Detecting Secure Memory Deallocation Violations with CBMC

Vinayak S. Prabhu*
Colorado State University
vinayak.prabhu@colostate.edu

Mohit Singh*
Colorado State University
mohit.singh.2.718@gmail.com

Indrajit Ray
Colorado State University
indrajit.ray@colostate.edu

Indrakshi Ray
Colorado State University
indrakshi.ray@colostate.edu

Sudipto Ghosh
Colorado State University
sudipto.ghosh@colostate.edu

ABSTRACT

Scrubbing sensitive data before releasing memory is a widely accepted but often ignored programming practice for developing secure software. Consequently, confidential 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, also known as Common Weakness Enumeration 244, in C programs. 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 combines taint analysis and model checking to detect this weakness. We have three main phases: (1) perform a coarse flow-insensitive inter-procedural static analysis on the program to construct a set of pointer variables that could point to sensitive data; (2) instrument the program with required dynamic variable tracking, and assertion logic for memory wiping before deallocation; and (3) invoke a model checker, the C-Bounded Model Checker (CBMC) in our case, to detect assertion violation in the instrumented program. We develop a tool, SecMD-Checker, implementing our instrumentation based algorithm, and we provide experimental validation on the Juliet Test Suite — the tool is able to detect all the CWE-244 instances present in the test suite. To the best of our knowledge, this is the first work which presents a solution to the problem of detecting unscrubbed secure memory deallocation violations in programs.

CCS CONCEPTS

• **Security and privacy → Software and application security**; *Security requirements*;

KEYWORDS

Secure memory deallocation, C-Bounded Model Checker (CBMC), Common Weakness Enumeration (CWE), taint analysis

ACM Reference Format:
Vinayak S. Prabhu, Mohit Singh, Indrajit Ray, Indrakshi Ray, and Sudipto Ghosh. 2022. Detecting Secure Memory Deallocation Violations with CBMC. In *Proceedings of the 8th ACM Cyber-Physical System Security Workshop (CPSS '22), May 30, 2022, Nagasaki, Japan.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3494107.3522779

1 INTRODUCTION

We consider the problem of sensitive heap memory data not being wiped before deallocation in C programs. Minimizing sensitive data lifetime in software memory is a widely recommended practice for security relevant software in order to reduce the chance of leakage of sensitive data, such as cryptographic keys, to the outside world [46, 47]. Software bugs that leak the contents of memory are common [17]. A recommended practice that addresses part of this vulnerability is to zero or wipe out heap memory before it is deallocated. In the absence of data being wiped before deallocation, we have to rely on the natural life-cycle of data, which ends when new data overwrites the old data during subsequent allocations 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 (defined as from the first write to the last read). The lifetime of sensitive data could be reduced to within 1.35 times the minimum possible data lifetime by zeroing the data at deallocation [14]. Consequently, not following this guideline is viewed as a software weakness. For instance, the Software Engineering Institute (SEI) CERT secure coding standards for C list "MEM03-C: Clear sensitive information stored in reusable resources" as a software recommendation [1]; this recommendation includes wiping dynamically allocated memory before being freed. The MITRE corporation similarly classifies code which does not follow this practice as suffering from the Common Weakness Enumeration-244 – CWE-244: Improper Clearing of Heap Memory Before Freeing ('Heap Inspection') [30]. Unfortunately this practice of wiping sensitive data before freeing is frequently not followed in many applications, including in embedded and Cyber-Physical Systems (CPS) in which C is the dominant language due to its efficiency and it being closer to hardware. Unfortunately these features that make C optimal for CPS domains also come at a cost – C is a memory unsafe language and thus C programs are prone to memory related bugs [44]. Consequently, sensitive data can be often be found leaked from C programs throughout the user and kernel memory and leaked data

*Joint first authors

CPSS '22, May 30, 2022, Nagasaki, Japan

ACM ISBN 978-1-4503-9176-4/22/05…$15.00
https://doi.org/10.1145/3494107.3522779

stays there for an indefinite period [13]. For example, all versions of Dell RSA BSAFE Crypto-C Micro Edition before 4.1.4 were affected by CWE-244, leaving them vulnerable to attacks where a malicious remote user could extract the sensitive information [35].

Leaks could also happen by exploiting other vulnerabilities. For example, Heartbleed is a serious vulnerability present in certain OpenSSL cryptographic software libraries. This library is used on a wide scale on for providing secure communication. The 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 RAM in affected systems [49]. A similar buffer overread problem has been encountered in curl [43]. Etherleak [6] was a vulnerability exposing portions of kernel memory that occurred in ethernet Network Interface Card (NIC) device drivers. It occurred due to incorrect implementations of RFC 1042 requirements and poor programming practices. An attack could be executed by sending an ICMP echo message to the vulnerable machine, which would return portions of kernel memory in the padding of the response messages.

IoT systems consist of diverse entities communicating over the network, this creates a large attack surface through which sensitive data can flow [12, 27]. Non-secure memory deallocation in software that reads encryption keys on Read-Only Memory (ROM) on CPS has an especially severe impact. The purpose of using read-only flash memory, ROM, EPROM, or EEPROM is to make sure that data stored in them is immutable. Such immutable elements are often used when system design requires root-of-trust which can be utilized during the software authentication procedures. Storing root-of-trust data in ROM ensures that the malicious entities cannot alter these keys. However, for authentication, we need to bring that key to the RAM, and if we carelessly free the memory without zeroing out, keys can be exposed when that memory is reallocated. Such cases would necessitate physical replacement of the chip holding the keys and/or reprogramming in the field through a particular upgrading procedure [32]. Both are highly expensive and highly undesirable options. This makes CWE-244 a serious concern for embedded and Cyber-Physical Systems.

Modern SCADA (Supervisory Control and Data Acquisition) systems are on the network; and use core programs written in C, *e.g.*, for the underlying system platform and for running control routines. They use standard network and operating systems libraries, and hence inherit corresponding memory related weaknesses. In the course of standard operation, these systems periodically send out data depending on the requests they receive. As this data is usually retrieved from RAM, this opens up the possibility of an attacker gaining access to highly sensitive data, such as root-of-trust keys, from an out-of-bounds read attack on CWE-244 vulnerable SCADA subsystems. Network library vulnerabilities such as Heartbleed and Etherleak can similarly be exploited on CWE-244 vulnerable systems to gain access to sensitive data. Typically, security has taken a back seat to control engineering concerns for such industrial control systems; and given the potential catastrophic infrastructure consequences involving entities such as the power grid or nuclear power plants, leakage of secure data is one of the urgent security concerns in this area [42, 54].

