DEICS: Data Erasure In Concurrent Software

Kalpana Gondi, A. Prasad Sistla, and V.N. Venkatakrishnan

Department of Computer Science, University of Illinois, Chicago

A well known tenet for ensuring unauthorized leaks of sensitive data such as passwords and cryptographic keys is to erase ("zeroize") them after their intended use in any program. Prior work on minimizing sensitive data lifetimes has focused exclusively on sequential programs. In this work, we address the problem of data lifetime minimization for concurrent programs. We develop a new algorithm that precisely anticipates when to introduce these erasures, and develop an implementation of this algorithm in a tool called DEICS. 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.

1 Introduction

Improper handling of data in C programs can often lead to its disclosure to unauthorized principals. This is because such sensitive data can often be stolen through various low-level attacks that C programs are prone to. The recent Heartbleed vulnerability [6] in OpenSSL is one such example. This security hole rendered millions of organizations on the Internet vulnerable to data theft attacks as it facilitated the risk of sensitive data being read from a OpenSSL connection through a buffer over-read. Such risks of sensitive data being stolen can be minimized if the program erased sensitive data that remained in memory beyond its intended use in the program.

Prior Work: The security issues in not erasing sensitive data in C programs are well documented in the systems community [17,18,23]. As documented in these works, a number of online and offline attacks could result from sensitive data that is resident in memory beyond its lifetime. In [23], authors had proposed an approach to minimize such disclosure of data in applications written in C through program analysis and transformation. The main idea is to identify the "first no use" points (`First-No-Use`) for any piece of sensitive data and introduce a zeroing instruction immediately before those points. The implementation of their approach was tested on several C programs and was demonstrated to work on large applications. One main limitation of their approach is that it cannot handle concurrent applications.

Problem Setting: In this paper, we consider the problem of sensitive data-lifetime minimization for concurrent applications. Implementing and understanding a concurrent application is relatively more difficult for a programmer due to the added complexity of programming such applications. While programming such applications, it is possible that a programmer may ignore other security considerations such as zeroing sensitive data after their intended use in the program. A recent low-level vulnerability (CVE-2011-0992) [1] in the popular Mono application (which implements the Silverlight API for Linux) allows remote attackers to obtain sensitive information that is unauthorized. Specifically, 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.

Such risk of disclosure would have been minimized by zeroing sensitive data values after use. Since a significant number of programmers are not aware [26] of these subtle security issues, we consider the problem of retrofitting a concurrent program with zeroing instructions that minimizes the lifetime of sensitive data used by that application.

Challenges: In order to minimize data lifetime, one must analyze the lifetime of sensitive data in concurrent programs. 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 data). 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.

Our Approach: Our approach is to transform concurrent applications with zeroing instructions (`memset` instructions in C) to erase shared data so that the exposure of data is minimized. The main challenge in doing this is to determine whether an erase instruction for a shared data object can be safely introduced at a given program point, by considering all possible concurrent executions of threads. The main contribution of this paper is to address the abovementioned challenge by introducing a formal notion of *RacyPairs*, which assists in determining whether a given erasure for a shared object is safe or not. In addition, we give an algorithm to effectively compute *RacyPairs*, by leveraging existing work [19] on concurrent dataflow analysis using a race-detection engine. 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 will not be erased by our approach if there can be a potential access by some thread in the program.

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. This paper makes the following contributions:

- A formal notion of *RacyPairs* that suggests when a given erasure is safe
- An algorithm to effectively compute *RacyPairs* using race detection engines
- Implementation of the algorithm in the form of a tool called SWIPE
- A detailed evaluation of our approach by transforming over 100k lines of C applications (combined, with the largest application consisting of 57k LOC).

The rest of the paper is organized as follows: Section 2 presents the challenges of introducing erase instructions for data lifetime minimization in concurrent applications with an illustrative example. We explain our approach in section 3. The main algorithm

behind our transformation scheme is described in section 4. Section 5 provides a detailed evaluation of our approach on set of real-life concurrent applications written in C. Section 6 presents the related work and we conclude in section 7.

