A Compiler-Based Approach to Data Security*

F. Li[1], G. Chen[1], M. Kandemir[1], and R. Brooks[2]

[1] Computer Science and Engineering Department,
The Pennsylvania State University, University Park, PA 16802
{feli, guilchen, kandemir}@cse.psu.edu
[2] Electrical and Computer Engineering Department,
Clemson University, Clemson, SC 29634
rrb@clemson.edu

Abstract. With the proliferation of personal electronic devices and embedded systems, personal and financial data is more easily accessible. As a consequence, we also observe a proliferation of techniques that attempt to illegally access sensitive data without proper authorization. Due to the severe financial and social ramifications of such data leakage, the need for secure memory has become critical. However, working with secure memories can have performance, power, and code size overheads since accessing a secure memory involves additional overheads for encryption/decryption and/or password checks. In addition, an application code may need to be restructured to work under such a memory system. In this paper, we propose a compiler-directed strategy to generate code for a secure memory based embedded architecture. The idea is to let the programmer mark certain data elements, called the seed elements, as secure (i.e., need to be stored in secure memory), and let the compiler determine the remaining secure elements automatically. We also address the problem of code size increase due to our strategy. The experimental results obtained through simulations clearly show that the proposed approach is effective in reducing the total secure memory size. The results also indicate that it is possible to reduce the resulting code size increase by clustering accesses to secure memory.

1 Introduction

Secure memories are those that provide a secure place for the storage of sensitive information to prevent undesired accesses. Such memories are becoming increasingly important in many embedded systems such as smartcards, health-monitoring devices, and PDAs that store vital personal/financial information. There are different ways of implementing secure memories. For example, crypto-memories store data in an encrypted form and require decryption for data access. The password-protected memories, on the other hand, require a handshaking protocol for verifying the identity of the requester.

Secure memories, while effective in providing data protection, have at least two problems associated with them. First, accessing a secure memory takes more execu-

* This work is supported in part by NSF Grants #0444158, #0406340, #0093082, and a grant from GSRC.

R. Bodik (Ed.): CC 2005, LNCS 3443, pp. 188–203, 2005.
© Springer-Verlag Berlin Heidelberg 2005

tion cycles than accessing a non-secure (conventional) memory. The number of additional cycles for the required security checks depends on the type of the secure memory employed, i.e., whether it is password-protected or crypto-memory. Second, secure memory accesses consume extra energy, which may or may not be tolerated depending on energy budget of the battery-operated embedded system under consideration. Fortunately, in a given embedded application, not all the data elements demand security (or at least the same level of security), and thus, not all the data elements need to be stored in a secure memory. As an example, in an image processing application that manipulates secure images, a certain portion of the frames can contain sensitive data (and need to be stored in a secure memory), whereas the remaining parts can be stored in a conventional (i.e., non-secure) memory. The problem then is to decide, given an embedded program, the set of data elements that need to be stored in the secure memory. Since an error in making this decision can have serious consequences (as it can compromise security of the application), it might be beneficial to automate this decision within an optimizer.

This paper proposes a compiler-directed approach to this problem. The idea is to let the programmer mark certain data elements, called the seed elements, as secure (i.e., need to be stored in the secure memory), and let the compiler determine the remaining secure elements automatically. The programmer can be conservative in determining the seed elements; however, more accurate she is in marking such elements, the smaller the total number of secure elements determined by the approach. It should be noticed that, since the seed elements can be assigned to other data elements in the application, the final set of secure elements determined by our approach is normally larger than the seed elements alone. We use a compiler-directed program analysis that captures such assignments of seed elements and keeps track of the elements that need to be stored in the secure memory. Since our approach determines the minimum set of secure elements, it reduces the secure memory space required by the application. In addition, reducing the number of secure elements both improves execution cycles and reduces memory energy consumption. However, since secure and non-secure memory accesses typically make use of different load/store operations, one needs to be careful in not excessively increasing the size of the generated code. To address this issue, this paper also proposes and evaluates a loop iteration scheduling scheme. The experiments with this scheduler indicate significant savings in the code memory space requirements.

We implemented the proposed approach within an optimizing compiler and performed several experiments with five embedded benchmark codes. Our experimental results obtained through simulations clearly show that the proposed approach is effective in reducing the secure memory size, and the overheads associated with working under secure memories.

The remainder of this paper is structured as follows. The next section gives an overview of secure memories. Section 3 presents the details of our code and data partitioning for secure memories. Section 4 presents results from our experimental evaluation, and Section 5 discusses related work. Finally, Section 6 concludes the paper.