Detection of CWE-244 in programs is challenging as it entails first ascertaining *which* memory locations contain sensitive data. In a typical use scenario, it is reasonable for developers to specify which variables *initially* get sensitive data. However, this sensitive data gets copied to *other* variables and memory locations, possibly through multiple function calls, both via shallow copies (copy by reference), and via deep copies (copy by value); with data being copied partially or wholly. The technical challenge is then to infer in a precise enough manner (1) how sensitive data gets copied and spread over the heap during the course of program execution, and (2) detect if *any* of these heap memory locations get freed without being wiped. We observe that simpler problems, such as that of alias detection, are already undecidable [37], hence any developed solutions must be approximate. At the same time, in order for a solution to be useful to developers, it needs to target a low false positive, and a low false negative rate.

Our Contributions. In this work, we propose a taint analysis [40], approach combining static analysis and model checking for detecting secure memory deallocation violations in C programs; and develop a prototype tool – SecMD-Checker (Secure Memory Deallocation Checker) – implementing the proposed algorithm. Our tool takes two inputs: (1) a compilable C program, and (2) the name of a pointer to a location holding sensitive data. The tool then conservatively checks if this original memory location, and other memory locations that have been copied from it, are always wiped out before they are freed. To the best of our knowledge, ours is the first work which presents a solution to the problem of detecting unscrubbed secure memory deallocation violations in programs.

Our algorithm has three main phases: (1) first, it performs a coarse flow-insensitive inter-procedural 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 both required dynamic variable tracking, and assertion logic for memory wiping before deallocation, 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 secure memory deallocation violation. Our tool implementation can handle the original tracked variable's shallow copies and also deep copies. We track both sorts of copies by keeping track of memory addresses, not names. Tracking this information in the program itself enables us to reduce potential false positives due to program flow and function calls. Note that we insert wiping checks before free statements. This leaves the possibility of memory leaks where sensitive data is left in memory that is not freed before the program terminates. Memory leaks, however, can be directly detected by analyzers such as CBMC.

Conceptually, our approach can be seen as implementing a form of taint analysis where taint propagation is only for tainted data copies, not for control or information flow. In addition, a data location being tainted at some point does not mean it is tainted when it is *freed*. We solve this taint sanitization problem by using CMBC in our final stage to check whether memory when being freed actually

contains sensitive data. While we use CBMC in our work in the last stage, our algorithm is also applicable to other model checkers and static analyzers, with appropriate minor changes to the tool-specific assertion logic code.

Experimental Validation of our Tool. For experimental validation of our technique, we utilized the Juliet Test Suite [33], a collection of test cases in C/C++ classified under different CWEs maintained by NSA. We ran SecMD-Checker on two groups of tests from the Juliet Test Suite. The first group contained all the tests that were designed for CWE-244 (there were 72 such tests in the suite). The second group consisted of 95 randomly chosen tests out of the 64099 tests in the test suite that were designed for *other* CWEs, and which had heap memory allocation. 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 deallocations. Both of these combined provide evidence of the analysis of our tool being reasonably precise.

Related Work. IoT operating systems are tested for CWE vulnerabilities using three static analysis tools in [3]. The work in [19] presents the lifetime issue for sensitive data. The work in [14] shows that 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; and it proposes a way of ensuring this by making changes in the compiler, libraries, and kernel, however changing the system environment to ensure this is not always feasible. Aliasing occurs when the memory location can be accessed by more than one name. Many techniques are present for alias analysis, such as inter-procedural pointer alias analysis [21], type-based alias analysis [16], and context-sensitive pointer analysis [52]. Detection of related memory related vulnerabilities such as buffer overflows is also a widely investigated area [2, 10, 25, 38]. Approaches based on taint analysis have been used for various problems related to information flow security [7, 18, 31, 40, 53]. In the context of industrial control systems, static and dynamic analysis have been used to detect security vulnerabilities in works [9, 50]. Works such as [4, 12, 27, 34, 39] address IoT security concerns using taint analysis.

2 PRELIMINARIES

2.1 Common Weakness Enumerations (CWEs)

Exploitable vulnerabilities in software systems are known as Common Vulnerabilities and Exposures (CVE). The CVE program has been developed at MITRE to identify, define, and catalog publicly disclosed cybersecurity vulnerabilities [29]. Each vulnerability in the CVE catalog is assigned a CVE Record. The causes of the vulnerabilities are categorized as Common Weakness Enumerations (CWEs). These causes are software and hardware weaknesses – faults, bugs, flaws, or other errors in software or hardware implementation, which if left unaddressed, could result in systems, networks, or hardware being vulnerable to attacks. The CWE list, which is a community list, is based in part on the CVEs, and is categorized into those by software development, by hardware design, and by research concepts. CWEs work as a common language, measuring matrix and baseline for weakness identification, mitigation, and prevention efforts. The CWE list has over 600 categories, including classes for race conditions, buffer overflows, non-secure random numbers, and hardcoded passwords. Several tools and services have been developed to find security weaknesses based on the CWE list [28, 51]. A detailed CWE list is available at the MITRE website [30]. The CWEs are organized in a hierarchical fashion, with parent/child relationships based on various concepts. Examples of CWEs are CWE-121 (Stack Based Buffer Overflow), CWE-122 (Heap Based Buffer Overflow), CWE-126 (Buffer Overread), CWE-401 (Memory Leak), CWE-415 (Double Free), and CWE-416 (Use After Free). Buffer overflow occurs when we write data more than the allocated memory size on a heap or stack. Buffer overread is a weakness where we read the data more than the size of the allocated memory buffer. CWE-401 occurs when the program does not deallocate memory. CWE-415 occurs if a user deallocates the memory twice. CWE-416 occurs if a user accesses the memory after deallocation.

2.2 CWE-244: Improper Clearing of Heap Memory Before Release

Security practices like zeroing out memory before freeing, also known as scrubbing or secure memory deallocation, are often neglected by the programmer when it comes to protecting sensitive data such as passwords and cryptographic keys. Operating systems do not usually clear the previously written information, and as a result sensitive data is leaked when memory is reallocated. This software weakness is part of CWE-244. This CWE is part of the related weakness CWE-226: sensitive information in resource not removed before reuse. CWE-226 includes other types of weaknesses (such as CWE-1239: Improper Zeroization of Hardware Register). In this work, we focus on detecting failures to remove sensitive information from heap memory before the memory being freed.

The Software Engineering Institute (SEI) has defined a secure coding standard for C programming language referred to as SEI CERT C coding standard [1]. The standard provide guidelines targeting insecure coding practices. These guidelines are defined in terms of rules and recommendations. One of the recommendations on Memory Management is to clear sensitive information stored in reusable resources (Mem03) before freeing, which is directly related to CWE-244.

