* ) malloc( 100 * sizeof( char ) );
6       memcpy( Y, X, 100 * sizeof( char ) );
7       char *Z = ( char* ) malloc( 100 * sizeof( char ) );;
8       memcpy( Z, Y, 100 * sizeof( char ) );
9       free(X);
10      free(Y);
11      free(Z);
12  }
```

  In the above example, if the input variable to the SecMD-Checker is Y, then Z will be considered a deep copy of Y thus will be tracked. However, our tool will not track X because it was not created/copied from Y.

3.2.2 Limitations

SecMD-Checker is a prototype focused on detecting secure memory deallocation. It does not support the following properties/features.

- Higher order functions and pointers to functions
- Double pointers
- Array of pointers

- Structure
- Class
- Polymorphism

3.3 Instrumentation Algorithm

Algorithm 1: CWE-244 Instrumentation Algorithm(input_c_file, variable_name)

```
   input  : input_c_file, var_name
   output: program_file
1  Begin
2  |  program_file ← CodeFormat(input_c_file);
   |  // Convert to certain brace syntax
3  |  AllCopiesSet ← FindCopies(program_file, var_name) ;    // Find
   |   all the deep and shallow copies of input var_name
4  |  program_file ← Add declaration and initialization __address_holder,
   |   __count, __AddAddress and __Memory_Wiping_Check;
   |  /* Extract all function (prototypes, body) pairs and add
   |     to ProgramFunctions                                    */
5  |  Program_Functions ← ExtractFunctions(program_file)
```

```
5
6   for (func_prototype, func_body) in Program_Functions do
7       func_name ←GetFunctionName(func_prototype);
8       for line in func_body do
            /* Variables from "AllCopiesSet", is being assigned
            something or initialized.  For instance
            var_name= f(), var_name= malloc()                          */
9           if IsAssignmentStatement(line) then
10              var_to, var_from ← GetCopyVars(line);
11              if AddProperSuffix(var_to, func_name) ∈ AllCopiesSet
                 then
12                  program_file← Insert __AddAddress(var_to) after line;

            /* Variables from "AllCopiesSet", is passed as an
            argument to a function call.  For instance
            f(.., var_name, ..)                                        */
13          if IsFunctionCall(AllCopiesSet, line) then
14              Var_Names ←
                 { var_name | var_name is a function argument on "line" and
                              AddProperSuffix(var_name,func_name)
                              ∈ AllCopiesSet };

15              if IsIfStatement(line) then
16                  for var_name ∈ Var_Names do program_file← Insert
                     __AddAddress(var_name) in beginning of if and else block;
17              else
18                  for var_name ∈ Var_Names do program_file← Insert
                     __AddAddress(var_name) after line;

19          if IsFreeStatement(line) then
20              freedvar← GetFreedVar(line);
                // The variable being freed
21              if AddProperSuffix(freedvar, func_name) ∈ AllCopiesSet
                 then
22                  program_file ← add
                     __Memory_Wiping_Check(freedvar) function call before
                     line;
```

Algorithm 1 describes instrumentation steps for tracking and monitoring variables for secure memory deallocation. The algorithm has four phases as follows:-

1. Algorithm takes a compilable C program file and variable name as an input.

2. Formats the code in a suitable format for regex processing routines (line 2). Note: Code Formatting is described in detail in section 3.3.1.2.

3. Algorithm does a flow insensitive static analysis on a program to extract the relevant information required to generate a graph based on deep and shallow copy relation between the variables. It utilizes the generated graph for determining the variables that may contain sensitive information based on input variable X, i.e., shallow and deep copy variables of input variable X.

4. Instruments the program by embedding the necessary assertion and monitoring logic into the code. This instrumentation allows CBMC to track all the addresses that were assigned to the determined variables during CBMC analysis and check if any of the tracked addresses are wiped by utilizing the wiping assertion before every "free()".

Phase 3 starts at line 3; Call to function FindCopies finds all the shallow and deep copy variables and stores the variable list in a `AllCopiesSet` variable. Every variable name present in the `AllCopiesSet` has the following format: (`Variable_Name + "_-_" + Scope`). If we have a global variable, then the algorithm adds "Global" as a suffix (Scope), and if it is local, then the algorithm adds "function_name" as a suffix (Scope). Using function names as suffixes resolves the same name clashes in different functions. We will be using the word "suffix" for representing appended scope throughout the algorithm. Inner functioning of FindCopies for extracting shallow and deep copies of the input variable name is described in section 3.3.1.1.

Phase 4 starts at line 4; Adds variable and function definitions in the program file. These definitions are as follows:

- **__address_holder**: It is a global array of the data type (void *) that stores the addresses of detected sensitive variables. During the program run, `__address_holder` holds the addresses assigned to an identified variable that is currently stored in `AllCopiesSet`. It is essential to know the address of the variable that contains sensitive information because during freeing of heap memory, it can help distinguish several memory addresses. Moreover, the assigned memory needs to be tracked as memory can be pointed by many pointer references and thus can be freed by any of them. So when any memory is freed, SecMD-Checker checks if the variable holds the same address as the `__address_holder`. If true, then it utilizes the assertion to verify if the memory is securely deallocated.

- **__count**: `__count` is a global integer variable; it contains the count of the number of addresses stored in a `__address_holder` at any given instance.

- **__AddAddress**: It is a function that takes a pointer as an argument, and it stores the address pointed by the passed pointer into `__address_holder`.

- **__Memory_Wiping_Check**: It is a function that takes a pointer as an argument and checks whether the memory pointed by the passed pointer is zeroed out or not. If the memory is not zeroed out, then `__Memory_Wiping_Check` throws an assertion error.

Next, in line 5, algorithm extracts all the user-defined functions from the program and stores them in prototype and body pairs (where both are extracted from the function definition). Next, it iterates the functions and body extracted from the program to analyze and embed the logic of tracking, as shown in lines 6-22. `line` represents statements that end with a semicolon(;) or open curly bracket({).

Algorithm utilizes regex to identify if the `line` is an assignment statement (line 9), function call (line 13), or a "free()" statement for freeing the heap memory (line 19). After checking if a current line is an assignment statement (line 9), it extracts the variable name

in which data will be assigned. Further, it checks if that variable name + suffix is present in the `AllCopiesSet` on line 11. If true, there is a chance that a new address could be assigned to a pointer. Therefore, the algorithm inserts a function call to `__AddAddress` after the assignment line, which adds the memory address pointed by the `var_from` to the `__address_holder` if already not present in the `__address_holder`. This way `__AddAddress` will always have all the addresses that were ever assigned to the detected variables. Note: If assignment statement assigns a reference to the identified variable or assignment is done using functions like "strcpy", "memcpy", "memcpy_s", "strncpy", "strcpy_s", "strncpy_s" then a call to `__address_holder` before the assignment line is added.

In line 13-18, if a function call is detected on the iterating line, then the algorithm extracts the variable name from the function arguments that are present in the `AllCopiesSet` set and stores them in `var_name`. Next, on line 15, the algorithm checks if a function call is present inside an "if condition" or not. If an "If block" is detected, then it inserts the `__AddAddress` function call for every variable present in `var_name` at the beginning of the "if block" and "else block" (if present). Moreover, if the "if statement" is not detected, then it embeds the data on the next line, as shown in lines 17-18. By doing this, the algorithm maintains the addresses that could have been changed during the execution of function calls from the third-party library or function calls with missing bodies or definitions. Note: we don't treat ("strcpy", "memcpy", "memcpy_s", "strncpy", "strcpy_s", "strncpy_s") as function calls.

Next, in lines 19-22, if a "free()" statement is detected, then the algorithm extracts the variable name and checks if the variable exists in `AllCopiesSet`. If true, it inserts a function call to the `__Memory_Wiping_Check` function with the extracted variable name before the line that is being analyzed. `__Memory_Wiping_Check` call allows checking the memory for non-secure memory deallocation (zeroed out before freeing or not).

Let us try to understand the algorithm by an example. Suppose we feed program 3.1 and variable name "password" as an input to the SecMD-Checker. SecMD-Checker performs flow-insensitive static analysis on the program to find all the shallow and deep copy variables that tend to store sensitive values at some instance of time. After the analysis, SecMD-Checker defines __count and __address_holder on line 4-6 and embeds the monitoring logic. `__holder_size` is a defined macro whose value can be set from the SecMD-Checker. The default value is 1000.

Program 3.1: Program without Instrumentation

```
1 #include <wchar.h>
2 #include <windows.h>
3 void FreeDuplicateMemory(char *copy)
4 {
5     ZeroOutMemory(copy);
6     free(copy);
7 }
8 void CWE244_Dummy()
9 {
10        char * password = (char *)malloc(100*sizeof(char));
11        char * password_deep_copy = (char *)malloc(100*sizeof(char));
12        char * password_shallow_copy;
13        password_shallow_copy = password_deep_copy;
14        if (fgets(password, 100, stdin) == NULL)
15        {
16            printLine("fgets() failed");
17            /* Restore NUL terminator if fgets fails */
18            password[0] = '\0';
19        }
20        memcpy(password_deep_copy, password, 100*sizeof(char));
21        ...
22        FreeDuplicateMemory(password_shallow_copy);
```