2 Secure Memory Background

The basic secure memory architecture consists primarily of a cryptographic engine and a normal memory unit. The cryptographic engine facilitates the encryption/decryption of the data transmitted between the CPU and the secure memory. Secure memories [5,11] can additionally support mechanisms for password protection and authentication in addition to the encryption functionality. The encryption and decryption are performed based on whether a secure load/store is desired. The instruction set architecture is augmented by special *secure load* and *secure store* operations. These secure memory operations can be implemented through the use of an additional bit in the instruction format, which can be set by the compiler during code generation. If this bit is set, a load operation requires the read data to be decrypted before it is fed to the datapath, and a store operation requires the data to be encrypted before it is written into the memory. However, normal loads and stores incur *no* additional performance penalty as they can identify that the encryption/decryption can be bypassed early in the instruction decode stage. The encryption scheme typically employs block encryption that translates a given plain text to a cipher text of the *same length*. Hence, there are no complexities involved with mapping the encrypted data on to the memory.

Since the data is stored in an encrypted form, the proposed technique counteracts other non-intrusive techniques such as microscope probing, determining electromagnetic flux, and using laser beams [22,8]. Circumventing other unauthorized programs from accessing the data locations is also possible in this technique. The compiler, in addition to marking load/stores as secure, can also generate a *unique encryption key* for use in the program. Consequently, even if another program uses a secure load or store operation on an illegal location, the operation will not be permitted since the keys would not match. The work presented in this paper can be also be defined as the problem of determining the type of each memory operation (secure vs. non-secure).

3 Code and Data Partitioning

The main problem that makes it difficult to generate code for a memory architecture that is composed of both secure and non-secure memories is that one does not want to compromise any security, but at the same time one does not want to incur severe performance or power overheads due to ensured security. In the following discussion, we first list the constraints under which our compiler-driven approach operates. Following that, we give the details for identifying the set of elements that need to be stored in the secure memory, and the details of a scheduling scheme that can be used for minimizing the code size increase.

3.1 Constraints

In restructuring an embedded application code for execution in a secure memory based environment, there are three major constraints that need to be addressed:

- Security Constraint: All sensitive data elements must be assigned to secure memory. Note that, this is a correctness issue since failing to satisfy this constraint can lead to serious consequences and is not acceptable.
- Overhead Constraint: The data memory space occupied by the secure data elements must be minimized. In addition, the performance (execution cycles) and energy overheads imposed by the secure accesses must be minimized. In fact, this is one of the main goals of this paper.
- Code Size Constraint: The size of the generated code should not be excessively large since this can increase the code memory demand. Both this and the previous constraint are important; but, if they are not satisfied, correctness is not affected (unlike the security constraint).

It must be noted that, some of these constraints can conflict with each other. For example, if one keeps the set of secure elements larger than necessary, this can increase overheads but can also reduce the increase in code size (as the accesses to secure and non-secure memories are not excessively interleaved and thus can be clustered in a compact manner), and thus a more compact code can be generated.

3.2 Details

3.2.1 Determining Secure Elements

Our approach tries to reduce performance/energy overheads and code size under the security constraint. That is, without compromising security, we want to reduce the overheads to the greatest extent possible.

Our focus is on array-based embedded applications, where multi-dimensional arrays are operated on using a series of nested loops. Let $seed(U_k)$ be the set of seed elements (as specified by the programmer) from array U_k. Then, the set of secure elements, denoted $secure(.)$, is initially set to $\bigcup_k seed(U_k)$, that is, the union of all $seed(U_k)$ sets. If an element s_i of $secure(.)$ is used on the right-hand-side (RHS) of a statement that assigns a new value to an s_j that does not currently belong to $secure(.)$, then we perform:

$$secure(.) \leftarrow secure(.) \cup s_{j.}$$

In other words, the set of secure elements is augmented by sj. The main reason for this is the possibility for some malicious entity to determine the value of si by looking at (i.e., observing) the value of sj as well as the values of the other non-secure elements used in the assignment statement in question, if sj is not treated as secure. Therefore, we conservatively add sj to secure(.).

However, the problem of determining whether si is actually used to update sj is not a trivial one. This is because we are dealing with array indices that are expressed in terms of loop iterators (and potentially loop-independent constants), which take multiple values during the course of execution. To identify such assignments carefully, we employ a polyhedral approach that expresses loop based computations and array accesses using Presburger formulas. Consider the following generic loop nest:

$$\text{for } I :: I_s, I_e$$

$$U_j[R_j(I)] \leftarrow \mathcal{H}\{U_k[R_k(I)]\}$$

```
Input:          loop nests $\mathcal{L}_1, \mathcal{L}_2, \ldots \mathcal{L}_n$;
                $seed(U_k)$ for each array $U_k$.
Output: $secure(.)$.

secure(.) = $\cup_k seed(U_k)$;
$changed$ = true;
while ($changed$) do
        $oldsecure = secure(.)$;
    foreach loop nest $\mathcal{L}_i$ do
        foreach statement $S_i$ in $\mathcal{L}_i$
                assume that $Ref_l$ is the LHS reference in $S_i$;
                foreach reference $Ref_r$ on the RHS of $S_i$
                        $E$ = the set of elements in $secure(.)$ that can be
                                accessed by $Ref_r$;
                        $L$ = the set of iterations at which $Ref_r$ accesses
                                elements in $E$;
                        $NewS$ = elements accessed by $Ref_l$ in iterations $L$;
                        $secure(.) = secure(.) \cup NewS$;
                endfor
        endfor
    endfor
    if ($oldsecure$ != $secure(.)$) then
        $changed$ = true;
    else
        $changed$ = false;
    endif
endwhile
```

Fig. 1. Algorithm for calculating *secure*(.)

