Program Transformation Techniques for Erasing Sensitive Data in Sequential and Concurrent Applications

BY

KALPANA GONDI
B.Tech (ISSCIT&EW, affiliated to JNTU, Hyderabad, India) 2002
M.Tech (University of Hyderabad, India) 2004

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2013

Chicago, Illinois

To my family,

who supported me at all times.

ACKNOWLEDGMENTS

I take this opportunity to express gratitude to all those who supported me through out my graduate studies.

First, I must thank my family who has been my strength through out my career. My parents have always been very encouraging and their love and support have brought me this far. I am indebted to my brother, Dr.Vijaya Bhasker Gondi, who encouraged me to pursue a doctoral degree in Computer Science and Engineering. I thank my husband and son who were very cooperative during the final phases of my graduate studies.

I acknowledge and extend my heartfelt gratitude to my advisors Dr.Aravinda Prasad Sistla and Dr.V.N.Venkatakrishnan for all help and support throughout my graduate study at UIC. Without their support, this dissertation would not have been possible. Their guidance helped me to come up with ideas and solutions for all the projects in this thesis. Dr.Aravinda Prasad Sistla and Dr.V.N.Venkatakrishnan were very understanding and caring and helped me during tough times I was going through personally. They also shared my emotions of stress and excitement for rejections and acceptance of my journal articles/papers. Their encouragement helped me resolve issues and come up with solutions for problems when I am stuck at times during my research. I am grateful to them for providing me an atmosphere of learning and spending their valuable time for me.

I would also like to thank my collaborators in my research: Patrick Zurek, Praveen Venkatachari and Prithvi Bisht, for their hardwork and constant support to develop my research work. I thank my lab members: Mike Ter Louw, Rigel Gjomemo, Maliheh Monshizadeh, Michelle zhou, for finding the time to attend my practice talks and give feedback.

I thank Dr.Jakob Erikkson, Dr.Mark Grechanik, Dr. Alan Jeffrey and Dr.Jon Solworth for being on my final defense committee and for their insightful comments on my dissertation, which have helped greatly to improve the quality of this dissertation.

The faculty and staff in the department of computer science at University of Illinois, Chicago, were very kind and helpful and I take this opportunity to thank them individually.

My research was partially supported by National Science Foundation grants CNS-0845894, DGE-1069311, CNS-1065537, CNS-1141863 CNS-1035914, CCF-0916438,CNS-0917229, CCF-1319754, CNS-1344485, and NSF-1018836. I extend my sincere gratitude to all these funding sources, which have been very helpful to support me economically throughout my graduate program.

Lastly, I express my sincere thanks to all those who have extended their co-operation directly or indirectly during the progress of my graduate study thereby ensuring the successful completion.

KG

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

CFG	Control Flow Graph
ECFG	Extended Control Flow Graph
DIECS	Data Erasure In Concurrent Software
CIL	C Intermediate Langues
OCaml	Objective Caml
GCC	GNU Compiler Collection
GDB	GNU Debugger
RAM	Random Access Memory
UIC	University of Illinois at Chicago.

SUMMARY

The memory footprints of sensitive data in security critical systems programs and end user applications is generally not ensured to be kept small. This results in data being left over in the memory long after it has been last used in the program. The prolonged presence of such memory resident data that is sensitive (such as passwords and cryptographic keys) poses a number of security risks.

The onus of ensuring that sensitive data does not remain in the program well beyond its intended use is left entirely to the programmer. Unfortunately this is only done by the most sophisticated of applications developers, as many developers are not simply aware of these issues. Consequently, there is a dire need to retrofit applications to minimize the prolonged life of sensitive data in memory. A promising approach is to delete sensitive data as soon as it becomes unnecessary.

In this thesis, we propose methods to reduce the lifetime of memory resident data by erasing the memory footprint of sensitive data after it's last usage in programs. Our approach uses program transformation techniques to achieve the goal of precise and effective minimization of data exposure in sequential and concurrent applications.

We first describe SWIPE, an approach to reduce the lifetime of sensitive, memory resident data in large-scale sequential applications written in C. In contrast to prior approaches that used a delayed or lazy approach to the problem of erasing sensitive data, SWIPE uses a novel *eager erasure* approach that minimizes the risk of accidental sensitive data leakage. SWIPE achieves this by transforming a legacy sequential C program to include additional instructions that erase sensitive data immediately after its intended use. SWIPE is guided by a highly scalable static analysis technique that precisely identifies locations to introduce erase instructions in the original program. In our experiments, SWIPE is able to successfully and robustly transform several large applications with minimal performance overheads. The sequential programs transformed using SWIPE enjoy several additional advantages: minimization of leaks that arise due to data dependencies; erasure of sensitive data with minimal developer guidance; and negligible performance overheads (averaging 1.35%).

We subsequently tackle the challenge of minimizing data lifetime in the realm of concurrent programs. In this case, the issues are comparatively more challenging as the interleaving of threads in concurrent applications makes the determination of lifetime statically difficult. In this dissertation, we also address the problem of data lifetime minimization for concurrent programs in addition to sequential programs. The major challenge for our approach is to ensure that the erasures introduced to the shared memory locations are done after anticipating all possible interleaving of threads, in order to ensure soundness. We develop a new algorithm that anticipates shared variable lifetimes, and correctly introduce erase instructions. This algorithm is implemented in a tool called DEICS (**D**ata **E**rasure **I**n **C**oncurrent **S**oftware). Through an experimental evaluation, we show that DEICS is able to reduce lifetimes of shared sensitive data in several concurrent applications (over 100k lines of code combined) with minimal performance overheads. This thesis is, as far as we know, the first study that applies "data lifetime minimization" techniques to concurrent programs. The method we present to erase data from shared memory is sound (i.e., data is erased only after it is no longer needed). Our experimental results show, DEICS enables data erasure efficiently.

Taken together, SWIPE and DEICS address the issues in minimizing sensitive lifetime in modern computing applications. They demonstrate that precise and effective minimization of data exposure in sequential and concurrent applications can be achieved through program transformation techniques.

CHAPTER 1

INTRODUCTION

Confidentiality of sensitive data is an important concern for modern enterprise organizations as well as end users. While a number of enterprise solutions have been proposed for protecting data residing in databases and permanent storage, the issue of addressing data confidentiality in running programs has received relatively less attention. In this thesis, we look at this issue of data confidentiality in the context of programs that operate on sensitive data residing in memory. In particular, we focus our efforts on programs that are written in C, a widely used systems programming language.

The memory footprint of sensitive variables in software applications written in C, is generally not ensured to be kept small. This state of practice results in the data being "left over" in memory long after its intended use in a program. The prolonged presence of sensitive data beyond its lifetime can lead to unauthorized disclosure.

Many security critical applications handle sensitive data such as passwords, credit-card numbers, social-security numbers, and various other forms of personally identifiable sensitive data. The use of sensitive data is not just restricted to personal systems, but to also in programs that are employed in mobile systems such as laptop computers and electronic voting systems that process voting information. Most often the use of such sensitive data has a certain locality; its use is restricted to code that is confined to a certain small region of the program text. A typical example is an authentication routine in a trusted system program that handles sensitive data such as passwords. Sensitive data is typically not required beyond their intended use portion of the program (in this example, beyond the authentication step), and therefore can be erased to have a limited lifetime. Typical programs hardly obey this tenet of erasing to restrict the lifetime of sensitive information. This results in the data being left over in the memory long after it has been last used.

In developing system software, it is often the programmer's responsibility to ensure that sensitive data does not remain beyond its intended use. Unfortunately, only the most skilled application developers think about these issues, while many developers are simply unaware of these issues. This state of practice has led to prolonged sensitive data lifetimes opening the door to sensitive data disclosure attacks. As shown further (Chapter 2), there is a dire need to retrofit applications to minimize sensitive data lifetimes.

The problem is exacerbated in a type-unsafe language such as C. Due to the lack of memory safety, developers receive no assistance to protect data variables from exposures, resulting in leakage of sensitive data. In addition, concurrency has become a natural phenomenon in the computing environment with the advent of graphical interfaces, phone applications and other hardware-software interfaces. However implementing and understanding a concurrent application is relatively more difficult for a programmer due to the added complexity of programming such applications. In order to develop concurrent applications, programmers often require to keep track of various concurrency constructs to ensure correct implementation. While following such disciplines, it is possible that a programmer may ignore other security considerations such as keeping track of usage of sensitive data in the program.

An interesting SecurityFocus discussion (1) examined problems faced by developers in writing code that handles sensitive data in C programs. Chow et al., (2) performed a whole system simulation study with virtual machines to analyze the execution of several system applications such as browsers and authenticators that handle sensitive data. Using this study they demonstrated their claim that "*. . . applications take virtually no measure to limit the lifetime of sensitive data they handle. . .* ".

The risks of sensitive data having an uncontrolled lifetime during the execution of a program are many fold. Such risks have implications not only during the lifetime of the program but even beyond that, sometimes even after an entire system shutdown. We discuss the risks under two broad categories:

1. **Data leakage through online attacks:** Sensitive data can be stolen by attackers who can employ various low-level attacks in order to steal such data from C programs. For instance, a format string attack can be employed to read any piece of sensitive data in program memory. In (3), the authors describe a mimicry attack that exploits an overflow attack to steal credit-card numbers and keys from a web server application. Another vulnerability (CVE-2011-0992) (4) allows remote attackers to obtain sensitive information from Mono (an implementation of Silverlight for Linux). Threads in Mono were not properly cleaned up upon finalization, so if one thread was resurrected, it would be possible to observe the pointer to freed memory, leading to unintended information disclosure. Recently, there were various vulnerabilities identified in the Linux kernel, where certain data structures were not initialized properly leading to obtain sensitive information from kernel heap and stack memory (CVE-2013-3222 to CVE-2013-3237) (4). In the above examples, sensitive data was made available in the program's memory well beyond its **lifetime** in the program. The memory should have been erased of its contents before returning the location for other use.

2. **Data leakage through offline attacks:** When resident in the memory space of the program long enough, sensitive information could leak through forensic examinations (or off-line attacks). For instance, offline examination of the swap space in disk can reveal passwords and other sensitive data, and it is not common for systems to have encrypted swap partitions. More recently, cold-boot attacks (5) have been demonstrated to be successful in recovering keys from memory long after a machine is powered down, by examining data resident in RAM even after a power recycle. A recent CVE citation CVE-2010-0551 (4) reports sensitive login credentials available in memory that could be recovered easily through a cold boot attack. Another source of offline leakage is from crash reports, as shown in (6) by demonstrating the presence of unmodified passphrases in the crash report (even when the cause of the crash itself was unrelated to sensitive data). In this case, sensitive information is available to a developer inspecting the crash report.

Prior Work Prior work in this area has been along several directions: residual data analysis in semiconductor devices (7; 8), use of memory management techniques such as garbage collection (9) and operating system / application-library level erasure methods (10; 11; 12). The closest work is by Chow et al., (12), which augments the *free* library function with erase instructions. In general, programmers do not place *free* instructions to free the memory at the earliest possible program location i.e., they generally do not aim for minimizing the lifetime of unused data in the memory. Thus any augmentation of *free* instructions for minimizing data lifetime does not effectively prevent all data leaks. As analyzed in Chapter 2, these approaches rely on a *deferred* or *pre-determined* location for erasing, introducing windows of instructions where sensitive data is present in the memory but no longer required by the program. Such exposures may lead to unauthorized data leakage.

1.1 Thesis Goals

Precise and effective minimization of data exposure in sequential and concurrent applications can be achieved through program transformation techniques. To support this thesis, we explore solutions to minimize data lifetime in applications that can be both sequential and concurrent. We adopt an *eager* strategy to aggressively reduce the lifetime of sensitive data items in legacy applications that are both sequential and concurrent. More precisely, our high level approach aims to erase data in an eager fashion after completion of its intended use in the program. For the password example discussed earlier, once the password verification step is done, the use of our technique will result in immediate erasure of the user-supplied password, ideally as the very next instruction after the password checking step. Through our experiments, we demonstrate that, our approach is more precise in introducing erasures in the program. We verified that erasures introduced by our approach are placed much earlier in the program compared to the developer inserted erase instructions. The effectiveness of our approach is demonstrated with the help of memory snapshots and coldboot attacks.

The underlying technique used in our approach is program transformation. Our approach transforms legacy applications by inserting erase (or scrub) instructions to clean-up sensitive data that is no longer required. This transformation is guided by a static analysis that precisely identifies the locations to which erase instructions are to be added. Our approach has the benefit that it can handle a majority of information leaks due to data dependencies in the program. By statically introducing erases, our approach avoids high performance overheads typical of runtime techniques (such as taint tracking) that track and eliminate sensitive information at runtime.

This dissertation makes contributions in two major directions. First, it provides an approach and algorithms to minimize the lifetime of sensitive data in sequential applications (Chapter 3). Second, it extends the approach of handling sequential programs to minimize the lifetime of sensitive data in concurrent applications (Chapter 4). A detailed analysis of data exposure in sequential and concurrent programs in given in the chapter 2.

1.1.1 Data Lifetime Minimization in Sequential Programs

In this thesis, we attempt to address the root cause of data lifetime problem in sequential programs. We plan to rewrite given sequential programs such that (a) it tracks the flow of sensitive data in the programs (b) identify the "last use" points of such data and (c) make subsequent access to this sensitive data unavailable through explicit zeroing.

Challenges: To minimize the sensitive data precisely, one of the challenge is to track the flow of sensitive data in the program. During program execution, presence of multiple references (aliases) to the same memory location makes it rather challenging to track all the references before the erasure is introduced. In general, application developers make use of functions to have a modular design and clean implementation of the program. Additionally, the "last use" of a sensitive data may fall in a different function than where it is defined in the program. Therefore, our analysis should keep track of all the function calls in the program and precisely identify the locations of erasures for the sensitive data. Another challenge is to handle and erase sensitive data that is present in dynamic data types. In general, programmers make use of dynamic data types to allocate data whose size is not know initially. This leaves a challenge for our analysis to precisely know the size of the data at the time of erasure. Note that the erase point for a dynamic data may not be in the same function where it is allocated.

Approach: The main idea is to locate the "first no use" point (First-No-Use) for any piece of sensitive data and erase it immediately before those points. Our high level approach consists of taking a specification of sensitive data in the program and then track the flow of sensitive data to accommodate the erasure of all possible copies of such data. For each sensitive datum, our analysis first identifies the region in the program where it is actually required and where it is available unnecessarily. The exposure window for sensitive data is then minimized with the help of erase instructions at the first possible no use point. Our static-analysis approach includes a novel summary-based analysis technique that enables the approach to be highly scalable and efficient. The summary captures the net effect on invoking a function and is applied wherever the function is called in the program thus providing an inter-procedural analysis, which is scalable to handle larger applications.

We note that our approach is conservative in its design. Specifically, if we are unable to correctly ascertain the "last use" points, we do not attempt to erase program data at these points, thus maintaining the original program semantics. Yet, our empirical results demonstrate that our approach is highly effective in erasing sensitive program data.

Results: We have implemented the basic approach in a tool called SWIPE, a source-to-source transformer that retrofits sequential C programs to minimize the lifetime of sensitive data. Using SWIPE, we transformed several commonly used open source desktop applications that handle sensitive data whose code sizes ranged from 3K to 137K lines of code. Many of these programs do not attempt to minimize sensitive data lifetimes. We also evaluated the correctness of our transformation using GCC test suit and assessed the effectiveness by analyzing the process memory for the presence of sensitive data.

1.1.2 Data Erasure in Concurrent Software

As mentioned earlier, the recent trend is geared towards using concurrent software. This thesis considers the problem of retrofitting a concurrent program with additional instructions that minimizes the lifetime of sensitive data used by that application. We extend the approach of SWIPE to handle concurrent applications.

Challenges: In order to minimize data lifetime, one must analyze the lifetime of sensitive data in concurrent programs. There are various challenges involved in analyzing the concurrent applications. The major challenge is to track the use of sensitive data in a concurrent application. The non-determinism involved in thread interleavings leaves a challenge for a static analysis to precisely reason the order of shared memory accesses by different threads. In general, programmers try to make use of locks to access any shared data to avoid conflicts in updating and accessing the data. Thus, any analysis should also keep track of all such synchronization constructs used in the program while accessing any shared data (i.e., the number of locks used for accessing a shared datum). The erase instructions that would be introduced by our analysis should also be well guarded by such locks. The number of threads that may access a shared data could be dynamic in nature. Often, it is not feasible to assume the number of threads statically. For example, if a thread is invoked within a loop, our analysis should also consider a possible interleaving of a thread with itself.

Approach: Our approach is to transform concurrent applications with a mechanism to erase shared data so that the exposure of data is minimized. For this, all the potential parallel executions of threads need to be considered to ensure that we do not erase a shared data before its access by any parallel thread (and thereby changing its original intended behavior). The approach of SWIPE is extended to handle shared sensitive data. The behaviour of shared data accesses by multiple threads is assessed with the help of a race detection engine. The high level approach is to identify potential erase points for a shared data and use the race detection engine to confirm the ideal erase point to actually introduce erasure.

Results: Our approach is implemented in a tool called DEICS, which transforms given concurrent C program into an equivalent C program with reduced lifetime of sensitive data used in the program. DEICS exclusively handles shared data and minimizes its lifetime by inserting erase instructions in a conservative way. By conservative we mean that any piece of shared data may not be erased by our approach if there can be a potential access by some thread in the program. Using DEICS, we transformed five concurrent applications (the largest application consists of 57K lines of code), each represents a different functionality being implemented using concurrency. Applications transformed by DEICS show minimum performance overhead (less that 1%) to erase shared data.

To the best of our knowledge, DEICS is the first known approach in the literature to bring data lifetime reduction technique to the realm of concurrent programs.

1.2 Thesis Contributions and Organization

This thesis makes the following main contributions:

- We provide a detailed security analysis of the threats due to *delayed* erasure of sensitive data. We also provide an analysis of data exposure in concurrent applications.
- We provide a novel, eager evaluation strategy for reducing data lifetime in sequential programs. we present a scalable static program analysis technique for identifying erase points in a program, that includes a novel summary-based analysis.

- We have implemented our approach to minimize data lifetime in sequential programs into a tool called SWIPE. We also provide a discussion of implementation challenges for SWIPE. Also a comprehensive evaluation of SWIPE on some commonly used desktop applications along the dimensions of scalability, effectiveness of erasure and performance.
- We present a scalable static analysis technique to transform concurrent applications to minimize sensitive data lifetime. We have implemented this technique into a tool called DEICS. We also discuss the implementation strategies for transforming concurrent C programs.
- We provide a detailed evaluation of the approach used in DEICS by transforming over 100k lines of C applications (combined, with the largest application consisting of 57k LOC).

Taken together, SWIPE *and* DEICS *address the issues in minimizing sensitive lifetime in modern computing applications. They demonstrate that precise and effective minimization of data exposure in sequential and concurrent applications can be achieved through program transformation techniques.*

The rest of the dissertation is organized as follows: Chapter 2 presents the importance of data lifetime minimization in applications with an illustrative example. Chapter 3 explains our approach used in the context of a simple imperative language that is modelled after the core constructs of the C language. Key algorithms of our approach and key implementation challenges are discussed Section 3.2. We also present an evaluation of SWIPE on many C programs that process sensitive data in Section 3.5. We explain our approach of data lifetime minimization for concurrent programs in Chapter 4. Algorithm of our approach with implementation details is given in section 4.3. Section 4.4 provides a detailed evaluation of our approach on set of real-life concurrent applications written in C. Chapter 5 presents the detailed comparison of our approach with related work. We conclude in chapter 6 with possible directions for future work.

CHAPTER 2

PROBLEM ANALYSIS

In this chapter, we provide an analysis of the problem space by first highlighting some issues in minimizing sensitive data lifetime by the concept of exposure windows. We then use these concepts in an empirical analysis of a few applications that handle sensitive data. Our analysis highlights the prevalence of the problem. We begin this chapter by introducing a running example for sequential programs that shows a typical authentication mechanism in the file encryption program (section 2.1). We explain the possible exposure windows for sensitive data using this running example in section 2.2. Also, we present a running example for concurrent applications and show possible exposure windows for the shared data. Our empirical analysis of exposure windows on various applications is given in 2.4. We also look at contemporary methods, which can be used to address the problem of minimizing data exposure and their limitations in section 2.5.

2.1 Running Example Representing Sequential Programs

We use a file encryption program shown in Figure 1 as a running example for explaining our analysis on sequential applications in this dissertation. We are not showing free instructions in this program for brevity. The *file_encrypt* function provides the functionality of encrypting the file (supplied as argument *fname*) with the private key of a user (identified by argument *userId*). The private key itself is obtained in the function *getPrivateKey* that first obtains and compares the user's password with the stored password. On successful authentication the function *getKeyfromDB* is invoked to retrieve the user's private key from a back-end database. The program makes three attempts to authenticate the user in a loop and aborts if the number of attempts exceeds three.

```
 1  int  file_encrypt ( char *fname, char *userId ){
 2     char key [255];
 3     if ( getPrivateKey( userId , key ) == 1 ){
 4          int fd = open( fname, O_RDONLY );
 5          if( fd == -1 ) exit( 2 ); // error
 6          encrypt( fd, key );
 7          // further processing 7-25 lines
26     } else return -1;
27  } // end of file_encrypt
28  int getPrivateKey( char *user, char *keyA ){
29   int attempts = 0; char *pass;
30   char shdwpwd [255];
31   readshdw( user, shdwpwd );
32   do{
33      pass = getpasswd();
34      char *epasswd = gethash( pass );
35      if (strcmp(epasswd, shdwpwd ) != 0) {
36          attempts++;
37      } else {
38          printf ("Authentication successful !");
39          getKeyfromDB( user, keyA );
40          return 1;
41      }
42   } while( attempts < 3 );
43   return 0; // end of getPrivateKey
44  }
```

Figure 1: Running example that illustrates the need for minimizing data lifetime (sequential application).