```
23          free(password);
24 }
25 ... More
```

To detect non-secure memory deallocation, the tool must validate any memory allocated to the identified variable X before it gets deallocated. During the analysis, as the password variable is deep copied and shallow copied, `AllCopiesSet` will contain four entries, i.e., password__CWE244_Dummy, password_deep_copy__CWE244_-Dummy, password_shallow_copy__CWE244_Dummy, copy__FreeDuplicateMemory.

During the Dummy function analysis, when lines 10, 11, 13, 18, 20 are read, SecMD-Checker detects the lines as assignment statements. As the password, password_deep_-copy, password_shallow_copy is present in the `AllCopiesSet` we insert the `__AddAddress` line after the respective line accordingly. Next, on line 14, the "if" statement is detected, and the `__AddAddress` call is inserted at the beginning of the "if" block. Lastly, lines 5 and 23 are detected as "free()" statements, and then SecMD-Checker inserts the `__Memory_Wiping_Check` call just before the respective lines. At the end of the instrumentation, our program looks like the program 3.2. SecMD-Checker also embeds `__Memory_Wiping_Check` and `__AddAddress` function definition in the program. However, we do not show it in the 3.2 for clarity.

Instrumented program 3.2 is fed to CBMC. During the CBMC analysis, the following steps occur

1. Line 16, 0x20 (For example) address is assigned to the password pointer.
2. Line 17, 0x20 address will be added to the `__address_holder`.
3. Line 18, 0x100 (For example) address is assigned to the password pointer.
4. Line 19, 0x100 address will be added to the `__address_holder`.
5. Line 25, 28 nothing will be added to `__address_holder` because previous lines do not change any addresses of the detected variables.

6. Line 33, FreeDuplicateMemory function is called, and then at line 10, memory starting from 0x100 (pointed by pointer "copy") is zeroed out.

7. Line 11, 0x100 will be checked for non-secure memory deallocation. Next, in line 12, memory 0x100 will be freed.

8. At line 34, nothing will be added to the `__address_holder` as the address is not changed.

9. Line 35, memory 0x20 will be checked for non-secure memory deallocation. As memory is not zeroed out, an alert will be raised.

10. Line 36, memory 0x20 will be freed.

Program 3.2: Program after Instrumentation

```
1  #include <wchar.h>
2  #include <windows.h>
3
4  #define __holder_size 1000
5  void *__address_holder[__holder_size];
6  int __count=0;
7
8  void FreeDuplicateMemory(char *copy)
9  {
10     ZeroOutMemory(copy);
11     __Memory_Wiping_Check(copy);
12     free(copy);
13 }
14 void CWE244_Dummy()
15 {
16         char * password = (char *)malloc(100*sizeof(char));
17         __AddAddress(password);
18         char * password_deep_copy = (char *)malloc(100*sizeof(char));
```

```
19          __AddAddress(password_deep_copy);
20          char * password_shallow_copy;
21          password_shallow_copy = password_deep_copy;
22
23          if (fgets(password, 100, stdin) == NULL)
24          {
25              __AddAddress(password);
26              printLine("fgets() failed");
27              password[0] = '\0';
28              __AddAddress(password);
29          }
30          memcpy(password_deep_copy, password, 100*sizeof(char));
31          ...
32          ...
33          FreeDuplicateMemory(password_shallow_copy);
34          __AddAddress(password_shallow_copy);
35          __Memory_Wiping_Check(password);
36          free(password);
37  }
38  ... More
```

3.3.1 Subroutines of the algorithm

In this section, we will be describing all the necessary procedures that are needed by our algorithm. A few of the procedures in this section are inserted in the instrumented program. The procedures only make updates to the newly defined variables inside them. Therefore, they do not affect the control flow of the original program.

3.3.1.1 FindCopies Function

Procedure 2 defines a method for finding all the potential sensitive variables that are a deep copy or shallow copy of the input variable X. FindCopies creates a graph by flow

insensitive static analysis of the program and uses it to find the sensitive variables. The algorithm explanation is as follows:

- Line 1, initializes the graph `G`. The nodes of graph G are variable names with the suffix (scope), and the edges are a deep or shallow copy relation between the variables. The naming convention of variables in the graph is discussed in the previous sections - the variable name will have a suffix as the "__function_name" or "__Global" depending on the declared scope.

- Line 2, extracts all the variable names that are declared in the global scope.

- Lines 3-5, adds all the extracted global variables as nodes in the graph with a "__-Global" suffix.

- Line 6, extracts all the user-defined procedures from the program in the form of function prototype and function body pairs and stores them in `Program_Functions`.

- Line 7, calls the function `AddVerticesAndEdges` which iterates the function bodies that are present in `Program_Functions` and adds the variable present in the function bodies and prototypes arguments in the graph as nodes with the proper suffix. It also adds the edges based on shallow and deep copy relation between the variables. The procedure `AddVerticesAndEdges` is discussed in more detail later in this section.

- All reachable nodes from input variable X are computed using Depth First Search (DFS) algorithm, and then all variable names present in those branches are added to `AllCopiesSet` set. As the edges in the graph represent the deep copy and shallow copy instances, all the variables present in the `AllCopiesSet` must contain sensitive information at some instance of the program run.

Procedure 2: FindCopies(c_file, variable_name)

```
input  : c_file, var_name
output: AllCopiesSet
1 G ← ∅;        /* Directed graph G for computing AllCopiesSet */
  // Extract all global variable names
2 Global_Vars← GetGlobalVariableDefinitions(program_file);
3 for global_var ∈ Global_Vars do
4 │ AddVertex(G, concat(global_var, "__Global"));
5 end
  /* Extract all function (prototypes, body) pairs and add to
     ProgramFunctions                                          */
6 Program_Functions ← ExtractFunctions(program_file)
  // Add Vetices and Edges by analysing function definitions
7 AddVerticesAndEdges(G, Program_Functions);
  /* All reachable nodes from vertex "var_name" in the
     directed graph G                                          */
8 AllCopiesSet← ComputeReach(G, var_name)
9 return AllCopiesSet;
```

Procedure 3 describes how the bodies of the functions that were extracted from the program are iterated in the procedure 2 for creating a graph G. AddVerticesAndEdges takes two inputs, 1) Graph and 2) (function prototype, function body) pairs. AddVerticesAndEdges algorithm explanation is as follows:

- Line 1, declares a set prototype which holds all the user-defined function names and arguments.
- Line 2, starts iterating the function (prototype, bodies) pairs that are extracted from the input program.
- Lines 3-5, extracts the function name and arguments from the prototype and stores them in the respective variables.

- Lines 6-8, adds all the arguments present in the function prototype in the graph by adding the proper suffix where the function name is extracted from the prototype.

- Next, the algorithm starts iterating lines from the body of each function where it checks if a line/statement is 1) a variable declaration, 2) an assignment statement, 3) a function call, and does the respective operations accordingly.

- Lines 10-14, if a line is a variable declaration statement, then the algorithm extracts the variable name and adds the proper suffix by calling the function `AddProperSuffix`. `AddProperSuffix` is described later in this section. After adding the proper suffix, it adds the recognized variable to graph `G` as shown in line 13.

- Lines 15-20, adds an edge between the nodes defined in graph `G`. Suppose variable X appears on the right-hand side of the equation/assignment. If true, the variable on the left-hand side must be considered for storing sensitive data because it could have stored the sensitive data partially or entirely. Therefore, we extract the variable from the left-hand and right-hand sides of the equation and add a directed edge between the RHS variables to the LHS variable by adding the respective suffix as shown in Lines 16-19. The algorithm considers memcopy, strncpy, function calls as assignment statements; in this scenario, "from" and "to" variables are extracted from the argument of the functions, and the edge is created as described in Lines 16-19.

- Lines 21-30, if a line contains a function call, an edge is added between the function call arguments and the user-defined function arguments with the appropriate suffix.

Procedure 3: AddVerticesAndEdges(G, Program_Functions)

