

# Securing Disk-Resident Data through Application Level Encryption

Ramya Prabhakar, Seung W Son, Christina Patrick, Sri Hari Krishna Narayana  
and Mahmut Kandemir

Department of Computer Science and Engineering  
Pennsylvania State University  
University Park, PA – 16801  
{rap244,sson,patrick,narayana,kandemir}@cse.psu.edu

## Abstract

*Confidentiality of disk-resident data is critical for end-to-end security of storage systems. While there are several widely used mechanisms for ensuring confidentiality of data in transit, techniques for providing confidentiality when data is stored in a disk subsystem are relatively new. As opposed to prior file system based approaches to this problem, this paper proposes an application-level solution, which allows encryption of select data blocks. We make three major contributions: 1) quantifying the tradeoffs between confidentiality and performance; 2) evaluating a reuse distance oriented approach for selective encryption of disk-resident data; and 3) proposing a profile-guided approach that approximates the behavior of the reuse distance oriented approach. The experiments with five applications that manipulate disk-resident data sets clearly show that our approach enables us to study the confidentiality/performance tradeoffs. Using our approach it is possible to reduce the performance degradation due to encryption/decryption overheads on an average by 46.5%, when DES is used as the encryption mechanism, and the same by 30.63%, when AES is used as the encryption mechanism.*

## 1 Introduction and Motivation

Many computing systems today generate, manipulate, transmit and store enormous amounts of data. End-to-end security refers to the process of guaranteeing confidentiality, integrity and availability of data during these activities. While several protocols such as IPsec [14] and SSL [7] exist and are widely used to secure data while in transit, efforts for securing data in rest (e.g., when stored in a disk subsystem) are relatively new. Lack of security provisions for disk-resident data can invite intrusions and other types of security related attacks, leading to significant damage to

computing systems.

In the context of storage systems, confidentiality refers to ensuring that disk-resident data are accessible only to those authorized to have access, and is one of the cornerstones of storage security [16]. One way of ensuring confidentiality is to employ *encryption*. The idea is to keep data in the disk subsystem in an encrypted form and decrypt only when it needs to be processed. Several file systems, which can collectively be called cryptographic file systems [2, 4, 8], provide support for encrypting disk-resident data and related provisions such as key management and key sharing. One of the problems with the file system based approaches to confidentiality is that the file system policies are generally heavy-handed with too much overhead and are not flexible enough so that they can be tuned based on the needs of individual applications.

While encrypting disk-resident data can help us ensure its confidentiality, it can also bring significant performance overheads at runtime, because encrypted data have to be decrypted to be operated on. While we want to keep data in the storage system in an encrypted form as much as possible, we may not want to pay a very heavy performance penalty for this. Prior research already pointed out this tradeoff between security and performance in the context of storage systems. However, it is also important to quantify this tradeoff and understand how it can be exploited using access characteristics of data.

Focusing on scientific applications, this paper proposes and evaluates an application-level approach for ensuring confidentiality of disk-resident data. While confidentiality of disk-resident data is important in many application domains, our focus here is on scientific computing domain, where multiple applications share the same disk subsystem, but owners of an application may not want others to access the contents of some of their data. Specifically, this paper makes the following contributions: Firstly, we present experimental data showing the tradeoffs between

confidentiality and performance. In particular, we report results from our experiments that compare two extreme approaches: never encrypt/decrypt (NED) and always encrypt/decrypt (AED). The results from these experiments motivate for selective encryption of disk-resident data. Secondly, we evaluate the potential for an approach that strikes a balance between the conflicting confidentiality and performance requirements. The goal of this approach is to explore the options between NED and AED. The proposed approach is based on the concept of *reuse distance* and applies encryption/decryption to a data block only if the reuse distance for that block is less than a certain value. Lastly, we describe how a profile-guided approach can be used for approximating the potential of the reuse based approach by deciding for each static I/O call in the program code whether it should encrypt (in case of writes) or decrypt (in case of reads) data, or use plain access (without any encryption/decryption). Our experiments with five applications that manipulate disk-resident data clearly show that our approach enables us to study the confidentiality/performance tradeoff.

The rest of this paper is organized as follows. Section 2 discusses related work on storage security. Section 3 describes our experimental setup and metrics of interest. Section 4 presents a quantitative analysis of the confidentiality-performance tradeoffs using two extreme schemes (NED and AED). Section 5 explains our reuse-oriented approach for analyzing these tradeoffs. Section 6 explains a profile-guided approach and discusses how it can be used for favoring confidentiality over performance or vice versa. Section 7 concludes the paper by summarizing our major observations, and giving a brief outline of future work.

## 2 Related Work

Many systems have been proposed to address the security related issues in modern file systems. To ensure confidentiality in the file system, most of these systems include several features for controlling access to files unless specifically authorized. Since these mechanisms do not protect data stored on disk, they offer weak security. Other systems, such as NASD (Network Attached Secure Disks) [10], iSCSI with IPSEC [14], NFS (Network File System) with secure RPC [23], AFS (Andrew File System) [11], and SFS (Secure File System) [17], provide stronger security by encrypting data on the wire as well as using considerably stronger authentication mechanisms. These encrypt-on-wire type of file systems protect the data from adversaries during communication. Therefore, all communication is secure, but the data is stored on disks in plain text. Since the data must be encrypted before sending it through the network, there is also an inherent encryption overhead associated with these approaches. There are also additional

**Table 1. Characteristics of Applications.**

Name	Source	# I/O Op	I/O Size(MB)
SWIM	SPEC2000 [15]	622900	85.3
MGRID	SPEC2000 [15]	600288	74
LU	LINPACK [24]	1431680	175
MxM	NAS Kernels [3]	787456	384.5
TSF	Perfect Club [1]	519176	126.75

overheads on the disk because the data must be decrypted before storing it on the disk.

While the file systems explained so far have focused on adding encryption on the wire, i.e., access control and protecting data in transit, several recent studies considered file systems that are designed to protect data stored on the disk. The representative efforts include CFS (Cryptographic File System) [2], Cryptfs [27], TCFS (Transparent Cryptographic File System) [4], EFS (Encrypted File System) [6], Cepheus [8], SFS-RO (SFS-Read Only) [9], and Plutus [13]. These file systems provide an end-to-end data confidentiality; that is, they encrypt data before it is sent over the network and data is stored in encrypted format in files. Only the users of that data (not the servers) have a privilege to decrypt it. Since all data is stored in an encrypted form on the disk, such systems ensure that the compromised servers have no control over the stored data. The main drawback of this encrypt-on-disk approach is that they may incur significant performance degradation on the client machines because all encryption/decryption burden is imposed on them. Strong security with poor usability and/or performance can make a system less attractive. Secure storage systems and cryptographic softwares are used less than one would expect in practice, mainly due to the overheads involved in encryption/decryption [25],[12].