In this nest, I represents a vector formed by the loop iterators from top to bottom; I_s and I_e are loop bounds (also expressed as vectors); U_j and U_k are arrays; R_j and R_k are array references to arrays U_j and U_k, respectively (which are functions of I); and $\mathcal{H}$ is a general function. Note that both U_j and U_k can be multi-dimensional.

Suppose now that an element of array U_k, say $U_k[x]$, are in the *secure*(.) set. To determine the corresponding element from array U_j, say $U_j[y]$, that also needs to be placed into the *secure*(.) set, we first determine the loop iteration L that accesses $U_k[x]$. That is, we find an L such that:

$$R_k(L) = x \quad \text{and} \quad I_s \leq L \leq I_e.$$

In the second step, we determine the element from array U_j accessed by L. In mathematical terms, we determine a y such that:

$$R_j(L) = y.$$

Finally, we perform:

$$secure(.) \;\leftarrow\; secure(.) \cup U_j[y].$$

Note that, this can easily be extended to the case where we have multiple array references on the RHS. In this way, starting with the seed elements, our approach keeps increasing the *secure*(.) set each time an assignment statement is processed. Therefore, the complexity of the approach is proportional to the number of the assignment statements in the application code being analyzed. It should be noticed that, after *secure*(.) is updated, we might have more secure elements referenced on the RHS. This is because, it is possible that the same array element can be accessed by both the LHS and RHS references within a given loop nest. Therefore, we need to repeat the above process until we cannot add more elements into *secure*(.). We want to emphasize that this approach tries to keep the size of the *secure*(.) set as small as possible. Fig. 1 gives our algorithm for calculating *secure*(.). In the innermost loop of this algorithm, we first calculate the iterations, *L*, in which the RHS of the statement being considered accesses some elements from the current *secure*(.) set. Then, we add all the elements accessed by the LHS of this statement in iterations *L* to *secure*(.). It can be seen that if *secure*(.) is changed after all the statements have been processed, which is indicated by a boolean variable *changed*, we repeat the same process again for all the statements until further processing would not add more elements to *secure*(). In our implementation, these elements are determined using a polyhedral tool called the Omega Library [13]. The Omega Library provides a set of routines for manipulating linear constraints over integer variables, Presburger formulas, and integer tuple relations and sets.

3.2.2 Reducing Code Size

It must be noted, though, our approach explained thus far does not do any specific optimization for reducing the code size. In fact, it just determines the smallest set of data elements that need to go to the secure memory (and this also helps reduce the runtime performance overheads of working with secure memory). An important point is that, if the accesses to secure and non-secure memories are interleaved (in the output code generated), this can increase the resulting code size dramatically since these two types of memories are typically accessed using different types of load instructions (distinguished using a bit in the instruction format), as explained in Section 2. As an example, consider a loop nest that accesses five one-dimensional arrays, U_1, U_2, U_3, U_4, and U_5:

```
for i :: 1, N
... U1[i], U2[i], U3[i], U4[i], U5[i] ...
```

P_i	U_1	U_2	U_3	U_4	U_5
P_1	*s*	*s*	*s*	*s*	*s*
P_2	*s*	*s*	*s*	*s*	*n*
P_3	*s*	*s*	*s*	*n*	*s*
…					
P_{32}	*n*	*n*	*N*	*n*	*n*

(a)

Iterations	U_1	U_2	U_3	U_4	U_5
1	*s*	*n*	*n*	*s*	*s*
2	*n*	*n*	*s*	*s*	*n*
3	*n*	*s*	*s*	*s*	*s*
…					
N	*s*	*s*	*n*	*n*	*s*

(b)

Fig. 2. (a) Possible load patterns (P_i) for a loop iteration that accesses five arrays. (b) Example load patterns for different iterations. In both (a) and (b), *s* represents $load_{secure}$, and *n* represents $load_{nonsecure}$