```
   input: G, Program_Functions
1  prototype ← ∅; /* Store all user defined functions prototypes */
   // Iterate the functions to extract pointer variable names
      and create link between pointer variables in a graph
2  for (func_prototype, func_body) in Program_Functions do
3      func_name ←GetFunctionName(func_prototype);
4      Arguments ← GetArguments(func_prototype);
5      prototype ← prototype ∪ {func_prototype} ;
6      foreach Argument in Arguments do
7          AddVertex(G, concat(Argument, "__", func_name));
8      end
9      foreach line in func_body do
10         if IsVariableDeclaration(line) then
11             var_name ← GetVariableName(line);
12             var_name= AddProperSuffix(var_name, func_name);
13             AddVertex(G, var_name);
14         end
15         if IsAssignmentStatement(line) then
16             var_from, var_to← GetCopyVars(line);
17             var_from= AddProperSuffix(var_from, func_name);
18             var_to= AddProperSuffix(var_to, func_name);
19             AddEdge(G, ⟨var_from, var_to⟩);
20         end
21         if IsFunctionCall(line) then
22             Arguments ← GetFunctionArguments(line);
23             func_call_name ←GetFunctionName(line);
24             fetched_arguments = fetchPrototype(func_call_name,
                prototype);
25             for arg, fetched_arg ∈ Arguments, fetched_arguments do
26                 arg = AddProperSuffix(arg, func_name);
27                 fetched_arg = AddProperSuffix(fetched_arg,
                    func_call_name);
28                 AddEdge(G, ⟨arg, fetched_arg⟩);
29             end
30         end
31     end
32 end
```

Let us try to understand the function call processing by an example 3.1. Suppose function CWE244_Dummy is used for analysis purposes. During analysis of function variables password, password_deep_copy, password_shallow_copy, a copy will be detected as pointer variables. Therefore, these variables will be added as nodes to the graphs with a proper suffix. The resultant graph will look like 3.3. Now we analyze the function bodies for creating edges in the graph. Line 13, adds an edge from password_deep_copy to password_shallow_copy because it is an assignment statement. Similarly, during the analysis of line 20, the algorithm adds an edge from password to password_deep_copy. Line 22, a function call to FreeDuplicateMemory is detected. As it is a user-defined function, the algorithm will create an edge from password_shallow_copy__CWE244_Dummy to copy__FreeDuplicateMemory. Here, the copy will have a suffix FreeDuplicateMemory, and the password_shallow_copy will have a CWE244_Dummy suffix. Now, the generated graph G will look like figure 3.4.

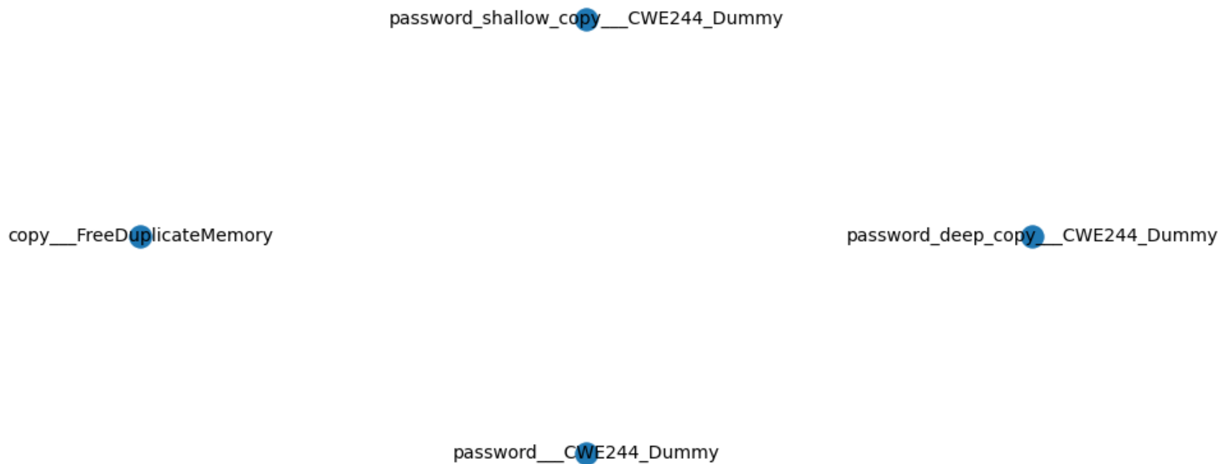

Figure 3.3: Graph nodes generation.

At the end of this process, FindCopies returns the graph that contains the nodes as variable names with appropriate suffix and edges as deep copy or shallow copy relation.

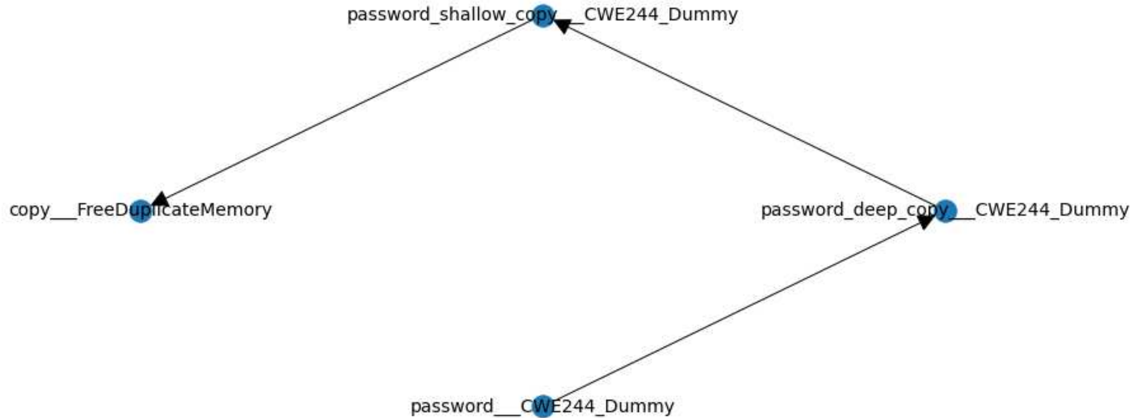

Figure 3.4: Graph edges generation.

After that, the algorithm computes all the reachable nodes from the variable "password", which will generate a `AllCopiesSet` containing variable names {password, password_deep_copy, password_shallow_copy, and copy}. The actual parser could be used to extract this information from the code, but a more straightforward approach using regex was followed for this research. In this algorithm elaboration, some of the code implementation details are omitted for the sake of clarity.

Procedure 4: AddSuffix(__variable, func_name)

```
  input : __variable, func_name
  output: __variable
1 if (IsGlobalVariable(__variable)) then
2 |   __variable= concat(__variable, "__Global");
3 else
4 |   __variable= concat(__variable, "__", func_name);
5 end
6 return __variable;
```

Procedure 4 (`AddProperSuffix`) adds the suffix "__function_name" or "__Global". It adds "__Global" to the passed variable based on the Scope resolution operator or depending on the context, i.e., if a variable is not declared and still used, then it has to be a

global variable because input was a compilable program as shown in line 1-3. If the line 1 condition is false, then the procedure simply concatenates the __function_name" to the variable name.

3.3.1.2 Function: CodeFormat

CodeFormat function changes the formatting of the code. Our instrumentation utilizes regex to find patterns and extract required data from the input program. Code following the same format throughout makes it easy to analyze/extract data using regex. We utilized the library "pyastyle", which allows us to change the format of the code. We utilize this library to add braces to unbraced one-line conditional statements such as 'if', 'while', 'for', and more as shown in 3.5. Moreover, we force the code to follow broken braces and split each statement by newline as shown in 3.6. We also removed all the single and multi-line comments from the input program.

```
1 if (variable)
2     variable = false;
```

(a)

```
1 if (variable){
2     variable = false;
3 }
```

(b)

Figure 3.5: (a) Before bracket formatting. (b) After bracket formatting

```
1 if (variable){variable = false; variable2 = random();}
```

(a)

```
1 if (variable)
2 {
3     variable = false;
4     variable2 = random();
5 }
```

(b)

Figure 3.6: (a) Before bracket Formatting. (b) After Formatting

3.3.1.3 Functions inserted into the instrumented program

__AddAddress Function

`__AddAddress` update a tool defined array `__address_holder` in the instrumented C program for storing memory addresses of all sensitive data variables. During the analysis of the C program, we search for three events/statements, after which we need to call `__AddAddress`. 1) Allocation statements (supports malloc, calloc, realloc) 2) Assignment operator when the input variable or identified variables are on Left Hand Side (LHS), and 3) the function calls that utilize detected variables. After the program scan, we embed a statement that makes a call to the function `__AddAddress` which takes a pointer as an argument.

Procedure 5 describes how `__AddAddress` adds the address pointed by the passed pointer to the `__address_holder`. The argument function takes the pointer variable that holds the current address of the identified variables. Line 1-4 works as a base condition where if no address is present in the `__address_holder` then procedure adds the address and increases the counter. Next, in lines 5-12, it checks if the memory address pointed by the passed pointer is already present in the `__address_holder` then it does make any changes. Otherwise, it adds the address to the `__address_holder` and

Procedure 5: __AddAddress(`__variable`)