2.3 Sensitive Data Replicates on the Heap

In C a *shallow copy* stores the reference of the object to the original memory address. Shallow copies only clone the reference, not the actual object; thus, any changes done on the shallow copy reflect in the original object. A shallow copy is efficient as it only copies a reference of the object, but sensitive data itself is not replicated. The allocated memory can be free using any reference object; after freeing the memory, none of the reference pointers can be used to access the original data. In contrast, a *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.

2.4 Objective: Detect Improper Removal of Sensitive Heap Data Replicates

Our objective in this work is two-fold. First, we track (conservatively) all possible data replicates, arising from both shallow and deep copies, of a given initial variable pointing to sensitive data on the heap; and second, we check if all of these data replicates are properly wiped before the corresponding memory is released to the operating system.

2.5 CBMC – C Bounded Model Checker

Model Checking [8, 20] computes the run-time states of an input program without actually running the program; these states are further utilized to check whether a specific property/specification holds for the system. One of the model checkers that have found widespread use is CBMC – C Bounded Model Checker created by Daniel Kroening at Carnegie Mellon University [24]. It can verify user defined assertions under a given loop unwinding bound, such as those relating to array bounds (buffer overflows), pointer safety, arithmetic exceptions and can provide a counterexample if an assertion is violated. Currently, it supports C89, C99, most of C11, and most compiler extensions provided by GCC and Visual Studio. Capabilities of CBMC have been showcased on Common Weakness Enumerations (CWE), and on major systems such as Amazon Web Services [11, 15]. While CBMC has built-in options for various memory safety related issues, it does not come with any options for our problem in Subsection 2.4.

3 OVERVIEW AND SCOPE OF OUR APPROACH

In this section, we present the overview of our approach implemented in our tool, SecMD-Checker (Secure Memory Deallocation Checker), for tracking sensitive data replicates on the heap, and detecting occurrences of deallocations without proper wiping of the data.

3.1 Problem Scope

We assume we are given a single file C-program, and a single variable, `X`, of this program. Our objective is to (1) track all shallow and deep copy replicates on the heap of `X`, and (2) check that these replicates are wiped whenever the corresponding heap memory is deallocated. The problem setting is best illustrated with an example. Consider the program in Listing 1. In the program, variable `X` is the source of the sensitive data, and we wish to track all heap replicates copied from `X`.

Listing 1: Program A

```
#include <wchar.h>
#include <windows.h>

void copy_process(char *Y)
{

  char * Y_deepcp = (char *)malloc(100*
  sizeof(char));
  memcpy(Y_deepcp, Y, 100*sizeof(char));
  char * other = (char *)malloc(99*sizeof(
  char));
  strcpy(other, ``non-sensitive data'');
  free(Y_deepcp);
  free(other);
}

void CWE244_dummy()
{
  char * X = (char *)malloc(100*sizeof(char
  ));
  char * X_shallowcp;
  X_shallowcp = X;
  if (fgets(X, 100, stdin) == NULL){
    printLine("fgets() failed");
    /* Restore NULL terminator if fgets
  fails */
    X[0] = '\0';
  }
  copy_process(X_shallowcp);
  zero_out_memory(X);
  free(X);
}
```

Let `CWE244_dummy()` be the starting point for the program execution. In this function, first a shallow copy is made to `X_shallowcp`, which is then passed to the function `copy_process()`. This function then makes a deep copy to `Y_deepcp`, does some further processing, and then finally frees `Y_deepcp` *without wiping it first.* The control then returns back to `CWE244_Dummy()`, which wipes `X` and frees it. The problem here is that a deep copy of `X`, namely `Y_deepcp` was *not* wiped before being freed. Our objective is to detect such secure memory deallocation violations. In our work, we treat a partial deep copy of a sensitive variable `Y` the same as a full deep copy of the variable.

3.1.1 Limitations. Our tool SecMD-Checker is a proof of concept prototype focused on detecting secure memory deallocation violation. It does not support the following C features: higher order functions and pointers to functions, double pointers, arrays of pointers, structures, classes, and polymorphism.

3.2 Overview of Our Approach

We implemented our tool SecMD-Checker in Python 3.7. It takes two arguments as an input 1) a compilable C program, and 2) the name of the variable holding sensitive data[1]. After we pass the inputs to the program, SecMD-Checker does a flow insensitive inter-procedural analysis on the C program for inferring information about sensitive variables, possible copy chains via both deep and shallow copies, function arguments, and possible function calls. Next, utilizing this information, SecMD-Checker instruments the program for detection of non-secure memory deallocation as follows. The instrumentation includes 1) declaring variables that track and monitor program variables containing sensitive data without changing the control flow logic of the original C program, 2) inserting logic for tracking copies of sensitive data into the program, and 3) embedding assertion logic (that we define) before every "free()" statement that deallocates memory containing potentially sensitive

[1]For multiple variables, we rerun the tool for each variable

data, The instrumented file has sufficient information to know when and how to validate potentially non-secure memory deallocation. CBMC is then invoked to analyze the instrumented C program file. If secure memory deallocation violations are present, an alert is raised due to assertion violations, and a failure trace obtained for CWE-244. Otherwise, CBMC indicates successful validation of the CWE-244 test.

Figure 1 shows the high level architecture of SecMD-Checker.

4 SECMD-CHECKER: DETECTING SECURE MEMORY DEALLOCATION VIOLATIONS

In this Section, we first present the various phases of the tool in detail. Subsection 4.1 presents the procedures implementing the program analysis phase. Subsection 4.2 presents the procedures which instrument the input program, based on information from the analysis phase. The instrumented program is then fed to CBMC, and described in Subsection 4.3.

Subsection 4.4 discusses issues related to the `realloc function`. Subsection 4.5 presents compiler optimization concerns. Finally, Subsection 4.6 discusses the soundness and completeness of our approach.

4.1 Program Analysis

The first phase of SecMD-Checker performs a flow insensitive static analysis of the program and creates a graph capturing the copy dependencies between variables. The nodes of this graph are the variables of the program. As the same variable names may occur in different functions, we add an appropriate suffix to indicate which function the variable scope belongs to. For example, if variable y belongs to function foo, then to y we add the suffix "__foo". If the variable y is a global variable, then we add the suffix "__Global". Once the nodes have been created, we add directed edges $y \longrightarrow z$ in the following instances: (a) if y is shallow or deep copied to z; (b) if z is an argument in a function definition type, and that function is called at some point with the value y for z. For example, consider the program in listing 1. Our first phase will generate the copy dependency graph in Figure 2.

Once the copy dependency graph has been created, we compute all the nodes that are reachable from the variable X, the originating variable pointing to sensitive data. This set of nodes is called `AllCopiesSet`. If there is any chain of shallow or deep copies, including across function calls, that could lead from X to Y, the variable Y (with the appropriate suffix) will be in `AllCopiesSet`. The pseudo-algorithm of this phase can be found in Procedure FindCopies, Procedure AddVerticesEdges, and Procedure AddProperSuffix.