After our approach has been applied, different loop iterations can have different load instruction patterns from each other (i.e., different combinations of secure and none secure loads). For each array reference, there are two possible load instructions: loadsecure and loadnonsecure. For all the five array references in this example, there are 32 (=25) possible load (instruction) patterns as shown in Fig. 2(a). In each row of this figure, s (or n) means that loadsecure (or loadnonsecure) is used to load the corresponding array element. The original loop nest might not be able to cover such cases due to different load patterns exhibited by the different iterations (i.e., in generating code, we cannot keep the original loop structure). A naïve way that the compiler can generate code is to fully unroll the original loop nest, and use the appropriate load instructions for each iteration. Fig. 2(b) presents such a scenario of complete unrolling. In this figure, each row represents an iteration, and there are a total of N iterations. The last five columns represent accesses to our five arrays, U1, U2, U3, U4, and U5. Each row gives the load pattern for the five array references in the corresponding iteration. In a real environment, iteration number N could be very large, e.g., more than one million. Consequently, the naïve solution leads to considerable code expansion in this case. At this point, one might point out that there could be some regularity in the load patterns across the iterations. But, since the seed sets for these arrays, seed(Uk) (1≤k≤5), are specified by the programmer (i.e., they can exhibit very irregular patterns), the compiler might not be able to extract any regularity from the load patterns and generate simple loop nest(s) to enumerate them. Even if this is possible, the compiler has to unroll all the iterations first before it can analyze the unrolled loop iterations, and this could be a significant overhead for the compiler in terms of both memory space and performance.

An alternative that we employ here is a load pattern centric approach. This approach can be explained as follows. Let us first assume that there is no loop-carried dependence in the original loop nest above. For each load pattern among the 32 possibilities, we calculate the set Gi (1≤i≤32) containing the iterations that have that load

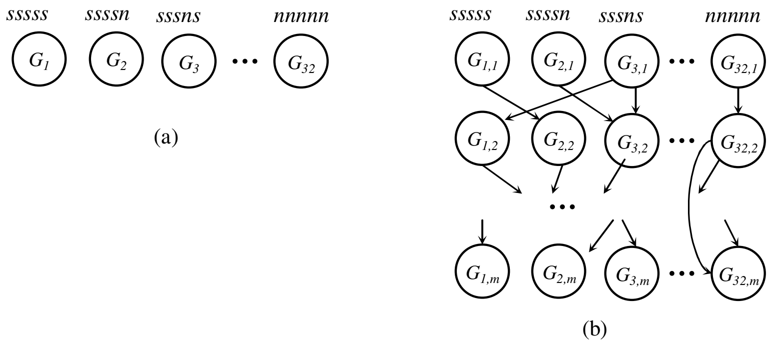

Fig. 3. (a) Sample iteration groups when there is no loop-carried dependence. Each G_i ($1 \leq i \leq 32$) contains the iterations that have the load pattern given above the corresponding node. (b) Example iteration groups when there are loop-carried dependences. The arrows indicate the dependences

pattern. For example, we can obtain the iteration set, G3, for load pattern P3 (sssns), using the Omega Library. That is, using the Omega Library, we build a set that con tains only the iterations that generate the pattern sssns. Fig. 3(a) illustrates this scenario. In this figure, each node represents a set of iterations with the same pattern (written above the node), and there are 32 nodes associated with 32 possible load patterns. Once we obtain all the iteration sets Gi, we make use of the "codegen" utility provided by the Omega Library. Given an iteration set, the Omega Library can generate the corresponding loop nest(s) that enumerates all the loop iterations in that set. Note that, in the ideal case, each node results in a single loop nest. However, if the iterations in a given group could not be enumerated using a single loop nest, the Omega Library can generate multiple nests with the same pattern, and this may lead to an increase in the size of the generated code.

The approach discussed above needs some modifications when there are loop-carried dependences. Fig. 3(b) illustrates such a scenario. In this figure, all the iterations (in the nest for which code is being generated) form a layered dependence graph. Each node G_{ij} represents a subset of G_i, the iterations that have the load pattern P_i given in Fig. 2(a). Note that, each iteration in node G_{ij} (j>1) depends on at least some iteration in $G_{i',j-1}$. The arrows in the graph indicate the dependences between the nodes. It should be emphasized that there cannot be any dependence from G_{ij} to $G_{i'j'}$, where $j' \geq j$. Obtaining such a graph can be done in an iterative way. For each G_i ($1 \leq i \leq 32$), we determine all iterations in G_i that do not depend on any other iteration, and add them to G_{i1}. After all the nodes in layer j have been built, we add $G_{i,j+1}$ the iterations (from the remaining iterations in G_i) that depend only on the iterations in $G_{i'j'}$, where $j' \leq j$. This process is repeated until all the iterations have been assigned to the nodes in the graph. When the entire process is complete, we can schedule the nodes of this graph using any scheduling algorithm (e.g., list scheduling). Then, using the "codegen" utility provided by the Omega Library, we generate code for each node (when we visit that node during scheduling). Fig. 4 gives our scheduling algorithm,

```
Input:          dependence graph G for iteration groups
Output:loop nests that load all the elements in the iteration groups

Ready = all nodes that have no predecessor;
Scheduled = Ø;
while (Ready ≠ Ø)
        remove a node n from Ready;
        codegen(n);
        added n to Scheduled;
        for each successor p of n in G do
                if (all of p's predecessors are in Scheduled) then
                        add p to Ready;
                endif
        endfor
endwhile
```

Fig. 4. Scheduling algorithm for the dependence graph of iteration groups

which is a variant of list scheduling, in the pseudo-code format. In this algorithm, at any scheduling step, we select a node whose all predecessors are already scheduled, so that the dependence requirements can be observed. After selecting a node, we use the "codegen" utility to generate the loop nest(s) that can enumerate the iterations in the group represented by this node. As can be seen, this code generation strategy is oriented towards reducing the number of static load operations in the generated code (through load pattern reuse).