2 Running Example

Example: Figure 1 gives a simplified version of producer-consumer example implemented using threads. In this program, the server thread accepts the connection from the client and collects the *Request* and places it in a shared memory (here the *request* variable declared at line 2) for the worker thread to process the same.

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 42. Shared variable, $request$ (defined 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 and logs the $request$ at line 10. The worker thread reads the same data on line 25. (Let us ignore line 25a for the moment as it is not part of the original but belongs to the transformed program that we will explain shortly). 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.

```
1   DEFINE LENGTH 20;
2   char *request;
3   mutex_type lockv;
    /* Server thread accepts
    connection
    and collects client HTTP requests
    */
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
    request */
23    lock(&lockv);
24    if(request != NULL){
25     process(request);
    // read(request)
    25a: memset(request, 0, LENGTH);
26    }
27    unlock(&lockv);
28  // generating response : Lines 28 - 36
37   }
38   int main(){
39    mutex_init(&lockv);
40    thread_create (server);
41    thread_create (worker);
42    // do some other work : Lines 42 - 52;
53   }
```

Fig. 1: Running example with erase instructions introduced for shared data.

3 Approach

Transformed Example: We first show how our approach transforms the original program given in figure 1. Our transformation simply introduces an erase for the shared variable $request$ after line 25 since $request$ is no longer required after line 25 in worker thread. This is accomplished using the call to `memset` introduced by our approach in line 25a. Note that, here we only highlight the erasure of shared data. Erasures for local variables within each function / thread could be introduced using the approach from existing work [23], and we do not discuss the erasure of locals further in our technical approach.

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 that, for each definition of a variable, identifies program locations after which no other thread 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 shared variable separately and tries to erase the contents of each definition after its intended use in the program (and thereby minimizes the exposure of data between definitions).

Introduction of an erase operation is safe only when the erase (which is a write operation) does not influence a read in another thread. Consider the running example given in figure 1. If we introduce an erase for $request$ in worker thread after line 25, we can clearly see that there is no parallel read that is influenced by it and is therefore safe. On the other hand, if the $request$ is erased in server thread after line 10, the worker thread may get a zero value for access at line 25, and is therefore not safe.

One may be tempted to identify parallel reads that may get influenced by the write we introduce by checking for data-races [1] caused by our write. However, absence of such data-races does not guarantee that our writes do not influence a parallel read. For example, in the running example given in 1, there was a last access for the $request$ in the server thread at line 10. Even though, the $request$ is last accessed at line 10 in server thread, it may still be required in worker thread based on the execution order. However, a race-detector may not identify the write we may introduce (before line 11) 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.

Alternatively, mere presence of parallel reads should not prevent the introduction of erasures. In the running example given in figure 1, there is a definition for the shared variable $request$ inside the worker thread at line 9. Introducing erasure for the shared data $request$ in the worker thread before line 26 does not influence the value read in server thread at line 10 (because of the definition at line 9), irrespective of thread

[1] There are various types of races as explained in [35], but we use a fairly general notion of a data-race which simply happens when two threads that access and modify shared data at the same time without any protection mechanism.

interleavings. Therefore our analysis should be more precise to check if the parallel reads in other threads are actually influenced by our erasure.

3.1 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. As observed earlier, it is not sufficient to check for parallel reads, we should also consider other writes which may actually influence those reads (in the running example given in figure 1, the read for $request$ in server thread at line 10 depends only on the definition at line 9 and does not get influenced by any write in worker thread). 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 erasure (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') specified as follows - 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 same shared variable, and l' is reachable from l without another definition to this shared variable in between. Absence of such critical pairs confirm that we can safely introduce erasures.

We explain our approach with the help of the running example given in figure 1. Consider the erase of the shared variable $request$ in $worker$ thread at new line $25a$ as shown in the figure 2.

```
22   int worker(){
     /* Worker thread processing 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   // generating response : Lines 28 − 36
37   }
```

Fig. 2: Erasing shared data $request$ in worker thread

We now introduce pseudo-reads for the shared variable $request$ after line 5 and line 11 in the $server$ thread as shown in the figure 3. Querying a race-detection engine identifies the pseudo-reads after lines 5 and 11 in $server$ thread to be in race with the write at line 25a 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

```
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  }
```

Fig. 3: Erasing shared data $request$ in server thread

the write we introduced at line 25a as these lines are inside a lock region (The number of pseudo-read instructions to be considered in a thread can be optimized as explained in section 4).