4.1.1 Procedure Details. We now explain the details of the procedures that need clarification.

Procedure FindCopies. This procedure calls Procedure AddVerticesEdges to create the graph with the variable dependencies, and invokes a reachability function `ComputeReach` to construct `AllCopiesSet`. In line 6, it calls a routine to extract and create a list of all the function definition prototype, function definition body pairs from the given input C program. This is needed in order to tag variables with the names of the function scope they belong to in other procedures.

```
Procedure 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← GetGlobalVarDefs(program_file);
3 for global_var ∈ Global_Vars do
4 |  AddVertex(G, concat(global_var, "__Global"));
5 end
  /* Extract all functions                        */
6 Program_Functions← ExtractFunctions(program_file);
  // Add Vertices and Edges by analyzing 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 AddVerticesEdges. In this procedure, first we create the variables in the graph, based on the pointer variables in the function prototype (lines 4-6), and the pointer variable declarations (lines 8-11). We add the suffix "`__func_name`" as described earlier. If further on in the procedure, there is a pointer variable in a statement for which a corresponding node has not been created in the function, that variable is interpreted as a global variable; nodes for global variables are created in Procedure FindCopies. The function `AddProperSuffix` does this analysis in order to tag variables with appropriate suffixes ("`__func_name`" or "`__Global`") in lines 13-30.

Edges from shallow and deep copy assignments are created in lines 13-18. Edges corresponding to argument instantiations in function calls are created in lines 19-29 (with appropriate suffixes added to the variable names). For example, in Program 1, on line 26, there is a function call `copy_process(X_shallowcp)`. The function definition prototype is on line 4 of the listing. As the variable `X_shallowcp` is an argument in the function call invocation `copy_process(X_shallowcp)`, we add an edge `X_shallowcp__CWE244_dummy` ⟶ `Y__copy_process`. If a statement is an assignment from a function returning a pointer, for example, `Z = foo(var)`, then the line is treated both for the function call, as well as for the assignment to `Z`, and edges are added for both (with one edge being from the return variable of `foo()` to `Z`).

The technical details of lines 20-23 are as follows. On line 20, we assign to `called_func` the called function from the list of program functions. This `called_func` assignment includes both the function definition prototype, as well as the function body. On line 21 we extract the arguments of the particular function call (which are in general different from the argument names in the function definition prototype). On line 22, we extract the name of the called function in order to add the appropriate suffixes. On line 23, we extract the argument names from function definition prototype

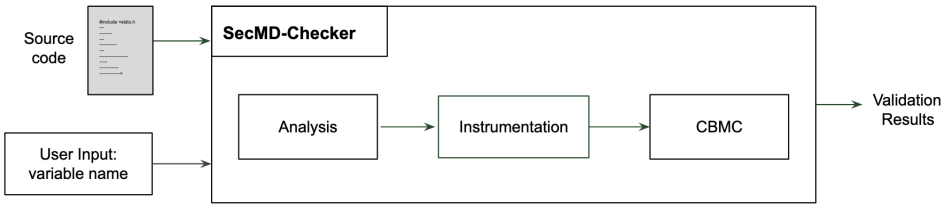

Figure 1: SecMD-Checker Architecture

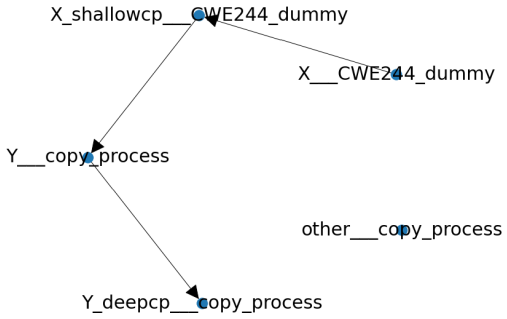

Figure 2: Generated graph edges

of `called_func`, in order to add edges to them from the appropriate variables in the function call in the current line of the input C program that is being analyzed.
Procedure AddProperSuffix. This function adds appropriate suffixes to the variable names ("`__func_name`" or " `__Global`") when generating the graph edges.

At the end of the analysis phase, the procedures described above will generate the copy dependency graph in Figure 2 for the program in listing 1. On line 8 of the program, this dependecy graph is utilized to find all the set of all reachable nodes, `AllCopiesSet`, from the input variable `X__CWE244_dummy` using a reachability algorithm:

$$\texttt{AllCopiesSet} = \left\{ \begin{matrix} \texttt{X\_\_CWE244\_dummy}, \texttt{X\_shallowcp\_\_CWE244\_dummy}, \\ \texttt{Y\_\_copy\_process}, \texttt{Y\_deepcp\_\_copy\_process} \end{matrix} \right\}.$$

4.2 Program Instrumentation

In this subsection we describe the instrumentation phase of SecMD-Checker. This phase incorporates two key ideas. First, we track actual memory addresses that could point to sensitive data during actual executions of the program (not variable names). Before every free of a heap pointer variable that has such a tracked address, we instrument the program, embedding an assertion, checking that the data pointed to by the variable has been wiped. Second, our tracking has a dynamic aspect. The previous phase computes a conservative set `AllCopiesSet` containing all the variable names which could point to sensitive data. However, it can happen that for the same variable Y in `AllCopiesSet`, one free occurrence of Y frees non-sensitive data, and another free occurrence statement over Y frees sensitive data. This happens, for example, due to different function call chains. In order to obtain call chains which indicate true positives for the assertion violation, we build a *dynamic* set of *memory addresses* (by instrumenting the input program) that could point to sensitive data, as opposed to `AllCopiesSet` which is a static set of variable names. However, the variables that point to this memory address set always belong to `AllCopiesSet`. We illustrate this issue in Section 5.

4.2.1 Procedure Details. First we describe two sub-procedures that are used by the main procedure. Subprocedure __AddAddress maintains a dynamic container `__address_holder` which can be viewed as a set; this container holds the current set of heap memory addresses that could contain sensitive data.

Subprocedure __MemoryWipingCheck when invoked over a pointer variable `__variable`, first checks if the address pointed to by `__variable` currently belongs to `__address_holder`, and if so checks if the data has been wiped. This sub-procedure is embedded before appropriate free statements by the main procedure of this phase. The sub-procedure contains two CBMC defined functions: `__CPROVER_OBJECT_SIZE` is a CBMC defined function which can be used to retrieve the size of the object a pointer points to; and `__CPROVER_assert` is the assertion function of CBMC [22].

The main procedure is Procedure CWE-244 Instrumentation Procedure. The procedure maintains a container `__address_holder` which contains memory addresses of potentially sensitive data. The container is updated dynamically in the instrumented program. We note that if a address indicated by a pointer variable is added to `__address_holder`, then it necessarily means that the variable belong to `AllCopiesSet` (with the appropriate suffixes); the other direction may not hold true. Thus `__address_holder` is more selective than `AllCopiesSet`. The procedure also embeds, in lines 19-22, a memory wiping check assertion just before frees of relevant variables. A variable is deemed relevant if its memory address *at that point* belongs to `__address_holder` (this also means the variable must belong to `AllCopiesSet`, a less restrictive condition).

Other parts which need clarifications follow. In lines 9-12, we check if an assignment statement to any variable in `AllCopiesSet` is such that the variable could get a new memory address, and if so we add it to to the container `__address_holder` by inserting a `__AddAddress`call after the assignment line in the program. Note that since `AllCopiesSet` has variable names concatenated with appropriate suffixes as described in Subsection 4.1, we have to do the same concatenation to a variable Y whenever we want to check if Y, with its proper scope, is in `AllCopiesSet`. Lines 13-18 handle function calls involving variables from `AllCopiesSet` as arguments. As a function call `foo(&Y)` could change the location to which `Y` could point to, we add the potentially new memory address contained in the pointer variable `Y` after every function call involving `Y` as a function argument when `Y` belongs to `AllCopiesSet` (accounting for the

Procedure AddVerticesEdges(G, Program_Functions)