Lets consider a particular execution of this program in which the user supplies correct password in the second attempt and the program exits at line 5. In terms of the program's control flow, the loop at lines 32 - 42 executes twice, and the control exits on line 5. This control flow is shown as a linear trace in Figure 2, through which we will explain the concept of *exposure windows* in the program.

2.2 Exposure Windows

Definition 1. *(Exposure Windows) In any given program P, for any object X, we coin the term exposure window as a sub-region of P that has access to X, when the value held in X is no longer required by P.*

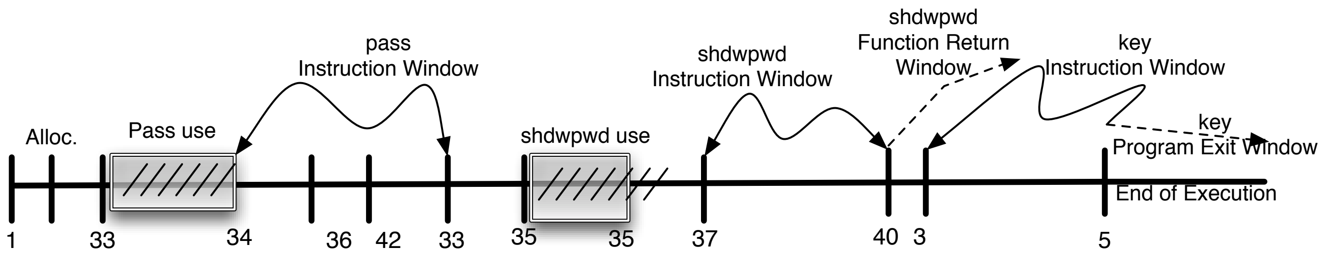


Figure 2: Windows of exposure for sensitive data

In the running example, consider the variable *pass* declared in line 29. This variable is initialized with the user-supplied password in line 33 (through the method *getpasswd*) and subsequently used to compute the hash of the password in line 34. Subsequently, the data in *pass* is no longer required by the program but remains available in the rest of the function (lines 35 to 41 as shown in Figure 2.

We define three types of exposure windows as illustrated in Figure 2 and their implications for leakage of sensitive data.

1. Instruction Window This type of exposure window was explained above using the *pass* example. Basically, it arises from instructions in a program having access to data beyond its last use. A program crash in this window will yield a crash report containing sensitive data that if transmitted to a developer, will result in data leaks.

2. Function Return Window The local variables of a function that hold sensitive data, if not erased before returning from the function, create another type of exposure window that we term *function return window*. To illustrate this, consider the *shdwpwd* variable defined in line 31 of the function *getPrivateKey*. The sensitive data held in this variable is not erased at any of the two return statements (line 40 and line 43). Thus invocation of the function *getPrivateKey* leaves this sensitive data in the function stack. Since function returns do not by default clear the stack (As observed by us in several contemporary operating systems including Linux), subsequent function call records will have access to this data. Furthermore, if the program stack does not subsequently grow any larger after *getPrivateKey* was called, the sensitive data in *shdwpwd* will remain in the stack memory until the program exits.

3. Program Exit Window Similar to function return windows, another type of exposure window is created at program exit points that we call *program exit window*. Consider the exit instruction in line 5. As shown in Figure 2, content of the *key* variable remains in program memory even after the program exits. In contrast to function exit windows, this type of exposure window has implications *beyond* the program's execution lifetime. Sensitive data from programs, if not erased, remains in system memory pages and may become the target of forensic attacks (In Section 3.5.4.2 we demonstrate one such successful attack on *OpenSSL*).

2.3 Running Example Representing Concurrent Programs

We present another running example to highlight the problem of data exposure in the context of concurrent applications. We use an abstract version of famous *producer-consumer* model implemented using threads. There are many real-world scenarios that implement the producer-consumer pattern. For example, a web service (producer) receives http requests for data, places the request into an internal queue. A worker thread (consumer) pulls the request from the queue and performs the work. Another scenario involves packet processing in networking applications. One thread polls the network and retrieves the packets and puts them in a buffer, and another thread picks packets from buffer and analyzes them.

```
1   DEFINE LENGTH 20;
2   char *request;
3   mutex_type lockv;
    /* Server thread which accepts the connection
    and collects HTTP requests from clients */
4   int server( ){
5   char * localdata ;
6    lock(&lockv);
7     request = malloc(LENGTH);
8     localdata = getRequestfromUser();
9     strcpy ( request , localdata , LENGTH);
10    log( request );   // read( request )
11    unlock(&lockv);
12    // do other work : Lines 12 − 20
21  }

22  int worker(){
   /* Worker thread to process HTTP request */
23    lock(&lockv);
24    if( request != NULL){
25      process( request );   // read( request )
26    }
27    unlock(&lockv);
28 // work like generating response : Lines 28 − 36
37  }
38  int mian(){
39    mutex_init(&lockv);
40     thread_create ( server );
41     thread_create (worker);
42    // do some other work : Lines 42 − 52;
53  }
```

Figure 3: Running example that illustrates the need for minimizing data lifetime (concurrent application).

Figure 3 gives a simplified version of webserver example implemented using threads. In this program, the server thread accepts the connection from the client and collects the *HTTPRequest* and places it in a shared memory (here the *request* variable declared at line 2) for the worker thread to process the same. In real scenario, there will be a *Queue* that would be used as a common buffer to collect and process requests made by clients. We use only one instance of the buffer for simplicity. However, we consider that both the server and the worker threads to run in parallel.

Threads are created in the $main$ function (which is the main thread) at lines 40 and 41 and all the three threads run in parallel starting from line 41. Shared variable, $request$ (declared at line 2) is accessed by server and worker threads and its access is protected by a lock variable $lockv$ for synchronization purpose. After receiving the request from client, the server thread, puts it in $request$ at line 9

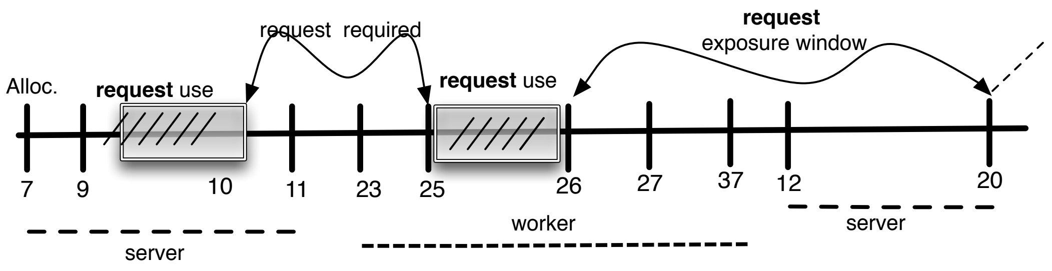


Figure 4: Windows of exposure for sensitive shared data

and logs the $request$ at line 10. The $request$ is not used after line 10 inside server thread. The worker thread reads the same data on line 25. After processing $request$ on line 25, it is no longer required in the program, but remains available in the rest of the worker thread and may be to other threads (lines 26-37 in worker, lines 12-21 in server, and lines 42-53 in main). Note that the actual execution at line 25 will depend on thread interleavings in general.

Similar to the sequential program, let us consider a particular execution of the program given in Figure 3. Note that, there can be different possible interleavings of worker, server and main threads based on how thread scheduler schedules the order of thread executions. Figure 4 shows one such execution line with the order of execution of instructions from each thread. If we consider this particular execution between server and worker thread, clearly there is a large window of exposure for the shared data held in $request$. The actual code between lines 12-20 and 28-36 is not shown. This code can include *exit* instructions or any other function calls, leading to potential exposure of program exit and function return windows respectively. We wish to note that, the window of exposure shown in this case is the maximum compared to other potential interleavings.

2.4 An Empirical Analysis

We performed an empirical analysis to assess the prevalence of exposure windows in open source applications. For this purpose, we chose five widely used open source applications such as *OpenSSL* that handle sensitive data and analyzed their source code to measure exposure windows. We found a large number of instruction, function return and program exit exposure windows in all the five applications. As these popular applications fail to minimize the sensitive data lifetime, this analysis emphasized the clear need to reduce such exposure windows. Our study is different from (2), which used an operating system simulation study to illustrate the impact of the data lifetimes across an operating system. In contrast, we study the effect of data lifetimes across *programs*, by making use of *program analysis techniques*. Thus our program-centric study can be viewed as complementary to the results reported in the system-centric study of Chow et al. (2). (Since exact computation of the exposure windows is an undecidable problem, we conservatively estimate them as given by our methodology below.)

2.4.1 Experimental Methodology

We choose five open source applications that handle sensitive data, as given in Table I. In order to estimate the exposure windows, we merged the source code for each application, inlining the source for all the function calls. Subsequently, using analysis tools, we built extended control flow graphs (ECFG), for these applications. We did not inline library calls, and instead treated each library call as a single instruction. We also approximated the effect of loops, by counting them only for zero or one iterations. In our analysis, we also did not consider the effect of data flows (sensitive data being copied around in the program). We note that all the above approximations are conservative, so the actual numbers of the metrics is likely to be worse than what we report here.

We then designed analysis techniques that analyzed these control flow graphs to compute the metrics for various exposure windows In measuring these metrics, we did account for any erasing instructions already in place in the program (such as *bzero* or *memset*).

For each program, we identified one important sensitive variable (manually) and provided that as input to our analysis. For all these programs it was the user-supplied password with the exception of *OpenSSL*, which operates on keys. To account for Exit Window, we count the presence of this (unerased) sensitive available at exit instructions and at the return statement(s) in main function. For Function Window, we count data of stack allocated sensitive local variables when a function returns. Instruction Window are accounted by counting number of instructions in which the data is available before it is erased or a return or exit instruction is encountered. Since instruction window sizes are measured for each control path, we present the average across all control paths in the table.

Application	**Size (eLoc)**	**# of Exit Wnd**	**# of Func Wnd**	**Instruction Windows**	
				# of Wnd	**Avg path (Max)**
SFTP	32970	60	238	804	79.73 (720)
OpenSSL	154056	70	265	1429	10.86 (80)
linux-utils	3862	2916	2652	12165	8.12 (102)
mpop	17995	11613	2153	24441	11.30 (104)
MySecureshell	10831	1583	169	10115	3.60 (42)

TABLE I: Empirical analysis of data exposures in popular open source applications

2.4.2 Results

Table I shows whether we were able to successfully identify exposure windows for each of the five programs analyzed. Of the five programs we analyzed, *SFTP* applies the most stringent data erasure policy on the respective data objects that were chosen, still leaving an instruction window open. The other applications do present significant instruction and exit windows.

We also obtained the metrics measured across all program variables that appear in Table I. Even though not all variables in a program are sensitive, the presence of multiple exposure windows in these programs is further highlighted by this table. It is interesting to compare the average instruction window sizes in the two tables. With the exception of *SFTP* (which does seem to implement stringent data erasure policies), the values for the sensitive variables is much higher than average, as expected.

Our simple analysis shows that indeed there are several windows of exposure for sensitive data in these programs, and there is a clear need to reduce such exposure. We now analyze some existing methods to tackle the problem.

2.5 Contemporary Methods

2.5.1 Erase on Function Returns or Exits or Free Method Calls

A simple way to erase data is to intercept program exit points and return statements (12). For data residing in dynamically allocated memory, a better place to erase is during the *free* method call. Chow et al., modify the *free* library call to erase data before freeing the memory. This technique is simple to implement as there is no analysis required to retrofit the application. However, the placement of a *free* instruction by the programmer may not aim to erase data at the earliest possible program location and hence may not reduce extended data lifetime, in general. Also, these techniques will not be able to close instruction exposure windows. In the running example given in Figure 1, the memory allocated for $pass$ would get deallocated (through free instruction) after the while loop. An erasing mechanism in free instruction after line 42 would clear the contents of the memory pointed to by pass. However, there will still be an exposure window of instructions present from lines 35 to line 42 for the data in pass. From our empirical analysis , we note that the size of instruction windows range from 42 to 720 instructions, without counting any library calls. In developing high assurance systems, closing such instruction windows will greatly mitigate the risks of sensitive data leaks.

2.5.2 Memory Management

It is possible to employ memory management methods such as garbage collection to address these issues. For instance, the Boer-Demers-Weiser (9) collector for C can be augmented with the task of erasing data, i.e., erase data as and when the garbage collector is reclaiming the memory. This also requires no application instrumentation. However, the effectiveness of this approach is dependent on how frequently the garbage collector is scheduled to run. One can in theory invoke the garbage collector quite frequently to erase sensitive data, but that would lead to very high performance overheads (13; 14). Additionally, a garbage collection mechanism for systems using region based memory management (15) would incur low performance overheads to clear memory contents. However, such a garbage collector also exhibits windows of exposure if references are retained to data which will never be used again (i.e instruction windows). An object may contain multiple definitions throughout its lifetime and a garbage collector would only clear the data present in the object at the end. All previous data definitions of this object may still have instruction exposure windows which need to be minimized.

2.5.3 Dynamic Taint Analysis

Dynamic taint analysis can be used to reason about the propagation of sensitive data in a program, and this together with techniques such as dynamic reference counting can be used to identify and erase sensitive objects in a program. The advantage of dynamic tracking is that it can lead to a precise approach in closing exposure windows in a program. However, such an approach is likely to have very high overheads. For instance, Xu et al. (16) report an overhead of 58% - 100% for taint tracking in C programs. Such overheads in production code will likely lead to resistance in acceptance of taint tracking in practice, and developers may seek less automated but faster methods to address the issue.

2.6 Summary

We provided a detailed analysis of the data lifetime problem in programs in this chapter. We identified possible windows of exposure for the data in both sequential and concurrent applications with the help of running examples. We also presented an overview of existing methods and their limitations in minimizing the lifetime. The next chapter provides our approach in minimizing the data lifetime in programs.

CHAPTER 3

SWIPE: EAGER ERASURE OF SENSITIVE DATA IN LARGE SCARE SYSTEM SOFTWARE

This chapter presents our approach to minimize lifetime of data in sequential applications. We provide a high-level overview of our approach with the help of transformed running example in section 3.1. A detailed technical description of our approach is given in section 3.2 and an algorithm of our implementation is given in section 3.4. We evaluate SWIPE for correctness, effectiveness, performance, and scalability in section 3.5.

3.1 Approach

In this section we describe our high level approach for reducing extended data lifetime. A detailed treatment of algorithms developed to realize our approach is also provided (Section 3.2).

3.1.1 Transformed Running Example

Our goal is to retrofit a given program's source code so that sensitive data lifetimes are minimized. Figure 5 illustrates the end result of transforming the running example (Figure 1) using our approach. Let us consider variables *key*, *userId*, *fname*, *fd*, *attempts*, *pass*, *shdwpwd* and *epasswd* as sensitive. The additional lines introduced (shown in gray) contain *memset* instructions to erase data objects.

We first note that the erase statement for *pass* is added at line 34a. Since *pass* is no longer required after the program computes the shadow password, the location at line 34a is the most precise location in source to erase this data item, addressing both the instruction window and the function return windows

```
1    int  file_encrypt ( char *fname, char *userId ){
2        char key [255];
3        if( getPrivateKey( userId, key ) == 1 ){
4            int fd = open( fname, O_RDONLY );
5            if( fd == -1 ) {
5a               memset( key, 0, keySize );
5b               memset( userId, 0, userIdSize );
5c               memset( fname, 0, fnameSize );
5                exit ( 2 ); // error }
6            encrypt( fd, key );
6a           memset( key, 0, keySize );
7            // further processing 7-25 lines
18a          memset( fd, 0, intSize );
26       } else return -1;
27   } // end of file_encrypt
28   int getPrivateKey( char *user, char *keyA ){
29       int attempts = 0; char *pass;
30       char shdwpwd [255];
31       readshdw( user, shdwpwd );
32       do {
33           pass = getpasswd();
34           char *epasswd = gethash( pass );
34a          memset( pass, 0, passSize );
35           if( strcmp( epasswd, shdwpwd ) != 0 ) {
35a              memset( epasswd, 0, epasswdSize );
36               attempts++;
37           } else {
37a              memset( shdwpwd, 0, 255 );
37b              memset(&attempts, 0, sizeOf( int ));
37c              memset(epasswd, 0,epasswdSize);
38               printf ( "Authentication successful!" );
39               getKeyfromDB( user, KeyA );
40               return 1;
41           }
42       } while( attempts < 3 );
42a      memset( shdwpwd, 0, 255 );
42b      memset( &attempts, 0, sizeOf( int ) );
43       return 0;
44   } // end of getPrivateKey
```

Figure 5: Transformed code of the running example from Figure 1. Newly added lines are highlighted with the gray color. The code for computing the size variables is not shown for brevity.

created by this piece of sensitive data. Another case is *epasswd*; the erase statements are introduced in a *path sensitive* manner, once at 35a in the *then* branch of the conditional, and the other along the *else* branch at line 37c, closing both instruction windows. A third example is the erase of *fname* and *userId* introduced only before *exit(2)* at line 5, to address the program exit window. We do not erase them at other places in the example as the caller of the function *file_encrypt*, which is not shown in the running example, may continue to use those values. For this example, the code for computing the size information of the variables is not shown in the Figure 5. For the variables whose type information can provide the size with the help C library functions, we use corresponding library methods to get the size. We discuss how we propagate the size information across functions in section 3.4.2.

3.1.2 Basic Idea

The transformation is performed based on the following idea.

In order to erase sensitive data precisely when they are no longer required by the program, our approach is to track and erase each sensitive data definition after its "last use".

Data definition can be through assignment statements or through function calls in a program. Notice that we make a distinction between the last use of a variable and the last use of a definition. A variable may be assigned many times in the program, and we do not attempt to find and erase the "last use" for the variable. Instead, we attempt to locate the "last use" for each definition of that variable, and erase after that use. Such a notion of "last use" is necessarily path sensitive, and there will be one such location along every control path of the program (The notion of "last use" is formalized in Section 3.2).

Why a Custom Analysis? It might seem that a standard static single assignment (SSA) transformation might give the benefits of identifying use of all definitions. One might think that it is possible to transform a given program into SSA form (17) and compute liveness information of variables to identify last use of each definition. However, mere liveness analysis on a program in SSA form is not sufficient to erase all definitions. For instance, a definition created inside a loop is considered as a single definition in an SSA form and a live variable analysis treats that variable as live throughout the loop. Consider the example given in Figure 6. In this case, for the definition of x at line 18 inside the loop, the SSA transformation does not add additional information about the dynamic nature of multiple definitions to x during loop iterations. And live variable analysis identifies x to be live throughout the loop. If the liveness information of variables is used to introduce the erasures, the erasure for the definition at line 18 will be placed after the loop, i.e., after line 23. Thus, we need additional analysis to precisely identify dynamic definitions and their ideal lifetimes (ideally, for the definition of x at line 18, the erasure should be immediately after line 20). Additionally, SSA form is useful only for register allocations in most production systems. To erase data from heap allocations, simple SSA form is not enough.

```
15    ...
16  while( condition ){
17     ...
18    x = definition ; // SSA does not add additional variables
19     ....
20    use(x);
21     ....
22     .... // live variable analysis identifies x to be live here
23  }
24    ....
```

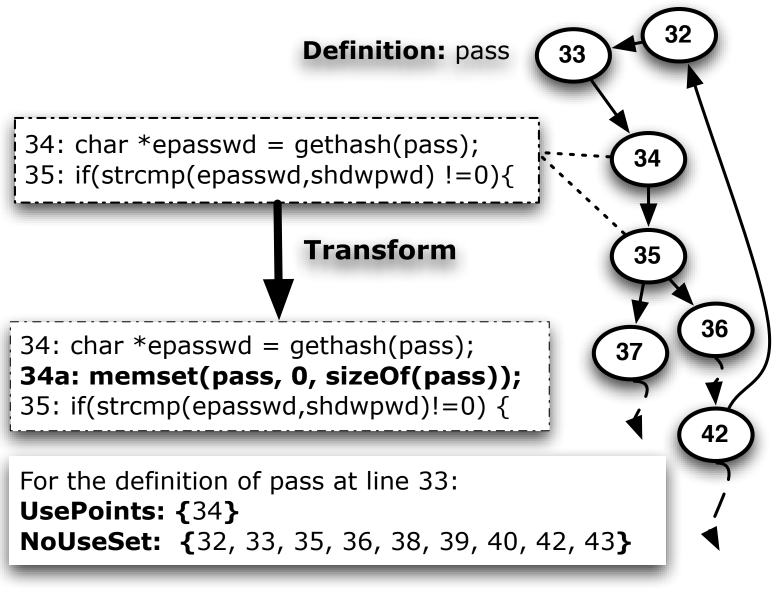
Figure 6: Definition inside a loop will have only one variable

The intuition behind the approach can be explained as follows: typically, the use of definition is often restricted to a certain locality in the program. The lifetime of this definition is often much shorter and often can be estimated more precisely using static analysis. More importantly, analysis and erasure of definitions is better for security; a variable may have many uses in a program and may have large "idle times" between definitions. Any residual sensitive data that remains between definitions remains exposed, and our approach to erase definitions essentially closes this gap.