Our work is different from these prior efforts on securing disk resident data because we focus on application level encryption to provide confidentiality, based on application needs. This allows our approach to strike a balance between two extreme options: poor security/good performance and poor performance/good security.

## 3 Experimental Setup and Metrics of Interest

We present experimental results obtained on a SUN Blade1000 machine running Solaris 10. The data arrays manipulated by our applications are stored in a remote file system accessible through NFS (Network File System). We used eight disks as a RAID for configuring the remote file system. We compiled our applications for our client system using gcc 3.4.3 under Solaris. The run-time layer of the encryption/decryption function is written in a separate file and built into a library, which is linked to an executable during compilation step.

As mentioned earlier, we focus on scientific applications that process disk-resident data sets. The default file cache used in our experiments can hold one data block at a time (we also report results with different file cache sizes). Our evaluation workload consists of five scientific applications: swim, mgrid, lu, mxm and tsf. Table 1 presents important characteristics of these applications. These applications are modified from their original forms such that they operate on disk-resident data. Apart from this change, however, the applications maintain the same behavior as their original versions. Therefore, one may think of these applications as I/O-intensive versions of the original codes. It needs to be mentioned however that we carefully hand-tuned the resulting codes so that they do not perform any unnecessary I/O (similar to the way an I/O-intensive application would be written by an experienced programmer). The files are written to read from the disk subsystem at a block granularity. The block sizes used in our experiments range from 512 bytes to 1024 bytes, depending on the application.

In our work, we focus on the following important metrics which help us study the tradeoff between confidentiality and performance.

- *I/O Time (IOT)*: This is the I/O latency when security (encryption/decryption) overheads brought by the scheme being evaluated are also included. Specifically, it includes the time spent in I/O plus the time spent in performing encryption/decryption and related activities and, in our presentation its value is normalized with respect to a *base version*. The I/O time contribution to overall execution latency in these benchmarks is between 64.2% and 96.6%. The base version does not employ any security features while reading and writing disk blocks. That is, it reads and writes the data blocks in plain form.<sup>1</sup>
- *Vulnerability Factor (VF)*: Percentage of data stored in plain text on disk subsystem at any given point in execution. In this work, we consider two variants of this metric. The *average vulnerability factor (AVF)* is the VF value, averaged over all execution points (all cycles). On the other hand, the *maximum vulnerability factor (MVF)* is the maximum value of the VF witnessed during the entire execution of the application.

Clearly, in the ideal case, we want to reduce the values of both IOT and VF as much as possible.

<sup>1</sup>It needs to be emphasized that this base version is different from NED to be discussed shortly, as it does not perform any encryption/decryption during execution, whereas NED performs encryption/decryption at the first read and last write operations.

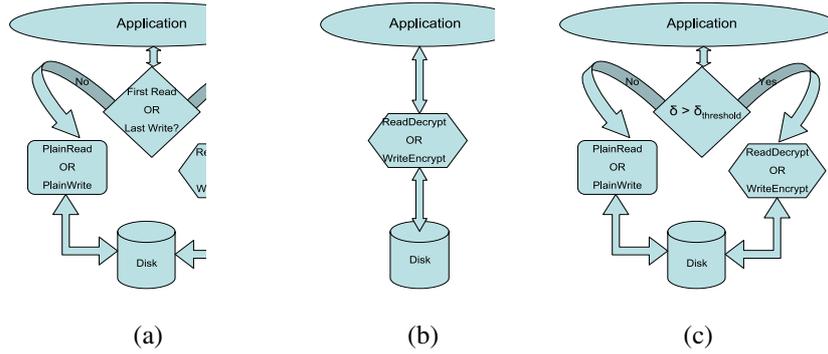
## 4 Quantitative Analysis of the Confidentiality-Performance Tradeoff

Applications using file-system based end-to-end security mechanisms in storage systems consistently incur the performance overhead of decryption/encryption on every I/O operation that go to the disk subsystem. In particular, frequently accessed application data is normally subject to multiple cycles of decryption and encryption. On one hand, we want to improve confidentiality by keeping data on the disk subsystem in encrypted form as much as possible; i.e., we want to reduce the value of the VF metric. On the other hand, we also want to minimize the overheads associated with encryption/decryption; i.e., we want to reduce the value of IOT metric. In this section, we evaluate the following two extreme schemes:

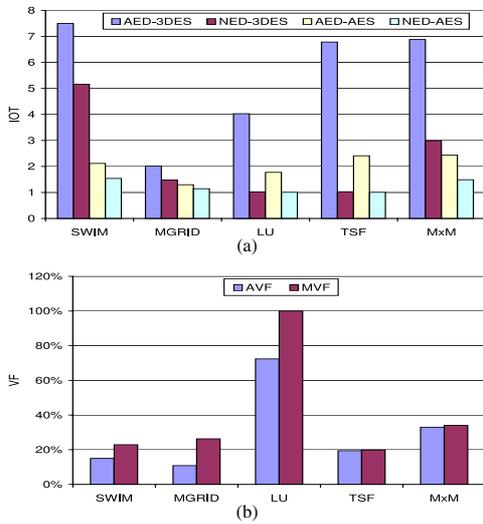
- **AED** : Always Encrypt/Decrypt refers to the default scheme where the data is encrypted and decrypted every time it is used. As depicted in Figure 1(b), every read from the disk requires data to be decrypted and similarly every write to the disk requires data to be encrypted. Although this scheme ensures complete end-to-end confidentiality of data (the best VF value possible), there is significant overhead involved in encrypting and decrypting data every time it is written to and read from the disk subsystem. As a result, this scheme incurs the largest possible performance overhead.
- **NED**: Never Encrypt/Decrypt refers to a scheme where the data is never encrypted or decrypted between its first read and last write. However, as depicted in Figure 1(a), this scheme requires the first read to a particular data to be a read with decryption as data in the disk is assumed to be initially encrypted and the last write requires encryption as data in the disk needs to be secured. As far as performance is concerned, this scheme scores far better than AED; however, it exposes data for the maximum duration of time.

We now present experimental results quantifying the tradeoffs between confidentiality and performance when these two extreme approaches are used. Specifically, we quantify our parameters of interest, IOT and VF, under these approaches. For each benchmark in our experimental suite, we implemented and performed experiments with two different encryption algorithms: Triple-DES (denoted as DES in this paper) [20] and AES [22]. All the results presented in Figure 2 are normalized against the base version (explained in Section 3).