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 generate a critical pair as there is a definition for $request$ at the line 9 that is on the control flow 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 it. 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 $worker$.

Now let us consider the introduction of an erasure for the shared data $request$ in $server$ thread immediately after line 10. However, a pseudo-read after line 22 in the $worker$ thread will be in race with this write on $request$ after line 10, furthermore there is an actual read at line 25 that is reachable from line 22 without any re-definition of $request$ in between. Hence, the pair $(22, 25)$ forms a critical pair and we cannot introduce erase for $request$ in the $server$ thread after line 10.

3.2 Technical Description

System Model We formalize the intuitive description given before using a subset of C language with concurrency constructs given in Table 1. Table 1 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 1. The labels of statements in all the functions including the main function are assumed to be distinct. We use sv to represent a typical shared variable and $lckv$ to represent a typical lock variable used for synchronization.

In a program P, threads (represented using t in the language shown in Table 1) are created by invoking the call $thread(t)$. Thread invocations are different from normal function calls. They represent 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

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*, ..., *E*)	[EXPRESSION]

Table 1: A small subset of C language with concurrency constructs

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.

Assumptions: We assume that ordinary functions invoked in distinct thread functions are different, i.e., the same ordinary function is not invoked in more than one thread function. 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 inlining of functions is assumed only for simplicity of presentation. Our actual algorithm does not do this. Instead it computes function summaries as in SWIPE [23] and uses them wherever a function is invoked.) 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 [39]). Authors of [23,38,10,9,40] 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.

A definition is a statement that updates the value of a variable, such as an assignment statement or a library call that reads external values and assigns it to a parameter. A definition is denoted by a unique identifier id. Standard definitions of $Aliases$, $must_definitions$, $may_aliases$, $succ$, and $preds$ are adopted from SWIPE [23]. 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.

Intra-procedural Analysis: 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 def-

inition id is reachable inside the function. We call this set as $\texttt{Reachability}(id)$ (as defined in [23]). We split the $\texttt{Reachability}(id)$ set into three different sets named $\texttt{UsePoints}(id)$, $\texttt{NoUse}(id)$ and $\texttt{ErasePoints}(id)$. $\texttt{UsePoints}(id)$ is the set of nodes where a definition id is required and $\texttt{NoUse}(id)$ is the set of nodes where the definition id is available but not required. For a given definition id, a transition from $\texttt{UsePoints}(id)$ to $\texttt{NoUse}(id)$ is the place where we can introduce erase instructions for id. We call this set of nodes as $\texttt{ErasePoints}(id)$ before which we may be able to safely introduce erase instructions (similar to [23]).

During our analysis, we identify each local variable x of each thread function. For each definition id of this local variable x, at each location l in $\texttt{ErasePoints}(id)$ 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 here onwards, we describe the approach for erasures of definitions of shared variables. As explained earlier, in the running example of figure 1, we treat the shared variable $request$ as an implicit formal parameter to each of the functions ($server$, $main$, and $worker$). There is exactly one definition, which is at line 9, for the location pointed to by $request$. We let $request_id$ denote this definition. Our analysis as given by SWIPE [23] for the $server$ thread, computes $\texttt{ErasePoints}(request_id) = 11$ and for the $worker$ thread it computes, $\texttt{ErasePoints}(request_id) = 26$.

Once the $\texttt{ErasePoints}(id)$ is computed for shared variables in each thread, our analysis needs to check that if introducing these erasures before the lines in $\texttt{ErasePoints}(id)$ would effect the reads in other threads. For each definition of a shared variable sv denoted by id, we consider the introduction of dummy writes (i.e., erasures) for sv, before each location in the set $\texttt{ErasePoints}(id)$ of a thread function. For each such dummy write, denoted by $\texttt{DummyWrite}(id)$, our analysis computes the set $\texttt{RacyPairs}(\texttt{DummyWrite}(id))$ defined below, of critical pairs discussed earlier.