Our approach first identifies sensitive inputs to a program through a policy. By default this policy treats any data received through a standard C library input function as sensitive and can be overridden by the program developer to precisely identify sensitive inputs to the program. Our approach then employs standard information flow tracking techniques to track propagation of sensitive inputs in the program. Finally, it identifies last use points for each sensitive data definition and conservatively adds erase instructions. The following are two additional important benefits of our approach.

- *Secure factoring of data dependencies:* Data dependencies in programs can often lead to sensitive residual data. For example, sensitive data may be propagated further in program's memory due to the effect of copying. Even if the original data is erased, this propagation will still leave residual data in memory. Since we capture all potentially sensitive definitions using information flow tracking, our approach offers the additional benefit of providing security factoring these data dependencies.

- *Usable without requiring specifications:* In the absence of user annotations to identify sensitive inputs to a program, our system employs a default, conservative policy that still ensures the sound erasing of all sensitive data. Thus our approach can still be readily applied to the source code of any program by a user unfamiliar with the program's source code.



```
34: char *epasswd = gethash(pass);
35: if(strcmp(epasswd,shdwpwd) !=0){
```

```
34: char *epasswd = gethash(pass);
34a: memset(pass, 0, sizeOf(pass));
35: if(strcmp(epasswd,shdwpwd)!=0) {
```

For the definition of pass at line 33:
UsePoints: {34**}**
NoUseSet: {32, 33, 35, 36, 38, 39, 40, 42, 43**}**

Figure 7: Illustration of *UsePoints* and *NoUse* sets and erase instruction for the definition of pass defined at line 33 (Erase is introduced after line 34)

3.1.3 Overview

To realize the basic idea explained previously, we need to identify program locations where each definition of a sensitive input is *available but not required*. We employ the standard reachability analysis to identify program locations where a definition is available. We then partition the reachable program locations into two sets: *UsePoints* and *NoUse* (§3.2.2). All program locations that use a definition are in *UsePoints* set. Further, all program locations that have at least one successor in the *UsePoints* set are included in the *UsePoints* set. Intuitively, the *UsePoints* set contains all program locations that either use the definition or must retain the definition for subsequent use. Rest of the reachable program locations are in the *NoUse* set. Figure 7 depicts *UsePoints* and *NoUse* sets for the definition of *pass* at line 33 in running example given in Figure 1.

Interestingly, the identified *UsePoints* and *NoUse* sets provide us program locations, we call *ErasePoints*, where erase instructions must be added. Specifically, nodes in *NoUse* set that have predecessors in the *UsePoints* set are the *ErasePoints*. Intuitively, once control flows from *UsePoints* nodes to *NoUse* nodes for a definition, that definition will never be used again and essentially is available unnecessarily in each program location in the *NoUse* set. In Figure 7 for the definition of *pass* at line 33, the *ErasePoints* set includes line 35. As shown, an erase is introduced before line 35 (line 34a).

Handling Aliasing: The presence of aliases (pointers in C) require care in reachability computation described above. Specifically, aliases may extend or reduce the lifetime of certain definitions thus may impact correct computation of *UsePoints* sets and *ErasePoints*. Our approach incorporates an alias analysis to ensure correctness of reachability, *UsePoints* and *ErasePoints* computation. The results from our alias analysis are used conservatively: the *must aliases* (i.e., those that only point to one memory location) are used to identify definitions that can be erased, while the *may aliases* are used for computing the locations where data is used. Thus being conservative in computing both erase and use locations, we are able to achieve correctness of transformation. We provide more precise details of aliasing in Section 3.2.

Handling Procedure Calls: The main challenge is to deal with the analysis of procedure calls to compute erase points. One straightforward way to deal with procedures is to inline procedure code and analyze the resulting program. However, this leads to a highly inefficient analysis that will not scale to large programs. We therefore develop a summary-based analysis (Section 3.2.3) that performs the analysis of a procedure *once* and reuses the results at every call site of the procedure.

3.2 Technical Description of the Approach

In this section, we discuss technical details of our approach. The details are explained in terms of algorithms, which are implemented by our tool SWIPE.

3.2.1 System Model and Terminology

Assumptions: Similar to previous works on alias analysis (18; 19), we assume that the use of pointer arithmetic to shift through objects (array, scalar or structure types) is only within their allocated memory sizes. This assumption is needed by the alias analysis currently implemented in SWIPE (20; 21). For those programs for which this assumption does not hold, bounds checking (22; 23) is needed to satisfy this assumption. We also assume that programs analyzed by SWIPE are single threaded, i.e., no concurrency.

P ::= S;	[PROGRAM]
S ::= l: `*px :=` E	[ASSIGN1]
| l: `x :=` E	[ASSIGN2]
| l: `px :=` E	[ASSIGN3]
| l: `if` E `then` S `else` S `endif`	[IF-ELSE]
| l: `while` E `do` S `done`	[LOOP]
| l: S `;` S	[LIST]
| l: `exit`	[EXIT]
| l: `return [x	px]`	[RETURN]
E ::= `c` | `x` | `&x` | `*px` | E `bop` E | `call f` ($E, \ldots, E$)	[EXPR]

TABLE II: A small subset of C language

Terminology: We consider programs in a subset of C whose grammar is given in Table XII. In this grammar, a statement is represented using S and each statement is prefixed with a label l that uniquely identifies its program location. We use the term *base variable* to denote a non-pointer variable that names a memory region. When memory allocation functions like *malloc* are invoked to assign memory to a pointer variable, SWIPE introduces a temporary variable as the base variable. SWIPE tracks only *base variables* and erases the data held by them. The variable x represents the base variable and px / py refer to pointer variables. The symbols $\&$ and $*$ represent operators *address of* and *pointer dereference* in C language respectively.

A definition at a location l is an assignment statement consisting of base variable (or a dereference of its alias) on the left hand side of the assignment instruction. We also identify definitions generated by certain library function calls such as *strcpy* or through user annotations. We say that a definition at l is a *must definition* of a base variable x (i.e., *mustdefinition(x)*) if the left hand side of the assignment is either x or $*px$ where $*px$ aliases only x at location l. Each definition in the program is represented with a unique identifier id. Table III lists the set of notations used in the standard program analysis literature that we will use. These values are computed using slightly modified standard algorithms (21; 20).

$A(v, l)$ & $A'(v, l)$	Returns a set of variables aliased to variable v just before (in-aliases) and just after execution (out-aliases) of the statement at location l, respectively.
$preds(l)$ & $succs(l)$	Returns a set of locations that represent immediate predecessors/successors of the program location l in program's control flow graph.

TABLE III: Standard terms used in SWIPE analysis

3.2.2 Intra-Procedural Analysis

For each given function SWIPE computes a Control Flow Graph (CFG) that contains a node for each instruction in the function. For each *must definition* of some base variable x, denoted by id, SWIPE computes three sets of locations namely, *UsePoints*(id), *NoUse*(id) and *ErasePoints*(id). *UsePoints*(id) denotes the set of locations where the definition denoted by id is used in statements at those locations. *NoUse*(id) is the set of locations where the definition denoted by id is available (or reachable) but not used. Effectively, *NoUse*(id) is the set of locations l such that, id is not used in l and is also not used on every control path from l until, and including the location l' such that $l' \in \{mustdefinition(x), exit, return\}$. Note that if $l \notin$ *NoUse*(id) then there is a location l', reachable from l, such that $l' \in$ *UsePoints*(id), i.e., the definition id is used at some location reachable from l.

ErasePoints(id) includes all those locations where the definition id can be erased immediately before those locations i.e., *ErasePoints*(id) is the set of locations l such that $l \in$ *NoUse*(id) and there exists an l' where $l' \in preds(l)$ and $l' \notin$ *NoUse*(id).

In the running example of Figure Figure 1, variable *pass* receives a new definition $pass\_d$ at line 33 which is only used in line 34, so *UsePoints*$(pass\_d)$ = {34}, *NoUse*$(pass\_d)$ = {32, 33, 35, 36, 38, 39, 40, 42, 43} and *ErasePoints*$(pass\_d)$ = {35}, i.e., the transition point from *UsePoints*$(pass\_d)$ to *NoUse*$(pass\_d)$.

Computing *UsePoints*(id): In order to compute *UsePoints*(id), we use the following functions ev and $DefsUsd$. Both of them take two arguments $E,\ l$ where E is an expression and l is a program location. $ev(E, l)$ defines the set of base variables that are directly referenced in E and $DefsUsd(E, l)$ defines the set of definitions that are used in E. The function ev handles *address-of* and *dereference* operators specially as application of these operators may effectively reference aliased variables. ev finds all base variables that the resulting expression is aliased to by using the alias relations at the current program location. The function is defined inductively and is self explanatory. $DefsUsd$ is defined using the function ev.

$$ev(E,l) = \left\{ \begin{array}{l} \phi \quad if E = c \\ x \quad if E = x \\ \&^{i+1}x \quad if E = \&y \; and \; \star^i y \in A(x,l) \\ \&^{i-1}x \quad if E = \star y \; and \; \&^i x \in A(y,l) \\ ev(E_1) \cup ev(E_2) \\ \qquad if E = E_1 \; bop \; E_2 | call f(E_1, E_2) \end{array} \right\}$$

$$DefsUsd(E,l) = \left\{ DefsHld(x,l) \middle| \begin{array}{l} x \in ev(E,l) \\ \text{and x is a} \\ \text{base variable} \end{array} \right\}$$

Where $DefsHld(x,l)$ is a set of identifiers representing definitions that the variable x may hold at the program location l. Now, *UsePoints*(id) is computed to be the set $\{\, l \,:\, id \in DefsUsd(E,l) \,\}$.

Alias Computation Exception: In the presence of *malloc* instructions inside a loop, the alias computation needs to be appropriately changed as follows.

SWIPE adds the following rule to the standard alias computation where $malloc\_base_l$ is the temporary base variable introduced.

$l : p = malloc(\ldots);$

$A'(x, l) = A(x, l) - \{*p\}, if * p \in A(x, l)$

$A'(\perp, l) = A(\perp, l) \cup A(malloc\_base_l, l)$

$A'(malloc\_base_l, l) = \{*p,\ malloc\_base_l\};$

This rule removes the LHS variable p from aliases of other variables. It updates alias information indicating $*p$ is an alias for $malloc\_base_l$.

We use the location specific $\perp$ i.e., $\perp_l$ to consider the definitions of $malloc\_base_l$ variables declared at each location l precisely. For malloc'd variables, the same algorithm to compute *NoUse* for non-malloc'd variables (described before) can be used with an exception to compute EP. The definition of $malloc\_base_l$ can not be erased at location l'' if -

- there exists a location l' where $\perp_l$ is referenced, i.e., there is a reference to $*p$ such that $*p \in A(\perp_l, l')$ and
- l appears in the path between l'' and l' .

This is to make sure that, we do not erase any previous definition of $malloc\_base_l$ inside a loop, before its use with the help of aliases of $\perp_l$. We can also erase data in memory locations allocated in earlier iterations (inside a $loop$) by tracking the references to $\perp_l$. This aids in erasing all the data without compromising the precision and soundness while applying rules to compute *ErasePoints*.

3.2.3 Inter-Procedural Analysis

Summary based approaches (24; 25) are used in the literature for inter-procedural pointer / alias analysis. Recent works such as (24; 25) generate summaries to capture the alias information purely for the purposes of analysis. In contrast, we employ the summary based approach for *transformation*. The main novelty of the technical approach developed here is to employ the summary-based approach to ensure that the instrumented code in a method is applicable for *all* calls to that method. To track and introduce erases for the data precisely, we build a summary-based approach with additional information that we describe in this section.

Intuitively, the summary-based analysis computes the "net effect" of calling a function as a summary. In doing so, it factors in issues such as

1. new definitions introduced by the function,
2. changes to aliases in the calling function caused by updates to formal parameters in the called function,
3. whether it is safe to erase some formal parameters inside the function, and
4. whether parameter values passed to the function are used inside it.

It is worth noting that traditional summary based analysis techniques usually consider only case 2 in the above list and not others. In the running example (Figure 1), after invoking the function *getPrivateKey* at line 3, the variable *key* would contain a new definition (case 1). Further, *key* passed to the function *encrypt* can be erased inside it provided that *key* is not used after the function call at line 6 (cases 3 and 4). Thus our detailed summary analysis readily facilitates eager erasure of sensitive data.

Once function summaries are available, our analysis computes the erase points and subsequently introduces erase instructions at these locations. For a given function f, SWIPE computes the summary of f into five components as shown in Table IV.

NewAllocVariables	Newly allocated memory variables inside called function which are accessible at call site.
NewDefinitions	New definitions that would propagate back to the caller. In Figure 1, after the call to *getpasswd* at line 33, *pass* points to a newly allocated variable. A new definition is also created (i.e., password).
AliasFunction	A function that captures changes to the alias relationship caused by the function. After the call to *gethash* at line 34 in Figure 1, *epasswd* becomes alias to newly allocated variable.
UsedIn	Denotes whether data passed to the function via the formal parameters is used inside. For *key* at line 6 in Figure 1, *UsedIn*$[2] = 1$, since *key* is used to encrypt the file.
UsedAfter	Denotes whether the definition held by a formal parameter is used after the function call in the caller. In Figure 1, if *key* is not used after the line 6, then *UsedAfter*$[2] = 0$.

TABLE IV: Summary captured by SWIPE

The component *AliasFunction* of the summary is obtained from aliases of variables at the return points inside the function. *UsedIn* ("used in") and *UsedAfter* ("used after") are m-bit vectors (where m is a natural number). *UsedIn*$[i] = 1$ if and only if i^{th} formal parameter is used inside the function, before being redefined. Similarly, *UsedAfter*$[i] = 1$ if and only if the i^{th} actual parameter is used after some call site, before being redefined.

Computing *AliasFunction*: Consider a function with formal parameters $f_1, f_2, ..., f_m$. Considering them as pointer variables, we let $level(f_i)$ denote the level of f_i. Thus, if f_i is a base variable then $level(f_i) = 0$. If it is a pointer to a base variable then $level(f_i) = 1$ and so on. Let rp be a special symbol that denotes a k-level pointer returned by the function. Let GPV denote the set of global variables. For each formal parameter we use f_i to denote the value of the formal before the execution and $f_i^{'}$ to represent the parameter after the execution of the function. Similarly, we use $x^{'}$ to represent the value of the global pointer variable x where $x \in GPV$, after the execution of the function.

AliasFunction is a function that denotes how the alias information of the formal parameters changes after execution of the function. Let D and R be sets as defined below.

- $D = NAV \cup \{*^i f_j \mid 1 \leq i \leq level(f_j),\ 1 \leq j \leq m\} \cup \{x \mid x \in GPV\} \cup \{\perp\}$.
- $R = \{\, *^i f_j^{'} \mid 1 \leq i \leq level(f_j)\ , 1 \leq j \leq m\} \cup \{x^{'} \mid x \in GPV\} \cup \{*^k\, rp\}$

Then, *AliasFunction* is a mapping from the domain D to the power set of R, i.e., $F\ :\ D\ \rightarrow\ 2^R$.

Significance of D and R: If $*f_2^{'} \in F(*f_1)$ then this denotes that after execution of the function, $*f_2$ aliases the same variable that $*f_1$ was aliasing before execution of the function. All references to formal parameters in *NewDefinitions*, refer to these values after the function has been executed. Observe that, for each x such that $x \in \{formals\ or\ GPV\}$, $x^{'}$ does not appear in D and x does not appear in R.

Note that the first four components of a function's summary are computed by analyzing the body of the function. The last component (*UsedAfter*) is contributed by call sites denoting whether an actual parameter is used after call sites. This "reverse feedback" from call sites is used to determine whether we could erase any formal parameters inside the functions. Use of an actual parameter after a call site would require that the corresponding formal parameter should not be erased inside the called function. If no such call site exists, we can erase the corresponding formal parameter inside the function.

```
1  char* fun(int *f1, int **f2){
2  int u = *f1 + **f2;
3  ...
4  *f2 = f1;
5  char *l3 = malloc(100);
6  if(u != 0) *f1 = newData;
7  ...
8  return l3; }
```

```
15  int *a1, **a2,*i1;
16  a1 = &x;
17  ...
18  i1 = &y;
19  a2 = &i1;
20  ...
21  a3 = fun(a1,a2);
22  int u = x;
```

Summary: $\textit{NewAllocVariables} = \{malloc\_base_5\}$
$\textit{AliasFunction}(*f_1) = \{**f_2', *f_1'\}$
$\textit{AliasFunction}(**f_2) = \{\}$
$\textit{AliasFunction}(malloc\_base_4) = \{*rp\}$
$\textit{NewDefinitions} = \{Def\ for\ (*f_1')\}$
$\textit{UsedIn} = [1, 1]$, $\textit{UsedAfter} = [1, 0]$

Figure 8: (a) Function *fun()* (lines 1-8) (b) Calling context of *fun()* (lines 15-22) (c) Summary of *fun()*

Observe that in Figure 8(a), at the end of the execution of the function *fun*, $**f_2'$ [1] will be aliasing the same location as $*f_1$ at the beginning of the function. Similarly, memory location aliased by $**f_2$ before the execution of the function has no aliases at the end of function. This is reflected in the summary of the function $fun()$ given in Figure 8(c). Also, observe that $malloc\_base_5$ is a newly allocated variable corresponding to $malloc()$ statement at line 5.

Applying Summaries: At the call site l, the invocation of function f should capture all the changes to aliases and definitions. Using the *AliasFunction* component of the summary, the corresponding aliases are updated. Let $f_1, f_2, \ldots, f_m$ represent formal parameters of a function f and $a_1, a_2, \ldots, a_m$ represent actual parameters passed to function f from the call site at location l, i.e.,

$$l : p = f(a_1, a_2, \ldots, a_m);$$

[1]For each formal parameter we use f_i and f_i' to represent the value of parameter before and after the execution of the function respectively. Also, $*^j a_i$ means applying dereference operator $*$, j times.

At the call site, an actual parameter may alias other variables. For a variable y at the call site, let $act(y)$, defined below, captures this aliasing relationship.

$$act(y) = \left\{ *^j a_i \middle| \begin{array}{c} *^j a_i \in A(y, l), \\ \\ 1 \leq j \leq level(a_i) \end{array} \right\}$$

Then the set *AliasFunction*(f_i) represents new aliases for a formal parameter f_i after execution of the function call. Replacing a formal parameter by the corresponding actual parameter in this set provides new aliases of the actual parameter at the call site. For each variable y at the call site, updates to alias relations due to function execution are computed as below where m is the number of formal parameters.

$$A'(y, l) = \left\{ \begin{array}{l} (A(y, l) - act(y)) \cup \\ \\ \bigcup\limits_{*^j a_i \in act(y)} \quad \mathit{AliasFunction}(*^j f_i)[p/rp][a_k/f'_k]_{k=1,\ldots,m} \end{array} \right\}$$

Intuitively, the above expression replaces appearances of actual parameters in alias relations before the function call, with new values as returned by *AliasFunction*. The notations a_k/f_k and p/rp represent replacement of formal parameter f_k by actual parameter a_k and return pointer variable rp by the LHS pointer p at the call site.

In the example given in Figure 8, at the call site of function fun (line 21), the alias information changes as shown in Table V.

Before call	$A(x,21) = \{*a_1\}, A(y,21) = \{*i_1, **a_2\}$, $A(i_1,21) = \{*a_2\}$
After fun call	$A(x,22) = \{*a_1, **a_2\}$, $A(y,22) = \{*i_1\}, A(a_1,22) = \{*a_2\}$ and $A(malloc\_base_4, 22) = \{*a_3\}$

TABLE V: Change to Alias information

Before calling the function, $act(x) = \{*a_1\}$, and $A(x,21) = \{*a_1\}$. After the call, aliases at the call site are computed as

$A'(x,21) = \{\{*a_1\} - \{*a_1\}\} \cup \{*a_1, **a_2\}$ which is $A(x,22)$.

The value of *UsedAfter* is updated at each call site for each actual argument a_i. If $*^j a_i \in A(z,l)$ and the definition of z at program location l is used at some l' that is reachable from l, set *UsedAfter*$[i] = 1$. Essentially the above computation checks if definitions passed as parameters to invoked functions are being used after the call site. In Figure Figure 8b, $*a_1$ is an alias to x and is not redefined in all paths of the function *fun*. Thus the same definition may be available and is used at line 22, hence *UsedAfter*$[1] = 1$. However, the second argument a_2 is aliased to variable y. Also, y is not used at line 22 and later, hence *UsedAfter*$[2] = 0$. The *UsedAfter* values of $[1,0]$ for function *fun* means that its first formal parameter cannot be erased within function *fun* but second can be erased as it is not used after the call site. Observe that for the function *fun* both parameters are used inside the function hence *UsedIn* = [1, 1].

3.3 Soundness

Soundness of our approach critically depends on the assumptions stated earlier and on the fact that we are erasing only *must definitions* and are using *may aliases* to compute *UsePoints*. Since our approach introduces erase statements only in the *NoUse* portion for each definition, it is clear that the addition of an erase instruction does not change the functionality of the original program. Now, with the help of following terminology, we introduce a theorem that states that our tool SWIPE does not affect the semantics of a given program even after introducing erase statements.