The graph in Figure 2(a) gives the value of the IOT metric for each application under the two schemes discussed above. The absolute IOT values measured for the base version are 2873.24, 2678.45, 5676.32, 5940.22 and



**Figure 1. Different approaches to confidentiality of the disk-resident data. (a) NED, (b) AED and (c) Our data reuse-oriented approach.**



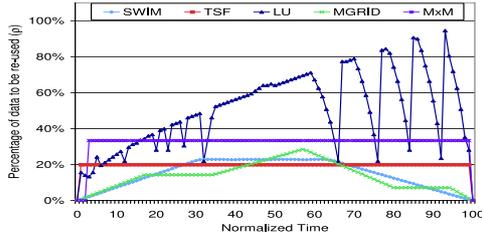
**Figure 2. (a) Normalized Time (IOT) with different schemes. (b) AVF and MVF of our applications with NED scheme.**

3453.79 msec for the applications swim, mgrid, lu, mxm and tsf, respectively. We can draw several conclusions from these results. First, we see that enforcing security on every read/write operation of data blocks with DES algorithm can increase IOT over the base version by 5.4 times when averaged across all five applications. The security overheads when we use the AES algorithm, however, increases the IOT by 2 times on average, because of the fact that AES is a much faster encryption algorithm than the DES algorithm. On the other hand, with the NED scheme, the security overheads increase IOT only by 2.3 times and 1.2 times using DES and AES, respectively, when averaged across our applications. We also observe that the NED-DES scheme de-

creases IOT over the AED-DES scheme by 84% and 74% in case of tsf and lu, respectively. Such a large improvement in performance is possible because the tsf and lu applications manipulate the same data in several loop nests repeatedly (high data reuse). However, in applications such as swim, mgrid and mxm, the improvement in performance of NED-DES scheme over the AED-DES scheme is by 31%, 26% and 56%, respectively. This is relatively lesser than the improvements observed with tsf and lu, because of the reduced fraction of data repeatedly used by these applications (low data reuse).

It should be noted that the performance gains presented for NED over AED are achieved at the expense of certain degree of confidentiality of intermediate data stored on the disk subsystem. That is, the NED scheme exposes data for the longest duration so as to reap maximum benefits on performance. In our next experiment, we gauge the percentage of data stored in plain form temporarily during the execution using our metrics for AVF and MVF.

The graph in Figure 2(b) plots the values of the AVF and MVF metrics for each application under the NED scheme. Notice that the results are the same irrespective of the encryption algorithm (DES or AES) used. Under AED scheme, data is always stored encrypted; so, we present results only for the NED scheme. We see from these results that AVF and MVF averages about 30% and 40%, respectively, across all five applications. Note that this is the result obtained after exposing data for the maximum duration. In case of MxM for example, only the result matrix (one out of three) which is written is exposed, leading to an AVF of 33.33%, while in lu the entire array is modified in place (by reads and writes while array is on disk) and hence AVF and MVF are about 72% and 100%, respectively, whereas for other cases with mixed behavior like swim, tsf and mgrid, AVF and MVF are about 15% and 20%, respectively.



**Figure 3. Data reuse potential in our applications. Different applications have different amounts of data reuses and also their reuse characteristics change during execution.**

Now, based on the results presented in Figures 2(a) and 2(b), one can clearly see the tradeoff between confidentiality and performance. In the rest of this paper, we explore possibilities between these two extreme schemes. Our main tool for this exploration is *data reuse*, as explained in the next section.

## 5 Data Reuse-Oriented Approach to Confidentiality

We observe that a significant fraction of data read/written by applications have reuse. In particular, scientific applications have distinctive and variable patterns of data reuse. This observation is crucial for reducing the performance overheads of encryption/decryption. By effectively utilizing the data reuse behavior of applications it may be possible to reduce the encryption/decryption overhead. More specifically, if a data block has a high degree of reuse, it can be stored in plain form to reduce access costs. If, on the other hand, it does not have a high degree of reuse, it can be stored in encrypted form.

In order to study the data reuse available in our scientific applications, we measured a program characteristic, which we call the *reuse potential* ( $\rho$ ), which is defined as the ratio between data previously read/written by the application which has a reuse and total data used by the application. Figure 3 plots the variation of  $\rho$  against normalized execution time. Note that, while this is certainly not the only parameter for quantifying data reuse, it is a suitable parameter for our purposes. We can see from Figure 3 that these applications have significantly different reuse potentials and, also data reuse changes, in a given application, during the course of execution. A high  $\rho$  at any given point, indicates that a large amount of data that has been previously used by the program would be reused in future. A high value of  $\rho$  is possible under two circumstances: 1) Data is used early in the program and reused late and 2) Same data is frequently reused over and over again during the execution

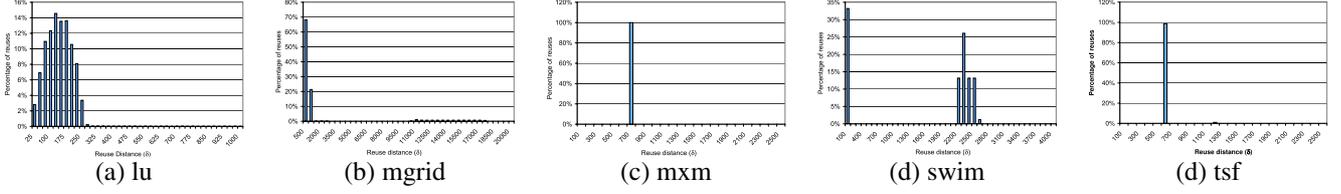
of the program. To distinguish between these two cases, we also define another parameter,  $\delta$ , which is the *reuse distance*; i.e., the number of intervening I/O requests between two requests to the same data block (same offset and file). Clearly,  $\delta$  also varies during the program execution. The data access pattern of applications gives rise to various values of reuse distance ( $\delta$ ) during the course of execution of the program. To characterize the variation of reuse distances in our applications, we experimentally determined the variation in the number of reuses across fixed intervals of reuse distance ( $\delta$ ). This distribution is depicted in Figure 4. As can be seen from this figure, lu and swim exhibit a more uniform distribution of reuse distances, whereas mgrid, tsf and mxm have a highly concentrated distribution in few ranges of  $\delta$ .

Clearly, a higher  $\delta$  value indicates higher vulnerability of data if left insecure (in plain form). Therefore, applications can choose to selectively secure such data. On the other hand, the data that has a high value of  $\rho$  but has a lower  $\delta$  can be temporarily written/read in the plain form to reduce performance overheads.