Definition 1. *For each dummy write* $\texttt{DummyWrite}(id)$ *of a definition of a shared variable sv, denoted by id,* $\texttt{RacyPairs}(\texttt{DummyWrite}(id))$ *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 dummy write* $\texttt{DummyWrite}(id)$*, 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 at least one element in $\texttt{RacyPairs}(\texttt{DummyWrite}(id))$ indicates that it may not be safe to introduce $\texttt{DummyWrite}(id)$. However, absence of such pairs guarantee that we can safely introduce $Erase(id)$ for shared data at the location of $\texttt{DummyWrite}(id)$.

As explained in section 3.1, for the running example of Figure 1, the set $\texttt{RacyPairs}(\texttt{DummyWrite}(request_id))$ is empty if $\texttt{DummyWrite}(request_id)$ denotes the dummy write on the variable $request$ just before line 26 in the worker thread. Hence this erase statement for $request$ can be safely introduced before line 26. On the other hand, if $\texttt{DummyWrite}(request_id)$ denotes the dummy write before the line 11 in the server thread then the pair $(22, 25) \in \texttt{RacyPairs}(\texttt{DummyWrite}(request_id))$. Thus an erase statement for $request$ cannot be introduced just before line 11.

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 Algorithm and Implementation

4.1 Algorithm

Algorithm 1 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 algorithm is given assuming that there is no recursion and all ordinary functions are inlined as explained in section 3 (the algorithm can be easily modified to avoid inlining the ordinary function and also to handle recursion by using function summaries as given in [23]).

Algorithm 1: DEICS Implementation

Notation: f - thread function, id - unique identifier for a definition ϕ - empty set

```
  for each f do
      attach shared variable to formals set;
      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;
      end
  end
  for each definition id of shared variable do
      for each ErasePoints(id) do
2         introduce DummyWrite(id) before;
      end
  end
  for each DummyWrite(id) do
3     Compute RacyPairs(DummyWrite(id));
  end
  for each function f do
      for each definition id of shared variable do
          if RacyPairs(DummyWrite(id)) = ϕ then
4             introduce Erase(id);
          end
      end
  end
```

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 consists 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 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 $\texttt{ErasePoints}(id)$ computed in step 1. Dummy writes are introduced first and actual erases are introduced based on our analysis on the changed program with dummy writes.

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

Step 4: We then transform the program by introducing erase instructions in place of those $\texttt{DummyWrite}(id)$ whose corresponding set $\texttt{RacyPairs}(\texttt{DummyWrite}(id))$ is empty. We provided a discussion on how one can prevent the compiler from optimizing our erasing instructions in [23].

4.2 Implementation

In our implementation, we assume that all global variables are shared variables, treated as additional formal variables to any function. 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 explained in section 3, 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.

Our implementation captures the effect of introducing erasures for the shared data at `ErasePoints` by leveraging on existing concurrent data flow analysis. In particular, we use and build on the RADAR [19] 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.

A straightforward use of RADAR approach does not suffice for our purpose of data erasure. This is because, in RADAR, the main focus is on $writes$ performed by all the threads. Whereas, our analysis introduces $writes$. RADAR considers write-write races in addition to write-read races. However, we only need to consider the later type of races, more specifically, races between dummy writes and pseudo-reads that are introduced.

Dummy Writes for Potential Erasures: We modified the RADAR framework to introduce dummy writes for shared data before the set $\texttt{ErasePoints}(id)$ as indicated earlier. In addition, pseudo-reads are introduced. These dummy writes are treated as original writes during the analysis inside RADAR. For each definition id of shared variable, instead of computing the set $\texttt{RacyPairs}(\texttt{DummyWrite}(id))$, our analysis computes a superset of `RacyPairs` which we call $WeakRacyPairs$ that we explain below.

WeakRacyPairs: $WeakRacyPairs$ is a set of pairs of locations (l, l') specified as follows : 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 same shared variable, and l' is reachable from l. Note that, the original criterion of the absence of redefinition of shared variable in the path between l and l' for $RacyPairs$, is relaxed for the computation of $WeakRacyPairs$.

Emptiness of the set $WeakRacyPairs$ confirms that we can safely introduce erasures and therefore is still sound.