```
   input: __variable
1  if __count == 0 then
2  |  __address_holder[__count++] = __variable;
3  |  return;
4  end
5  present ← false;
6  for x ∈ __address_holder do
7  |  if (__variable == x ) then
8  |  |  present = true
9  |  end
10 end
11 if present == false then
12 |  __address_holder[__count++] = __variable;
13 end
```

increments the `__count` value; this way `__address_holder` will have all the unique addresses that were ever assigned identified variables at any time in the code. `__address_holder` list is utilized by the __Memory_Wiping_Check function for the assertion check described in the next section.

Assertion Function: __Memory_Wiping_Check

Assertion Function contains the assertion logic for verifying secure memory deallocation. This function contains C syntax and CBMC predefined function. Hence, CBMC can recognize it. Call to this function will check if the passed address is present in the `__address_holder`. If yes, then non-secure memory deallocation assertion logic will be triggered, as shown in line 6-8. Here the "free()" statement works as an anchor for embedding the `__Memory_Wiping_Check` call just before any encountered "free()" statement in the program. A function definition is as follows:

```
Procedure 6: __Memory_Wiping_Check(__variable)
   input: __variable
1  if __CPROVER_OBJECT_SIZE( __variable ) == 0 then
2  |   return;
3  end
4  for x ∈ __address_holder do
5  |   if (x == __variable) then
6  |   |   foreach bit y of __variable do
7  |   |   |   __CPROVER_assert(y == 0, "Error Message");
8  |   |   end
9  |   |   __DelAddress(x) ;
   |   |   // Delete x from __address_holder data structure as it
   |   |      has now been freed.
10 |   end
11 end
```

__Memory_Wiping_Check checks that the size of the allocated memory pointed by the passed variable is not zero, as shown in lines 1-3. After that, from lines 4-10, it checks whether any memory addresses present in the __address_holder that are currently being tracked match with the passed pointer pointed memory address. When the address is matched, it checks if the memory is wiped out, as shown in lines 6-7. If all the bits are not zeroed, it concludes that the memory was not securely deallocated. For the assertion, we have utilized the predefined functions provided in the CBMC [66]. They are as follows:

- __CPROVER_OBJECT_SIZE(): Used to fetch the size of the allocated memory that is pointed by the variable. CBMC makes it possible because CBMC considers allocated memory as an array.

- __CPROVER_assert() - Checks if the mentioned Boolean condition in the passed argument is satisfied.

More information about the function can be found at CBMC developer documentation [68]. __Memory_Wiping_Check function is called just before every "free()" statement

that is present in the code. Doing so confirms that the tracked variable is wiped just before freeing the data utilizing the assertion logic described above.

3.4 Realloc Handling

As discussed in section 3.1, using "realloc()" function may lead to CWE-244. The "realloc()" function is utilized to extend or shrink the allocated memory block size. It may allocate the memory to the new address because space after the end of the old memory block may be in use [67]. This results in copying data from the old memory block to the new one - making the old memory block with the data intact inaccessible to the program. Therefore, using "realloc()" makes code susceptible to heap inspection.

We can handle the "realloc()" function in two ways. First, not allowing a user to reallocate the memory because that will lead to non-secure memory deallocation. Second, making sure that reallocation is performed only after zeroing out the memory. We did not pursue the first option because it will restrict the user from using a "realloc()", even if they are clearing the memory before calling the "realloc()" function. Juliet Test Suite test cases with good implementation of CWE-244 perform memory wiping before utilizing "realloc". Therefore, we implemented the second option. In order to enable CBMC to handle the "realloc()" scenario, we treat it as a "free" and "assignment" statement. Consequently, we embed a `__Memory_Wiping_Check` function call before any "realloc()" call because it may deallocate an old memory block (Free Statement). Moreover, we embed a `__AddAddress` function call after the "realloc()" statement because it may allocate a new memory block (Assignment Statement). Therefore, if memory is not cleared and the user tries to reallocate it, the assertion will fail, indicating "Non-Secure memory Deallocation".

Note: Implementation details of "realloc()" handling are omitted from the CWE-244 instrumentation algorithm for the sake of clarity.

3.5 Example: Instrumenting multi-function program

In this section, we will look at a bit more complicated example than the program 3.1. The example includes multiple function calls, shallow copy, and deep copy instances of input variable X. We will give a rough explanation of how the program will be instrumented by SecMD-Checker. We will also discuss how our instrumentation helps to track variables and detect non-secure memory deallocation at a high level.

We input the program file 3.3 and a variable name "password" to the SecMD-Checker. In the program, we create a shallow copy (line 23) and deep copy (line 33, 22) of the variable "password". We pass these pointer variables to different functions (line 34, 13) and make deep copies inside those functions (line 7) as well. We zero out some deep copies (line 14) and leave a few copies (ref, password, bit_store) unwiped to cause non-secure memory deallocation. After this instrumentation, the program 3.3 will look like the program 3.4.

Program 3.3: Multi-function Program Example

```
1 #include <wchar.h>
2 #include <windows.h>
3
4 void Dummy_Function(char *arg)
5 {
6     char * key = (char *)malloc(100*sizeof(char));
7     strncpy(key, arg, 100);
8     char *ref = key;
9     free(ref);
10 }
11 void FreeDuplicateMemory(char *copy)
12 {
13     Dummy_Function(copy);
14     ZeroOutMemory(copy);
15     free(copy);
```