```
input : G, Program_Functions
// Iterate the functions to extract pointer
variable names and create link between pointer
variables in a graph
1  for (function in Program_Functions do
2      func_name←GetFuncName(function);
3      Arguments← GetFuncArgs(function);
4      func_body← GetFuncBody(function);
5      foreach argument in Arguments do
6          AddVertex(G, concat(argument, "__", func_name));
7      end
8      foreach line in func_body do
9          if IsVariableDeclaration(line) then
10             var_name ← GetVariableName(line);
11             var_name= concat(var_name, "__", func_name);
12             AddVertex(G, var_name);
13         end
14         if IsAssignmentStatement(line) then
15             var_from, var_to← GetCopyVars(line);
16             var_from= AddProperSuffix(var_from,
                func_name);
17             var_to= AddProperSuffix(var_to, func_name);
18             AddEdge(G, ⟨var_from, var_to⟩);
19         end
20         if IsFunctionCall(line) then
21             called_func← GetFunction(line,
                Program_Functions);
22             CArgs← GetFuncArgs(line);
23             name_cfunc ←GetFuncName(called_func);
24             Fetched_Args ← GetFuncArgs(called_func);
25             for matching (carg, fetched_arg) in (CArgs,
                Fetched_Args) do
26                 carg←AddProperSuffix(carg, func_name);
27                 fetched_arg ←
                    AddProperSuffix(fetched_arg,
                    name_cfunc);
28                 AddEdge(G, ⟨carg, fetched_arg⟩);
29             end
30         end
31     end
32 end
```

proper suffixes). Line 16 takes care of the case when such a function call is part of a predicate of an `if` statement condition. This strategy of updating `__address_holder` after function calls also accounts for calls to third party library function calls with missing bodies or definitions. Note: we do not treat `strcpy`, `memcpy`, `memcpy_s`, `strncpy`, `strcpy_s`, `strncpy_s` as function calls for this purpose, as they do not result in new memory locations being generated, they just change the data.

Procedure AddProperSuffix(var_name, func_name)

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

Procedure __AddAddress(__variable)

```
input : __variable
/* Functionality: if __variable is not present in
   container __address_holder, then add __variable
   to it                                          */
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
```

Procedure __MemoryWipingCheck(__variable)

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

Consider again the program in listing 1. The `AllCopiesSet` for this program was computed at the end of Subsection 4.1. The procedures in the current subsection result in the instrumented program in listing 2. The added function calls are shown indented.

Listing 2: Instrumented Program A

Procedure CWE-244 Instrumentation Procedure(input_c_file, variable_name)

input : input_c_file, var_name
output: program_file

```
program_file ← CodeFormat(input_c_file);
// Convert to certain brace syntax
AllCopiesSet ← FindCopies(program_file, var_name) ;
 // All deep and shallow copies of input var_name
program_file ← Add declaration and initialization
 __address_holder, __count, __AddAddress() and
 __Memory_Wiping_Check();
// Extract all functions; prototypes and bodies
Program_Functions← ExtractFunctions(program_file);
for (function in Program_Functions do
    func_name←GetFuncName(function);
    func_body← GetFuncBody(function);
    for line in func_body do
        /* Variables from ''AllCopiesSet'', are
        being assigned something or initialized. For
        instance var_name= f(), var_name= malloc()   */
        if IsAssignmentStatement(line) then
            var_to, var_from ← GetCopyVars(line);
            if AddProperSuffix(var_to, func_name) ∈
             AllCopiesSet then
                program_file← Insert
                 __AddAddress(var_to) after line;
        /* Variables from ''AllCopiesSet'', are
        passed as arguments to a function call. For
        instance f(.., var_name, ..)                 */
        if IsFunctionCall(AllCopiesSet, line) then
            Var_Names ←
              { var_name | var_name is a function argument on "line",
                           & AddProperSuffix(var_name,func_name)
                           ∈ AllCopiesSet } ;
            if IsIfStatement(line) then
                for var_name ∈ Var_Names do
                 program_file← Insert
                 __AddAddress(var_name) in beginning of if
                 and else block;
            else
                for var_name ∈ Var_Names do
                 program_file← Insert
                 __AddAddress(var_name) after line;
        if IsFreeStatement(line) then
            freedvar← GetFreedVar(line);
            // The variable being freed
            if AddProperSuffix(freedvar, func_name) ∈
             AllCopiesSet then
                program_file ← add
                 __Memory_Wiping_Check(freedvar)
                 function call before line;
```