Let P be a program generated by grammar shown in Table XII. Further, assume that P does not use recursion or pointer arithmetic.

A $state\ s$ of P consists of two functions $(Address, Value)$. $Address$ maps variables to memory locations and $Value$ maps memory locations to its values. We assume that there is a set of base variables and base variables are mapped to fixed memory locations that do not change. An initial configuration is a pair (s_0, i) where s_0 is a state and i is the $label$ of the first statement of the program. Let $successor(s_j, i_j)$ be a $configuration$ (s_{j+1}, i_{j+1}) where s_{j+1} is obtained by executing the statement i_j in the state s_j and $i_{j+1} \in succ(i_j)$ be the next command to be executed in state s_{j+1}.

An $execution$ is a sequence $(s_0, i_0), \ldots, (s_n, i_n)$ such that (s_0, i_0) is an initial configuration and for each $j, 0 \leq j < n$, $successor(s_j, i_j) = (s_{j+1}, i_{j+1})$ and i_n is an $exit$ statement.

A base variable never appears on the left hand side of any assignment statement. As a consequence, in any execution, the address of base variable does not change.

Given a configuration (s, i), a memory location m is referenced in the configuration, if i is an evaluation of an expression e that has a variable y such that $s.Address(y) = m$.

Given an $execution$ $\rho = (s_0, i_0), \ldots, (s_n, i_n)$, a base variable x and a $configuration$ (s_j, i_j) in ρ, we say that the value of memory location pointed to by base variable x (i.e., the value of $*x$) is not needed from $configuration$ (s_j, i_j) onwards if either $*x$ is not referenced in the configuration (s_j, i_j) for $0 \leq j \leq n$ or there is an integer $j' > j$ such that $i_{j'}$ is a $Must\_definition$ of $*x$ and for every l where $j \leq l \leq j'$, the value of memory location pointed to by x is not referenced in configuration (s_l, i_l).

Lemma 3.3.1. *Suppose that* SWIPE *introduced an erase statement for the memory location pointed to by base variable x after the statement i in program P, then in every execution of P containing a configuration (s, i), the value of $*x$ is not needed from the configuration (s', i') onwards where (s', i') is the successor of (s, i).*

***Proof Sketch*.** Lemma 3.3.1 follows from the correctness of algorithms to compute aliases , *UsePoints* set, *NoUse* set, and *ErasePoints* and follows from the fact that if a definition at a memory location is erased at a node n in the CFG, then on all the paths of the CFG starting from the node n, that particular definition pointed to by the given memory location is never referenced. □

For any statement i, let $active(i)$ be the set X of all program variables x , such that there is a path in CFG from i to another statement j that references the memory location pointed to by x and there is no $Must\_definition$ of $*x$ on this path. Note that if $x \notin active(i)$, then every definition of memory location pointed to by x, that is live when control is at i, will never be referenced further in the execution.

We assume certain statements of programs are input statements that read input values from the environment (for example through files, or end user input). Certain statements are output statements which output value of a particular variable. In general, any execution of a given program would include the execution of input and out statements. In an output statement, the variable whose value is being output, is considered referenced in that statement.

Let P be the original program and P^T be the transformed program. Let $\rho = (s_0, i_0), \ldots, (s_m, i_m)$ be an $execution$ of P. Let $\rho' = (t_0, j_0), \ldots, (t_n, j_n)$ be an $execution$ of P^T.

We say that ρ' corresponds to ρ if the following conditions are satisfied -

1. There exists integers $u_0 < u_1 < \ldots u_m$ such that $u_0 = 0$, $u_m = n$ and for every l, $0 \leq l \leq m$, $i_l = j_{u_l}$ and $t_0 = s_0$ and for every memory location pointed to by a based variable x that is referenced in statement i_l, the values of $*x$ in s_l and t_{u_l} are the same, and every statement $j_k \in P^T$ and $j_k \notin P$ where $u_l < k < u_{l+1}$, j_k is an erase statement of memory location pointed to by the same base variable x

2. The interleaved sequence of input and output values (i.e., results of executing input and output statements) in ρ and ρ' are the same.

Th first condition above ensures that, excepting for the erase statements, the sequence of statements executed in ρ and ρ' are same and their results are the same. The second condition ensures that, the values read and output by both the executions ρ and ρ' are same. Whenever a variable is erased, it is no longer required by the program. Accordingly, the erased value has no effect on the values to be read by the program and results generated.

The soundness of our transformations is given by the following theorem.

Theorem 3.3.2. *For every execution ρ of P there exists an execution ρ' of P^T such that ρ' corresponds to ρ and vice versa.*

***Proof Sketch*.** Consider an execution ρ of P. Let $\rho = (s_0, i_0), \dots, (s_m, i_m)$. Let $\rho^{(k)}$ be the prefix of ρ given by $(s_0, i_0), \dots, (s_k, i_k)$.

We define an execution $\rho'^{(k)}$ of P^T that corresponds to $\rho^{(k)}$ by induction on k -

- Base case: $\rho'^{(0)} = \rho^{(0)} = (s_0, i_0)$
- Inductive case: For $k \geq 0$ assume we have defined $\rho'^{(k)}$ where $\rho'^{(k)}$ is $(t_0, j_0), (t_1, j_1), \dots, (t_{u_k}, j_{u_k})$. $\rho'^{(k)}$ satisfies the property that $j_{u_k} = i_k$. Our inductive hypothesis states that $\rho'^{(k)}$ corresponds to $\rho^{(k)}$ and also for every $x \in active(i_k)$, the values of memory locations pointed to by base variable x in s_k and t_{u_k} are the same.

Suppose SWIPE introduced erasures for the memory location pointed to by base variables in the set X_k after statement i_k. Let t' be the state obtained by executing the statement i_k in state t_{u_k}. Then, $\rho'^{(k+1)}$ is obtained by extending $\rho'^{(k)}$ with execution of these erase statements, followed by the configuration $(t_{u_{k+1}}, j_{u_{k+1}})$ where $t_{u_{k+1}}$ is the result of executing erase statements in state t' and $j_{u_{k+1}} = i_{k+1}$. By inductive hypothesis, for every $x \in active(i_k)$, the values of memory locations pointed to by x in states s_k and t_{u_k} is the same. If the statement i_k is an output statement, then the value output by the states s_k and t_{u_k} are the same. If the statement i_k is an input statement, then we make sure that the value read in executions ρ is same as the value read in $\rho'^{(k)}$. If the statement i_k is a definition, then the value that is assigned in both the executions is going to be the same, because every memory location pointed to by base variable x referenced in i_k has identical values in states s_k and t_{u_k}. Suppose the memory location pointed to by a base variable x is erased during the execution of all erase statements in t' to get to state $t_{u_{k+1}}$. Because of lemma 3.3.1, we see that the erased value of the memory location pointed to by x is not needed after i_k. Using this property and the inductive hypothesis it follows that - for every $x \in active(i_{k+1})$, the values of memory locations pointed to by x in s_{k+1} and $t_{u_{k+1}}$ are the same. Therefore, the inductive hypothesis holds for $\rho'^{(k+1)}$ also.

Similarly, it can be shown that for every execution ρ' of P^T, there exists an execution ρ of P that corresponds to ρ'. $\square$

The above theorem implies that the interleaved sequences of inputs and outputs in ρ and ρ' are the same and therefore they are going to produce the same interaction with the environment. Hence the programs P and P^T produce same results.

3.4 Implementation

3.4.1 Implementation Overview & Algorithm

This section provides an overview of the implementation and presents the algorithm employed by SWIPE. The first and key requirement to generate all the required information described in Section 3.1 is to have the callgraph for an application to be transformed. SWIPE computes the callgraph of the application to process all the functions. The callgraph is traversed bottom-up to generate function summaries and use them at their invocation points. Also, bottom-up traversal of the callgraph is required to track the flow of sensitive data through formal parameters.

Once the call graph is generated, SWIPE uses a three step process as shown in Algorithm 1 to transform a given application. In the first step of the algorithm, SWIPE performs a bottom-up traversal of the callgraph to compute aliases, reaching definitions, reachability and other components of the summary of the function. SWIPE also sets *UsedAfter* values of invoked functions, using the information at the call sites in step one and two. The second step requires a top-down traversal of callgraph to track definitions that flow across functions through formal parameters. In the final step, SWIPE computes erase points for all the sensitive definitions in each function and introduces erases for the same.

3.4.2 Handling Special Cases

In this section, we discuss various challenges we have encountered while implementing SWIPE.

Dynamic Allocations: To erase dynamically allocated memory regions, their size information is required at erase points. SWIPE introduces unique temporary variables to hold the size information until erase points. (If a bounds checking approach is integrated with SWIPE, these variables will not be required.) To propagate the size information across function boundaries a size stack is used that mirrors function call stack i.e., for each entry in the function call stack it's size information is available in the size stack.

Global Variables: A global variable g which is an array is construed as an implicit parameter to every procedure. Otherwise, g is handled by introducing a temporary pointer variable pg (initialized to point to g) which is considered as an implicit parameter to all functions. During the analysis, any references to g are considered as references to $*pg$. This way, we identify erase points for global variables in a manner similar to handling formal parameters to a function.

Arrays and Complex Data Structures: Standard alias analysis is not sufficient for arrays and complex data structures such as linked lists, to precisely erase data at its ideal lifetime. There are two techniques to tackle this problem:

Algorithm 1: SWIPE Implementation

```
Notation: $f$ - Function
          $n$ - Node in CFG of a function
          $C$ - Called function
          $id$ - unique identifier for a definition
          $CG$ - callgraph
  for each $f$ during bottom-up traversal of $CG$ do
      repeat
          update Aliases as in §3.2.2 and use AliasFunction of $C$ to update Aliases in the presence
          of call to $C$;
      until fixpoint ;
1     repeat
          update Reaching Definitions as in §3.2.2 and in the presence of call to $C$ use
          NewDefinitions of $C$ to update them;
      until fixpoint ;
      for each definition $id$ do
          update reachability information;
          compute UsePoints($id$);
          compute NoUse($id$);
      update UsedAfter of each $C$ called in $f$;
      compute NewAllocVariables, NewDefinitions UsedIn and AliasFunction of $f$ as in §3.2.3;
  for each $f$ during top-down traversal of $CG$ do
2     update UsedAfter of each $C$ called in $f$;
  for each $f$ do
      for each definition $id$ do
3         compute ErasePoints($id$);
          introduce $Erase(id)$;
```

1. A conservative approach where any reference to an element inside the data structure is considered as a reference to the entire data structure. Erases are introduced only after the use of all the elements. Thus this technique compromises precision to achieve soundness. This is employed by SWIPE.

2. Augment our approach with other sophisticated methods such as shape analysis (26) to reason about individual data items of complex data structures.

Libraries and APIs: Our underlying transformation infrastructure (CIL (27)) has limitations in dealing with glibc. We therefore do not support analysis and transformation of library code. To make the analysis correct, SWIPE applies over-approximated summaries for functions whose sources are not being analyzed or are not available at call sites. The over-approximated summary makes all pointer variables used at the call site aliases to each other and assumes that the invoked function does not introduce any new definitions. For a small number of cases (13), we overrode these assumptions with human generated summaries through manual analysis of source.

Recursion: The presence of recursion in a program yields cyclic call graphs. Such cycles may prevent fixed point computation from terminating. SWIPE does not handle recursion and conservatively summarizes (as given above) the nodes of call graph that lead to cycles.

3.4.3 Policy Specification

In our approach, a policy dictates which data items are sensitive in a given program. This is desirable for cases where a developer who is familiar with the program can specify its sensitive variables. However, we do not expect all users of our tool to be developers, they could be system administrators who have a high-level knowledge of the tool but are not intimately familiar with the program source code. For this reason, we have also provided a default policy that is conservative. In our default policy, we treat all C library functions which take data from external sources such as user, network or file system as sensitive. This default policy may often be used as a starting point for constructing actual policies by refinement.

Given the policy, we statically identify entry points for sensitive data in the program and then identify all definitions that may contain sensitive data because of propagation. This is done using a static data-flow analysis that merges results over paths, thus overapproximating the actual amount of sensitive data and thus is conservative. Being conservative here may add additional erase instructions for non-sensitive data but ensures that we do not miss erasing any sensitive data.

3.5 Evaluation

Prototype Our tool SWIPE is implemented as a source-to-source transformer and uses the CIL framework (27) and consists of approximately 5K lines of OCaml code. The CIL framework has limitations in dealing with glibc, especially with code that is written using GNU C extensions. Our implementation therefore does not analyze library code. To ensure that our analysis is correct despite this limitation, SWIPE applies *over-approximated* summaries for functions whose sources are not being analyzed or available at the call site. The over-approximated summary makes all pointer variables used at the call site aliases to each other and assumes no new definitions because of invoked function. For a small number of cases (13), where this assumption did not hold, we used summaries generated through manual analysis.

3.5.1 Correctness

Our first experiment with SWIPE aims to ensure that the behavior of the transformed program is not altered through our transformation. For this purpose, we transformed the regression testsuite of GCC that is often used in testing of changes to the C code bases. This testsuite provides a large collection of

Testsuite	**# of files**	**# of tests taken**	**# of tests passed**
gcc.c-torture	2076	20974	20974
compile	*811*	*5354*	*5354*
execute	*992*	*13912*	*13912*
unsorted	*273*	*1708*	*1708*
gcc.dg	2206	4068	4068
gcc.misc-tests	21	73	73
gcc.target	2	88	88
Total	4305	25203	25203

TABLE VI: GCC testsuite results for SWIPE

tests that cover different aspects of the C language evaluating functionality and correctness of any program optimizations. Given the comprehensiveness of the GCC testsuite, it provided a rigorous empirical benchmark for testing the correctness of our approach.

The GCC testsuite is organized as directories. The GCC C-torture tests contain particular code fragments which have historically uncovered fragile program transformations. These tests are run with multiple optimization options. *gcc.c-torture/compile* directory contains test cases that should compile but do not need to link or run while the *gcc.c-torture/execute* directory contains test cases that should compile, link and run. The *gcc.dg* directory contains tests for specific features of the compiler while *gcc.misc-tests* contains tests that require special handling and the *gcc.target* contains architecture specific tests. For *gcc.target*, we chose only x86 specific tests.

Table VI shows the split of the tests in the testsuite that were performed on CIL and SWIPE. From the collection of all GCC tests, we only chose those that passed successfully through CIL. It lists the number of files and the number of tests for each set of tests. The tests were run on a linux machine running Ubuntu 10.04 with GCC 4.3.3 and CIL 1.3.7. Out of 25203 tests performed on CIL across 4305 files, *programs transformed through* SWIPE *also passed all of them*. This experiment empirically demonstrates that our changes to the original program do not alter the original program semantics.

3.5.2 Scalability

Our next experiment is to check if SWIPE can scale to transform larger applications. We chose eight frequently used Linux applications of different sizes most of which handle sensitive information such as passwords, cryptographic keys, etc. We took three utilities from *OpenSSH(5.6)* - *SFTP*, *SSH* and *SCP*.

Application	**C**	**CIL**	**Erase sens defs**	
	eLOC	**eLOC**	**#eras**	**%sz ovrhd**
Bftpd	3348	62692	229	0.60
MySecureShell	4455	91817	109	0.36
SCP	18882	612672	196	0.28
SFTP	20835	632782	138	0.34
SSH	25619	722648	236	0.33
KeyRing	39230	1077533	59	0.40
GnuPg	68497	1006392	106	0.25
OpenSSL	137225	1085876	208	0.30

TABLE VII: Effect of SWIPE transformation on application sizes

Size Overhead Table VII presents the changes in size of applications because of our transformation. We used the RSM tool (28) to measure application codebase sizes in effective lines of code (eLOC). As shown in Table VII, CIL pre-processing of the original source code (column 2) led to significant increases in application source sizes (column 3). SWIPE transformation adds additional instructions as shown by erase instructions (column 4), and overall increase in the code size due to erase instructions (column 5). In all applications the size overhead due to SWIPE erase instructions was low (less than 1%).

3.5.3 Static Measure of Data Leak

As a definition is redundantly present in all statements of a *NoUse* set, its size provides a static metric of data leak. Care must be taken while interpreting this metric, as it may under- or over-represent leaks. In this metric, instructions appearing in loop bodies are counted once and function calls are counted as one instruction. At runtime such loops may execute more than once thus increase the actual number of instructions which leak a data definition. Similarly, function calls could leak a definition in more instructions. Thus the actual window of data exposure at runtime may be higher than the *NoUse* set size. At the same time, *NoUse* set may comprise of statements from alternate control paths and at runtime actual window of exposure may be less than the size of *NoUse* set.

Using SWIPE, we measured the *NoUse* set by considering all the definitions in the program. Along with the applications that are transformed earlier by SWIPE, we transformed *gzip* and *Oggenc*. Although, they do not involve any sensitive information, there were transformed primarily to assess the number of additional instructions inserted by SWIPE transformations and corresponding overhead. The results of this experiment are shown in Table VIII. Column 1 shows the application transformed. Sizes of the applications are shown in columns 2 and 3 (before and after CIL transformation respectively). Column 4 shows the number of erasures introduced for all the definitions in the program. The size overhead due to size tracking instructions is given in column 10.

Application	C	CIL	Erases	defs considered			Avg	Max	size	% size
	eLOC	eLOC	Intrd	Locals	Frmals	Total	NoUse	NoUse	trk instr	ovrhd
(1)	**(2)**	**(3)**	**(4)**	**(5)**	**(6)**	**(7)**	**(8)**	**(9)**	**(10)**	**(11)**
Bftpd	3348	62692	1262	956	33	989	30	672	147	2.25
MySecureShell	4455	91817	1960	1509	60	1569	37	398	220	2.37
SCP	18882	612672	6561	3886	375	4261	34	501	1532	1.32
SFTP	20835	632782	7035	4355	451	4806	33	501	1999	1.43
SSH	25619	722648	8955	5740	461	6201	42	1229	2167	1.54
KeyRing	39230	1077533	24815	17963	1395	19358	23	331	4249	2.69
GnuPg	68497	1006392	21923	16892	1050	17942	45	792	2448	2.42
OpenSSL	137225	1085876	98964	40207	2707	42914	319	2845	3038	9.39
gzip	4620	15335	1361	1172	8	1180	53	310	68	9.31
Oggenc	43246	108112	5390	4761	275	5036	32	569	656	5.59

TABLE VIII: Measure of *NoUse* sets for data in applications

Column 7 of Table VIII shows the number of definitions considered and columns 8 and 9 show average and maximum sizes of *NoUse* sets, respectively. Column 5 shows number of analyzed definitions that were local to functions and column 6 presents definitions that were passed to functions through formal parameters. The maximum number of instructions in which a definition can be exposed after its last use varied from 310 to 2845. Though the application *gzip* is smaller in terms of eLOC when compared to *Oggenc* or *GnuPg*, its *NoUse* set size is more. This is due to the presence of nested function calls in *gzip* source which are invoked many times in different parts of the program.

3.5.4 Effectiveness in Erasing Data

3.5.4.1 Process Memory Snapshot Analysis

To assess effectiveness of SWIPE in erasing sensitive data, we analyzed runtime process memory of the original and the transformed applications. We used Gnu Debugger (GDB) to obtain backtraces and coredumps of running processes.

Our experiments were limited to single snapshots of the process memory. Some of the applications (*SFTP*, *Bftpd*) exited immediately after their execution and required artificial time delays to ensure that the program executed long enough to capture memory snapshots.

Other applications (*OpenSSL*, *MySecureShell*, *GnuPg*, *SSH*) were designed for manual termination through user interfaces and enabled us to attach to the running process after executing them with GDB. The processes were frozen at the point when the snapshot were taken. The backtraces provided valuable information on variable values whereas the coredumps gave us data not erased from the memory from the start of the program execution to that point.

We analyzed all backtraces and coredumps for the presence of known sensitive information and found that in many of the untransformed applications, sensitive data such as passwords, keys or file contents were present in the memory. Table IX shows the occurrences of sensitive information found during analysis that was not being erased by original applications. The SWIPE transformed applications were then run through the same set of experiments and no traces of sensitive information were found in the memory.

An alternate technique called secure deallocation (12) which is close to our work, erases data at exit points, return instructions and augments free method calls with erase instructions. This technique is simple to implement as there is no analysis required to retrofit the application. However, this technique will not be able to close instruction exposure windows. A program may spend more time in such instruction windows (because of loops or function calls) thus exposing the data for a longer duration. This is supported by our empirical analysis (Table I in section 2.4) which found significantly large instruction windows in the studied applications even without counting the effect of loops and any library calls.

Application	**Sensitive Variable**	**In Original?**	**In Sec Dealloc?**	**In SWIPE?**
Bftpd	password	Yes	Yes	No
MySecureShell	filename	Yes	Yes	No
	password	Yes	No	No
SFTP	hostname	Yes	Yes	No
	filename	Yes	Yes	No
SSH	hostname	Yes	Yes	No
	password	Yes	Yes	No
GnuPg	passphrase	Yes	Yes	No
OpenSSL	Key	Yes	No	No
	location	Yes	Yes	No

TABLE IX: Memory analysis for sensitive data

Closing such instruction windows will greatly mitigate the risks of sensitive data exposure. To perform a detailed comparison with the above method, we implemented the secure deallocation approach of (12), by transforming C applications using CIL for a comparison with our tool SWIPE. Table Table IX also shows the occurrences of sensitive data in the memory snapshot of the applications transformed through secure deallocation approach. Note that in most applications, we noted presence of sensitive data in the secure deallocation approach that was absent in the SWIPE transformed application. For *Bftpd* and *SFTP*, there was no deallocation mechanism (such as free) for the data of interest. The program *SSH* still leaves a copy of user password in memory which is not being deallocated. In case of *GnuPg*, only one of the two occurrences of passphrase is erased by secure deallocation approach. Thus, the results from this experiment demonstrate that the secure deallocation is not completely effective in removing sensitive data from typical systems programs.