```
16 }
17 void CWE244_Dummy()
18 {
19         char * password = (char *)malloc(100*sizeof(char));
20         char * password_deep_copy = (char *)malloc(100*sizeof(char));
21         char * password_shallow_copy;
22         char * bit_store = (char *)malloc(1*sizeof(char));
23         password_shallow_copy = password_deep_copy;
24 
25         if (fgets(password, 100, stdin) == NULL)
26         {
27             printLine("fgets() failed");
28             /* Restore NUL terminator if fgets fails */
29             password[0] = '\0';
30         }
31 
32         bit_store[0] = password[0];
33         memcpy(password_deep_copy, password, 100*sizeof(char));
34         FreeDuplicateMemory(password_shallow_copy);
35 
36         free(password);
37         free(bit_store);
38 }
39 
40 void main()
41 {
42     CWE244_Dummy();
43 }
```

Let us understand the instrumentation done by our tool on program 3.3. It is as follows:

1. Code is formatted according to section 3.3.1.2.

2. Functions are iterated for finding pointer variables.

3. Graph `G` is created

4. Dummy_Function is iterated and the variables (arg__Dummy_Function, key__Dummy_Function, ref__Dummy_Function) are added to the graph `G` as vertices.

5. FreeDuplicateMemory is iterated and the variable (copy__FreeDuplicateMemory) is added to the graph `G` as vertices.

6. CWE244_Dummy is iterated and the variables (password__CWE244_Dummy, password_deep_copy__CWE244_Dummy, password_shallow_copy__CWE244_Dummy, bit_store__CWE244_Dummy) are added to the graph `G` as vertices.

7. Graph `G` is shown in figure 3.7 after analysing the program for vertices.

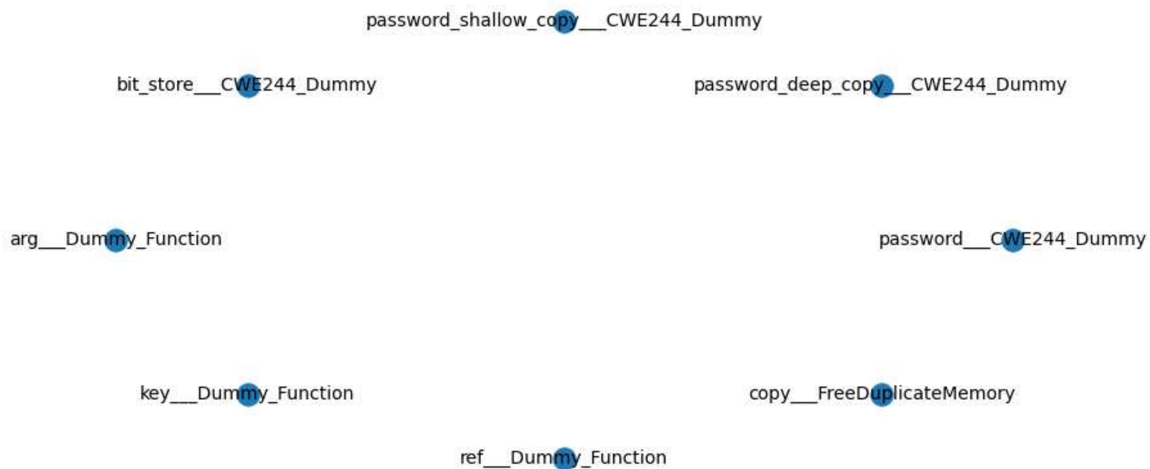

Figure 3.7: Example: Generated graph vertices.

8. Now, all functions are iterated to find the variables' deep copy and shallow copy relation.

9. Assignment statement is detected at lines 7, 8 during Dummy_Function iteration. Therefore, an edge is created between (arg -> key) and (key -> ref) variables.

10. No shallow copy and deep copy relation are detected during FreeDuplicateMemory iteration. However, at line 13, a function call is detected, so we create an edge (copy__FreeDumplicateMemory -> arg__Dummy_Function). Note: ZeroOutMemory function zero out the memory, and we are considering it as a library call.

11. Assignment statements are detected at line 23, 32, 33 during CWE244_Dummy iteration. Therefore, edges are created between (password_deep_copy -> password_shallow_copy), (password -> bit_store) and (password -> password_deep_copy) variables. At line 34, function call is detected, so we create an edge from (password_shallow_copy__CWE244_Dummy -> copy__FreeDuplicateMemory).

12. Complete graph `G` is shown in figure 3.8 after analysing the program for edge creation.

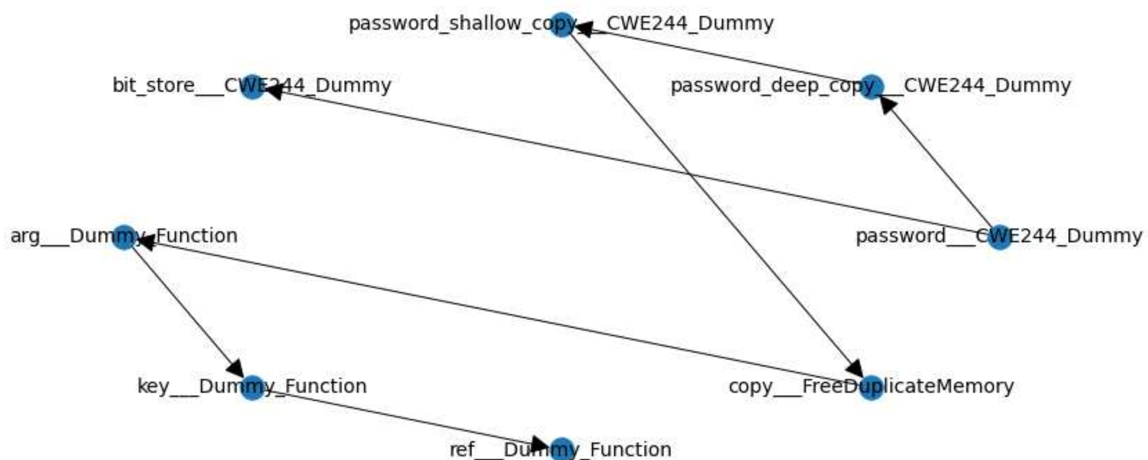

Figure 3.8: Example: Generated graph edges.

13. SecMD-Checker computes all reachable nodes from vertex name password__AnyFuncName. Resultant `AllCopiesSet` set will be { password_deep_copy__CWE-244_Dummy, password_shallow_copy__CWE244_Dummy, password__CWE244_-

Dummy, bit_store__CWE244_Dummy, copy__FreeDumplicateMemory, arg__Dummy_Function, key__Dummy_Function, ref__Dummy_Function }

14. Next, SecMD-Checker embeds the tracking logic for all the variables present in `AllCopiesSet`.

15. Assignment statements are detected at line 6, 7, 8 during Dummy_Function iteration. SecMD-Checker inserts the `__AddAddress` function call after the assignment lines where "to" variables are present in `AllCopiesSet`. At line 9, the "free()" statement is detected. Therefore, it inserts `__Memory_Wiping_Check` before the free statement.

16. Function calls are detected at line 13, 14 during the FreeDuplicateMemory iteration. SecMD-Checker inserts `__AddAddress` after function call lines where their arguments are present in `AllCopiesSet`. At line 15, "free()" is detected. Therefore, it inserts `__Memory_Wiping_Check` before the free statement.

17. Assignment statements are detected at line 19, 20, 22, 23, 29, 32, 33 during CWE-244_Dummy iteration. SecMD-Checker inserts `__AddAddress` function call after respective line as per the algorithm. At line 25, 34, function calls are detected where function calls include arguments that are present in `AllCopiesSet`. Therefore, it inserts the `__AddAddress` function call after the respective lines. At line 36, 37, "free()" is detected. Therefore, it inserts `__Memory_Wiping_Check` before the free statement.

18. Lastly, we add the global variable `__address_holder`, function `__AddAddress` and `__Memory_Wiping_Check` definitions in the program.

19. Program 3.4 is the resultant of SecMD-Checker instrumentation.

Program 3.4: Output after instrumentation

```
1 #include <wchar.h>
2 #include <windows.h>
3
4 void Dummy_Function(char *arg)
5 {
6     char * key = (char *)malloc(100*sizeof(char));
7     __AddAddress(key);
8     strncpy(key, arg, 100);
9     char *ref = key;
10    __Memory_Wiping_Check(ref);
11    free(ref);
12 }
13 void FreeDuplicateMemory(char *copy)
14 {
15    Dummy_Function(copy);
16    __AddAddress(copy);
17    ZeroOutMemory(copy);
18    __AddAddress(copy);
19    __Memory_Wiping_Check(copy);
20    free(copy);
21 }
22 void CWE244_Dummy()
23 {
24    char * password = (char *)malloc(100*sizeof(char));
25    __AddAddress(password);
26    char * password_deep_copy = (char *)malloc(100*sizeof(char));
27    __AddAddress(password_deep_copy);
28    char * password_shallow_copy;
29    char * bit_store = (char *)malloc(1*sizeof(char));
30    __AddAddress(bit_store);
31    password_shallow_copy = password_deep_copy;
32
```

```
33      if (fgets(password, 100, stdin) == NULL)
34      {
35          __AddAddress(password);
36          printLine("fgets() failed");
37          password[0] = '\0';
38          __AddAddress(password);
39      }
40
41      bit_store[0] = password[0];
42      __AddAddress(bit_store);
43      memcpy(password_deep_copy, password, 100*sizeof(char));
44      FreeDuplicateMemory(password_shallow_copy);
45      __AddAddress(password_shallow_copy);
46
47      __Memory_Wiping_Check(password);
48      free(password);
49      __Memory_Wiping_Check(bit_store);
50      free(bit_store);
51  }
52
53  void main()
54  {
55      CWE244_Dummy();
56  }
57  #define __holder_size 1000
58  void *__address_holder[__holder_size];
59  int __count=0;
60
61  void __Memory_Wiping_Check(void* wiped)
62  {...}
63  void __AddAddress(void* change)
64  {...}
```

Lets understand how CBMC will detect non-secure memory deallocation in the above program:

1. main function is analyzed first. At line 59, function call to CWE244_Dummy is detected.
2. At line 24, address 0x10 (for example) is added to the `__address_holder`.
3. At line 26, address (for example) 0x110 is added to the `__address_holder`.
4. At line 29, address (for example) 0x210 is added to the `__address_holder`.
5. At lines 35, 38, 42 45, nothing will be added to the `__address_holder` because the address (0x10, 0x110) pointed by the passed pointer is already present in the `__address_holder`.
6. Line 44, control transfers to FreeDuplicateMemory function.
7. Line 15, control transfers to Dummy_Function function.
8. Line 7, address 0x310 (for example) is added to the `__address_holder`.
9. Line 12, memory starting from 0x310 is checked for non-secure memory deallocation. In this case, non-secure memory deallocation is detected, and an alert message is generated.
10. Line 11, 0x310 memory is freed.
11. Control transfers back to the FreeDuplicateMemory function.
12. Line 16, 18 nothing will be added to the `__address_holder`.
13. Line 19, memory starting from 0x110 is checked for non-secure memory deallocation. In this case, memory is wiped at line 19, so the test is validated.
14. Line 20, memory starting from 0x110 is freed.

15. Control transfer back to CWE244_Dummy function.

16. Line 45, nothing will be added to `__address_holder`.

17. Line 47, 49 memories starting from 0x10, 0x210 are checked for non-secure memory deallocation. In this case, as the memory is not zeroed out, an alert is generated for both cases.

18. Line 48, 50 memory (0x10, 0x210) is freed.

Note: We can utilize an option $--trace$ in CBMC for analyzing the code. It will generate a failure trace for all failed assertions; In our case, failure traces of non-secure memory deallocation. For more information refer [66].

3.6 Dynamic tracking SecMD-Checker

Our algorithm enforces CBMC to dynamically track the addresses that are allocated to the identified variables. Absence of dynamic tracking will increase the detection of some false positives. For example, if we do not save the addresses in the `__address_holder`, then we must check non-secure memory deallocation for all the variables present in `AllCopiesSet` before freeing the memory. This will remove the dynamic address tracking aspect from our algorithm. Therefore, this automatically changes our algorithm to use non-secure memory deallocation assertion logic to all the addresses passed to `__Memory_Wiping_Check` function.

Let us look at the dummy program 3.5 that shows why missing dynamic tracking capability could lead to false positives. Variable "password" is passed as input to SecMD-Checker. SecMD-Checker does an interprocedural static analysis on the program resulting in a `AllCopiesSet` = { password__main, arg__Dummy_Func }. Therefore, line 3 is added during instrumentation in the program because variable arg will hold the reference of password (sensitive data) in one of the execution branches. If we are not dynamically tracking the addresses, then during one execution branch (line 14), the "arg" (reference

of temporary) will be checked for non-secure memory deallocation. This leads to a false positive because the memory pointed by "arg" (temporary reference) does not hold any sensitive data. However, in the case of dynamic tracking `__address_holder` (current algorithm), the address pointed by password will be added to the `__address_holder` at line 9. Therefore, when line 3 is executed for a branch where "arg" is a reference to a temporary variable. CBMC will not use the assertion logic for a memory pointed by an "arg" (temporary reference) variable because its address will not be present in the `__address_holder`. Therefore, having a `__address_holder` allows us to dynamically track the addresses of sensitive variables and decrease such false positives in the presence of may-aliases.

Program 3.5: Dynamic tacking