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 behavior is same throughout the region. For the running example given in figure 1, 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.

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

5 Evaluation

We implemented our tool DEICS in Ocaml using the CIL [34] and RADAR.

Applications: Using our tool, we transformed five multi-threaded applications written in C. The common feature of all these applications is that they use Pthreads library for the multi-threading functionality. Of the five, three of the applications (zebedee [5], retawq [41], mtdaapd [2]), handle sensitive data such as ftp passwords and database records. In order to further illustrate the precision and performance aspects of DEICS, we chose two additional applications (pfscan [3], knot [11]) from the RADAR benchmarks suite [4].

Application	Size (LOC)	no.of funcs	Xfrmtion time(sec)
pfscan	1259	24	15
knot	2255	56	21
zebedee	11682	220	55
mtdaapd	57102	637	3150
retawq	38750	638	2753

Table 2: application size and transformation time taken

Application	#of globs	#of thrds	erasures globs	#of locals	erasures locals
pfscan	18	2	11	156	114
knot	43	6	10	62	160
zebedee	61	3	928	3776	2465
mtdaapd	326	5	176	5359	3755
retawq	444	2	342	3511	4387

Table 3: Effect of transformation on Applications

Table 2 shows the application sizes and transformation time taken. The largest application consists of 57K lines of code (LOC). Column 2 in table 2 shows how DEICS scales well to transform applications from 1K LOC to 57K LOC, whose sum of lines of code is more than 100K. Total number of functions in each application is given in column 3 and the transformation time taken for each application is shown in column

4. We observed a correlation between the number of functions and transformation time. The transformation time includes the time taken by race-detection engine as well.

5.1 Effectiveness

Table 3 shows the effect our transformation. For each application (shown in column 1), we identified the number of global variables (excluding pthread mutex variables) which is shown is column 2 of the table 3. Minimum number of threads required to run 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 one 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. Also, DEICS introduces erases for globals before all termination points in the program, covering all possible paths an execution can take. The number of local variables in each application is shown in column 5. The number of erasures DEICS introduced for definitions of local variables is shown in column 6 of table 3.

We also evaluated the effectiveness of our tool to check if the erasures are introduced for sensitive information in local and shared global variables. For the application *zebedee*, DEICS erased sensitive information such as keys used for data encryption. 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, by introduction of such erase instructions in an application, DEICS reduces 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. After analyzing each application, it is clear that, most of the applications do handle sensitive data in local and shared variables and erasure of such data is important.

Precision: We illustrate the utility and precision of DEICS using *pfscan* and *knot* applications. The *pfscan* application already had $free$ instructions for some sensitive objects. DEICS introduced erase instructions for these objects much before these free instructions, thus reducing the exposure of the data. For the *knot* application, DEICS our manual review of certain key program variables indicates that DEICS is precise in identifying the erase points for data held in these variables.

5.2 Performance

We measured the performance overhead of transformed applications (shown is figure 4). We ran our experiments on x86 Linux platform with configuration of 8 GB of memory and a 3 GHz AMD Phenom processor. To capture the execution time, we ran each application multiple times with the same input and using same number of threads. For each application, we used minimum set of required threads during the execution of original and transformed program (this varied between 2 and 6 threads).

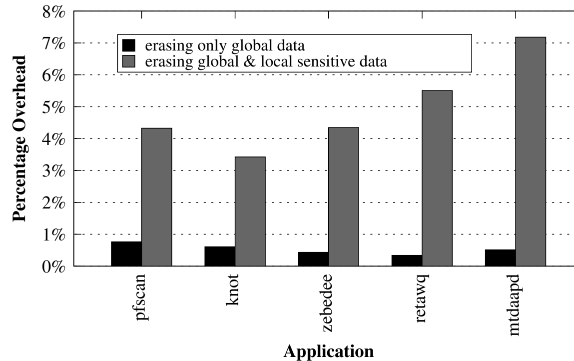

Fig. 4: Performance overhead of concurrent applications

The overhead caused due to the erase instructions for sensitive data in global variables as well as local variables of individual functions In this case, the maximum runtime overhead is less than 8% and average is around 5%.