These experiments demonstrate that even popular applications fail to erase sensitive data from memory that, if leaked, can have serious implications. As observed earlier, SWIPE transformed applications did not leak any such information, thus demonstrating the security benefit provided by our transformation.

3.5.4.2 Cold Boot Attack

To underscore the importance of erasing data, we conducted an experiment to demonstrate how a malicious user can gain access to the sensitive data leaked in the memory. We used the approach devised by Halderman et. al (5) to perform the cold boot attack. In a cold boot attack, the attacker with physical access to a computer can retrieve encryption keys from a running operating system after using a cold reboot (typically by pressing and holding the power button until the machine switches off and then switching it back on) to restart the machine from an off-state. The attack exploits data remnant property of RAM, minutes after power is switched off. The application we chose for the demonstration was *OpenSSL*. The untransformed *OpenSSL* application was run on a Windows machine to generate a private-public key pair. Immediately after this, the machine was cold rebooted and the RAM snapshot was dumped into a USB memory stick using their tool (5). This snapshot was then analyzed with Volatility (a memory forensic tool). We were able to confirm that the keys were indeed available. We performed the same experiment on the SWIPE transformed *OpenSSL* application by following the exact same steps. The subsequent RAM snapshot obtained from the cold boot attack did not contain any keys. This demonstrates the effectiveness of our approach as a strong countermeasure against a very stealthy attack.

3.5.5 Performance

Transformation Time Table X depicts the time taken by SWIPE to transform applications. We also show the total number of files and functions SWIPE analyzed for each application. We can observe that the time taken for the analysis (transformation time) increases with the number of functions being analyzed.

We also compared the time taken to generate the compilation unit of an application with SWIPE and with an extended CFG that is described in section 2.4. As expected, the summarization based SWIPE implementation was able to handle larger programs in less time compared to the extended CFG (ECFG) implementation.

Application	**No. of C Files**	**No. of Funs**	**Call Depth**	**Call Sites**	**Xform Time (sec)**
Bftpd	12	147	8	609	18
MySecureShell	26	189	9	1660	25
SCP	76	749	15	4775	120
SFTP	78	799	15	5577	133
SSH	84	906	24	7979	152
KeyRing	106	2589	15	12518	1544
GnuPg	140	1788	54	8224	498
OpenSSL	626	5669	33	53486	4109

TABLE X: SWIPE transformation metrics

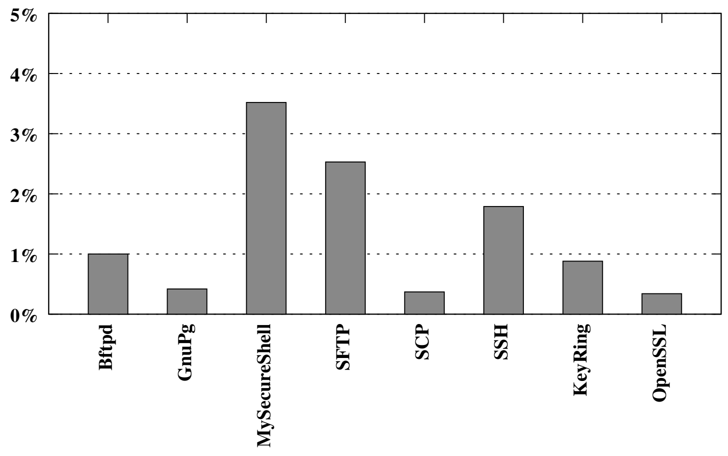


Figure 9: SWIPE performance overhead in percentage

One of the main reasons for this reduction is that performing fixed-point computation on large CFGs is costly. Further, in summarization based implementation each function body is analyzed only once (to generate summary), as opposed to the ECFG implementation where each function is analyzed once for each call site.

Runtime Overhead: Figure 9 shows the runtime overhead of the transformed applications. Erasing sensitive data led to low overheads ranging from 0.34% to 3.5% and averaging 1.35%. We captured the overhead on a particular run for each application. Also, we disabled optimizations so that the optimizer does not remove instrumented instructions. In the presence of optimization flags, one could use compiler directives to retain erase instructions (29).

Improving Precision of Erases in Original Applications We further analyzed applications to assess SWIPE's precision in placing erase instructions. For this purpose, we identified applications that were already erasing some of their sensitive variables, and compared the placement of the programmer-introduced erase instruction and the SWIPE-introduced erase instruction. Table XI lists application (column 1), analyzed variable (column 2) in the application, mechanism used to erase/free sensitive data (column 3) and SWIPE action (column 4).

Application - Variable	**Original**	**SWIPE**
SFTP- password	memset	memset (before 10 instr)
SSH- password	memset	memset (before 5 instr)
GnuPg- passphrase	free	memset (avg window 6)
OpenSSL- key	free	memset (avg window 10)
MySecureShell- password	free	memset (avg window $>$ 10)
KeyRing-login_password	erases	No erase

TABLE XI: Erasing precision of SWIPE

Table XI shows that for five of the six analyzed applications, SWIPE was able to improve the precision by shortening the window of exposure. SWIPE identified locations before *free* calls to introduce erases (rows 4, 5 and 6). The *KeyRing* application (row 7) stored the identified sensitive data in a global variable, and could not be erased due to limitations in our current implementation.

For the programs that did erase sensitive information (rows 2 and 3), SWIPE improved precision by identifying an earlier point in the program to introduce erases, thus closing the exposure windows further. This is interesting because programmer introduced erase instructions are typically expected to be more precise than an automated tool such as ours. For example, the original *SSH* application erases sensitive password value. However, SWIPE identified erase point for *password* in a function (*ssh_put_password*) which is called prior to the original erase (*memset*) instruction. This experiment indicates that SWIPE can safeguard applications that fail to erase sensitive data as well as improve precision of the existing erases in applications.

3.6 Summary

In this chapter, we presented SWIPE, an automated approach and tool for reducing lifetime of sensitive variables. Our approach and tool employs static analysis for tracking sensitive information and automatically transforms the program with instructions that erase all sensitive data after intended use. The effectiveness of our approach was demonstrated with a set of real world C programs that handle sensitive information such as passwords and keys.

CHAPTER 4

DEICS: DATA ERASURE IN CONCURRENT SOFTWARE

Reasoning about the scope of shared data in concurrent applications is difficult in general. As shared data is used by different threads simultaneously, statically identifying ideal lifetimes of such shared data is a challenge. Program analysis techniques that were developed to analyze data-lifetime earlier, mainly focus sequential programs. Concurrent programs contain an element of non-determinism, which is the challenge to reason the behavior of shared data. In this chapter, we explain our approach to minimize data lifetime for concurrent applications written in C.

4.1 Revisit the Running Example with Concurrency

We revisit the the running example for concurrent programs (in Figure 3) with the help of a control flow diagram given in Figure 10, which represents the typical execution scenario. Each node represents a line number in the program. Dotted arrows represent the parallel execution of threads. Nodes where $request$ is initialized (node with number 9) and where it is actually used (node with numbers 10 and 25) are shown with double layers. We also show where the $request$ is actually required (with strong lined arrows), and where it is available but not required (with thin lined arrows). From the diagram (in Figure 10), it is clear that, there are nodes where the shared data is not required, but is available.

4.1.1 Challenges

- **Ensuring Soundness:** The main challenge in analyzing a concurrent application is the non-determinism involved in thread interleavings. Therefore, it is difficult to identify the sequence

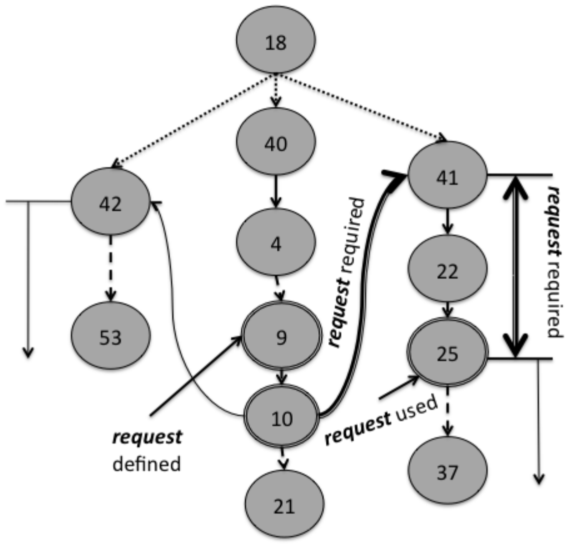


Figure 10: Control flow diagram of example in Figure 3

of accesses for a shared data from different threads. For example, consider the example given in Figure 3. Following a sequential approach (such as SWIPE (30)) to identify location where $request$ data is no longer required may locate program points where data can be erased. However, it may not be correct as the sequential approach does not consider potential thread interleavings. In the example given in Figure 3, assuming the thread invocations as normal function calls, for the shared variable $request$, an approach like SWIPE, would correctly identify it to be immediately after line 25 in worker thread. On the other hand, if the programmer invokes the consumer thread first and then the producer thread (i.e. interchanging the code on lines 40 and 41), the sequential analysis would identify the final use point of $request$ to be at a line 10 in server thread where the $request$ is being logged. Thus, one of the main challenge for our analysis is to consider all potential thread interleavings (irrespective of how they are created) before we identify locations where a piece of shared data is not required.

- **Presence of Function Calls:** Our analysis should also consider function invocations from threads as they may change the behavior of lock regions inside a thread. For example, consider that there is call to function $fcall$ in a thread $T1$ as shown in Figure 11. The use of $data$ is inside $lock$ and $unlock$ calls. However, there is a call to function $fcall$ on line 5 which unlocks and locks on the $lockv$. Although, the call to $fcall$ may not change the fact that the lock is held before accessing the $data$, there is still a possibility of another thread $T2$ acquiring the lock before the $fcall$ reaquires it. If our analysis erases the $data$ in $T2$ thread, the $data$ accessed at line 6 in thread $T1$ will be zero thus changing the program behaviour. Therefore, our approach should identify any such function calls and analyze accordingly.

4.2 Approach

In this section, we first present how our approach transforms an application with the help of running example (Figure 3) given before. We then explain the basic idea and discuss challenges involved. A formal description of the approach is provided with the help of an abstract version of the language with concurrency feature.

4.2.1 Transformed Example

We first show how our approach transforms the original program given in Figure 3. Figure 12 shows the result of transforming the running example (Figure 3) using our approach. The shared variable $request$ is erased after line 25 (shown with a label 25a). Since $request$ is no longer required

```
int T1(){
   ...
  lock(&lockv);
   ...
   fcall ();
  consume(data);
  unlock(&lockv);
   ...
}
   ...
void fcall (){
    unlock(&lockv);
     ...
    lock(&lockv);
  }
```

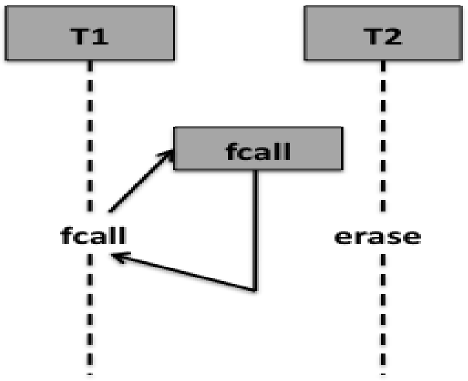


Figure 11: Invoking a function inside a thread

after line 25 in worker thread. Note that, here we only highlight the erasure of shared data. There will be erasures for local variables within each function/thread based on the policy we adopt for sensitive data.

Main Idea: Our approach to minimize lifetime of shared data in concurrent applications is to identify a location in the application, after which a particular definition of shared data is no longer required. If a shared definition is available at a location in a thread and is no more required further in that thread or any other thread running in parallel, we can safely erase such data after that location. Since execution of threads is not predictable statically, we adopt a conservative approach of identifying a location for a definition after which no other threads accesses that definition. Note that, we use the word *definition*, to differentiate between a variable and its values at different times during the program execution. A shared variable can hold multiple definitions in a program. Our analysis treats each definition of a

```
 1  DEFINE LENGTH 20;
 2  char *request;
 3  mutex_type lockv;
    /* Server thread which accepts the connection
    and collects HTTP requests from clients */
 4  int server( ){
 5  char * localdata ;
 6   lock(&lockv);
 7    request = malloc(LENGTH);
 8    localdata = getRequestfromUser();
 9    strcpy ( request , localdata , LENGTH);
10    log( request );   // read( request )
11   unlock(&lockv);
12   // do other work : Lines 12- 20
21  }

22   int worker(){
   /* Worker thread to process HTTP request */
23     lock(&lockv);
24     if( request != NULL){
25       process( request );   // read( request )
   25a: memset( request , 0, LENGTH );
26     }
27     unlock(&lockv);
28  // work like generating response : Lines 28 - 36
37   }
38   int mian(){
39     mutex_init(&lockv);
40     thread_create ( server );
41     thread_create (worker);
42     // do some other work : Lines 42 - 52;
53   }
```

Figure 12: Transformed running example with erase instructions for shared data.

shared variable separately and tries to erase the contents of each definition after its intended use in the program.

To erase any definition, we need to insert a write operation on the shared variable (i.e. the erasure) in a particular thread. To perform a check that there are no other threads that need this data further, it is sufficient to confirm that, the write that we inserted does not influence any read. Such writes should not have any significance from the program's perspective. Consider the running example given in Figure 3. If we introduce erase (i.e., a write) for $request$ in worker thread after line 25, we can clearly see that there is no parallel read for $request$. Clearly, the erase for $request$ in worker thread is insignificant. Hence, our analysis should introduce insignificant writes to erase any data so that the program does not depend on those writes at any point during the execution. At the same time, our analysis should identify potential read operations that would get influenced by the write we introduce. For instance, in the running example given in Figure 3, if the $request$ is erased in server thread after line 10, the worker thread may get a zero value for access at line 25. Hence, we can not introduce erase for $request$ in server thread.

```
 5  mutex mutexlk;
 6  char *data;
 7  int producer(){
 8    lock(&mutexlk);
 9     ...
10    data = produce();
11    unlock(&mutexlk);
12     ...
13    // other task
14    lock(&mutexlk)
15    access(data);
16    unlock(&mutexlk);
17  }
18   ...
19  int consumer(){
20     ...
21    lock(&mutexlk);
22     ...
23    consume(data);
24    unlock(&mutexlk);
25     ...
26  }
```

Figure 13: Access data in producer thread

One may be tempted to identify parallel reads that may get influenced by the write we introduce by checking for data-races[1] because of our write. However, this is not always true, especially when locks are used for every shared data access. Consider another producer-consumer example given in Figure 13.

[1]There are various types of races as explained in (31), but we use the generic notion of a data-race as - two threads access and modify a shared data at the same time, i.e., without any protection mechanism.

In this case, even if the $data$ is no longer required after line 23 in the consumer thread (Figure 13), it may still be required in producer thread based on when the code on line 15 gets executed (during the execution, depending on thread scheduling, thread interleaving may make the line 15 execute after line 23). However, a race-detection does not identify the write we may introduce before line 24 as a data-race as all the accesses are protected by the locks. This leaves us with a challenge to keep track of all the accesses to shared variable irrespective of the program being data-race free. Note that we do not show $free$ instructions and erasures for the shared data in $producer$ thread for brevity. However, the case for erasing in server (producer) thread for the example Figure 3 is discussed in section 4.2.2.

```
 1  char *data;
 2  mutex lockvr;
 3  int threadA(){
 4    lock(&lockvr);
 5     ...
 6    data = definition1 ;
 7    unlock(&lockvr);
 8     ...
 9    // other task
10    lock(&lockvr)
11    data = definition2 ;
12     ...
13    access(data);
14    unlock(&lockvr);
15  }
16  ...
17  int threadB(){
18     ...
19    lock(&lockvr);
20     ...
21    read(data);
22    unlock(&lockvr);
23     ...
24  }
```

Figure 14: Multiple definitions inside a thread

Additionally, mere presence of parallel reads should not prevent the introduction of erasures. Consider the example given in Figure 14, where there are multiple definitions for the shared variable $data$ inside the thread $threadA$. Introducing erasure for the shared data $data$ in thread $threadB$ before line 22 does not influence the value read in thread $threadA$ at line 13. Therefore our analysis should verify if the parallel reads in other threads are actually influenced by the erasure we introduced. In this example, both the definitions of shared data $data$ at lines 6 and 11 in threadA get erased if an erasure is introduced in threadB.

4.2.2 Approach Overview

Given two threads $T1$ and $T2$ (and the non-determinism in their interleaving during execution), we need to ensure that for a shared data, the erasure point identified in thread $T1$ is safe, i.e., thread $T2$ will not need this data anymore. One simple way is to analyze thread $T2$ to check if there are any $read$ operations on the shared data appear. This is not sufficient as there could be other writes in thread $T2$, which actually influence the read operation (as shown in example Figure 14). However, if we can identify a location l in thread $T2$ which would get influenced by our write in thread $T1$, and an actual read operation in $T2$ is reachable from l without another write operation in between, then the actual read is influenced by our write. To identify such locations that would get influenced by our erase (i.e., write operations), we can make use of a *pseudo-read* at those locations and check for a data-race (in particular, Write (our erasure)-Read (pseudo-read) races). We use the term pseudo-reads for imaginary reads, which are just used to query the race-detection engine, but are not actual reads in the program. Then, our analysis can be reduced to the problem of identifying critical pairs of locations (l, l'), where l is location in a thread where a pseudo-read of the shared data is in race with the write we introduced and l' is another location which consists of an actual read operation on the shared variable and it is reachable from l without another definition to the shared variable in between. Absence of such critical pairs confirm that we can safely introduce erasures.

```
22    int worker(){
     /* Worker thread to process HTTP request */
23     lock(&lockv);
24     if( request != NULL){
25       process( request );    // read( request )
     25a: memset( request , 0, LENGTH ); // write( request )
26     }
27     unlock(&lockv);
28   // work like generating response : Lines 28 − 36
37    }
```

Figure 15: Worker thread with the erase instruction

We explain our approach with the help of the running example given in Figure 3. Consider the erase of the shared variable $request$ in $worker$ thread at new line $25a$ as shown in Figure 15.

We now introduce pseudo-reads for the shared variable $request$ after line 5 and line 11 in the $server$ thread as shown in Figure 16.

Querying a race-detection engine identifies the pseudo-reads after lines 5 and 11 in $server$ thread to be in race with the write after line 25 in $worker$ thread. Note that we show pseudo-read instructions only after lines 5 and 11. Pseudo-reads after lines ranging from 6-10 will not be in race with the write

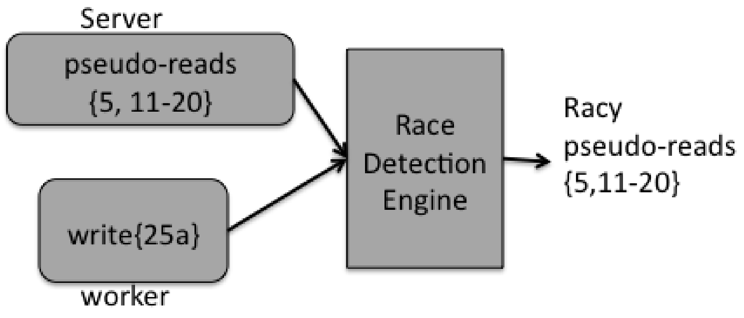
```
 4  int  server( ){
 5  char * localdata ;
    pseudo−read(request);
 6   lock(&lockv);
 7    request  = malloc(LENGTH);
 8    localdata  = getRequestfromUser();
 9    strcpy ( request ,  localdata , LENGTH);
10   log( request );       // read( request )
11   unlock(&lockv);
    pseudo−read(request);
11   // do other work : Lines 12 − 20
21  }
```

Figure 16: Pseudo-reads in server thread

we introduced at line 25a as these lines are inside lock region. However, we can optimize the number of pseudo-read instructions required in a thread, which we explain in section 4.3.

There is an actual read for the shared data $request$ inside the $server$ thread at line 10, which is reachable from the pseudo-read after line 5. However, it cannot become a critical pair as there is a definition for $request$ at the line 9 which is in the path from line 5 to line 10. Also, for the pseudo-read after line 11, there is no actual read on $request$ in $server$ thread reachable from line 11. In this scenario, the set of critical pairs is empty for the write at line 25. Therefore, we can introduce the erase instruction for the shared variable $request$ inside the thread $worker$.

Similarly, we could introduce an erasure for the shared data $request$ in $server$ thread after the last use at line 10. However, for this write on $request$ after line 10, there is a pseudo-read after line 22 in the $worker$ thread in race and there is an actual read at line 25 reachable from line 22 without any



definition on $request$. In this case, the pair $(22, 25)$ is a critical pair. Hence we cannot introduce erase for $request$ in $server$ thread.

In the rest of this section, we extend the intuitive idea given above with the help of a detailed technical description. A high-level algorithm of our implementation is provided in next section.