4 Experimental Evaluation

We implemented the proposed strategy within an optimizing compiler built upon SUIF [2], and made experiments with five different embedded benchmark codes. Basically, our compiler reads the input code using SUIF, and fills the Omega Library data structures with necessary information. After this, the Omega Library determines the set of secure elements and secure loop iterations, and the collected information is used by the compiler to construct the dependence graph between the iteration groups. The algorithm given in Fig. 4 is invoked on this dependence graph. Each node of this graph is processed by the Omega Library and the generated enumeration codes are translated into the SUIF internal structures, which in turn is used for emitting the output code. Table 1 shows the five embedded benchmark codes used in this study. The second column gives a brief description of each benchmark, and the third column gives the total data sizes (i.e., the total number of array elements manipulated by the benchmark).

In our experiments, we use the memory access latency values shown in Table 2, which are typical of those memories used in 3.5MHz smartcards [21]. We define the

Table 1. The benchmarks used in this study

Benchmark	Brief Description	Dataset Size
Med-Im04	medical image reconstruction	825.55KB
MxM	triple matrix multiplication	1,173.56KB
Radar	radar imaging	905.28KB
Shape	pattern recognition and shape analysis	1,284.06KB
Track	visual tracking control	744.80KB

Table 2. The latency values used in our experiments

Access Type	Non-secure	Secure
Read	25msec	42msec
Write	50msec	67msec

seed size as follows: (the total number of seed elements marked by the programmer)/(the total number of input elements). We performed experiments with different seed sizes.

The graph in Fig. 5 gives the secure and non-secure memory sizes (percentages) determined by our approach for different sizes of the seed set (as specified by the programmer), namely, seed sizes of 10%, 25%, and 50%. We see from these results that the average secure memory sizes (across all applications) are 29.6%, 55.7%, and 66.9% for the seed sizes of 10%, 25%, and 50%, respectively. Note that, if we conservatively assume that all data elements are secure (i.e., without any compiler analysis), their total sizes (as given in the last column of Table 1) would determine the required capacity

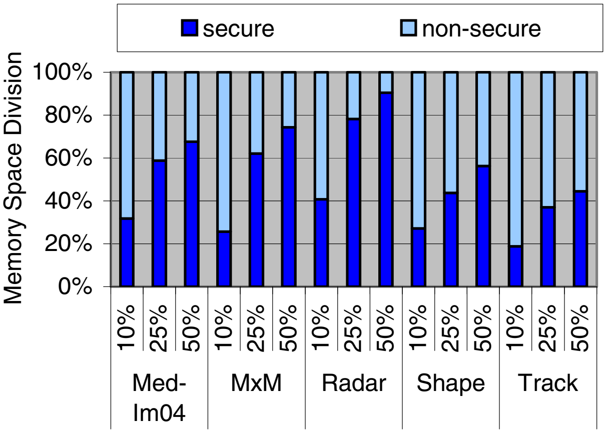

Fig. 5. Memory space division between secure and non-secure components

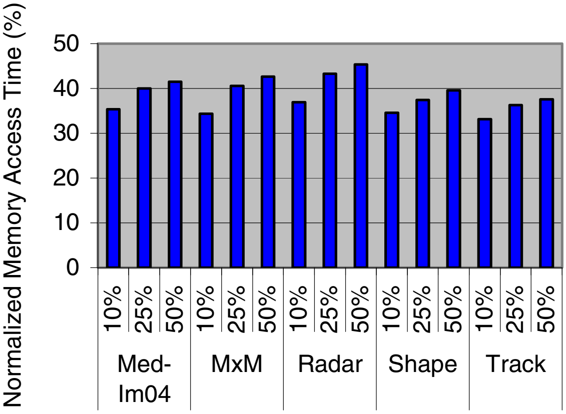

Fig. 6. Memory access time

of the secure memory (i.e., for each bar in Fig. 5, the secure portion would be 100%). In other words, through our compiler-directed approach, we are able to reduce the required secure memory size significantly. The graph in Fig. 6 illustrates how our approach impacts the memory access time. The experiments have been performed in a simulation environment that models a simple five-stage pipelined embedded machine. The values, again given for different seed sizes, are normalized with respect to the case when all data elements are stored in and accessed from the secure memory. We see that our approach reduces the memory access time of this naïve scheme by 64.8%, 60.2%, and 58.7% for the seed sizes of 10%, 25%, and 50%, respectively. Although not quantified here explicitly, one can also expect similar savings in energy consumption as well.