```c
#include <wchar.h>
#include <windows.h>

#define __holder_size 1000
void *__address_holder[__holder_size];
int __count=0;

void copy_process(char *Y)
{
  char * Y_deepcp = (char *)malloc(100*
  sizeof(char));
            __AddAddress(Y_deepcp);
  char * other = (char *)malloc(99*sizeof(
  char));
  strcpy(other, "non-sensitive data");
  memcpy(Y_deepcp, Y, 100*sizeof(char));
            __MemoryWipingCheck(Y_deepcp);
  free(Y_deepcp);
  free(other);
}

void CWE244_dummy()
{
  char * X = (char *)malloc(100*sizeof(char
  ));
            __AddAddress(X);
  char * X_shallowcp;
  X_shallowcp = X;
            __AddAddress(X_shallowcp);
  if (fgets(X, 100, stdin) == NULL){
            __AddAddress(X);
    printLine("fgets() failed");
    /* Restore NULL terminator if fgets
  fails */
    X[0] = '\0';
            __AddAddress(X);
  }
  copy_process(X_shallowcp);
          __AddAddress(X_shallowcp);
  zero_out_memory(X);
          __AddAddress(X);
          __MemoryWipingCheck(X);
  free(X);
}
```

4.3 CBMC Assertion Checking

Once we have the instrumented C program, SecMD-Checker feeds it to CBMC for assertion violation detection. CBMC analyses the instrumented program to see if any `__CPROVER_assert` assertion violations are possible for corresponding free statements (under a set iteration bound for the program). If any such violations are detected, then these free operations need to be checked as the pointer variables that are being freed could point to sensitive data. *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 possible secure memory deallocation violations.

4.4 Realloc Handling

The `realloc()` function is utilized to extend or shrink the allocated memory block size. It may allocate the memory to a new address as space after the end of the old memory block may be in use, or due to internals of library implementing the function, for instance this can happen even if the memory is being shrunk [26]. 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, by not allowing a user to reallocate the memory at all. Second, by making sure that reallocation is performed only after original memory has been wiped. We did not pursue the first option as it would restrict the user from using `realloc()` altogether, even if they are clearing the memory before calling the function. Therefore, we followed the second option. In order to enable SecMD-Checker to handle the `realloc()` functions, we treat each `realloc()` call as a free followed by an assignment statement. Consequently, we embed a `__Memory_Wiping_Check` function call before every `realloc()` call because the call may deallocate an old memory block (free statement). Moreover, we also embed a `__AddAddress` function call after every `realloc()` statement because a a `realloc()` call 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 a possible secure memory deallocation violation.

Note: Implementation details of `realloc()` handling are omitted from the CWE-244 instrumentation algorithm in Subsection 4.2 for the sake of clarity.

4.5 Compiler Optimization Issues

Compilers change code automatically in order to optimize the running performance. Such changes sometimes change the logic written by the programmer. Memory wiping code is one candidate for compilers to remove during their optimization. We illustrate the problem in Listing 3. Therefore, we should make sure that the object file consists of all the critical parts of the code.

Listing 3: Sample Program

```
1  Function decrypt_memory()
2  {
3      char[] key = get_key();
4      decrypt_data(key);
5      wipe_key(key); //Compiler removes this
6      free(key);
7    }
8
9  Function wipe_key(char[] key)
10 {
11     for(int i=0;i<key.length;i++)
12         key[i] = '';
13 }
```

Program 3 wipes sensitive memory before freeing it. However, a compiler might remove line 5 from the code due to the compiler optimizer inferring that writes due to the call on line 5 do not have any use as the following line frees the memory (line 6).

One option when dealing with sensitive information is to cross verify the object code when operations are being performed that have the potential to be removed by compilers. Another option is to selectively disable compiler optimizations; for instance, we can modify our code from Listing 3 as in Listing 4.

Listing 4: Disabling Compiler Optimizations for Code Block

```
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
```

4.6 Soundness and Completeness Analysis

Our instrumentation approach is complete – if the original program could free a variable pointing to an unwiped sensitive memory area, then the instrumented program will insert an appropriate assertion before the free statement. This can be seen as follows. The computation of `AllCopiesSet` from Subsection 4.1 ensures that if there is any sequence of shallow or deep copies from the originating sensitive variable `X` to any pointer variable `Y`, then `Y` gets included in `AllCopiesSet`. The instrumentation phase from Subsection 4.2 ensures that when new addresses are generated for variables in `AllCopiesSet`, they are tracked; and when any of these tracked addresses are freed, inserted assertions check if the corresponding memory has been wiped. Note that since the final stage involves invoking CBMC, the full chain of our tool is not complete, as CBMC may fail to detect assertion violations that would occur due to an insufficient iteration bound (assertion violation detection is in general an undecidable problem [41]). However, if we define our problem as that of checking for secure memory deallocation violations within a given bound on the number of program steps, then our full approach is complete.

If a memory wiping check assertion fails, this does not necessarily mean that the program is freeing unwiped memory with sensitive data (hence SecMD-Checker may be unsound when it indicates a secure memory deallocation violation). This is because our computation of `AllCopiesSet` is flow and context insensitive, and hence could be overly conservative. We do not consider this to be a significant negative – as such assertion violations indicate that had the program flow been different involving the same variables, then a true positive of deallocation of sensitive data would have occurred. Since a program update could change the program flow, this conservative analysis provides useful feedback to the developers.

5 EXAMPLE – DYNAMIC TRACKING

Our algorithm enables CBMC to dynamically track the memory addresses that are allocated to the identified variables in `AllCopiesSet` for better precision in avoiding false positives. An absence of such dynamic tracking would lead to CBMC possibly generating spurious assertion violations. We illustrate this case by an example.

Consider Listing 5 (the listing shows the instrumented program). Suppose our approach was not to dynamically track addresses as described in Subsection 4.2; instead suppose we were inserting the `__MemoryWipingCheck()` call before every free of a variable that belonged to `AllCopiesSet` that was computed statically in Subsection 4.1; and this call was blindly checking if the `__MemoryWipingCheck()` function argument `__variable` was being wiped (thus only lines 6–8 in Procedure __MemoryWipingCheck). It can be checked that for Listing 5, `arg__dummy_func` belongs to `AllCopiesSet`, hence in this modified approach we would get an assertion violation from CBMC before the free in `dummy_func`; with the function call chain trace `main()` ⟶ `dummy_func(temporary)`. But this trace is spurious as `temporary` does not point to sensitive data. A developer could then view this assertion violation as a false positive of the tool.

However, with our dynamic tracking approach, the call `dummy_func(temporary)` would not lead to an assertion violation as `temporary` does not belong to the dynamically tracked address list, and hence `__MemoryWipingCheck()` does not reach the assertion statement. In contrast, the next function call `dummy_func(X)` *does* cause `__MemoryWipingCheck()` to reach the assertion statement as the memory address pointed to by `X` will be in the set of dynamically tracked addresses. In this case, CBMC returns the function call chain trace `main()` ⟶ `dummy_func(X)` which demonstrates a true positive.

Listing 5: Example: Dynamic Tracking