```
1  void Dummy_Func(char * arg)
2  {
3      __Memory_Wiping_Check(arg);
4      free(arg);
5  }
6  void main()
7  {
8      char * password =  (char *)malloc(100*sizeof(char));
9      __AddAddress(password);
10     char * temporary =  (char *)malloc(100*sizeof(char));
11
12     Dummy_Func(password);
13     __AddAddress(password);
14     Dummy_Func(temporary);
15 }
```

3.6.1 May-Alias: False Witness to a Positive

Our algorithm occasionally shows false positives occurrences in case of the presence of may-aliases of input variable X, even with dynamic tracking. Let us discuss a scenario in which such a false positive occurs in the program 3.6. In the program 3.6, memory is allocated in Dummy_Func itself. Therefore, line 4 is added during instrumentation after the assignment statement (line 3) because "arg" is a may-alias of password in one branch where "arg" is a "password" reference. Hence, during the CBMC analysis of line 4, addresses of password and temporary (arg reference) are added to the `__address_-holder`. This compels CBMC to use the non-secure memory deallocation assertion at both the memory addresses. `__Memory_Wiping_Check` assertion will fail for variable "arg" when it is a reference of password (True Positive) and when arg is a reference of a temporary variable (which does not contain sensitive information), leading to a false positive. In the second scenario, the call-chain witness of CBMC to the assertion violation may be viewed as a false witness for the alarm.

Program 3.6: False Witness because of may-alias

```
1  void Dummy_Func(char * arg)
2  {
3      arg = (char *)malloc(100*sizeof(char));
4      __AddAddress(password);
5      __Memory_Wiping_Check(arg);
6      free(arg);
7  }
8  void main()
9  {
10     char * password;
11     char * temporary;
12
13     Dummy_Func(password);
14     __AddAddress(password);
```

```
15      Dummy_Func(temporary);
16  }
```

3.7 Bad Compiler Optimization

The compiler changes the code automatically for optimizing it for a performance gain. However, many times, the logic written by the programmer is removed during the compiler optimization because these changes are not essential for the compiler and will unnecessarily slow the program. Therefore, we should make sure that the object file consists of all the critical parts of the code.

```
1  Function DecryptMemory()
2  {
3      char[] key = getKey();
4      DecryptData(key);
5      wipeKey(key); // Compiler removes this line
6      free(key);
7  }
8  Function wipeKey(char[] key)
9  {
10     for(int i=0;i<key.length;i++)
11         Key[i] = '';
12 }
```

Suppose the above code is being compiled and follows the guideline of clearing the sensitive information before freeing to avoid the issue discussed in the section 3.1. The compiler might remove line 5 from the code because the optimizer might think that wiping of information does not lead to any action/operation as the following line frees the memory (line 6). Therefore, when dealing with sensitive information, object code should be cross verified when operations are being performed that does not perform any operation/action that does not change the program's logic. For disabling the compiler from optimizing the code, we can write our code in the block as shown in 3.7.

Program 3.7: Disabling compiler optimization for block of code