After having presented our secure memory size and performance results, we next focus on the code size increase due to our compiler-based approach. As mentioned earlier, this increase occurs due to the requirement that we have different types of load operations for different types of memories (i.e., secure vs. non-secure). The first bar for each benchmark in Fig. 7 gives the percentage increase in code size when our code re-ordering strategy explained in Section 3.2.2 is used. We see that the code size increase incurred (with respect to the code size where all the data is stored in and accessed from non-secure memory) is about 84% when averaged over all five benchmarks. On the other hand, the second bar for each benchmark represents the code size increase if we use our approach without code reordering (iteration group scheduling). In this case, we observe an average of 243% increase in code size. These results clearly emphasize the importance of our code reordering component.

It is to be noted that the number of secure elements typically determines the capacity of the secure memory in an embedded system. This size of the memory is an important consideration when the underlying secure memory employs additional fabrication steps such as metal shielding that add to the cost of the secure memory. While the approach presented so far is very effective in achieving a reduction in the number of secure elements, one can further reduce the secure memory space needed by considering the lifetimes of secure data elements.

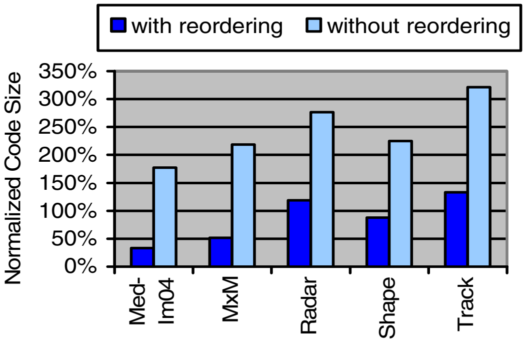

Fig. 7. Impact of code reordering

The main rationale behind the approach studied in this part of the paper is that not all the secure elements are needed for the entire duration of the program execution. In this part of our presentation, we evaluate a framework based on linear algebra to determine (and increase the number of) the cases where the lifetime of two secure array elements do not overlap with each other. When this happens, these two array elements can share the same location in the secure memory, thereby reducing the demand in the secure memory capacity. For array-based embedded applications, in a given loop nest, one can define the lifetime of an array element as the difference (in loop iterations) between the time it is first assigned (written) and the time it is last used (read). For a given array index a (which might be multi-dimensional), the start of its lifetime is referred to as $S(a)$, whereas the end of its lifetime is denoted using $E(a)$ - both in terms of loop iterations. Using these definitions, the lifetime vector for this array element can be given as $s = E(a) - S(a)$, where "-" denotes vector subtraction. Note that the lifetime of a is expressed as a vector as in general there might be multiple loops in the nest, and expressing lifetime as a vector allows the compiler to measure the impact of loop transformations on it. As an example, if an array element (that is accessed in a nest with two loops) is produced in iteration $(2\ \ 2)^T$ and consumed (i.e., last-read) in iteration $(6\ \ 7)^T$, its lifetime vector is $s = (6\ \ 7)^T - (2\ \ 2)^T = (4\ \ 5)^T$. It should be noted that before $S(a)$ and after $E(a)$ the secure memory location allocated to this array element could be used for storing another array element (which belongs to the same array or to a different array). Obviously, the shorter the difference between $E(a)$ and $S(a)$, the better, as it leaves more room for other secure elements.

As stated earlier in Section 3.2.1, the loops in an array-based program surrounding any statement can collectively be represented using a column vector (called the iteration vector) $I = (i_1\ i_2\ \dots\ i_n)^T$, where n is the number of the enclosing loops. Here, i_k is the k^{th} loop index from top. The loop range or affine bounds of these loops can be described by a system of inequalities, which define a polyhedron. The integer values that can be taken on by I collectively define the iteration space of the nest. In a similar fashion, data (memory) layout of an array can also be represented using a polyhedron. This rectilinear polyhedron, called the index space, is delimited by array bounds, and each integer point in it, called an array index, is represented using an index vector $a = (a_1\ a_2\ \dots\ a_m)^T$, where m is the number of dimensions of the array. Based on the iteration space and index space (data space) definitions, an array access (i.e., an array reference) can be defined as a mapping from iteration space to index space, and can be described as $GI + o$. Assuming a nest with n loops that accesses an array of m dimensions, in the expression above, I denotes the iteration vector, G is an $m \times n$ matrix (called the access matrix or the reference matrix, and o is an m-entry constant vector (called the offset vector) [26]. As an example, in a nest with two loops (i_1 and i_2) that contains array reference $X_1[i_1+2][i_2-3]$, G is two-by-two identity matrix and $o = (2\ \ -3)^T$.