```
1  void dummy_func(char * arg)
2  {
3          __MemoryWipingCheck(arg);
4      free(arg);
5  }
6
7  void main()
8  {
9      char * X =  (char *)malloc(100*sizeof(
       char));
10         __AddAddress(X);
11     char * temporary =  (char *)malloc(100*
       sizeof(char));
12
13     dummy_func(temporary);
14     dummy_func(X);
15         __AddAddress(X);
16 }
```

6 EXPERIMENTS

The Juliet Test Suite. For our experiments, we utilized the Juliet Test Suite for evaluating SecMD-Checker. The test suite is a Common Weakness Enumerations (CWEs) test suite which contains 64099 C/C++ test cases organized under different CWEs. It is a part of a software assurance reference dataset (SARD) and was created by NSA's Center for Assured Software (CAS). The purpose of the Juliet Test Suite is to have a codebase of categorized 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. This test suite has been used for experimental validation by various researchers [5, 36, 45, 48]. Thus, while we did not run our tool on industrial code, we believe examining on the Juliet Test Suite provides initial evidence for the validity of our approach on other code. The code in the test suite includes functions, as well as loops. Our experiments were run on a machine with an i7 six core 2.60GHz processor and 8GB of memory.

CBMC: Direct Detection of memory related CWEs. CBMC allows direct checking of a few predefined properties through its various options [23]; some of these properties directly map to memory related CWEs, such as buffer overflow and memory leaks. There are however no options to directly check secure memory deallocation violations, for this our instrumentation is required. We first evaluated CBMC's effectiveness on the Juliet Test Suite. for the CWEs that were directly testable by the CBMC options. We set 200 as the CBMC loop unrolling bound (from manual inspection, there are loops in the tests that go to 100). This also indirectly served as a measure of the examination capability of the test suite for detection of CWEs. Table 1 presents the results.

Table 1: CBMC: Direct 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

SecMD-Checker Results. We evaluated our tool on the CWE-244 test cases over two classes in the Juliet Test Suite. The first group consisted of test cases designed for CWE-244; there were 72 such test cases. The second group consisted of 95 other randomly chosen programs designed for other CWEs (we restricted these tests to involve heap memory allocation). Test cases in the first class consist of functions with suffixes __good and __bad. A __good suffix indicates a correct implementation of code where the respective CWE is absent, and a __bad suffix indicates a bad implementation of code where the respective CWE is present. Thus in the first group there were 72 function instances of CWE-244 at deallocation, and 72 instances where CWE-244 was supposed to be absent at deallocation. Upon closer analysis 18 of the good functions had `realloac()` calls without scrubbing, and hence these also had CWE-244. In the second group of randomly chosen tests, 51 out of the 95 tests did not have CWE-244, and 44 had CWE-244; we inferred these from manual inspection of the code.

Our test results for the experiments for the experiments are shown in Tables 2, 3, and 4. Each test program had "good" and "bad" marked functions (for the CWE the test corresponded to). The CBMC analysis times are further split for good and bad marked function subgroups. We kept the CBMC loop bound the same at 200

for all experiments except for 5 test cases in group two for which the SAT solver in CBMC ran out of memory – for these 5 test cases, we reduced the loop bound to 50. Our instrumentation approach had a 100% true positive rate, and a 100% true negatives rate on all the tests in our experiments. .

We point out that, although, technically we were not able to guarantee lack of false positives in Subsection 4.6 (due to theoretical unsoundness), the zero false positives and zero false negatives on the Juliet Test Suite demonstrate experimentally the precision of our algorithm in practice. On larger sized programs, we expect we will have to reduce the loop bound in order to avoid out of memory issues for CBMC which could possibly result in false negatives.

Table 2: SecMD-Checker Results: Detection of CWE-244 (All Groups)

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

Table 3: SecMD-Checker Running time (First Group)

	Average Time	Maximum Time
Instrumentation	0.23s	0.95s
CBMC Analysis (good marked functions)	14.69s	62.93s
CBMC Analysis (bad marked functions)	3.86s	4.61s

Table 4: SecMD-Checker Running time (Second Group)

	Average Time	Maximum Time
Instrumentation	4.31ss	47.01s
CBMC Analysis (good marked functions)	70.91s	578.68s
CBMC Analysis (bad marked functions)	71.30s	632.10s

7 CONCLUSION

Deallocating secure memory before wiping poses a security risk by exposing confidential data to attackers. In this work we developed a prototype SecMD-Checker implementing our approach for detecting secure memory deallocation violation (CWE-244) instances in C programs. We demonstrated our approach to be an effective and practical way of detecting CWE-244 instances by evaluating our tool on the Juliet Test Suite — we had a 100% true positive rate, and a 0% false negative rate. Future directions for our research are 1) engineering the tool further to remove the current limitations, *e.g.*, our tool cannot currently handle structs, or arrays of pointers – this entails refining our `AllCopiesSet`data structure to be more than a simple set (alternatively, we can conservatively track these additional constructs using overtainting); and 2) improvement of the tool algorithm by leveraging flow and context-sensitive approaches and alias analysis for better accuracy.

ACKNOWLEDGMENTS

The work was supported in part by funding from NSF under Award Number CNS 1822118 and from our industry partners AMI, NIST, Cyber Risk Research, Statnett, and ARL.

REFERENCES

[1] 2016. Software Engineering Institute CERT C Coding Standard. https://resources.sei.cmu.edu/downloads/secure-coding/assets/sei-cert-c-coding-standard-2016-v01.pdf.
[2] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. 2009. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against out-of-Bounds Errors. In *Proceedings of the 18th Conference on USENIX Security Symposium (SSYM'09)*. USENIX Association, USA, 51âĂŞ66.
[3] Abdullah Al-Boghdady, Khaled Wassif, and Mohammad El-Ramly. 2021. The Presence, Trends, and Causes of Security Vulnerabilities in Operating Systems of IoT's Low-End Devices. *Sensors* 21, 7 (2021).
[4] Abdullah Mujawib Alashjaee, Salahaldeen Duraibi, and Jia Song. 2019. IoT-Taint: IoT Malware Detection Framework Using Dynamic Taint Analysis. In *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*. 1220–1223.
[5] Richard Amankwah, Jinfu Chen, Alfred Adutwum Amponsah, Patrick Kwaku Kudjo, Vivienne Ocran, and Comfort Ofoley Anang. 2020. Fast Bug Detection Algorithm for Identifying Potential Vulnerabilities in Juliet Test Cases. In *2020 IEEE 8th International Conference on Smart City and Informatization (iSCI)*. 89–94.
[6] Ofir Arkin and Josh Anderson. [n. d.]. EtherLeak: Ethernet frame padding information leakage. https://dl.packetstormsecurity.net/advisories/atstake/atstake_etherleak_report.pdf.
[7] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, USA, 259âĂŞ269.