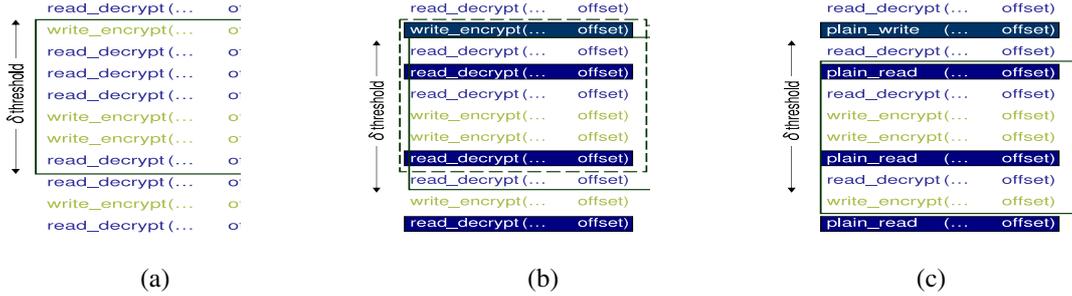
Recall that, in NED, data is exposed for the entire duration in a manner which is oblivious of the magnitude of reuse distance. This results in high VF values, as shown in Figure 2(b). In the rest of this section, we introduce a *data reuse-oriented approach* and quantify its behavior in terms of IOT and VF. While this approach is not directly implementable (as it requires knowledge on future I/O patterns), it gives us potential of a reuse-based approach to confidentiality of disk-resident data. Later (in Section 6), we present an implementable approach that approximates the behavior of this reuse-oriented approach. The reuse oriented approach, shown pictorially in Figure 1(c), uses a threshold value of reuse distance ( $\delta_{threshold}$ ). If the reuse distance  $\delta$  for the data being written is greater than  $\delta_{threshold}$ , data is encrypted and written in order to minimize the VF. Otherwise data is written in plain form and any subsequent reads to the same data are performed in the plain form.

To arrive at a  $\delta_{threshold}$  value we first start by defining two ratios to account for both *IOT* and *VF*. The first is the *Performance Ratio* and the second is called the *Security Ratio*. For any value of  $\delta$ , called  $\delta_k$ , the performance ratio is defined as the ratio of the *IOT* value of the most secure case (value with lowest value of  $\delta$ ) to the *IOT* with  $\delta_k$  as the reuse distance. The performance ratio in effect measures the effect of increasing the increasing performance afforded by increasing the reuse distance. The performance ratio is a value that is greater than or equal to 1.

The security ratio is defined for a reuse distance,  $\delta_k$ , as the ratio of the portion secure (unexposed) data when a reuse distance of  $\delta_k$  is used to the portion of secure data in the most insecure case (with the highest value of  $\delta$ ). The security ratio measures the benefit to the security of data



**Figure 4. Distribution of reuses across intervals of  $\delta$ . Values on x-axis represent ranges of reuse distance ( $\delta$ ) for various applications, and the y-axis captures the percentage of reuse with particular distances.**



**Figure 5. Look-ahead window for the reuse-oriented encryption/decryption scheme. *write\_encrypt* and *read\_decrypt* correspond to encrypted write and decrypted read, respectively. *plain\_write* and *plain\_read* are write and read operations, respectively, without any encryption/decryption. (a) Sliding window of size  $\delta_{threshold}$ . (b) Offset of evicted write instruction compared with offsets of other read instructions in the window. (c) Instructions with matching offsets (if any) are replaced by *plain\_read*, and the evicted instruction is itself made a *plain\_write*.**

stored in memory when a lower value of reuse distance is used compared to the most insecure case. It is a value that is at least 1.

Using the performance ratio and the security ratio, a suitable combined metric value can be determined by dividing performance ratio by security ratio for a reuse distance of  $\delta_k$ . This value represents the unit gain in performance ratio for every unit loss in security ratio. If the combined metric is less than 1, it means that the performance gained in exploring reuse at  $\delta_k$  is not proportionate to the increased exposure of data. A combined metric of exactly 1 means that the reuse distance at which this occurs allows the application to gain in terms of performance what is lost in term of security. The ideal case is when the combined metric is  $> 1$ , as this indicates a performance gain that is disproportionately larger than the security loss. A suitable  $\delta_{threshold}$  value is determined as the value of  $\delta_k$  for which combined metric is  $> 1$ .

To evaluate the values of our metrics with this reuse oriented approach, we implemented it as a library which we call *reuse-window library*. The reuse-window library maintains a *sliding look-ahead window* of size  $\delta_{threshold}$ ,

as shown in Figure 5(a). All I/O calls in our applications were instrumented such that they call the reuse-window library. The library in turn evicts the first I/O call in the window and appends the latest I/O call to the reuse-window. When a write call is being evicted from the reuse-window, its offset is compared with the offsets of the read instructions present within the window, as pictorially depicted in Figure 5(b). If a match is found, the particular offset is marked to be in the plain form (this means that the reuse distance for that disk block is short). Any subsequent read (before a *write\_encrypt*) to the same offset is interpreted as a *plain\_read*. Any other read/write is interpreted as a *read\_decrypt*/*write\_encrypt* as shown in Figure 5(c).

Clearly,  $\delta_{threshold}$  is a critical parameter in the reuse-oriented approach. As explained before, every program exhibits a characteristic pattern of data reuse with a lot of variation in the reuse distance,  $\delta$ . Therefore, in the remainder of this section, we evaluate the variation of our metrics IOT and VF for different values of  $\delta_{threshold}$  using this reuse-oriented approach.

We present results for a range of values for  $\delta_{threshold}$  for every application in our experimental suite. It is important