4.2.3 Technical Description

4.2.3.1 System Model

We formalize the intuitive description given before using a subset of C language with concurrency constructs given in Table XII. Table XII gives the syntax of the executable part a program or a function. Declaration of a function is given by specifying the function name, return type, the formal parameters, and the function body specified using the syntax of Table XII. The labels of statements in all the functions including the main function are assumed to be distinct.

In a program P, threads (represented using t in the language shown in Table XII) are created by invoking the call $thread(t)$. Thread invocations are different from normal function calls. They represent

P ::=	S;	[PROGRAM]
S ::=	l: `*px := ` E	[ASSIGN1]
|	l: `x := ` E	[ASSIGN2]
|	l: `px := ` E	[ASSIGN3]
|	l: `if` E `then` S `else` S `endif`	[IF-ELSE]
|	l: `while` E `do` S `done`	[LOOP]
|	l: S `;` S	[LIST]
|	l: `exit`	[EXIT]
|	l: `return [x | px]`	[RETURN]
|	l: `lock (lckv)`	[LOCK]
|	l: `unlock (lckv)`	[UNLOCK]
|	l: `thread (t)`	[THREAD-CREATE]
E ::=	`c` | `x` | `&x` | `*px` | E `bop` E	
|	`call f (`$E, \dots, E$`)`	[EXPRESSION]

TABLE XII: A small subset of C language with concurrency constructs

a parallel execution during runtime. Functions in the program are classified into two distinct sets called $ordinary\ functions$ and $thread\ functions$. A thread function is only invoked when a thread is created (i.e., using the call $thread(t)$). Whereas an ordinary function is not invoked in a thread creation statement. Inside a function, ordinary functions or other thread functions can be invoked.

We use sv to represent a typical shared variable and $lckv$ to represent a typical lock variable used for synchronization. We make the following assumptions for our analysis.

Assumptions: We assume that functions invoked in distinct thread functions are different. For ease of presentation we assume that there is no recursion. In this model by inlining all the ordinary function invocations, we can convert the program into a form that contains only the main function and thread functions. The bodies of the main function and the thread functions invoke only thread functions. We treat all the global variables as shared variables (similar to (32)). Authors of (30; 33; 19; 21; 20) assume that a pointer variable in the program accesses the data within its allocated memory bounds. We make similar assumptions about pointer variables for our analysis. Further, we assume that the shared memory model of the programs we analyze exhibit the sequential consistency (34) behaviour, i.e., the memory operations are executed in the same order in which they appear in the program. Instructions of different threads are executed in a sequential order which is an interleaving of the sequence of instructions of each thread specified by the thread function. If the concurrent program does not have any data races, this property is usually satisfied.

A definition is denoted by a unique identifier id. Standard definitions of $Aliases$, $must\_definitions$, $may\_aliases$, $succ$, and $preds$ are used. A definition of a variable x/sv at location l is a $must\_definition$ if the left hand side of the assignment consists of x/sv or $*p$ for a pointer variable p where $*p$ aliases only to x/sv at location l.

In our analysis, we treat a shared variable sv as a formal parameter. Throughout our analysis we differentiate between local data and shared/global data. We also identify different definitions of sv.

4.2.3.2 Intra-Procedural Analysis

Our tool, DEICS first computes a control flow graph (CFG) for each thread function in the program. The CFG represents each instruction of the program as a node. For each definition (including the definitions of shared variables which are added as formals) denoted by id, DEICS computes all the nodes where the definition id is reachable inside the function. We call this set as $Reachability(id)$. We split the $Reachability(id)$ set into three different sets named *UsePoints*(id), *NoUse*(id) and *ErasePoints*(id). *UsePoints*(id) is the set of nodes where a definition id is required and *NoUse*(id) is the set of nodes where the definition id is available but not required. For a given definition id, a transition from

UsePoints(id) to *NoUse*(id) is the place where we can introduce erase instructions for id. We call this set of nodes as *ErasePoints*(id) before which we may be able to safely introduce erase instructions (similar to (30)).

During our analysis, we identify each local variable x for each thread function. For each definition id of this local variable x at each location l in *ErasePoints*(id) we can safely introduce erase instructions before l provided there is no global pointer p pointing to x, i.e., for each global pointer variable p, $*p$ does not alias x. From this point, we only provide the treatment for definitions of shared variables.

In the running example in Figure 3, for the shared variable $request$, treating it as formal parameter to each function ($server$, $main$, and $worker$), our analysis identifies *ErasePoints*$(request\_d)$, where $request\_d$ is the identifier we use for definition of $request$ inside each function. In $server$ thread for the shared variable $request$, *ErasePoints*$(request\_d) = 11$ and in $worker$ thread, *ErasePoints*$(request\_d) = 26$.

Once the *ErasePoints*(id) is computed for shared variables, our analysis needs to check that if introducing these erases(which are nothing but write instructions) before *ErasePoints*(id) would cause any data-races in the program. For this, we first introduce dummy writes (*DummyWrite*(id)) for each definition id of shared variable inside a thread function before each location in the set *ErasePoints*(id). We use dummy writes to denote another set of imaginary writes we introduce in the program for querying the race-detection engine for potential races because of these imaginary writes. Note that, we use both pseudo-reads and dummy writes to identify safe erasures. Our analysis then reduces to identify the set of *RacyPairs*(*DummyWrite*(id)).

Definition 1. *For each DummyWrite(id) of a definition id of a shared variable sv, RacyPairs(DummyWrite(id)) is the set of all pairs (l, l') of locations in some thread function t such that a pseudo-read immediately after location l would be in race with the DummyWrite(id) we introduced, and there is a path from l to l' in the CFG of t such that there is no $must\_definition$ of sv on this path and there is a $read$ of sv at l'.*

Existence of such a pair of locations indicate that it may not be safe to introduce *DummyWrite*(id). However, absence of such pairs confirm that we can safely introduce $Erase(id)$ for shared data.

In the running example Figure 3 for the shared variable $request$, with the set *ErasePoints*$(request\_d) = \{26\}$, a dummy write, *DummyWrite*$(request\_id)$ is introduced before line 26 in the $worker$ thread. As explained in section 4.2.2, in $server$ thread, for $request\_d$ defined at location 9, pseudo-reads after lines 5 and lines ranging from 11-20 can race with *DummyWrite*$(request\_id)$ we introduced in $worker$ thread. There is a $read$ access on $request$ at line 10 reachable from line 5, however there is another definition of $request$ falls in the path from line 5 to line 10. For the pseudo-read after line 11, there is no actual $read$ on the $request$ reachable from line 11. Thus, the set *RacyPairs*(*DummyWrite*$(request\_id)$) is empty. Hence we can safely introduce erase in $worker$ thread before line 26.

Similarly, for the definition of $request\_d$ defined at line 9 in the $server$ thread, *ErasePoints*$(request\_id) =$ 11. Inserting a pseudo-read after line 22 in the $worker$ thread and querying the race detection engine for race with the write we introduce in the $server$ thread before line 11, would indicate a warning of data-race. We can clearly see that, the read at line 25 is reachable from location 22 without killing the definition. That is, there exists a pair of locations $(22, 25)$ such that pseudo read after line 22 is in race with *DummyWrite*$(request\_id)$ and an actual read at line 25 is reachable from 22, i.e., the set

RacyPairs(*DummyWrite*($request\_id$)) includes the pair $(22, 25)$. Hence, we cannot introduce erase for $request$ in $server$ thread.

4.2.3.3 Inter-Procedural Analysis

When ordinary functions are involved we use the summary of the function at each invocation. We follow the approach given in SWIPE to compute summaries of ordinary functions. For thread functions, we do not require the computation of such summaries since we use *RacyPairs*(*DummyWrite*) for determining whether erase instructions for shared variables can be safely introduced in thread functions.

4.3 Algorithm and Implementation

4.3.1 Algorithm

Algorithm 2 shows the outline of our approach which we have implemented into the tool DEICS. The algorithm is divided into four major steps. For simplicity of presentation the algorith is given assuming that there is no recursion and all ordinary functions are inlined as explained in section 3 (the algorithm can be easiliy modified to avoid inlining the ordinary function and also to handle recursion by using function summaries as given in (30)).

Step 1: As mentioned in the approach, the first step is to treat each global variable as formal variable. We then identify definitions and aliases using fix-point computation. For each definition of local variables and formal variables, the set Reachability is computed considering aliases. The Reachability is then split into two sets *UsePoints* and *NoUse*. Note that the *UsePoints* set consisits of all the locations where the definition is actually used and also the locations which need to retain the definition for an actual use at later point in the program. Summaries of ordinary functions are used wherever they are

Algorithm 2: DEICS Implementation

```
Notation: $f$ - thread function or main function
          $id$ - unique identifier for a definition
          $\phi$ - empty set
for each $f$ do
    attach shared variable to formals set;
    /* So that the definition of shared variable can be treated as local to a function */
    for each definition $id$ do
1       Compute $Reachability(id)$;
        Spilt $Reachability(id)$ into $UsePoints(id)$ and $NoUse(id)$ ;
        Compute $ErasePoints(id)$ ;
        introduce $Erase(id)$ for all local variables;
for each definition $id$ of shared variable do
    for each $ErasePoints(id)$ do
2       introduce $DummyWrite(id)$ before;
for each $DummyWrite(id)$ do
3   Compute $RacyPairs(DummyWrite(id))$;
    /* set of pairs of locations identified using race-detection engine */
for each function $f$ do
    for each definition $id$ of shared variable do
        if $RacyPairs(DummyWrite(id)) = \phi$ then
4           introduce $Erase(id)$;
```

invoked in the program to cover the inter-procedural analysis. The *ErasePoints* set is computed for the locations where the definitions can be erased. All the local definitions can be erased at the appropriate *ErasePoints* if there is no global pointer p such that $*p$ aliases to the variable of definition.

Step 2: For each definition id of shared variable, or a definition whose alias is a shared variable, we introduce $DummyWrite(id)$ before each location in the set *ErasePoints*(id) computed in step 1.

Dummywrites are introduced first and actual erases are introduced based on our anlysis on the changed program with dummywrites.

Step 3: We compute *RacyPairs*(*DummyWrite*(id)) using the technique outlined in approach section 4.2. For each shared variable corresponding to definition id, pseudo-reads are inserted and race-detection engine is invoked to identify racy pseudo-reads and *RacyPairs*(*DummyWrite*(id)) is computed.

Step 4: We then transform the program by introducing erase instructions in place of those *DummyWrite*(id) whose corresponding set *RacyPairs*(*DummyWrite*(id)) is empty.

4.3.2 Soundness

Using the approach discussed so far, DEICS introduces erase statements for definitions of the shared variables in the program. Since our approach introduces erase statements only after considering the fact, that the erasure of a shared data has no effect on the other thread accesses, it is clear that the addition of an erase instruction does not change the functionality of the original program. With the help of the following terminology, we state and prove, that our tool DEICS does not change the semantics of a given program even after introducing erase statements.

Let P be a program generated by the grammar shown in Table XII. As indicated above we assume that P does not use recursion or pointer arithmetic. All lock variables are global variables and are accessed using the atomic operations $lock$, $unlock$.

We define a $thread\ state\ TS$ of a thread function f in P to be an $Address$ function that assigns addresses to the local variables of f. Let GS be the shared state consists of two functions, $Address$, and $Value$. $Address$ function maps shared variables to memory locations and $Value$ function maps all the memory locations to values. We define a $thread\ configuration\ TC$ of a thread function f to be a triple (GS, TS, i) where TS is a thread state of f, GS is the shared state and i is the $label$ of a thread command. Intuitively i denotes the next statement that will be executed in the thread f. For a thread configuration (GS, TS, i), where i is not an $exit$ statement, let $successor(GS, TS, i)$ be the unique thread configuration (GS', TS', i') where, TS', GS' are obtained by executing the statement i in the thread state TS and the shared state GS and $i' \in succ(i)$ is the next command of the thread function f to be executed in state TS'. If i is an $exit$ command, then there is no such $successor$ i.e., $successor(GS, TS, i)$ is undefined.

A $state$ S is a sequence $(GS, (TS_0, i_0), (TS_1, i_1), \ldots, (TS_n, i_n))$ where GS is a shared state and for $0 \leq j \leq n$, TS_j is a thread state of a thread j and i_j is a label of a statement in the thread j. Furthermore, The range of $Address$ functions in $GS, TS_0, TS_1, \ldots, TS_n$ are disjoint. TS_0 is a thread state representing the $main$ thread execution. Note that $(n+1)$ denotes the number of threads in the state S. An initial state is of the form $(GS_0, (TS_0, i_0))$ where TS_0 is the initial state of the $main$ thread, GS_0 specifies the initial values of addresses of global variables and values of memory locations and i_0 is the first statement in the $main$ thread.

Let $S = (GS, (TS_0, i_0), (TS_1, i_1), \ldots, (TS_n, i_n))$ be a state. Another state
$S' = (GS', (TS'_0, i'_0), (TS'_1, i'_1), \ldots, (TS'_m, i'_m))$ is called a $successor$ of S if S' is obtained by executing some thread j where $0 \leq j \leq n$, that is enabled in S. Thread j is enabled in S if the statement i_j is not a $lock$ statement or is a $lock$ statement such that the corresponding lock is available in state S. The successor state S_{j+1} of executing statement i_j of thread j is as follows -

- If i_j is not an $exit$ or a thread invocation, then the number of threads in S' and S are same. That is, $m = n$, and the thread configuration (GS', TS'_j, i'_j) is the $successor$ of (GS, TS_j, i_j), and $\forall k \neq j, TS'_k = TS_k$.
- If i_j is an $exit$ statement, then the number of threads in S' decreases by 1 and all the other thread states together with the shared state remain unchanged, i.e., $m = n - 1$, $GS' = GS$ and $\forall k < j$, $TS'_k = TS_k$, $i'_k = i_k$ and for $j < k \leq n, TS'_{k-1} = TS_k$, $i'_{k-1} = i_k$.
- If i_j is a thread invocation, then the number of threads in S' increases by 1, i.e., $m = n+1$. In this case, $TS'_j = TS_j$, $i'_j = succ(i_j)$. TS'_{n+1} is the initial thread state of the new thread created and i'_{n+1} is the first statement of this new thread. The range of $Address$ function of TS'_{n+1} is disjoint from the ranges of $Address$ functions of $GS', TS'_0, TS'_1, \ldots, TS'_n$ (i.e., $Address$ functions of other threads including shared states). $\forall k 0 \leq k \leq n$ and $k \neq j$, $TS'_k = TS_k$, $i'_k = i_k$. Also $GS' = GS$.

An execution is a sequence of states $S_0, S_1, \ldots, S_l, \ldots$, where S_0 is the initial state, and for each $i \geq 0$, S_{i+1} is a $successor$ of S_i. We say that the program is data-race free if it is not possible to reach a state S from an initial state such that no two threads access the same shared data at the same time in state S with one thread accessing it in the $write$ mode and another accessing it in the $read$ mode.

We assume that a shared variable never appears on the left hand side of any assignment statement (similar to base variables in section 3.3). As a consequence, in any execution, the address of shared variable does not change. Given a thread configuration (GS, TS, i), a memory location m is referenced in the configuration, if i is an evaluation of an expression e that has a variable y such that $TS.Address(y) = m$.

Consider an $execution$ $\rho = (S_0, S_1, \ldots, S_n)$ where, for $0 \leq i \leq n$, $S_i = (GS_i, (TS_{(0,i)}, j_{(0,i)}), (TS_{(1,i)}, j_{(1,i)}), \ldots, (TS_{(\ell_i,i)}, j_{(\ell_i,i)}))$. For each i, $0 \leq i < \ell_i$, let $f(i)$, $0 \leq f(i) \leq n_i$, be the thread that is executed in the step from S_i to S_{i+1}. For $i \geq 0$, we say that memory location pointed to by a shared variable sv is not needed after S_i in the execution ρ, if the current definition of the memory location pointed to by sv is not needed in every thread configuration of each thread that is executed after S_i in ρ; i.e., either $\forall k$, $i \leq k < n$, the memory location pointed to by sv is not referenced in the thread configuration $(GS_k, TS_{(f(k),k)}, j_{(f(k),k)})$ of the thread $f(k)$ or $\exists k$ $i \leq k < n$, such that the program statement $j_{(f(k),k)}$ of thread $f(k)$ is a $must\_definition$ of the memory location pointed to by sv and $\forall p$ $i \leq p < k$, the memory location pointed to by sv is not referenced in the the thread configuration $(GS_p, TS_{(f(p),p)}, j_{(f(p),p)})$ of thread $f(p)$.

Lemma 4.3.1. *Suppose that* DEICS *introduced an erase statement for a memory location pointed to by a shared variable* sv *after statement* l *in a thread function* f *of program* P*, then, in every execution* $\rho = (S_0, \ldots, S_n)$ *of* P *and for every* i*,* $0 < i < n$ *the following condition holds -*

if S_{i+1} *is the result of executing statement* l*, of a thread that is executing the thread function* $f()$ *in state* S_i*, then the memory location pointed to by variable* sv *is not needed after* S_i *in* ρ*.*

***Proof Sketch*.** Lemma 4.3.1 depends on the correctness of algorithms to compute *ErasePoints* for memory locations pointed by shared variables (along with the correctness of computation of *UsePoints NoUse*), and the robustness of race detection engine to identify potential accesses on a particular definition of a shared variable. It also follows from the fact that if a definition of a memory location pointed to by shared variable is erased at a node n of the CFG of a thread, then on all the paths of the CFG starting from node n, that particular definition of memory location pointed to by sv is never referenced and none of the accesses on $*sv$ in other threads do not depend on this definition of memory location pointed to by sv. $\square$

For any statement i, let $active(i)$ be exactly be the same as defined in sections 3.3. These are all the variables which point to memory locations whose definitions at statement i will likely be referenced once before they get redefined. Note that in an execution, if $sv \notin active(i)$ and every definition of memory location pointed to by sv, that is alive when control is at i, will never be referenced within the same thread that is being executed. However, it may be referenced in other threads.

Let P be the original program and P^T be the transformed program. Let $\rho = (S_0, S_1, \ldots, S_m)$ be an execution of P where, $\forall i, 0 \leq i \leq m, S_i = (GS_i, (TS_{(0,i)}, j_{(0,i)}), (TS_{(1,i)}, j_{(1,i)}), \ldots, (TS_{(p_i,i)}, j_{(p_i,i)}))$. For $0 \leq i < m$, let $f(i)$ be the thread executed in the step from S_i to S_{i+1}. Let $\rho' = (S'_0, S'_1, \ldots, S'_n)$ be an execution of P^T where, $\forall i, 0 \leq i \leq n, S'_i = (GS'_i, (TS'_{(0,i)}, j'_{(0,i)}), (TS'_{(1,i)}, j'_{(1,i)}), \ldots, (TS'_{(r_i,i)}, j'_{(r_i,i)}))$. For $0 \leq i < n$, let $g(i)$ be the thread executed in the step from S'_i to S'_{i+1}.

We say that an execution ρ' of the transformed program P^T *corresponds* to an an execution ρ of the original program P if the number of threads , the thread scheduling order, and the sequence of statements other than the new erase instructions executed within those threads are the same and the following condition also holds. If ρ' contains additional steps involving execution of the newly introduced erase instructions, then the definitions erased by these instructions are not needed any further in both the executions.

Formally, ρ' $corresponds$ to ρ if the following conditions are satisfied -

1. There exist integers $u_0 < u_1 < \ldots < u_m$ such that $u_0 = 0, u_m = n$ and $S_0 = S'_0$, $\forall l, 0 \leq l \leq m$, the number of threads in S_l is same as the number of threads in S'_{u_l}, i.e., $p_l = r_{u_l}$, and for each i, $0 \leq i \leq p_l$, the thread functions of the i^{th} threads in S_l and S'_{u_l} are the same, and $f(l) = g(u_l)$, and $j_{(f(l),l)} = j'_{(g(u_l),u_l)}$. And the following condition is also satisfied -
   - For each k, $u_l < k < u_{l+1}$, the statement $j'_{(g(k),k)}$ of thread $g(k)$ is an erase statement of a memory location pointed to by a shared variable sv introduced in P^T such that the memory location pointed to by sv is not needed after state S_l in ρ.
2. The interleaved sequence of input and output values (i.e., results of executing statements input and output statements) in ρ and ρ' are the same.

First condition above ensures that, excepting for the erase statements, the sequence of statements executed in ρ and ρ' are same. The second condition ensures that, the values of the memory locations read and output by both the executions (ρ and ρ') are same. Whenever the memory location pointed to by a shared variable is erased, that definition of the memory location pointed to by shared variable is no longer required by the program. Accordingly, the erased value has no effect on the values to be read by the program and the results generated. Following theorem states the soundness of our approach.

Theorem 4.3.2. *For every execution ρ of P there exists an execution ρ' of P^T such that ρ' corresponds to ρ and vice versa.*

Proof Sketch. The proof of this theorem is on the same lines as the proof of theorem 3.3.2 excepting that this proof uses lemma 4.3.1 instead of lemma 3.3.1.

□

The above theorem implies that the interleaved sequences of input outputs in ρ and ρ' are the same and therefore they are going to produce the same interaction with the environment. Hence both the programs P and P^T produce same results ensuring the correctness of our transformation.