The application of a loop transformation represented by a square, non-singular matrix T can be accomplished in two steps [26]: (i) re-writing loop body and (ii) re-writing loop bounds. For the first step, assuming that I is the original iteration vector and $J = TI$ is the new (transformed) iteration vector, each occurrence of I in the loop body is replaced by $T^{-1}J$ (note that T is invertible as the transformation must be one-

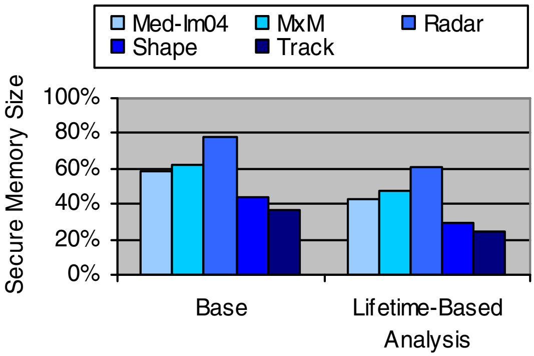

Fig. 8. Impact of lifetime analysis

to-one). In other words, an array reference represented by *GI+o* is transformed to $GT^{-1}J+o$. Determining the new loop bounds, however, is more complicated and, in general, may require the use of complex methods such as Fourier-Motzkin elimination (a method for solving an affine system of inequalities [26, 3]). One can see that application of a loop transformation changes the execution order of loop iterations and, consequently, the order in which array elements are accessed. As a result, a loop transformation *T* changes the lifetime vectors as well.

In more technical terms, let $s = I_e - I_s$ be the original lifetime vector for an array element, where I_s is the first iteration that accesses the array element, and I_e the last access. After applying *T*, we have $I_e' = TI_e$ and $I_s' = TI_s$. Now, we have:

$$s' = I_e' - I_s' = TI_e - TI_s = T\,(I_e - I_s).$$

That is, if *s* is the original lifetime vector, *s'* is the transformed lifetime vector. Our objective is then to select a suitable *T* such that $s' = (0\ 0\ 0\ \ldots\ 0\ 0\ 1)^T$ for as many array references that access secure elements as possible. In other words, we want to achieve the minimum lifetime vector. Note that, while a more sophisticated implementation can try other lifetime vectors as well (for *s'*) – as long as they are smaller than the original lifetime vector *s* – in this paper we restrict ourselves to $s' = (0\ 0\ 0\ \ldots\ 0\ 0\ 1)^T$. Obviously, a loop transformation (*T*) must also preserve the data dependences in the code. In our approach, when we determine a candidate loop transformation, we check whether it preserves data dependences in the code; if it does not, we drop it from consideration.

The memory space results with the lifetime analysis are presented in Fig. 8. All the applications in our experimental suite show significant improvements when the lifetime analysis for secure array elements explained above is used. That is, when applicable, the lifetime analysis of secure elements can be very effective in practice. On an average, using lifetime analysis reduces the secure memory size by 15.3% over our base approach that does not employ any lifetime analysis.

5 Related Work

Several prior efforts address the problem of secure remote execution using a circuit based model as part of the general problem of confidentiality [1,9,27,28]. The integrity of the computation is the ability of the circuit owner to verify the correctness of the execution of the circuit. This problem has been widely studied from the general reliability angle but not from the viewpoint of a malicious server. In a framework proposed by [23,24], the privacy of a function is assured by an encrypting transformation on that function. Yee [29] suggested proof-based techniques in which the untrusted host has to forward a proof of correctness of execution together with the result. In [17,18], a function is encrypted using error coding and sent to the untrusted host that provides the cleartext input. The enciphered output generated by the host is then sent back to the original host, where it is decrypted and the result is verified. The decrypted result matches the result which would have been obtained if the original function had been directly applied to the cleartext input. The authors argue for the need of tamper proof hardware (TPH) to store and provide the control flow between the numerous functions that make up a program. Control flow is located on the TPH and is supplied to the untrusted host. In contrast to these studies, our work is more oriented towards using secure memory in an embedded system. Techniques similar to our use of the Omega Library have been suggested in [19] and [20] in an entirely different context (optimizing data cache locality and interprocessor communication). Slicing [25] is a well-studied program analysis technique that can be used for different optimization goals. In comparison to these studies, our approach targets secure access to data with minimal performance and power overheads.

6 Conclusions

The need for protecting sensitive data from illegal access has resulted in the adoption of secure memories in embedded systems such as smartcards. It is anticipated that securing data will become important for other embedded systems and applications as well. This is because ensuring data security can impose overheads such as increased memory cost, code size, reduced performance or higher power consumption. This work focuses on transforming code structures, with the help of a polyhedral tool, to minimize these overheads when selectively protecting sensitive data identified by the programmer. Experimental results demonstrate that the proposed approach provides required data security by keeping the performance and code size overheads under control.