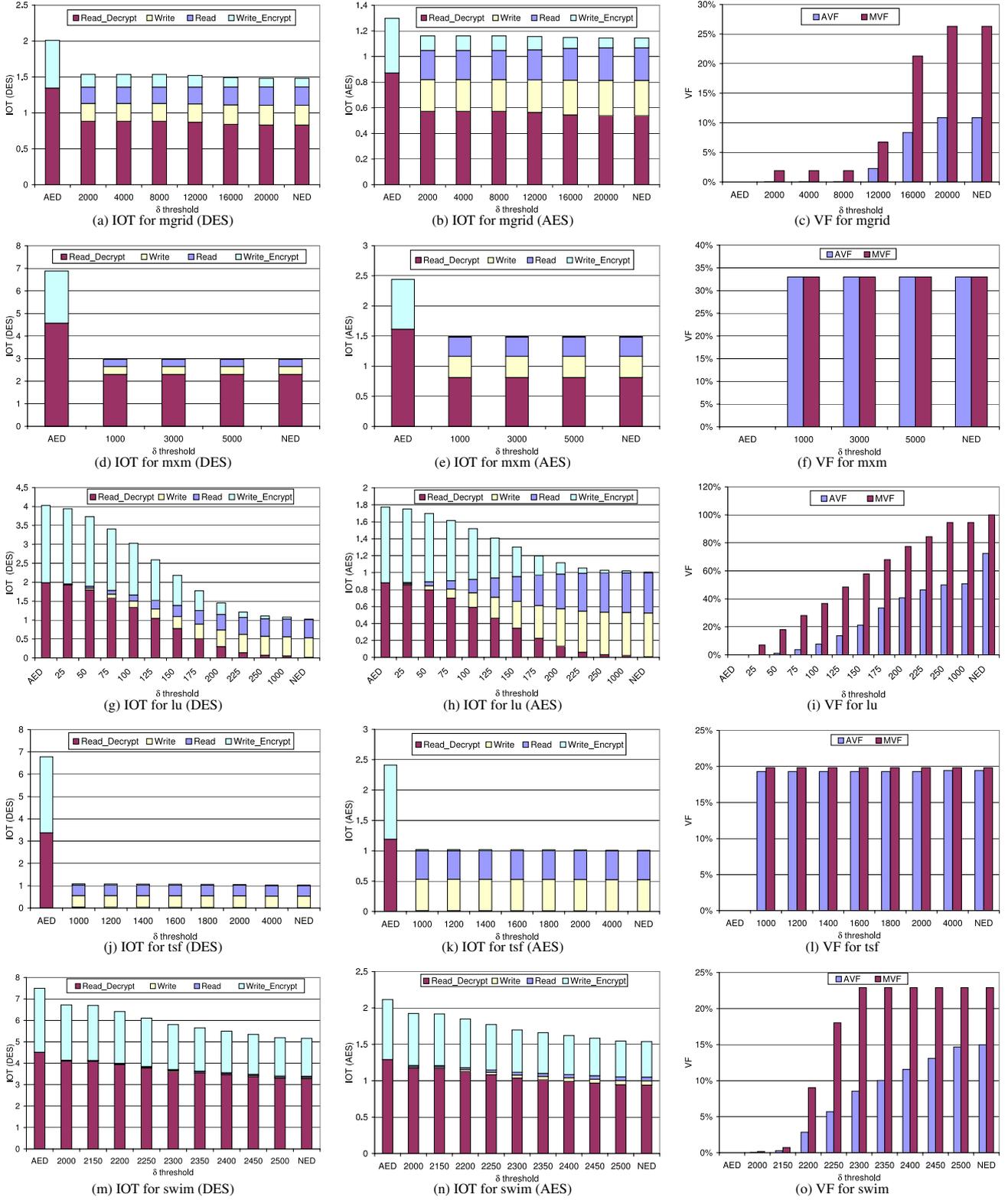


Figure 6. IOT and VF variations with different  $\delta_{threshold}$  values.

to note that whenever  $\delta_{threshold}$  is increased, say from  $\delta_1$  to  $\delta_2$ , all reuses that were exploited for  $\delta_1$  are exploited for  $\delta_2$  as well. With a threshold of  $\delta_2$  ( $\delta_2 > \delta_1$ ), we can exploit some additional reuses that fall in the distance between  $\delta_1$  and  $\delta_2$ . Therefore, IOT either remains the same (in case no additional reuse is detected) or decreases with increase in  $\delta_{threshold}$ . Figures 6(a) through 6(o) show the variation of IOT and VF for our applications with the different values of  $\delta_{threshold}$ . On an average, we observe an improvement of 21.8%, 45.1%, 24.8%, 84.4% and 56.5% in performance (IOT metric) over the AED scheme for swim, lu, mgrid, tsf and mxm, respectively, with DES as the encryption algorithm. Also, on an average, we observe an improvement of 19.1%, 26.1%, 11.0%, 57.9% and 38.9% in performance over the AED scheme for swim, lu, mgrid, tsf and mxm, respectively, with AES as the encryption algorithm. Similarly, with an increase in  $\delta_{threshold}$ , vulnerability factor VF increases due to increased duration of exposure of data. On an average, we observe an increase of 30.13% in AVF and 33.80% in MVF over that in AED across the applications.

For certain programs like lu and swim, the variation of IOT and VF is continuous. The continuous variation is because of the uniform distribution of the data reuses across the different reuse distances. For other programs, like mxm, tsf and mgrid, most of the reuses occur at fixed reuse distances (see Figure 4), and consequently, increasing  $\delta_{threshold}$  beyond a certain value does not bring any additional benefits. For instance, a  $\delta_{threshold}$  of 2000 improves the performance of mgrid, using DES as the encryption mechanism, by 23.6% (over AED), this causes a MVF increase of only 1.9%. We could explain this behavior by observing that most reuses in case of mgrid, happen at reuse distances lesser than 2000, as shown in Figure 4(b). This result clearly shows that, by sacrificing only a little confidentiality, our reuse-oriented approach can reduce the performance overhead brought by AED dramatically. Beyond this value (2000), increasing  $\delta_{threshold}$  to even 20000, improves the performance by only 2.72%, for an increase of 24.36% in the vulnerability (MVF). However, in lu, due to the uniform distribution of reuses, increasing  $\delta_{threshold}$  from 250 to 1000 continuously leads to gain in performance. Again, since lu has few reuses at  $\delta$  greater than 1000 (as seen from Figure 4(a)), the improvement in performance by increasing  $\delta_{threshold}$  beyond this value is negligible. Therefore, it is important to select a reasonable range of  $\delta_{threshold}$  for an application based on its reuse characteristics, specifically, the distribution of reuses with different reuse distances.

Our approach is also a very promising option when compared to NED. Again, focusing on mgrid, we see that working with  $\delta_{threshold}$  of 2000 reduces the AVF and MVF values by 99.4% and 92.6%, respectively, over the NED scheme, while generating a similar IOT value as the latter. Based on this discussion, we can conclude that our data

reuse-oriented approach can be used for either 1) Obtaining a much better performance than AED with little confidentiality sacrifice, or 2) Obtaining a much better confidentiality than NED with similar performance.

So far in our experiments, for each application, we used a file cache that can hold a single data block at a time. We now report results with different file cache sizes. To evaluate the effectiveness of our approach in presence of larger file caches, we performed additional experiments by varying the default size of the file cache. We report results of the variation of our metrics, IOT and VF, with different values of  $\delta_{threshold}$ . It is important to emphasize that our reuse based approach supplements the performance benefits achieved using a larger file cache.

Figures 7(a) and 7(b) depict the variation of AVF with respect to the file cache size in applications tsf and lu, respectively. As the file cache size (FCS) increases, there is an increase in AVF with increase in  $\delta_{threshold}$ . This is because the file cache buffers the data, possibly residing on disk in plain form, thereby increasing the duration for which the data is kept in plain form on disk.