Another set of experiments are conducted to measure the overhead due to erasing only shared / global data (without erasing local data) (measuring only the overheads of the work reported in this paper). In this case, the runtime overhead ranged from 0.3% to 0.7%, averaging 0.5%. We show this in Figure 4 (black bars). We observe that transforming applications with a tool like SWIPE would also have the overhead of around 5%. Our transformation in DEICS to erase shared data adds only an additional performance overhead of 0.5% to minimize the lifetime of shared data.

6 Related Work

Data Lifetime Minimization There have been various works in this area of data lifetime minimization in the systems community by employing techniques from operating systems [24,25,13,8,31,18]. Our approach uses program transformation techniques. In addition, we handle concurrent programs which is not handled by prior work.

Static Analysis of Concurrent Applications: There are tools and frameworks developed to perform dataflow analysis [20,19]. In [32,22,39,20], a graph to represent the parallelism is built and a modified version of sequential analysis is performed. [21,14] provide a generic approach for static analysis of concurrent programs. Qadeer et. al. proposed a technique to transform concurrent programs to sequential programs [36] 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.

Garbage Collection and Region-based Management Our approach for reducing sensitive data-lifetimes is related to approaches for garbage collection.A key difference between our approach and garbage collection is that our approach uses a tight, dynamic criterion for erasing sensitive data whereas garbage collectors use a more relaxed criteria. We could augment such a garbage collector with memory erasing routines to ensure that freed objects are erased in memory. However, such a solution may still be impre-

cise 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, however, this frequent calling can introduce overheads that are unpredictable. Free-me [27] aims to insert deallocation instructions by conservatively estimating object lifetimes.

Extensive work in the area of region-based memory management has been performed [37,7,28,12,42]. The main goal of these works is to have an economic usage of memory and reduce the need for invoking garbage collector. A region based approach could be used for erasing sensitive data, however it might result in poor precision.

Data Erasure and Memory Safety Chong et.al [16,15] provide a formal treatment to information erasure in their paper. Their approach is targeted towards new applications, whereas our approach can transform existing applications as well. Memory management techniques have been proposed to minimize the risk of data exposure [8,31]. However data lifetime minimization is not achieved with these approaches.

Privilege Separation Approaches such as [29,33,30,43] rely on changes at operating system and hardware level to maintain the sensitive data separately so that there are no privilege escalation attacks. Our approach focuses on modifying applications to have built-in mechanisms for minimizing data exposure.

7 Conclusion

In this paper, we 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 the runtime overhead. We have evaluated a set of real world concurrent applications written in C to show its effectiveness.

Acknowledgements. This material is based on research sponsored in part by DARPA under agreement number and by National Science Foundation under grants FA8750-12-C-0166, CCF-0916438, CNS-1035914, CCF-1319754, CNS-1314485, CNS-0845894, DGE-1069311, and NSF-1241685. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, NSF or the U.S. Government.

References

1. Common vulnerability exposures. https://cve.mitre.org/.
2. Mtdaapd. http://sourceforge.net/projects/mt-daapd/.
3. Pfscan. http://freecode.com/projects/pfscan.
4. Radar. http://cseweb.ucsd.edu/~lerner/radar.html.
5. Zebedee. http://www.winton.org.uk/zebedee/index.html.
6. Heartbleed. http://en.wikipedia.org/wiki/Heartbleed, 2014.
7. Alexander Aiken, Manuel Fahndrich, and Raph Levien. 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.