[8] Thomas Ball and Sriram K. Rajamani. 2001. Automatically Validating Temporal Safety Properties of Interfaces. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software* (Toronto, Ontario, Canada) *(SPIN '01)*. Springer-Verlag, Berlin, Heidelberg, 103âĂŞ122.
[9] Carlo Bellettini and Julian L. Rrushi. 2007. Vulnerability Analysis of SCADA Protocol Binaries through Detection of Memory Access Taintedness. In *2007 IEEE SMC Information Assurance and Security Workshop*. 341–348.
[10] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. 2005. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14 (SSYM'05)*. USENIX Association, 17.
[11] Minjae Byun, Yongjun Lee, and Jin-Young Choi. 2020. Analysis of software weakness detection of CBMC based on CWE. In *2020 22nd International Conference on Advanced Communication Technology (ICACT)*. IEEE, 171–175.
[12] Z. Berkay Celik, Leonardo Babun, Amit K. Sikder, Hidayet Aksu, Gang Tan, Patrick McDaniel, and A. Selcuk Uluagac. 2018. Sensitive Information Tracking in Commodity IoT. In *Proceedings of the 27th USENIX Conference on Security Symposium*. USENIX Association, 1687âĂŞ1704.
[13] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. 2004. Understanding data lifetime via whole system simulation. In *USENIX Security Symposium*. 321–336.
[14] Jim Chow, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. 2005. Shredding Your Garbage: Reducing Data Lifetime Through Secure Deallocation.. In *USENIX Security Symposium*. 22–22.
[15] Byron Cook, Kareem Khazem, Daniel Kroening, Serdar Tasiran, Michael Tautschnig, and Mark R. Tuttle. 2018. Model Checking Boot Code from AWS Data Centers. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, Cham, 467–486.
[16] Amer Diwan, Kathryn S McKinley, and J Eliot B Moss. 1998. Type-based alias analysis. *ACM Sigplan Notices* 33, 5 (1998), 106–117.
[17] Dawson R. Engler, David Yu Chen, and Andy Chou. 2001. Bugs as Inconsistent Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings*

of the 18th ACM Symposium on Operating System Principles, SOSP 2001, Keith Marzullo and Mahadev Satyanarayanan (Eds.). ACM, 57–72.

[18] Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. 2016. FlowFence: Practical Data Protection for Emerging IoT Application Frameworks. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 531–548.

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

[20] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. 2003. Software Verification with BLAST. In *Model Checking Software*, Thomas Ball and Sriram K. Rajamani (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 235–239.

[21] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. 1999. Interprocedural Pointer Alias Analysis. *ACM Trans. Program. Lang. Syst.* 21, 4 (July 1999), 848âĂŞ894.

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

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

[24] Daniel Kroening and Michael Tautschnig. 2014. CBMC – C Bounded Model Checker. In *Tools and Algorithms for the Construction and Analysis of Systems*, Erika Ábrahám and Klaus Havelund (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 389–391.

[25] David Larochelle and David Evans. 2001. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *10th USENIX Security Symposium (USENIX Security 01)*. USENIX Association.

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

[27] Amit Mandal, Pietro Ferrara, Yuliy Khlyebnikov, Agostino Cortesi, and Fausto Spoto. 2020. *Cross-Program Taint Analysis for IoT Systems*. Association for Computing Machinery, 1944âĂŞ1952.

[28] Daniel Marjamäki. 2013. Cppcheck: a tool for static C/C++ code analysis.

[29] MITRE. [n. d.]. Common Vulnerabilities and Exposures. https://cve.mitre.org/.

[30] MITRE. [n. d.]. Common Weakness Enumeration list. https://cwe.mitre.org/data/index.html.

[31] James Newsome and Dawn Xiaodong Song. 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2005, San Diego, California, USA*. The Internet Society.

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

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

[34] Weina Niu, Xiaosong Zhang, Xiaojiang Du, Lingyuan Zhao, Rong Cao, and Mohsen Guizani. 2020. A deep learning based static taint analysis approach for IoT software vulnerability location. *Measurement* 152 (2020), 107139.

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

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

[37] G. Ramalingam. 1994. The Undecidability of Aliasing. *ACM Trans. Program. Lang. Syst.* 16, 5 (1994), 1467–1471.

[38] Olatunji Ruwase and Monica S. Lam. 2004. A Practical Dynamic Buffer Overflow Detector. In *Proceedings of the 11th annual Network and Distributed System Security Symposium*. 159–169.

[39] Florian Schmeidl, Bara Nazzal, and Manar H. Alalfi. 2019. Security Analysis for SmartThings IoT Applications. In *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. 25–29.

[40] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP '10)*. IEEE Computer Society, USA, 317âĂŞ331.

[41] Michael Sipser. 2012. *Introduction to the Theory of Computation* (third ed.). Cengage Learning.

[42] Ioannis Stellios, Panayiotis Kotzanikolaou, Mihalis Psarakis, Cristina Alcaraz, and Javier Lopez. 2018. A Survey of IoT-Enabled Cyberattacks: Assessing Attack Paths to Critical Infrastructures and Services. *IEEE Communications Surveys Tutorials* 20, 4 (2018), 3453–3495.

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

[44] Victor van der Veen, Nitish dutt Sharma, Lorenzo Cavallaro, and Herbert Bos. 2012. Memory Errors: The Past, the Present, and the Future. In *Research in Attacks, Intrusions, and Defenses*, Davide Balzarotti, Salvatore J. Stolfo, and Marco Cova (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 86–106.

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

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

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

[48] Andreas Wagner and Johannes Sametinger. 2014. Using the Juliet Test Suite to compare static security scanners. In *2014 11th International Conference on Security and Cryptography (SECRYPT)*. 1–9.

[49] Jun Wang, Mingyi Zhao, Qiang Zeng, Dinghao Wu, and Peng Liu. 2015. Risk Assessment of Buffer "Heartbleed" Over-Read Vulnerabilities. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 555–562. https://doi.org/10.1109/DSN.2015.59

[50] Yong Wang, Dawu Gu, Daogang Peng, Shuai Chen, and Heng Yang. 2012. Stuxnet Vulnerabilities Analysis of SCADA Systems. In *Network Computing and Information Security*, Jingsheng Lei, Fu Lee Wang, Mo Li, and Yuan Luo (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 640–646.

[51] David A Wheeler. 2013. Flawfinder.

[52] Robert P. Wilson and Monica S. Lam. 1995. Efficient Context-Sensitive Pointer Analysis for C Programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation* (La Jolla, California, USA) *(PLDI '95)*. Association for Computing Machinery, New York, NY, USA, 1âĂŞ12.

[53] Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. 2015. Automatic Inference of Search Patterns for Taint-Style Vulnerabilities. In *2015 IEEE Symposium on Security and Privacy*. 797–812.

[54] Bonnie Zhu, Anthony Joseph, and Shankar Sastry. 2011. A Taxonomy of Cyber Attacks on SCADA Systems. In *2011 International Conference on Internet of Things and 4th International Conference on Cyber, Physical and Social Computing*. 380–388.