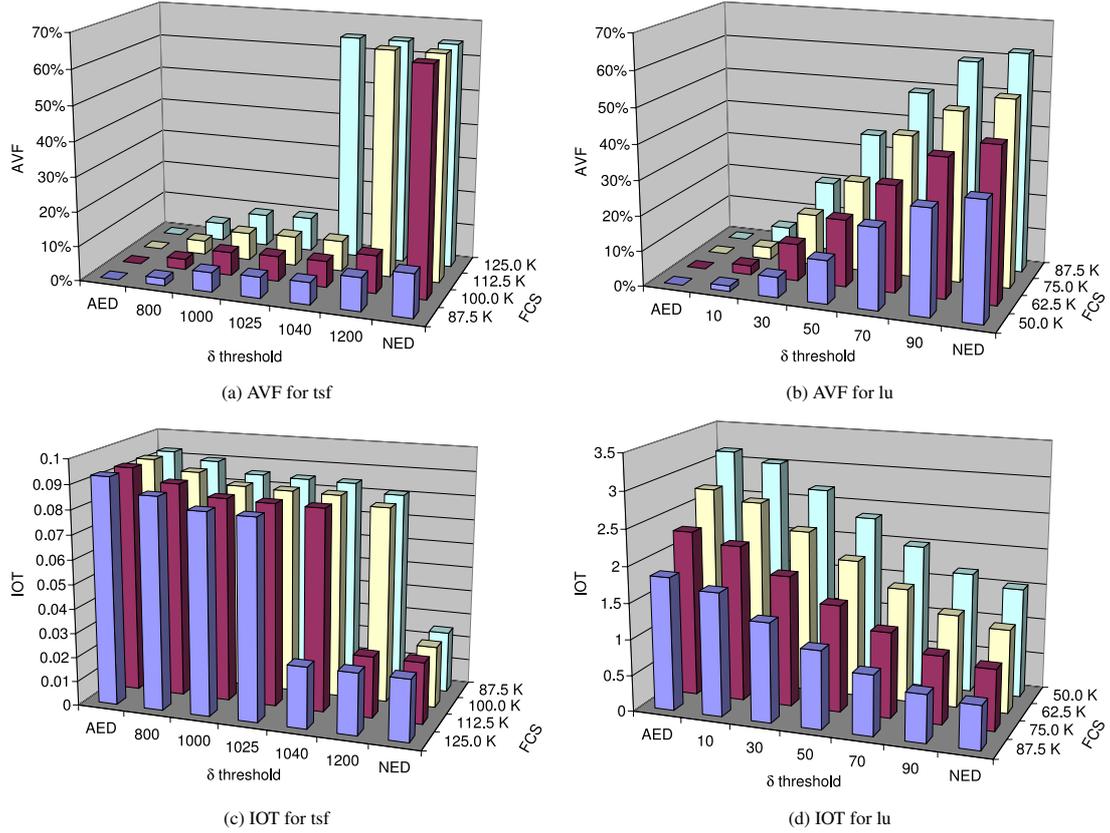
Figures 7(c) and 7(d) present the variation of IOT with respect to file cache size (FCS) in our applications tsf and lu, respectively. We observe that, with an increase in the file cache size (FCS), performance continuously improves (IOT decreases) as  $\delta_{threshold}$  is increased. The file cache temporarily stores the recently-accessed disk data. This reduces the total number of disk accesses made by the application, thereby reducing IOT. Additionally, an increase in  $\delta_{threshold}$  reduces the overhead of encryption/decryption on some file cache misses, contributing to further improvement in performance.

The results presented in this section clearly show that our data reuse-oriented approach is useful in analyzing the options between the two extreme schemes, AED and NED. In particular, by making experiments with different  $\delta_{threshold}$  values, a designer can determine the most suitable value for a given application under specified performance and confidentiality requirements.

## 6 Profile-Guided Approach to Confidentiality

While the reuse oriented approach presented in Section 5 helps us study the confidentiality-performance tradeoff, it is not a directly implementable scheme, as it requires future knowledge of data access patterns. In order to approximate the behavior of this reuse based approach, we propose in this section a *profile-based* approach, which is implementable.

Even I/O-intensive applications typically have a small number of static I/O calls in their program codes. Each of these static I/O calls is executed multiple times during



**Figure 7. Sensitivity of IOT and VF to the file cache size (FCS).**

the execution of the program. For instance, a static I/O call enclosed in a loop results in many dynamic instances of the call, possibly with different parameters each time. Based on such variable parameters and using knowledge of data access patterns, our reuse-oriented approach explained in Section 5 decides, for each dynamic instance of an I/O call, whether it is to be a plain access (without any encryption/decryption) or an encrypted access (in case of writes) and decrypted access (in case of reads). Since the reuse-oriented approach proposed above works with the dynamic I/O instructions, it has flexibility in marking different dynamic instances of the same static instruction differently (i.e., plain versus encrypted/decrypted). However, any approach that works with static I/O calls have limited flexibility compared to the reuse-based scheme (because we have fewer static calls than dynamic instances, as illustrated in Figure 8). In our profile-guided approach, profiler inserts *hints* in every static I/O call. Note that the hint attached to a static I/O call indicates whether that call works with plain or encrypted data, and effects the behavior of *all* dynamic instances of that static call. Therefore, the hint attached to a static call should be determined very carefully. Based on this observation, we can divide the static calls in a given

program code into three groups:

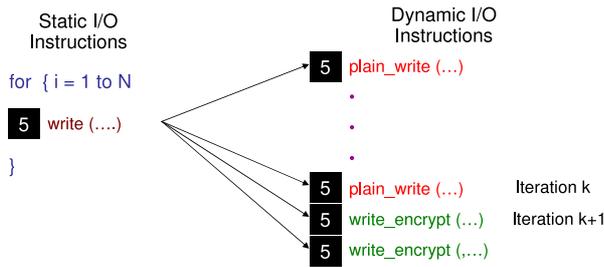
- Group I: The static I/O call (read/write) can always be interpreted as *read\_decrypt/write\_encrypt*. This holds true when there is no frequent reuse on the data manipulated by these I/O calls; i.e., when all dynamic instances (in the reuse-oriented scheme) are marked as encrypted/decrypted write/reads. Examples of this are the static I/O calls numbered 1 and 4 in Figure 8.
- Group II: The static I/O call (read/write) can always be interpreted as *plain\_read/plain\_write*. This holds true when the data manipulated by these I/O calls are repeatedly reused within a short period of time; i.e., when all corresponding dynamic instances (in the reuse-oriented scheme) are marked as plain reads/writes. An example of this is the static I/O call numbered 3 in Figure 8.
- Group III: Decision varies dynamically for the same static I/O call in the program. This group captures the occurrences that do not fall into the first two groups. Examples of this are the static I/O calls numbered 2 and 5 in Figure 8.