```
1 #pragma GCC push_options
2 #pragma GCC optimize ("O0")
3
4 // Code that should not be optimized goes here
5
6 #pragma GCC pop_options
```

3.8 Recommended practices

3.8.1 Checking Data after Null Character

In order to secure our system, we validate the input before using it in our program. These validations usually are done till we reach the null character in the data stream because it represents the end of the data. It is also one of the weaknesses that the attacker can exploit to gain access to the system. Attackers always search for opportunities or memory gaps to store their malicious code; one place for storing the malicious code is after the null character. This allows attackers to remove the null character just before executing the code, which leads to the execution of the malicious/unwanted code. Execution of malicious code could lead to a sensitive data leak, create a backdoor, or escalate the user privilege. The above-described attack falls in the category of the Time-Of-Use and Time-of-Check. Therefore, input or the allocated memory should be validated until the end of the assigned memory length rather than validating until the null character is encountered.

For example: If the input is read from allocated memory where the pointer is passed to the function by the program. This function validates the pointer data before executing it to make sure the malicious program is not executed. Now suppose a pointer is passed to the program, which is pointing at the address 0x21 as shown in the figure 3.9. When validated by the application till the null character, it should I validate the passed argument

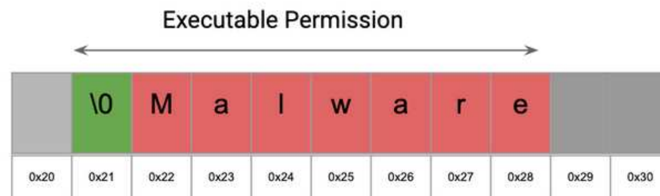

Figure 3.9: Checking Data after Null Character.

without any issues because it considers the input is blank or has valid input. However, if just before the execution of the input command, the attacker was able to manipulate the memory and get rid of the null character present in the 0x21 address. Then in such a scenario, the malicious code will be executed, and the system will be exploited. Therefore, weaknesses like this should be absent in the system as it leaves the system vulnerable.

3.8.2 Storing Hard coded or unprotected Key

Many vendors hard-code the passwords, directories, and other confidential information, thinking source code will not be present in the public domain, thus, making it transparent. They assume that hardcoded sensitive information cannot be extracted from binary files. However, such information can be extracted by reverse engineering and can lead to information leakage. It can be further exploited if the attacker can create a debugging environment and change those values, leading to unwanted program behaviours. Many firmware does have hard-coded sensitive information which is exploited by the attacker, like DLink Router DIR-878, HP Printer PageWide Pro 452dw, etc. Thus, sensitive information should not be hardcoded in the program. This is a weakness also known to the industry as CWE-256 "Plaintext Storage of Password" [69] and CWE-259 "Hard Coded Password" [70]. As it is included in the CWE list and OWASP Top Ten 2017 Category A2 - Broken Authentication, it showcases that this weakness is dangerous to the system.

3.8.3 Wiping non-volatile sensitive data on Restart

System stores sensitive information in the non-volatile memory so the system can run smoothly/efficiently throughout. However, the information that is not required by the system on a restart should be wiped as it can be exposed to the attacker. Generically, this sensitive information is the data that is generated during the intermediate stage of the firmware. Sensitive information is wiped when the system is shut down properly. If forceful/unintended restart occurs, then there is a chance that the programmer might miss doing these checks and wipe the data.

Chapter 4

Experiments

4.1 Environment

For our experiments, we have created a Linux and Windows environment for running the CBMC; both run the latest version (5.11) of CBMC. The Linux operating system is Ubuntu 18.04.5 LTS (Bionic Beaver) with 12 core processors @ 3.20GHz and 8GB memory. Windows operating system is Windows 10 with i7 6 core processor @ 2.60GHz and 16GB memory.

Table 4.1: Environment

Software	Versions
Operating System	Ubuntu 18.04.5 LTS
gcc	7.5.0
g++	7.5.0
flex	2.6.4
bison	3.0.4
make	4.1
patch	2.7.6

4.2 Juliet Test Suite

For our experiments, we have utilized Juliet Test Suite for benchmarking SecMD-Checker. Juliet Test Suite is a Common Weakness Enumerations (CWEs) test suite which contains 64099 test cases in the C/C++ organized under different CWEs. It is a part of a

software assurance reference dataset (SARD) and was created by the NSA's Center for Assured Software (CAS). The purpose of the Juliet Test Suite is to act as a set of security flaws enabling users to evaluate tools to test their methods. It includes files, scripts, and headers needed to compile the test cases, either as one program per test case or all CWE test cases together. These test cases contain programs for both Linux and Microsoft Windows environments. Utilizing the Juliet Test Suite does limit the scope of our approach for testing it on real examples. However, in the absence of a definitive source of code containing the vulnerability, we consider Juliet Test Suite the best available repository. Juliet Test Suite has been used for experimental validation by different research [71–74]. Table 4.2 represents a list of CWEs that are relevant to memory errors present in the Juliet Test Suite.

Table 4.2: Relevant Common Weakness Enumerations

Relevant Common Weakness Enumerations	
CWE-121: Stack Based Buffer Overflow	CWE-369: Divide by Zero
CWE-122: Heap Based Buffer Overflow	CWE-127: Buffer Underread
CWE-259: Hard Coded Password	CWE-126: Buffer Overread
CWE-590: Free Memory Not on Heap	CWE-244: Heap Inspection
CWE-482: Comparing Instead of Assigning	CWE-191: Integer Underflow
CWE-319: Cleartext Tx Sensitive Info	CWE-416: Use After Free
CWE-226: Sensitive Information Uncleared Before Release	CWE-401: Memory Leak
CWE-256: Plaintext Storage of Password	CWE-124: Buffer Underwrite
CWE-481: Assigning Instead of Comparing	CWE-190: Integer Overflow
CWE-321: Hard Coded Cryptographic Key	CWE-415: Double Free
CWE-476: NULL Pointer Dereference	

4.3 Result for detecting other memory related CWEs

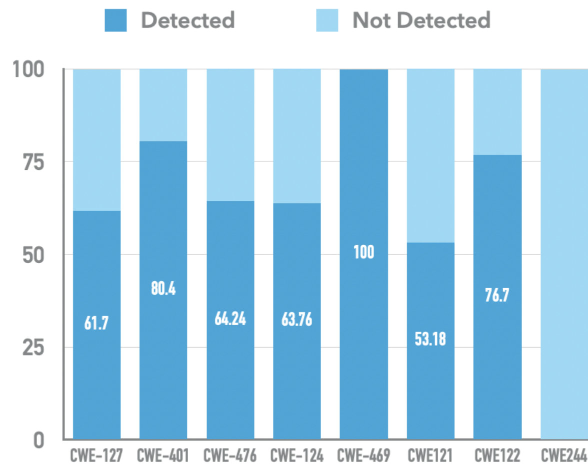

Figure 4.1: CBMC Test Result on Juliet Test Suite.

We tested CBMC for detecting other memory-related CWEs using the predefined CBMC properties (section 4.3.1). We ran CBMC directly on the test cases (without any instrumentation) that are present in Juliet Test Suite. Figure 4.1 illustrates the results depicting how many CWEs were detected by the CBMC when bounded to 200 iterations. The X-axis represents the CWEs that we picked from the Juliet Test Suite, and Y-axis represents the number of CWEs detected. Detection of CWE in test cases depicts that some of the predefined properties were violated during the CBMC analysis. CBMC could not detect any of the CWE-244 test cases with available options as predefined properties in CBMC do not target specific CWEs.

Detailed results of the test run are shown in 4.3. The first column represents the CWE, the second column is the CWE name, the third column represents the number of files in which CBMC detected a memory error, the fourth column represents the number of files on which CBMC does not report any violation, and the last column represents the total number of files on which test was run for the particular CWE. These results show the potential of CBMC in detecting memory-related CWEs.

Table 4.3: Test Results: Detection of memory related CWEs

CWE	CWE Name	Detected	Not Detected	Total
CWE-127	Buffer Underread	1170	726	1896
CWE-401	Memory Leak	988	240	1228
CWE-121	Stack Based Buffer Overflow	3140	2766	5906
CWE-124	Buffer Underwrite	1209	687	1896
CWE-476	NULL Pointer Dereference	239	133	372
CWE-122	Heap Based Buffer Overflow	2807	849	3656
CWE-244	Heap Inspection	0	18	0

4.3.1 CBMC Properties

CBMC has predefined properties which can be accessed using various options [66]. We have utilized the following parameters to check CBMC capability on Juliet Test Suite to check memory-related errors.

- Buffer OverFlow
  - --pointer-check checks errors related to pointers like double free, invalid pointer, dead object etc.
  - --pointer-overflow-check checks error related to pointer arithmetic overflow and underflow
  - --bounds-check checks error related to array bounds
- Memory Leak
  - --signed-overflow-check checks error related to signed arithmetic overflow and underflow checks

– --memory-leak-check checks for memory leak in scenarios where allocated memory is never freed, memcpy src/dst overlaps, memcpy destination regions are writeable and memcpy source regions are readable beyond their bounds.

4.4 SecMD-Checker Test Results

We verified our proposed solution on the CWE-244 (Improper Clearing of Heap Memory Before Release) test cases present in the Juliet Test Suite. We were able to detect all the existing 72 test cases for CWE-244 using our approach. SecMD-Checker has set the CBMC iteration bound to 200 for the experiments. Test cases present in Juliet Test Suite consist of functions with suffix 1) __good: Correct implementation of code where respective CWE is absent, and 2) __bad: Incorrect implementation of code where respective CWE occurs. Our test results for the experiments are shown in Table 4.4. Our instrumentation approach detected 100% True Positive and 100% True Negatives with the examples present in the Juliet Test Suite. However, Juliet Test Suite does not demonstrate the full potential of our tool as SecMD-Checker can handle more complex programs.

Table 4.4: Test Results: Detection of CWE-244

	Positives	Negatives
True	100% (True Positives)	100% (True Negatives)
False	0% (False Positives)	0% (False Negatives)

Chapter 5

Conclusions and Future Directions

We developed a prototype SecMD-Checker implementing our approach for detecting non-secure memory deallocation (Heap Inspection). We showed that our approach is an effective and practical way of detecting CWE-244. As the experimental section states, by utilizing our approach, we were able to instrument the code and detect all the examples present in the Juliet Test Suite CWE-244 test cases that were initially undetected by the CBMC.

Future directions for our research would be 1) use of an actual parser for extracting information instead of using a regex routine as it could be inefficient for general C programs, and 2) improvement of the tool algorithm by including fine-grain analysis and alias analysis for better accuracy.

Bibliography

[1] Phil Koopman. A case study of toyota unintended acceleration and software safety. *Presentation. Sept*, 2014.

[2] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, Tadayoshi Kohno, et al. Comprehensive experimental analyses of automotive attack surfaces. In *USENIX Security Symposium*, volume 4, pages 447–462. San Francisco, 2011.

[3] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, et al. Experimental security analysis of a modern automobile. In *The Ethics of Information Technologies*, pages 119–134. Routledge, 2020.

[4] Trammell Hudson and Larry Rudolph. Thunderstrike: Efi firmware bootkits for apple macbooks. In *Proceedings of the 8th ACM International Systems and Storage Conference*, SYSTOR '15, New York, NY, USA, 2015. Association for Computing Machinery.