4.3.3 Implementation

In our implementation, we assume that all global variables are shared variables, treated as additional formal variables to any function. This step is useful, as it would propagate the definition of a variable defined in one thread to an access in another thread. We perform a sequential analysis to compute Reachability sets for all the definitions within each function along with the definitions of shared variables in the form of formal variables. Following the method explain in section 4.2, we then compute *UsePoints* and *NoUse* points by splitting the reachability set. We compute *ErasePoints* for all definitions inside each function and erase all the local data that is not being pointed to by any of the global/shared variables.

Handling shared variables: In order to simplify our implementation, we treat each each global variable as a pointer type variable. If a shared variable is not of pointer type, then a temporary shared pointer variable can be introduced, and all occurrences of the global variable can be referenced through this temporary pointer.

The reason we make this transformation is to make use of existing concurrent program analysis infrastructures for our analysis of erasures. In particular, we use and build on the RADAR (35) framework for concurrent program analysis. RADAR is a data-flow analysis framework, which converts a sequential analysis into the one that is sound for concurrent programs. This framework has a built-in race-detection engine (RELAY), which identifies racy accesses on shared data.

For our implementation, we modified one of the instantiations of RADAR which performs the null-pointer dereference analysis. Since (through the above description), we have already translated all global accesses to pointer dereferences, we can now use the null-pointer dereference analysis of RADAR. However, a straightforward use of RADAR approach does not suffice here. This is because, in RADAR, the main focus is on $writes$ performed by all the threads. However, our analysis need to track $reads$ in all the threads, and therefore we modify the original RADAR framework to keep track of reads as described below. Additionally, our implementation also assesses the impact of the writes that we introduce for erasing data.

Dummy writes for potential erasures: We modified the RADAR framework to introduce dummy writes for shared data before the set *ErasePoints*(id). These dummy writes are treated as original writes during the analysis inside RADAR. We check for the emptiness of the set *RacyPairs*$($*DummyWrite*$(id))$ by using the null-pointer dereferencing analysis of the modified framework . If no null-pointer dereference warnings are generated on the modified program, then the set *RacyPairs*$($*DummyWrite*$(id))$ is empty and we safely introduce the erasure corresponding to id. Actually our modified RADAR framework introduces some pseudo-reads and employs the race-detection engine RELAY ((36)) to identify pseudo-reads that are in race with *DummyWrite*(id). Further, it checks if from any such racy pseudo-read an actual reference to the shared variable at another location is reachable in the CFG without any intermediate definition to the variable.

Reducing invocations of race-detection engine: Instead of introducing pseudo-reads at each program location, the program can be divided into race equivalence regions. A representative program location is chosen from each region to introduce pseudo reads. A race equivalence region is a region in the program where the raciness behaviour is same throughout the region. For the running example given in Figure 3, instead of introducing pseudo-reads at each location in the $server$ thread, it is sufficient to introduce pseudo-reads after lines $5, 6$ and 11. For each definition of shared variable inside a function, after identifying a representative location for each race equivalence region, a pseudo read is introduced for that definition using the modified RADAR framework. Once all functions are populated with dummy writes and pseudo-reads, a race-detection engine is invoked to identify possible races between these accesses. As described above, we then check for the emptiness of $\mathit{RacyPairs}(\mathit{DummyWrite}(id))$,by checking for potential null-pointer dereference warnings.

Analysing warnings: Not all null-pointer dereference warnings reported by RADAR are critical. Since our main goal is to identify parallel reads, races because of actual writes can be filtered. For example, warnings based on writes, i.e., dereference to the shared variable occurring on left hand side of an assignment instruction, can be safely ignored. Only those warnings due to reads, i.e., dereferences to the shared variables occurring in places other than left hand side of the assignment instruction are important and are considered. Our modifications to the RADAR framework implement this filtering of warnings.

Aliases to shared variables: Our analysis also requires consideration of alias information in identifying dereferences. RADAR's original null-pointer dereference implementation does not consider aliases of a global pointer as its requirement is only to identify null-pointer dereference warnings. For our application, any access, even if it is through aliases, should be captured. Therefore in our modified RADAR framework, we added this step of including aliases for null-dereference warnings.

```
1   char *data;
2   mutex lockvr;
3   int T1(){
4   char * localAlias ;
5    lock(&lockvr);
6     ...
7     writeinto (data);
8     localAlias  = data;
9    unlock(&lockvr);
10    ...
11   // other task
12   lock(&lockvr);
13    ...
14   readDatain( localAlias );
15   unlock(&lockvr);
16  }

17   ...
18  int T2(){
19    ...
20   lock(&lockvr);
21    ...
22   readDatain(data);
23   unlock(&lockvr);
24    ...
25  }
```

Figure 17: Alias to shared data

Consider the example given in Figure 17. Thread $T1$ has a local variable $localAlias$, which becomes an alias to the memory pointed to by the shared variable $data$. Original null-pointer dereference analysis does not consider access of the shared data at line 14 and does not throw any warning if we introduce a write before the line 23 in thread $T2$. In our implementation we track all the accesses to the shared data through aliases as well.

We give a detailed evaluation of this implementation by transforming various concurrent applications in next section.

4.4 Evaluation

We implemented our tool DEICS in Ocaml language (37) using CIL (27) and RADAR frameworks. CIL is a front-end for the C programming language that facilitates program analysis and transformation. CIL parses and type checks a program, and compiles it into a simplified subset of C which reduces number of cases to be considered to manipulate a C program. Along with the basic sequential analysis to compute *ErasePoints* for each definition in the program, we added and modified the source of RADAR to suit the requirements of our approach. We evaluated our implementation with four goals in mind: 1. Correctness, 2. Scalability, 3. Effectiveness, and 4. Performance.

Correctness: We developed our own test harness consisting of various cases (around 15 test cases) of concurrenct programming scenarios to check the correctness of our transformation. We tried to cover different patterns of usage of shared data with concurrency constructs in these test cases. Our tool transformed all the programs in the test suite by placing erase instructions for the shared data at the intended locations after which that particular definition of shared data is no longer required. We manually verified the correctness of our approach by checking each shared data erasure in each program in the test suit and executing the same to test the functional correctness. This approach of checking for the correctness manually is far from perfect, however it provides us a starting point to verify our transformation on larger applications. This experiment with a test harness provided a high degree of confidence that our tool does not change the program's behaviour and minimizes the lifetime of data.

$Pfscan$	Pfscan (38) is a parallel file scanner utility from pftools set. It uses multiple worker threads to search through directories in parallel. This application combines the functionality of find, xargs, and grep.
$Knot$	Knot (39) is a small multithreaded web server distributed with the Capriccio threads package.
$Zebedee$	Zebedee (40) is a simple multithreaded application to establish an encrypted, compressed tunnel for TCP/IP or UDP data transfer between two systems. It allows traffic such as telnet, ftp and X to be protected from snooping as well as potentially gaining performance over low-bandwidth networks from compression.
$Mtdaapd$	Mtdaapd (41) is an open-source audio media server (or daemon) which serves media files to users. It is developed for POSIX systems.
$Retawq$	Retawq (42) is a small interactive, multi-threaded text based web browser used for text terminals on computers with Unix-like operating systems.

TABLE XIII: Applications transformed by DEICS

4.4.1 Applications and Scalability

Using our tool, we transformed six multithreaded applications written in C. All these applications use Pthreads library for the multithreading features. The applications we chose handle sensitive data such as ftp passwords and database records. Some of the applications we used in our experiments are taken from RADAR benchmarks (43). Table XIII gives a brief description for each application we have transformed.

We merged each application into a single source file which for convenience. We used the CIL merger utility for this. CIL merger is a tool that combines all of the C source files in a project into a single C file. The merged file also includes additional preprocessed code.

Application	**Size (LOC)**	**no.of functions**	**transformation time(sec)**
pfscan	1259	24	28
knot	2255	56	39
zebedee	11682	220	200
mtdaapd	57102	637	12451
retawq	38750	638	12260

TABLE XIV: application size and transformation time taken

We chose concurrent applications of various sizes to check if DEICS can scale to transform larger application. The largest application consists of 57K lines of code (LOC). Column 2 in Table XIV shows how DEICS scales well to transform applications from 1K LOC to 57K LOC, totaling more than 100K LOC. Table XIV also shows the total number of functions in each application (column 3). The transformation time taken for each application is shown in column 4. For initial erase point computation using SWIPE, we observed a correlation between the number of functions and transformation time. However, for RADAR, there is a correlation between number of strongly connected components (SCC) computed by RADAR and transformation time. RADAR takes more time for applications with larger SCCs. For instance, retawq has more SCCs and RADAR spends more time for analysis. The transformation time includes the time taken by race-detection engine as well.

4.4.2 Effectiveness

Table XV shows the effect our transformation. We use RSM (28) tool to measure the application codebase sizes. For each application (column 1), we identified the number of global variables (excluding pthread mutex variables) which is shown is column 2 of the Table XV. Minimum number of threads in each application is also shown (column 3). The effect of our approach is given in column 4 as the number of erases introduced for globals. For a given global, there can be more than definition and for each definition there can be more than one erase point as the size of *ErasePoints* set can be greater than one. For example, in $zebedee$ application, there are only 61 globals, but number of erasures are 928. We observed that, there is a switch case in the program with different cases and the globals are getting erased in each case of the switch statement. Furthermore, the simplified program (by CIL) contains additional temporary variables and our transformation introduces erasures for each potential sensitive definition, which could be a copy of data from global variable. Also, DEICS introduces erases for globals before all termination points in the program, covering all possible paths an execution can take.

We also evaluated the effectiveness of our tool to check if the erasures are introduced for sensitive information in shared global variables. For the application *pfscan*, DEICS erased local copies of pathname, location and position of the search string in files (i.e. result of the execution stored in those variable). DEICS also erased two global variables (of *pfscan* application) which contain the path information and error messages thus minimizing their exposure.

Application	**no. of globals**	**no. of threads**	**no. of erasures for globals**
pfscan	18	2	11
knot	43	6	10
zebedee	61	3	928
mtdaapd	326	5	176
retawq	444	2	342

TABLE XV: Effect of transformation on Applications

After the transformation, for application *knot*, DEICS introduced erases for shared variables which are used to store the statistics information (which is also being updated by threads). The statistics information consists of the connection information, the number of clients, and the amount of data transfer between server and clients. The main thread resets this information every time for client threads to update. DEICS introduced erases for these shared variables before the main threads resets them, treating the reset instructions as new definition. Although the erasure instructions do not save any extra lifetime of the data, but they show the precision of our approach in identifying erase locations thus, minimizing data lifetime.

For the application *zebedee*, DEICS erased copies of token information and also counter data. Erase instructions are inserted in the code for many of the global variables. Although, most of the global variables are not protected by mutexes, they were set once and used by all the threads. DEICS introduced erasures for such data at potential program exit points; hence minimizing the the data exposure that may happen even after the program termination.

In the text-based browser application *retawq*, most of the global data has already been erased by the developer. For the sensitive data like, *FTP_login_password* and *current_keymap_keystr*, the application has erase instructions. DEICS also introduced erases at the same location. This clearly shows that, in the absence of such erase instructions in an application, DEICS minimizes the data exposure by introducing erases automatically.

The audio media server application *mtdaapd*, uses the database to store the music information which is retrieved by the users connected to the server. DEICS introduced erasures for the global shared variables, which contain sensitive information. For example, *db_songs* and *db_removed* variables in the program contain the database header and status information and DEICS has placed erases in the code for these variables. Also, individual datum is erased by DEICS after their last usage in the application (as part of local erasures). We observed that, the erase instructions of DEICS are placed after the library call to *gdbm_delete*. Similar to C library function *free*, such built-in functions may not actually erase, but simply deallocate the memory. If we instruct the tool to ignore such library calls, DEICS will erase the data immediately after its lifetime, i.e. much before such library calls.

After analyzing each application, it is clear that, most of the applications do handle sensitive data in shared variables and erasure of such data is important. Our tool DEICS effectively introduces erasures for some of the shared sensitive data as explained above.

4.4.3 Performance

We measured the performance overhead of transformed applications. Figure 18 shows the overhead of each application. We ran our experiments on x86 Linux platform with configuration of 8 GB of memory and a 3 GHz AMD Phenom processor. For each application, we used minimum set of threads

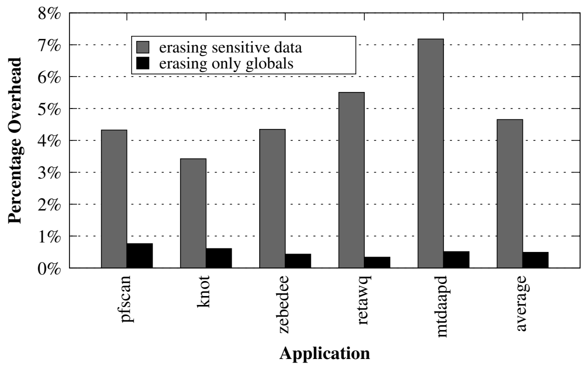


Figure 18: Performance overhead of concurrent applications

during the execution of original and transformed program. The runtime overhead (gray region in each bar) is less than 8% and average is around 5%. This overhead also includes overhead caused due to the erase instructions for sensitive data in local variables of individual functions.

Another set of experiments are conducted to measure the overhead due to erasing only shared/global data. The aim of this experiment is to verify that the performance overhead to erase only shared data is minimal (less than 1%). The runtime overhead is ranged from 0.3% to 0.7% averaging 0.5%. Figure 18 also shows the overhead only due to erasing global variables (the black region in each bar). We observed that transforming applications with a tool like SWIPE would also have the overhead of around 5%. Our transformation to erase shared data adds only an additional performance overhead of 0.5% to minimize the lifetime of shared data.

4.5 Summary

In this chapter, we presented our approach of data lifetime minimization in concurrent applications. We presented our implementation of the tool called DEICS for reducing lifetime of shared variables that are accessed by multiple threads concurrently. We employ static analysis techniques for tracking shared data and identifying potential erase points, and transform the program with erase instructions that are confirmed safe by a race detection engine. DEICS effectively transformed a set of real world concurrent applications that handle sensitive data in shared variables and minimized their lifetimes.

CHAPTER 5

RELATED WORK

5.1 Data Remanence

The broad area of *data remanence* in systems development has been devoted to the problem of handling residual data in computer systems. This has been an important concern in secure operating system development and of MLS systems. Gutmann (7; 8) surveyed earlier work in this area, and specifically looked at the problem of secure deletion of information from magnetic disks (7) and semiconductor devices (8). Work from Gutmann (7; 8) is considered as the first work addressing the issue of residual data from memory. Our work of erasing data from legacy programs is complementary to these works which analyze issues related to erasing at the hardware level.

There have been various works in the area of data lifetime minimization by employing different techniques at various layers of memory where data can reside (9; 10; 11; 12). In a whole system analysis performed by Chow et.al., several concurrent applications like, browsers and webservers, leave the sensitive data in the memory for a long time. In their subsequent work, Chow et al., (12), came up with an approach to reduce the lifetime of sensitive data. They changed the free routines to have erasing mechanism. However, a developer may not always insert free instructions in the program, thus leaving the scope for data to reside in the memory for a long time. Chow et.al., also proposed erasing the stack immediately after the function is returned, this minimizing the function return windows. However, their approach to reducing lifetimes by erasing the stack does not work for concurrent applications, where shared variables often carry sensitive data.

5.2 Garbage Collection Approaches

Our approach for reducing sensitive data-lifetimes is related to approaches for garbage collection. Garbage collection techniques have a long history of research, as surveyed by Paul Wilson (44). A key difference between our approach and garbage collection is that our approach uses a tight, dynamic criterion for erasing sensitive data . As noted in a detailed survey on garbage collection (45), garbage collectors use more relaxed criteria. Indeed, a tracing garbage collector (such as the Boehm-Demers-Weiser (46; 9) garbage collector) for the C language, relies on reachability of an object from a *root* set of objects. We could augment such a garbage collector with memory erasing routines to ensure that freed objects are erased in memory. This benefit of this approach is that it requires no application instrumentation. Free-me (13) aims to insert deallocation instructions by conservatively estimating object lifetimes and they indeed propose change to deallocation routines to have erasing mechanism as well. However, such a solution may still be imprecise in addressing our goal, namely to erase contents of sensitive memory *immediately* after their lifetime. By calling the garbage collector more often, this gap can be narrowed; nevertheless this frequent calling of the garbage collector has the problem of introducing overheads that are unpredictable.One can, in theory, invoke the garbage collector quite frequently to erase sensitive data but that would lead to very high performance overheads (13; 14).

5.3 Region-Based Memory Management

An extensive work in the area of region-based memory management has been performed (47; 48; 49; 50; 51; 52) . The main goal of these works is to have an economic usage of memory and reduce the need for invoking garbage collector. Lifetime of objects is computed to precisely decide which regions in the memory can be allocated to those object and if any of the allocated regions can be reused. Using these approaches, the memory allocation and deallocation can be done in a constant time. Although the analysis techniques used in these approaches are similar to our approach, our goal is different from theirs. The analysis we developed is specific to lifetime minimization of data residing in the memory, whereas, authors of (47; 48; 49; 50; 51; 52), mainly focus on allocation and deallocation of memory based on object lifetime. Additionally, larger regions in systems using region based memory management, would lead to extended windows of exposure for large portion of data whose lifetimes are much smaller than the lifetime of the memory region in the program. To minimize these windows, a program restructuring is required (15). Our approach, on the other hand erases data as soon as its requirement in the program is finished.

5.4 Information Erasure Policies

The property of secure data deletion has been studied formally by Chong and Myers (53). In (54), the authors extend the Jif programming language with information erasure policies. In (53; 54), a developer explicitly specifies conditions under which data has to be erased whereas our approach automatically infers points of erasure through static analysis. The proposed framework in (53; 54) is suitable for developing new applications. In contrast, our approach can retrofit legacy applications with erase instructions. Additionally, Jif being a type safe programming language automatically eliminates some of the reasons for sensitive data leakage through memory safety attacks. In contrast, our approach is targeted towards C, a weakly typed language, that is almost exclusively used for operating systems code where concerns about sensitive data lifetime have greater significance.

5.5 Memory Safety of C Programs

Another closely related line of work is memory safety of C programs. Most of these approaches have been based on program analysis and transformation. Approaches for memory safety for C programs (23) have contributed to research in the area but these approaches mostly rely on garbage collection for recollecting memory which is inherently unsuitable for our purpose. *Safe-C* (55) makes use of runtime checks to detect all memory errors. However, this approach introduces compatibility problems due to the use of fat pointers. Kelly (22) introduced techniques that stressed the importance of backwards compatibility in retrofitting programs.

Several program analysis and transformation techniques have been devised to identify errors in C programs (56; 57; 58; 59; 33; 22; 55; 23). All these works focus on identifying bugs in the program. Whereas our work aims at minimizing lifetime by transforming programs to have data erasure mechanism. Work on managing memory to prevent security attacks is also related to our approach. Cling (10) is a memory allocator that aims to prevent use-after-free vulnerabilities by making use of address spaces. Automatic pool allocation (11) segregates heap memory into different pools for improving performance. However the main goal of lifetime minimization to ideal data lifetimes is not achieved with these approaches.

5.6 Preventing Unauthorized Access

There have been a number of works in operating systems to prevent unauthorized access or disclosure of sensitive data. Work related to DFIC operating systems (60; 61), Data Sandboxing (62), aim to minimize exposure of sensitive data by mitigating the effects of privilege escalation attacks on operating systems. Authors of (62) proposed a technique to partition the code so that the sensitive operations are executed in a sandboxed environment. TrustVisor (63) additionally provides data secrecy assurances for sensitive data by use of a micro-trusted platform module. Our work is focused on legacy operating systems implementation and the secure data erasure principles analyzed in this thesis further minimize the risks in these systems.

5.7 Static Analysis of Concurrent Applications

There are tools and frameworks developed to perform data-flow analysis on concurrent applications (64; 35). In (65; 66; 32; 64), a graph to represent the parallelism is built and a modified version of sequential analysis is performed. (67; 68) provide a generic approach for static analysis of concurrent programs. Qadeer et. al. proposed a technique to transform concurrent programs to sequential programs (69) for finding errors in concurrent programs. All these works mainly focus on identifying bugs in programs. Our objective of minimizing sensitive data lifetimes is different from all of the above works.

This chapter compared contributions of this thesis with the related works broadly in the area of securing the data. Discussion about the future work based on this thesis is given in next chapter.

CHAPTER 6

CONCLUSION AND FUTURE WORK

The data lifetime reduction problem is a pressing issue for most security critical real world applications written in C. In this dissertation, we presented SWIPE, an automated approach and tool for reducing lifetime of sensitive variables in sequential applications. Our approach and tool employs static analysis for tracking information and automatically transforms the program with instructions that erase all the sensitive data after intended use. The effectiveness of SWIPE was demonstrated through a comprehensive evaluation over a set of real world C programs that handle sensitive information such as passwords and keys. SWIPE provides support for programmers and system administrators to retrofit software in order to minimize the memory footprint of sensitive information.

We also presented an approach to minimize lifetime of data in concurrent applications. Our approach is implemented as a tool called DEICS, which automatically transforms concurrent programs with instructions to erase data after its intended use. Our tool is based on static analysis to minimize runtime overhead. We presented a novel algorithm to reduce sensitive data lifetimes and discussed its implementation. We have evaluated a set of real world concurrent applications written in C to show its effectiveness.

Together SWIPE and DEICS provide a solution to retrofit C programs with the security feature of minimized data lifetime. With our experiences from this thesis, we have identified potential future directions of this research.