**Table 2. Breakdown of the occurrences of Group I, II, and III for each application in percentage in Performance Oriented (PO) scheme.**

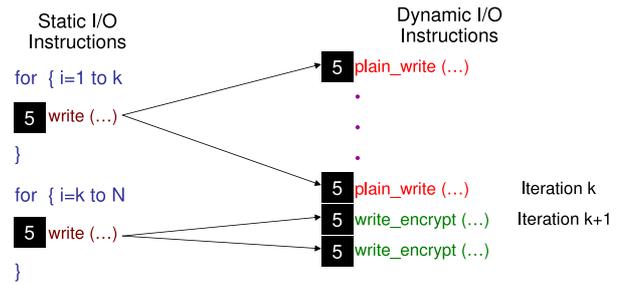
PO	$\delta_{threshold}$	Group I	Group II	Group III
swim	2450	12.72%	67.23%	20.05%
mgrid	2000	2.58%	52.26%	45.16%
tsf	4000	0.78%	0.0%	99.22%
mxm	1000	0.0%	0.59%	99.41%
lu	125	0.02%	0.59%	99.39%

**Table 3. Breakdown of the occurrences of Group I, II, and III for each application in percentage in Security Oriented (SO) scheme.**

SO	$\delta'_{threshold}$	Group I	Group II	Group III
swim	2000	0.0%	10.91%	89.09%
mgrid	2000	2.58%	52.26%	45.16%
tsf	500	0.0%	100%	0.0%
mxm	500	0.0%	100%	0.0%
lu	50	0.03%	0.6%	99.37%

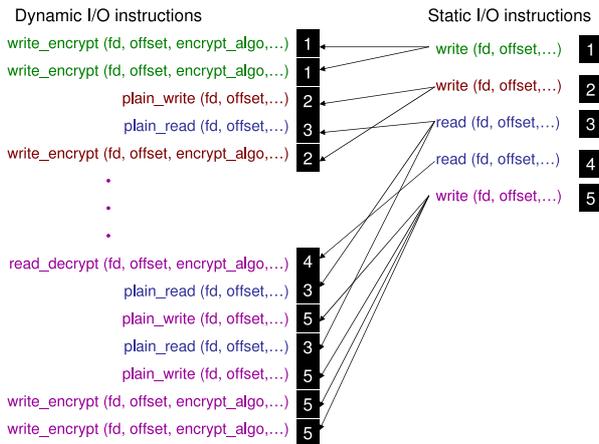


(a) Before I/O call splitting.



(b) After I/O call splitting.

**Figure 9. I/O call splitting for handling references in Group III.**



**Figure 8. Mapping between static I/O calls and dynamic instances.**

In the first and second cases, a compiler can mark the static I/O call as the particular type indicated by the dynamic instances. We refer to a static I/O call with this type of uniform behavior as *deterministic*. However, in the third case when a static I/O (read/write) call is dynamically made a *read\_decrypt* (*write\_encrypt*) some times and *plain\_read* (*plain\_write*) the rest of the times, the compiler cannot mark it to be of any particular type easily. In such cases, the com-

piler may want to modify the code such that it approximates to the dynamic behavior as closely as possible. More specifically, using code transformations, we may want to create two (static) I/O calls from the same original I/O call such that these new calls have more deterministic behavior (see Figure 9 for an example). This transformation is called *I/O call splitting* and is used in both the profile-guided schemes explained below. But, even after this transformation, for a given static call, we may still have dynamic instances with different behavior. However, one may also play with the  $\delta_{threshold}$  value to make most of the I/O calls in Group III more deterministic. Consequently, two schemes are feasible with the profile-guided approach described above.

- Performance Oriented: In the performance oriented approach (PO), the compiler profiles the code with a  $\delta_{threshold}$ , such that performance is favored over confidentiality in the tradeoff between performance and confidentiality.
- Security Oriented: In the security oriented approach (SO), compiler profiles the code with a  $\delta'_{threshold}$ , such that confidentiality is favored over performance in the tradeoff between performance and confidentiality.

Note that the main difference between PO and SO is in how the references in Group III are marked. This is achieved using different values of  $\delta_{threshold}$  and applying I/O call

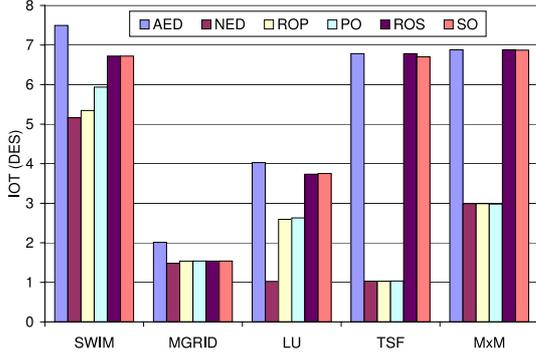


Figure 10. IOT(DES) variation with different approaches.

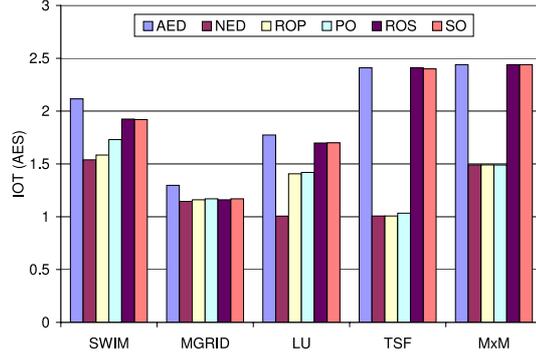


Figure 11. IOT(AES) variation with different approaches.

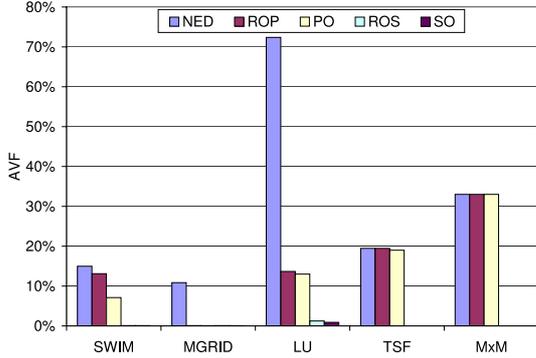


Figure 12. AVF variation with different approaches. Note that the AVF values for ROS and SO are very close to zero for all benchmarks.

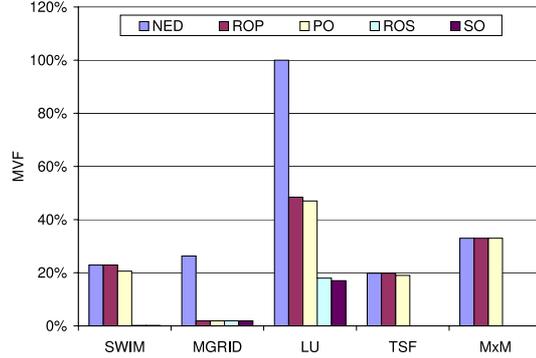


Figure 13. MVF variation with different approaches. Note that the MVF values for ROS and SO are very close to zero for swim, tsf and mxm.

splitting as explained above. From our experiments we find that, in most of the applications, a large portion of the static I/O calls belong to Group III. Tables 2 and 3 give a breakdown of the occurrences of Groups I, II, and III for each application in the Performance Oriented (PO) and the Security Oriented (SO) schemes. In such cases where the mixed behavior of static I/O calls needs to be analyzed, we carry out intra-loop reuse analysis [18, 21, 26] and apply I/O call splitting whenever possible.