References

[1] M. Abadi and J. Feigenbaum. Secure circuit evaluation. Journal of Cryptology, 2(1):112, 1990.

[2] S. P. Amarasinghe, J. M. Anderson, C. S. Wilson, S.-W. Liao, B. R. Murphy, R. S. French, M. S. Lam and M. W. Hall. Multiprocessors from a Software Perspective, IEEE Micro, June 1996.

[3] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 39-50, 1991.

[4] R. J. Anderson, M. G. Kuhn. Low Cost Attacks on Tamper Resistant Devices. In M. Lomas, et al. (eds.), Security Protocols, In Proceedings of the 5th International Workshop, LNCS 1361, pp. 125-136, Springer-Verlag, 1997.

[5] Atmel Secure Memories. http://www.atmel.com/products/SecureMem/.

[6] C. Collberg, C. Thomborson, and D. Low, A Taxonomy of obfuscating transformations. Technical Report #148, Department of Computer Science, University of Auckland, July 1997.

[7] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In Proceedings of the 25th ACM Symposium on Principles of Programming Languages, pages 184-196, January 1998.

[8] J.-F. Dhem and E. Faber. Built-in hardware security: smart cards and crypto-processors. Embedded tutorial. In Proceedings of International Conference on Computer Design, 2001.

[9] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In Proceedings of the 19th Annual ACM Symposium on Theory of Computing,} pages 218--229, New York City, May 1987.

[10] F. Hohl. An approach to solve the problem of malicious hosts. Universitaet Stuttgart Fakultaet Informatik, Bericht Nr. 1997/03, 1997.

[11] Infineon Technologies. Security Chips and ICs. http://www. infineon.com/products

[12] W. Jansen and T. Karygiannis. Mobile agent security. NIST Special Publication, 800-19, http://csrc.nist.gov/mobileagents/publication/sp800-19.pdf, August 1999.

[13] W. Kelly and W. Pugh. *A* Framework for Unifying Reordering Transformations. Technical Report, University of Maryland Institute for Advanced Computer Studies. Dept. of Computer Science, Univ. of Maryland, April 1993.

[14] P. Kilpatrick, D. Crookes, and M. Owens. Program slicing: A computer aided programming technique. In Proceedings of the Second IEEE / BCS Conference on Software Engineering, pp. 602-604, 1988.

[15] C. Linn and S. Debray. Obfuscation of wxecutable code to improve resistance to static disassembly. In Proceedings of the 10th ACM Conference on Computer and Communication Security, October 2003.

[16] S. Loureiro. Mobile Code Protection, Ph. D.Dissertation, Institut Eurecom, 2001.

[17] S. Loureiro, L. Bussard, and Y. Roudier. Extending tamper-proof hardware security to untrusted execution environments. In Proceedings of CARDIS, 2002.

[18] S. Loureiro and R. Molva. Function hiding based on error correcting codes. In Proceedings of the International Workshop on Cryptographic Techniques and Electronic Commerce, pages 92--98, City University of Hong-Kong, July 1999.

[19] W. Pugh and E. Rosser. Iteration space slicing and its application to communication optimization. In Proceedings of the International Conference on Supercomputing, 1997.

[20] W. Pugh and E. Rosser. Iteration space slicing for locality. In Proceedings of Languages and Compilers for Parallel Computing, 1999.

[21] W. Rankl and W. Effing. Smart Card Handbook. pp.71,421. John Wiley and Sons, 1997.

[22] J. Quisquater and D. Samyde. Electromagnetic Analysis: Measures and Countermeasures for smart cards. E-Smart 2001, LNCS 2140, pp. 200-210, 2001.

[23] T. Sander and C. F. Tschudin. Towards mobile cryptography. In Proceedings of the 1998 IEEE Symposium on Security and Privacy, pp. 215--224, Oakland, California, May 1998.

[24] T. Sander and C. Tschudin. On software protection via function hiding. In Proceedings of the Second Workshop on Information Hiding, Portland, Oregon, USA, April 1998.

[25] M. Weiser. Program slicing. IEEE Transactions on Software Engineering, pages 352-357, July 1984.

[26] M. Wolfe. High Performance Compilers for Parallel Computing, Addison-Wesley Publishing Company, 1996.

[27] A. C. Yao. Protocols for secure computations. In Proceedings of the IEEE Symposium on Foundations of Computer Science, pages 160--164, Chicago, 1982.

[28] A. C. Yao. How to generate and exchange secrets. In Proceedings of the IEEE Symposium on Foundations of Computer Science, pages 162--167, Toronto, 1986.

[29] B. Yee. A sanctuary for mobile agents. Technical Report CS97-537, Department of Computer Science and Engineering, UCSD, April 1997.

[30] X. Zhang and R. Gupta. Hiding Program Slices for Software Security. First Annual IEEE/ACM Symposium on Code Generation and Optimization, pp. 325-336, San Francisco, CA, March 2003.