Making SWIPE and DEICS More Precise: The immediate future direction of our work is to extend our tools to be more precise. Our implementations do not handle individual data elements in complex and composite data structures (arrays, linked lists etc.). Using approached like shape analysis (26) to reason the behaviour of individual data elements and minimize their lifetimes would become the first extension.

Erasing All Copies of Data: Another major future direction is to identify and erase the data at all the storage places, not just memory. This requires a clear understanding of how the data propagates in the program, in the RAM, and in the secondary storage, i.e., the data erasure should happen at all possible these layers. Our approach does not consider erasing data from the secondary storage (for example, propagated to secondary storage during paging) which is also equally important concern. In general, it is difficult to identify propagation of a particular sensitive data in the secondary storage. One simple approach is to erase each page from the secondary storage right before it is paging in. Recent work (70) has addressed this problem in part. Still, a whole system approach may be needed to estimate when the data is no longer required. Our tools SWIPE and DEICS provide an application level approach to minimize lifetime of data. One can envision a similar approach that can scale to entire system. Infrastructural limitations prevented us from deploying our tools over the entire linux kernel, however in theory our approach could become a whole system approach.

Minimizing the Locality of Sensitive Data: In general, application developers do not consider the 'locality of use' when handling the sensitive data in the programs. There might still be exposure regions if the use of a variable appears much after it has been defined. Considering program transformation techniques to reduce these regions of data exposure would become another potential direction to our research. The code in the program can be moved inside so that all the sensitive data handling (i.e., definitions and their use) is limited to smaller regions. This analysis can be considered as an application of code motion technique. Once the sensitive data handling is restricted to smaller regions, its exposure can be reduced further along with helping developers to enforce any policies on the sensitive data (with this, developers need not worry about identifying all the places where a particular sensitive data is being handled and apply policies on the same).

Using SWIPE for Solving Privilege Escalation Problems: The main theme of this thesis is to retrofit legacy applications with erase mechanism to minimize data lifetime. Our approach of data life time minimization can be used to introduce other security features in applications. For example, one such problem is to reduce privilege exposure in programs. Each instruction of the program can be tagged with the privilege level that is being applied. Using our analysis of exposure window, one can identify the privilege exposure in the program. And using code motion techniques these exposure windows can be minimized.

Data Lifetime Minimization in Mobile Environment: Applying our approach to minimize data lifetime in mobile applications would be another dimension of future work. With the recent trend rapidly shifting towards the usage of smart-phones, the security of the data being used in mobile applications becomes critical. According to a SANS (71) survey, 68% of companies are creating mobile version of web applications and another 32% are developing internal mobile applications for business units. Software vulnerabilities in mobile applications that access sensitive data and transact business critical operations are a significant concern. Work along the lines of mobile application security to investigate the challenges in mobile environment would be an useful extension. Extending techniques of our approach to programming languages used in mobile environment (Java byte code, ObjectiveC, etc.), we will be able to minimize the lifetime of data in mobile applications.

CITED LITERATURE

1. Guttman, P.: Software Leaves Encryption Keys, Passwords Lying around in Memory. Security Focus Vuln Dev Mailing List, 2002.

2. Chow, J., Pfaff, B., Garfinkel, T., Christopher, K., and Rosenblum, M.: Understanding Data Lifetime via Whole System Simulation. In USENIX Security Symposium, San Diego, CA, 2004.

3. Parampalli, C., Sekar, R., and Johnson, R.: A practical mimicry attack against powerful system-call monitors. In ASIACCS, pages 156–167, 2008.

4. Common vulnerability exposures. https://cve.mitre.org/.

5. Halderman, J. A., Schoen, S. D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J. A., Feldman, A. J., Appelbaum, J., and Felten, E. W.: Lest We Remember: Cold Boot Attacks on Encryption Keys. In Usenix Security Symposium, San Jose, CA, 2008.

6. Broadwell, P., Harren, M., and Sastry, N.: Scrash: A System for Generating Secure Crash Information. In USENIX Security Symposium, Washington, DC, 2003.

7. Gutmann, P.: Secure Deletion of Data from Magnetic and Solid-state Memory. In USENIX Security Symposium, San Jose, California, 1996.

8. Gutmann, P.: Data Remanence in Semiconductor Devices. In USENIX Security Symposium, Washington, DC, 2001.

9. Boehm, H.-J.: A Garbage Collector for C and C++. http://www.hpl.hp.com/personal/Hans_Boehm/gc, 2002.

10. Akritidis, P.: Cling: A Memory Allocator to Mitigate Dangling Pointers. In USENIX Security Symposium, Washington, DC, 2010.

11. Lattner, C. and Adve, V.: Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. In Programming Language Design and Implementation, Chicago, IL, 2005.

12. Chow, J., Pfaff, B., Garfinkel, T., and Rosenblum, M.: Shredding Your Garbage: Reducing Data Lifetime through Secure Deallocation. In USENIX Security Symposium, Baltimore, MD, 2005.

13. Guyer, S. Z., McKinley, K. S., and Frampton, D.: Free-Me: A Static Analysis for Automatic Individual Object Reclamation. In Programming Language Design and Implementation, Ottawa, Ontario, Canada, 2006.

14. Cherem, S. and Rugina, R.: Uniqueness Inference for Compile-time Object Deallocation. In International Symposium on Memory Management, Montreal, Quebec, Canada, 2007.

15. Region Based Memory Management. http://en.wikipedia.org/wiki/Region-based_memory_management.

16. Xu, W., Bhatkar, S., and Sekar, R.: Taint-enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In USENIX Security Symposium, Vancouver, B.C., Canada, 2006.

17. Static Single Assignment form. http://en.wikipedia.org/wiki/Static_single_assignment_form.

18. Zheng, X. and Rugina, R.: Demand-driven Alias Analysis for C. In Principles of Programming Languages, San Francisco, CA, 2008.

19. Avots, D., Dalton, M., Livshits, V. B., and Lam, M. S.: Improving Software Security with a C Pointer Analysis. In International conference on Software engineering, St. Louis, MO, 2005.

20. Steensgaard, B.: Points-to Analysis in Almost Linear Time. In Principles of Programming Languages, St. Petersburg Beach, FL, 1996.

21. Andersenm, L. O.: Program Analysis and Specialization for the C Programming Language. Technical report, 1994.

22. Jones, R. W. M., H J Kelly, P., and Most C, and Uncaught Errors: Backwards-compatible Bounds Checking for Arrays and Pointers in C Programs. In HP Labs Tech Report, 1997.

23. Necula, G. C., McPeak, S., and Weimer, W.: CCured: Type-safe Retrofitting of Legacy Code. In Principles of Programming Languages, Portland, OR, 2002.

24. Aiken, A., Bugrara, S., Dillig, I., Dillig, T., Hackett, B., and Hawkins, P.: An Overview of the Saturn Project. In Program Analysis for Software Tools and Engineering, San Diego, CA, 2007.

25. Nystrom, E. M., Kim, H.-S., and Hwu, W.-M. W.: Bottom-Up and Top-Down Context-Sensitive Summary-Based Pointer Analysis. In Static Analysis Symposium, Verona, Italy, 2004.

26. Sagiv, M., Reps, T., and Wilhelm, R.: Parametric Shape Analysis via 3-valued Logic. In Principles of Programming Languages, San Antonio, TX, 1999.

27. Necula, G. C., McPeak, S., Rahul, S. P., and Weimer, W.: CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In Conference on Compiler Construction, Grenoble, France, 2002.

28. Resouce Standard Metrics. http://msquaredtechnologies.com.

29. StackOverflow. http://stackoverflow.com/questions.

30. Gondi, K., Bisht, P., Venkatachari, P., Sistla, A. P., and Venkatakrishnan, V. N.: Swipe: eager erasure of sensitive data in large scale systems software. In Proceedings of the second ACM conference on Data and Application Security and Privacy, CODASPY '12, pages 295–306, New York, NY, USA, 2012. ACM.

31. Netzer, R. H. B. and Miller, B. P.: What are race conditions?: Some issues and formalizations. ACM Lett. Program. Lang. Syst., 1(1):74–88, March 1992.

32. Sinha, N. and Wang, C.: Staged concurrent program analysis. In Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering, FSE '10, pages 47–56, New York, NY, USA, 2010. ACM.

33. Rugina, R. and Rinard, M.: Symbolic Bounds Analysis of Pointers, Array Indices, and Accessed Memory Regions. In Programming Language Design and Implementation, Vancouver, British Columbia, Canada, 2000.

34. Lamport, L.: How to make a correct multiprocess program execute correctly on a multiprocessor. IEEE Trans. Comput., 46(7):779–782, July 1997.

35. Chugh, R., Voung, J. W., Jhala, R., and Lerner, S.: Dataflow analysis for concurrent programs using datarace detection. In Proceedings of the 2008 ACM SIGPLAN conference

on Programming language design and implementation, PLDI '08, pages 316–326, New York, NY, USA, 2008. ACM.

36. Voung, J. W., Jhala, R., and Lerner, S.: Relay: static race detection on millions of lines of code. In Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, ESEC-FSE '07, pages 205–214, New York, NY, USA, 2007. ACM.

37. Ocaml. http://ocaml.org/.

38. Pfscan. http://freecode.com/projects/pfscan.

39. Behren, R. v., Condit, J., Zhou, F., McCloskey, B., Brewer, E., and Necula, G.: Knot. http://capriccio.cs.berkeley.edu/.

40. Zebedee. http://www.winton.org.uk/zebedee/index.html.

41. Mtdaapd. http://sourceforge.net/projects/mt-daapd/.

42. Thomaen, A.: Retawq. http://retawq.sourceforge.net/.

43. Radar. http://cseweb.ucsd.edu/~lerner/radar.html.

44. Wilson, P. R., Lam, M. S., and Moher, T. G.: Caching considerations for generational garbage collection. SIGPLAN Lisp Pointers, V(1):32–42, 1992.

45. Jones, R.: Garbage Collection: Algorithms for Automatic Dynamic Memory Management. John Wiley and Sons, July 1996. With a chapter on Distributed Garbage Collection by Rafael Lins. Reprinted 1997 (twice), 1999, 2000.

46. Boehm, H.-J. and Weiser, M.: Garbage Collection in an Uncooperative Environment. 18(9):807–820, 1988.

47. Ruggieri, C. and Murtagh, T. P.: Lifetime analysis of dynamically allocated objects. In Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '88, pages 285–293, New York, NY, USA, 1988. ACM.

48. Hanson, D. R.: Fast allocation and deallocation of memory based on object lifetimes. Software: Practice and Experience, 20(1):5–12, 1990.

49. Aiken, A., Fahndrich, M., and Levien, R.: Better static memory management: improving region-based analysis of higher-order languages. In Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation, New York, NY, USA.

50. Hallenberg, N., Elsman, M., and Tofte, M.: Combining region inference and garbage collection. In Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, PLDI '02, pages 141–152, New York, NY, USA, 2002. ACM.

51. Birkedal, L., Tofte, M., and Vejlstrup, M.: From region inference to von neumann machines via region representation inference. In Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '96, pages 171–183, New York, NY, USA, 1996. ACM.

52. Tofte, M. and Talpin, J.-P.: Implementation of the typed call-by-value &#955;-calculus using a stack of regions. In Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '94, pages 188–201, New York, NY, USA, 1994. ACM.

53. Chong, S. and Myers, A. C.: Language-Based Information Erasure. In Computer Security Foundations Workshop, Aix-en-Provence, France, 2005.

54. Chong, S. and Myers, A. C.: End-to-End Enforcement of Erasure and Declassification. In Computer Security Foundations Symposium, Pittsburgh, PA, 2008.

55. Austin, T. M., Breach, S. E., and Sohi, G. S.: Efficient Detection of All Pointer and Array Access Errors. In Programming Language Design and Implementation, Orlando, FL, 1994.

56. Dor, N., Rodeh, M., and Sagiv, M.: CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C. In Programming Language Design and Implementation, San Diego, CA, 2003.

57. Ganapathy, V., Jha, S., Chandler, D., Melski, D., and Vitek, D.: Buffer Overrun Detection using Linear Programming and Static Analysis. In Computer and Communications Security, Washington D.C., 2003.

58. Xie, Y., Chou, A., and Engler, D.: ARCHER: Using Symbolic, Path-sensitive Analysis to Detect Memory Access Errors. In European Software Engineering Conference, Helsinki, Finland, 2003.

59. Larochelle, D. and Evans, D.: Statically Detecting Likely Buffer Overflow Vulnerabilities. In USENIX Security Symposium, Washington, D.C., 2001.

60. Krohn, M., Yip, A., Brodsky, M., Cliffer, N., Kaashoek, M. F., Kohler, E., and Morris, R.: Information Flow Control for Standard OS Abstractions. In Symposium on Operating Systems Principles, Washington, WA, 2007.

61. Zeldovich, N., Boyd-Wickizer, S., Kohler, E., and Mazières, D.: Making Information Flow Explicit in HiStar. In Symposium on Operating Systems Design and Implementation, Seattle, WA, 2006.

62. Khatiwala, T., Swaminathan, R., and Venkatakrishnan, V.: Data Sandboxing: A Technique for Enforcing Confidentiality Policies. In Annual Computer Security Applications Conference, Miami Beach, FL, 2006.

63. McCune, J. M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V., and Perrig, A.: TrustVisor: Efficient TCB Reduction and Attestation. In IEEE Symposium on Security and Privacy, Oakland, CA, 2010.

64. De, A., D'Souza, D., and Nasre, R.: Dataflow analysis for datarace-free programs. In Proceedings of the 20th European conference on Programming languages and systems: part of the joint European conferences on theory and practice of software, ESOP'11/ETAPS'11, pages 196–215, Berlin, Heidelberg, 2011. Springer-Verlag.

65. Lee, J., Padua, D. A., and Midkiff, S. P.: Basic compiler algorithms for parallel programs. In Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming, PPoPP '99, pages 1–12, New York, NY, USA, 1999. ACM.

66. Dwyer, M. B. and Clarke, L. A.: Data flow analysis for verifying properties of concurrent programs. In Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering, SIGSOFT '94, pages 62–75, New York, NY, USA, 1994. ACM.

67. Duesterwald, E. and Soffa, M. L.: Concurrency analysis in the presence of procedures using a dataflow framework. In Proceedings of the symposium on Testing, analysis, and verification, TAV4, pages 36–48, New York, NY, USA, 1991. ACM.

68. Bouajjani, A., Esparza, J., and Touili, T.: A generic approach to the static analysis of concurrent programs with procedures. In Proceedings of the 30th ACM SIGPLAN-SIGACT symposium

on Principles of programming languages, POPL '03, pages 62–73, New York, NY, USA, 2003. ACM.

69. Qadeer, S. and Wu, D.: Kiss: keep it simple and sequential. SIGPLAN Not., 39(6):14–24, June 2004.

70. Onarlioglu, K., Mulliner, C., Robertson, W., and Kirda, E.: PRIVEXEC: Private Execution as an Operating System Service. http://seclab.ccs.neu.edu/static/publications/sp2013privexec.pdf.

71. SANS institure. http://www.sans.org.

72. citi bank data breach. http://news.softpedia.com/news/Citi-Exposes-Details-of-150-000-Individuals-Who-Went-into-Bankruptcy-369979.shtml.

73. Musuvathi, M. and Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, PLDI '07, pages 446–455, New York, NY, USA, 2007. ACM.

74. Grunwald, D. and Srinivasan, H.: Data flow equations for explicitly parallel programs. In Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming, PPOPP '93, pages 159–168, New York, NY, USA, 1993. ACM.

VITA

811 S Lytle St, Apt 312, Chicago, Illinois 60607

KALPANA GONDI

☎ 312-504-1509 ✉ kgondi2@uic.edu

www.cs.uic.edu/∼kgondi

EDUCATION

Doctor of Philosophy, CS	University of Illinois at Chicago	**CPA:** 4.0/10.0
	Aug 2006 - Dec 2013 (expected)	
Master of Technology, CS	University of Hyderabad, Hyderabad	**CPA:** 9.42/10.0
	2002 - 2004	
Bachelor of Technology, CS	SSCIT&EW, JNTU, Hyderabad	**CPA:** 82.5/10.0
	1998 - 2002	

PUBLICATIONS

REFEREED CONFERENCE PAPERS

1. **Monitoring the Full range of Omega-regular properties of Stochastic Systems**. Kalpana Gondi, Yogesh Kumar Patel and Aravinda P. Sistla. In *VMCAI '09: Proceedings of the 10th International conference on verification, Model Checking and Abstract Interpretation (VMCAI 2009), Savannah, Georgia, 2009*, Savannah, Georgia, USA, 2009.

2. **Swipe: Eager Erasure of Sensitive Data in Large Scale Systems Software**. Kalpana Gondi, Praveen Venkatachari, Prithvi Bisht, Aravinda P. Sistla and V.N. Venkatakrishnan. In *CODASPY'12:*

Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy, San Antonio, TX, USA, Feb 2012, **Acceptance Rate = 21 / 113, 18.5%**.

INVITED PAPERS

3. **WebAppArmor: A Framework for Preventing Web-based Attacks**. V.N. Venkatakrishnan, Prithvi Bisht, Mike Ter Louw, Michelle Zhou, Kalpana Gondi and K. T. Ganesh. In *ICISS'10: Proceedings of the 6th International Conference on Information Systems Security*, Gandhinagar, Gujarat, India, 2010.

PRESENTATIONS

- Data lifetime minimization in concurrent applications.
  - ✦ Work-in-progress presentation, Midwest Verification Day, UIC, Chicago, Sep 2013
- SWIPE: Eager erasure of sensitive data in large scale system software.
  - ✦ Paper presentation, CODASPY Conference, San Antonio,Texas, USA, Feb 2012
- Monitoring safety properties of Stochastic Systems
  - ✦ Paper presentation, Midwest Women in Computing Conference, Chicago, USA 2009
- Wireless Application Security, at Computer Society of India , vizag Chapter,India 2000

ADDITIONAL RESEARCH PROJECTS

- Data lifetime minimization in concurrent applications (with Aravinda P.Sistla, V.N. Venkatakrishnan - UIC)

Professional Experience

Research Assistant Fall, 2007 – Oct, 2013

University of Illinois at Chicago, Chicago, USA

- Proposed novel solutions to minimize data lifetime in C Programs for data confidentiality.
- Studied monitoring Techniques for program verification.
- Peer reviewed academic conference papers and journal articles.
- Prototyped and evaluated several research ideas.

Teaching Assistant Fall, 2006 – Fall, 2008

University of Illinois at Chicago, Chicago, USA

- Teaching assistant for for courses Languages and Automata, and Compiler Design.
- Graded assignments for courses Operating Systems and Computer Networks.
- Mentored students and graded assignments of students in courses Database Management Systems and Software Engineering.

Research Student June, 2003 – June, 2004

University of Hyderabad, Hyderabad, India

- Studied Role Based Access Control mechanisms.
- Designed and developed Privilege Management Infrastructure (PMI),an authorization framework to certify privileges for individuals.

IT Engineer June, 2004 – June, 2006

CMC Ltd. R&D, Hyderabad, India

- Active member of the project which provides PKI software NIC, India.
- Worked on Cheque Truncation System (CTS) to handle security aspects.
- Developed security module for Firmware Upgradation System (FUS).
- Worked on Single Sign On (SSO) module for MIRO Technologies.
- Part of development team for Fingerprint Analysis and Criminal Tracking System (FACTS) .

Professional Activities

- Peer-reviewed research articles for:
  - ✦ IEEE Security & Privacy (Oakland): 2010, 2011, 2012
  - ✦ ACM Computer & Communications Security (CCS): 2009
  - ✦ Annual Computer Security Applications Conference (ACSAC): 2008, 2009, 2010
- Conferences attended
  - ✦ CODASPY 2012, CCS 2010, 2011, and 2012
  - ✦ USENIX Security 2009, MidWic 2009
  - ✦ Midwest Workshop 2007, 2008

Awards and Activities

Awards

- Graduate Student Travel award to attend CODASPY 2012

- Travel Grant to attend Grace Hopper Conference in Florida, 2007.
- Travel Grant for 18th USENIX Security Symposium, Montreal Canada in 2009.
- Second Topper in Class in Bachelors and Masters
- GE Fund Scholarship for 2002-04
- First price for technical talk on wireless application security at CSI, vizag Chapter.

EXTRA CURRICULAR ACTIVITIES

- Member of WISE at UIC
- Represented the state (in India) at sub-junior level kho-kho game Championship
- Active participant in school level singing competitions.
- Third price in state-level vedic knowledge exam conducted by TTD-India in 1996.

RELEVANT COURSEWORK

AT UIC

Advanced Web and Electronic Voting Security	Formal Methods in Concurrent and Distributed Systems	Security & Privacy: Ethical, Legal and Technical Consideration
Compiler Design	Network and Distributed Systems Security	Computer Systems Security
Secure Computer Systems		

AT STANFORD UNIVERSITY

Securing Web Applications, and Emerging threats & Defenses

SKILLS

Research:	Developing solutions, Identifying research problems, Collaborations.
Computer Languages:	OCaml, C, C++, HTML, LaTeX, Java, JSP, Perl, Shell script, SQL
Tools:	SVN, Vim, Eclipse
Operating Systems:	DOS, Linux (Gentoo/Ubuntu), UNIX, Windows (98/ME/2000/XP)
Software Engineering:	SDLC for short and long-term software development projects