We present a comparison of our results by plotting the variation of our metrics (IOT (DES), IOT (AES), AVF and MVF) with different applications using various approaches described so far. Figures 10 through 13 compare the variation of our metrics with six different approaches (AED, NED, ROP, ROS, PO and SO). AED and NED are the two extreme cases, explained in Section 4. PO and SO are Performance Oriented and Security Oriented approaches using profiling, for different values of  $\delta_{threshold}$  given in the second columns of Table 2 and 3, respectively. ROP and ROS are the reuse-oriented approaches (discussed in Section 5) for values of  $\delta_{threshold}$  same as that of PO and SO, respec-

tively. Our most important observation from these results is that the profile-guided approach follows the reuse-oriented approach *very closely* in all benchmarks. In fact, the two approximations give almost the same result for four of our five applications. In swim, our intra-loop static calls from a given original static call, using the approach in Figure 9. Consequently, there is a slight difference between the reuse-oriented approach and the profile-guided approach. One way of having better approximation in swim is to employ inter-loop reuse analysis [5, 19], which is in our future research agenda.

## 7 Conclusion and Future Work

Securing disk-resident data is very important in the context of storage systems. Applications using file system-based end-to-end security mechanisms consistently incur performance degradation because of the overheads involved in encryption/decryption. This paper describes an appli-

cation level solution to secure disk-resident data, based on data reuse concept, that helps us effectively utilize the data access/reuse patterns of the application. We quantify the tradeoffs between performance and confidentiality and experimentally analyze the effect of our proposed reuse-oriented approach on performance and confidentiality of data with various scientific applications that manipulate disk-resident data sets. From our experimental evaluation, we can conclude that using our approach it is possible to reduce the performance degradation due to encryption/decryption overheads by 46.5%, when DES is used as the encryption mechanism, and the same by 30.63%, when AES is used as the encryption mechanism. In our future work, we plan to devise a suitable (combined) metric in order to find an optimum value for  $\delta_{threshold}$ , under given performance and confidentiality requirements. Work is also underway in employing inter-loop reuse analysis for bridging the gap between the reuse-oriented scheme and the profile-guided schemes. Finally, we are also planning to explore suitable (encryption/decryption) key management and key sharing schemes that operate with our approaches.

## References

- [1] The PERFECT club benchmarks: Effective performance evaluation of supercomputers. *The International Journal of Supercomputer Applications*, 3(3):5–40, 1989.
- [2] M. Blaze. A cryptographic file system for UNIX. In *ACM Conference on Computer and Communications Security*, pages 9–16, 1993.
- [3] R. Carter. Nas kernels on the connection machine. Technical Report RND-90-005, Computer Sciences Corporation, NASA Ames Research Center, 1991.
- [4] G. Cattaneo, L. Catuogno, A. D. Sorbo, and P. Persiano. The design and implementation of a transparent cryptographic file system for unix. In *USENIX Annual Technical Conference*, 2001.
- [5] K. D. Cooper, K. Kennedy, and N. McIntosh. Cross-loop reuse analysis and its application to cache optimizations. In *Languages and Compilers for Parallel Computing*, 1996.
- [6] M. Corporation. Encrypting file system for windows. 2000.
- [7] A. O. Freier, P. Karlton, and P. C. Kocher. The ssl protocol version 3.0. <http://wp.netscape.com/eng/ssl3/ssl-toc.html>, 1996.
- [8] K. Fu. Group sharing and random access in cryptographic storage filesystems. Master's thesis, MIT, 1999.
- [9] K. Fu, M. F. Kaashoek, and D. Mazières. Fast and secure distributed read-only file system. *Computer Systems*, 20(1):1–24, 2002.
- [10] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. *SIGOPS Oper. Syst. Rev.*, 32(5):92–103, 1998.
- [11] J. H. Howard. An overview of the andrew file system. In *USENIX Winter Technical Conference*, 1988.
- [12] J. Kaiser and M. Reichenbach. Evaluating security tools towards usable security: A usability taxonomy for the evaluation of security tools based on a categorization of user errors. In *Usability*, 2002.
- [13] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus — scalable secure file sharing on untrusted storage, 2003.
- [14] A. Kent and R. Atkinson. Security architecture for the internet protocol. <http://www.ietf.org/rfc/rfc2401.txt>, 1998.
- [15] A. KleinOsowski, J. Flynn, N. Meares, and D. J. Lilja. Adapting the spec 2000 benchmark suite for simulation-based computer architecture research. *Workload characterization of emerging computer applications*, 2001.
- [16] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (sundr). In *OSDI*, 2004.
- [17] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *The 17th ACM symposium on Operating systems principles*, pages 124–139, Charleston, USA, 1999.
- [18] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4), 1996.
- [19] K. S. McKinley and O. Teman. Quantifying loop nest locality using SPEC'95 and the Perfect benchmarks. *ACM Transactions on Computer Systems*, 17(4), 1999.
- [20] R. C. Merkle and M. E. Hellman. On the security of multiple encryption. *Commun. ACM*, 24(7), 1981.
- [21] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., 1997.
- [22] J. Nechvatal, E. Barker, L. Bassham, W. Burr, M. Dworkin, J. Foti, and E. Roback. Report on the development of the advanced encryption standard (AES). Technical report, National Institute of Standards and Technology, 2000.
- [23] B. Pawlowski, S. Shepler, C. Beame, B. Callaghan, M. Eisler, D. Noveck, D. Robinson, and R. Thurlow. The NFS version 4 protocol. *Proceedings of the 2nd international system administration and networking conference*, page 94, 2000.
- [24] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. Hpl - a portable implementation of the high-performance linpack benchmark for distributed-memory computers. <http://wp.netscape.com/eng/ssl3/ssl-toc.html>.
- [25] A. Whitten and J. D. Tygar. Why johnny can't encrypt: a usability evaluation of pgp 5.0. In *8th conference on USENIX Security Symposium*, 1999.
- [26] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *PLDI*, pages 30–44, 1991.
- [27] E. Zadok, I. Badulescu, and A. Shender. Cryptfs: A stackable vnode level encryption file system. Technical Report CUCS-021-98, Computer Science Dept, Columbia University, 1998.