8. Periklis Akritidis. Cling: A Memory Allocator to Mitigate Dangling Pointers. In *USENIX Security Symposium*, Washington, DC, 2010.
9. Lars Ole Andersenm. Program Analysis and Specialization for the C Programming Language. Technical report, 1994.
10. Dzintars Avots, Michael Dalton, V. Benjamin Livshits, and Monica S. Lam. Improving Software Security with a C Pointer Analysis. In *International conference on Software engineering*, St. Louis, MO, 2005.
11. Rob von Behren, Jeremy Condit, Feng Zhou, Bill McCloskey, Eric Brewer, and George Necula. Knot. http://capriccio.cs.berkeley.edu/.
12. Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. 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.
13. Hans-J Boehm. A Garbage Collector for C and C++. http://www.hpl.hp.com/personal/Hans_Boehm/gc, 2002.
14. Ahmed Bouajjani, Javier Esparza, and Tayssir Touili. 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.
15. Stephen Chong and Andrew C. Myers. Language-Based Information Erasure. In *Computer Security Foundations Workshop*, Aix-en-Provence, France, 2005.
16. Stephen Chong and Andrew C. Myers. End-to-End Enforcement of Erasure and Declassification. In *Computer Security Foundations Symposium*, Pittsburgh, PA, 2008.
17. Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding Data Lifetime via Whole System Simulation. In *USENIX Security Symposium*, San Diego, CA, 2004.
18. Jim Chow, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Shredding Your Garbage: Reducing Data Lifetime through Secure Deallocation. In *USENIX Security Symposium*, Baltimore, MD, 2005.
19. Ravi Chugh, Jan W. Voung, Ranjit Jhala, and Sorin Lerner. 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.
20. Arnab De, Deepak D'Souza, and Rupesh Nasre. 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.
21. Evelyn Duesterwald and Mary Lou Soffa. Concurrency analysis in the presence of procedures using a data-flow framework. In *Proceedings of the symposium on Testing, analysis, and verification*, TAV4, pages 36–48, New York, NY, USA, 1991. ACM.
22. Matthew B. Dwyer and Lori A. Clarke. 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.
23. Kalpana Gondi, Prithvi Bisht, Praveen Venkatachari, A. Prasad Sistla, and V. N. Venkatakrishnan. 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.
24. Peter Gutmann. Secure Deletion of Data from Magnetic and Solid-state Memory. In *USENIX Security Symposium*, San Jose, California, 1996.
25. Peter Gutmann. Data Remanence in Semiconductor Devices. In *USENIX Security Symposium*, Washington, DC, 2001.

26. Peter Guttman. Software Leaves Encryption Keys, Passwords Lying around in Memory. Security Focus Vuln Dev Mailing List: http://www.securityfocus.com/archive/82/298001/30/0/threaded, 2002.
27. Samuel Z. Guyer, Kathryn S. McKinley, and Daniel Frampton. Free-Me: A Static Analysis for Automatic Individual Object Reclamation. In *Programming Language Design and Implementation*, Ottawa, Ontario, Canada, 2006.
28. Niels Hallenberg, Martin Elsman, and Mads Tofte. 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.
29. Tejas Khatiwala, Raj Swaminathan, and V.N. Venkatakrishnan. Data Sandboxing: A Technique for Enforcing Confidentiality Policies. In *Annual Computer Security Applications Conference*, Miami Beach, FL, 2006.
30. Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information Flow Control for Standard OS Abstractions. In *Symposium on Operating Systems Principles*, Washington, WA, 2007.
31. Chris Lattner and Vikram Adve. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. In *Programming Language Design and Implementation*, Chicago, IL, 2005.
32. Jaejin Lee, David A. Padua, and Samuel P. Midkiff. 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.
33. Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *IEEE Symposium on Security and Privacy*, Oakland, CA, 2010.
34. George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Conference on Compiler Construction*, Grenoble, France, 2002.
35. Robert H. B. Netzer and Barton P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, March 1992.
36. Shaz Qadeer and Dinghao Wu. Kiss: keep it simple and sequential. *SIGPLAN Not.*, 39(6):14–24, June 2004.
37. C. Ruggieri and T. P. Murtagh. 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.
38. Radu Rugina and Martin Rinard. Symbolic Bounds Analysis of Pointers, Array Indices, and Accessed Memory Regions. In *Programming Language Design and Implementation*, Vancouver, British Columbia, Canada, 2000.
39. Nishant Sinha and Chao Wang. 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.
40. Bjarne Steensgaard. Points-to Analysis in Almost Linear Time. In *Principles of Programming Languages*, St. Petersburg Beach, FL, 1996.
41. Arne Thomaen. Retawq. http://retawq.sourceforge.net/.
42. Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-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.
43. Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making Information Flow Explicit in HiStar. In *Symposium on Operating Systems Design and Implementation*, Seattle, WA, 2006.