[5] John Viega. Protecting sensitive data in memory. https://www.cgisecurity.com/lib/protecting-sensitive-data.html.

[6] John Viega and Gary R McGraw. *Building secure software: How to avoid security problems the right way, portable documents*. Pearson Education, 2001.

[7] MITRE. CWE 244 Heap Inspection. https://cwe.mitre.org/data/definitions/244.html.

[8] NVD. CVE-2019-3733. https://nvd.nist.gov/vuln/detail/CVE-2019-3733.

[9] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *USENIX Security Symposium*, pages 321–336, 2004.

[10] Jim Chow, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Shredding your garbage: Reducing data lifetime through secure deallocation.

[11] Adrian Bednarek. Password managers: Under the hood of secrets management, 2020.

[12] Jun Wang, Mingyi Zhao, Qiang Zeng, Dinghao Wu, and Peng Liu. Risk assessment of buffer "heartbleed" over-read vulnerabilities. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 555–562, 2015.

[13] Daniel Stenberg. curl: RTSP bad headers buffer over-read. https://seclists.org/oss-sec/2018/q2/116.

[14] Ofir Arkin and Josh Anderson. EtherLeak: Ethernet frame padding information leakage. https://dl.packetstormsecurity.net/advisories/atstake/atstake_etherleak_report.pdf.

[15] Kiavash Satvat and Nitesh Saxena. Crashing privacy: An autopsy of a web browser's leaked crash reports. *arXiv preprint arXiv:1808.01718*, 2018.

[16] Michal Zalewski. two browser mem disclosure bugs. https://seclists.org/fulldisclosure/2014/Oct/66.

[17] Wade Mealing. uninitialized kernel data leak in userspace coredumps. https://seclists.org/oss-sec/2020/q2/88.

[18] Solaris (and others) ftpd core dump bug . https://insecure.org/sploits/ftpd.pasv.html.

[19] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *17th USENIX Security Symposium (USENIX Security 08)*, San Jose, CA, July 2008. USENIX Association.

[20] Peter Gutmann. Secure deletion of data from magnetic and solid-state memory. In *Proceedings of the Sixth USENIX Security Symposium, San Jose, CA*, volume 14, pages 77–89, 1996.

[21] NIST. Platform Firmware Resiliency Guidelines. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-193.pdf.

[22] Khaled M Mustafa, Rafa E Al-Qutaish, and Mohammad I Muhairat. Classification of software testing tools based on the software testing methods. In *2009 Second International Conference on Computer and Electrical Engineering*, volume 1, pages 229–233. IEEE, 2009.

[23] David Maynor. *Metasploit toolkit for penetration testing, exploit development, and vulnerability research*. Elsevier, 2011.

[24] Jay Beale, Haroon Meer, Charl van der Walt, and Renaud Deraison. *Nessus Network Auditing: Jay Beale Open Source Security Series*. Elsevier, 2004.

[25] Brian Caswell and Jay Beale. *Snort 2.1 intrusion detection*. Elsevier, 2004.

[26] Michael D Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, 2003.

[27] Stephan Merz. Model checking: A tutorial overview. In *Proceedings of the 4th Summer School on Modeling and Verification of Parallel Processes*, MOVEP '00, page 3–38, Berlin, Heidelberg, 2000. Springer-Verlag.

[28] Dawson Engler and Madanlal Musuvathi. Static analysis versus software model checking for bug finding. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 191–210. Springer, 2004.

[29] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software*, SPIN '01, page 103–122, Berlin, Heidelberg, 2001. Springer-Verlag.

[30] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Software verification with blast. In Thomas Ball and Sriram K. Rajamani, editors, *Model Checking Software*, pages 235–239, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[31] Daniel Kroening and Michael Tautschnig. Cbmc – c bounded model checker. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–391, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[32] Byron Cook, Kareem Khazem, Daniel Kroening, Serdar Tasiran, Michael Tautschnig, and Mark R. Tuttle. Model checking boot code from aws data centers. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 467–486, Cham, 2018. Springer International Publishing.

[33] Minjae Byun, Yongjun Lee, and Jin-Young Choi. Analysis of software weakness detection of cbmc based on cwe. In *2020 22nd International Conference on Advanced Communication Technology (ICACT)*, pages 171–175, 2020.

[34] Kostyantyn Vorobyov and Padmanabhan Krishnan. Comparing model checking and static program analysis: A case study in error detection approaches. 01 2010.

[35] NIST. Juliet Dataset. https://samate.nist.gov/SRD/testsuite.php, 2017. [Online; accessed 12-Dec-2020].

[36] Yaniv David, Nimrod Partush, and Eran Yahav. Firmup: Precise static detection of common vulnerabilities in firmware. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 392–404, New York, NY, USA, 2018. Association for Computing Machinery.

[37] Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Karonte: Detecting insecure multi-binary interactions in embedded firmware. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1544–1561, 2020.

[38] Zafeer Ahmed, Ibrahim Nadir, Haroon Mahmood, Ali Hammad Akbar, and Ghalib Asadullah Shah. Identifying mirai-exploitable vulnerabilities in iot firmware through static analysis. In *2020 International Conference on Cyber Warfare and Security (ICCWS)*, pages 1–5, 2020.

[39] D. Wagner and R. Dean. Intrusion detection via static analysis. In *Proceedings 2001 IEEE Symposium on Security and Privacy. S P 2001*, pages 156–168, 2001.

[40] Steven P. Miller, Michael W. Whalen, and Darren D. Cofer. Software model checking takes off. *Commun. ACM*, 53(2):58–64, February 2010.

[41] B. Schwarz, Hao Chen, D. Wagner, G. Morrison, J. West, J. Lin, and Wei Tu. Model checking an entire linux distribution for security violations. In *21st Annual Computer Security Applications Conference (ACSAC'05)*, pages 10 pp.–22, 2005.

[42] Hao Chen and David Wagner. Mops: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 235–244, 2002.

[43] Marius Bozga, Conrado Daws, Oded Maler, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. Kronos: A model-checking tool for real-time systems. In Anders P.

Ravn and Hans Rischel, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 298–302, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[44] C. Daws, A. Olivero, and S. Yovine. *Verifying ET-LOTOS programs with KRONOS*, pages 227–242. Springer US, Boston, MA, 1995.

[45] C. Daws and S. Yovine. Two examples of verification of multirate timed automata with kronos. In *Proceedings 16th IEEE Real-Time Systems Symposium*, pages 66–75, 1995.

[46] O. Maler and S. Yovine. Hardware timing verification using kronos. In *Proceedings of the Seventh Israeli Conference on Computer Systems and Software Engineering*, pages 23–29, 1996.

[47] Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. Using abstractions for the verification of linear hybrid systems. In *Proceedings of the 6th International Conference on Computer Aided Verification*, CAV '94, page 81–94, Berlin, Heidelberg, 1994. Springer-Verlag.

[48] Kenneth L. McMillan. *Symbolic Model Checking*, pages 25–60. Springer US, Boston, MA, 1993.

[49] W. Chan, R.J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J.D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, 1998.

[50] R.W. Ritchey and P. Ammann. Using model checking to analyze network vulnerabilities. In *Proceeding 2000 IEEE Symposium on Security and Privacy. S P 2000*, pages 156–165, 2000.

[51] G.J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[52] Jiří Barnat, Luboš Brim, Ivana Černá, Pavel Moravec, Petr Ročkai, and Pavel Šimeček. Divine–a tool for distributed verification. In *International Conference on Computer Aided Verification*, pages 278–281. Springer, 2006.

[53] Gerard J. Holzmann. Software model checking with spin. volume 65 of *Advances in Computers*, pages 77–108. Elsevier, 2005.

[54] Koushik Sen and Mahesh Viswanathan. Model checking multithreaded programs with asynchronous atomic methods. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification*, pages 300–314, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[55] Kees Verstoep, Henri E Bal, Jiri Barnat, and Lubos Brim. Efficient large-scale model checking. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12. IEEE, 2009.

[56] Jiří Barnat, Luboš Brim, and Ivana Černá. Cluster-based ltl model checking of large systems. In *International Symposium on Formal Methods for Components and Objects*, pages 259–279. Springer, 2005.

[57] Tal Garfinkel, Ben Pfaff, Jim Chow, and Mendel Rosenblum. Data lifetime is a systems problem. In *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, pages 10–es, 2004.

[58] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Trans. Program. Lang. Syst.*, 21(4):848–894, July 1999.

[59] Amer Diwan, Kathryn S McKinley, and J Eliot B Moss. Type-based alias analysis. *ACM Sigplan Notices*, 33(5):106–117, 1998.

[60] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for c programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming*

Language Design and Implementation, PLDI '95, page 1–12, New York, NY, USA, 1995. Association for Computing Machinery.

[61] MITRE. CWE list. https://cwe.mitre.org/data/index.html.

[62] Rachid Hadjidj, Xiaochun Yang, Syrine Tlili, and Mourad Debbabi. Model-checking for software vulnerabilities detection with multi-language support. In *2008 Sixth Annual Conference on Privacy, Security and Trust*, pages 133–142, 2008.

[63] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[64] Benjamin Schwarz, Hao Chen, David Wagner, Jeremy Lin, Wei Tu, Geoff Morrison, and Jacob West. Model checking an entire linux distribution for security violations. In *Proceedings of the 21st Annual Computer Security Applications Conference*, ACSAC '05, page 13–22, USA, 2005. IEEE Computer Society.

[65] Dirk Beyer, Adam J. Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The blast query language for software verification. In Roberto Giacobazzi, editor, *Static Analysis*, pages 2–18, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[66] D. Kroening. Cprover manual. https://www.cprover.org/cbmc/doc/manual.pdf. [Online; accessed 12-Dec-2020].

[67] Sandra Loosemore, Roland McGrath, Andrew Oram, and Richard M Stallman. *The GNU C library reference manual*. Free software foundation Boston, 2001.

[68] Daniel Kroening. CProver Developer Documentation. http://cprover.diffblue.com/index.html. [Online; accessed 12-Dec-2020].

[69] MITRE. CWE 256 Unprotected Storage of Credentials. https://cwe.mitre.org/data/definitions/256.html.

[70] MITRE. CWE 259 Use of Hard-coded Password. https://cwe.mitre.org/data/definitions/259.html.

[71] Andreas Wagner and Johannes Sametinger. Using the juliet test suite to compare static security scanners. In *2014 11th International Conference on Security and Cryptography (SECRYPT)*, pages 1–9, 2014.

[72] Richard Amankwah, Jinfu Chen, Alfred Adutwum Amponsah, Patrick Kwaku Kudjo, Vivienne Ocran, and Comfort Ofoley Anang. Fast bug detection algorithm for identifying potential vulnerabilities in juliet test cases. In *2020 IEEE 8th International Conference on Smart City and Informatization (iSCI)*, pages 89–94, 2020.

[73] Lakshmi Manohar Rao Velicheti, Dennis C Feiock, Manjula Peiris, Rajeev Raje, and James H Hill. Towards modeling the behavior of static code analysis tools. In *Proceedings of the 9th Annual Cyber and Information Security Research Conference*, pages 17–20, 2014.

[74] Vadim Okun, Aurelien Delaitre, Paul E Black, et al. Report on the static analysis tool exposition (sate) iv. *NIST Special Publication*, 500:297, 2013.