On Confidentiality and Algorithms
or
Programming Under the Constraints of Noninterference

Johan Agat*
Gatespace AB
agat@gatespace.com

David Sands
Department of Computing Science
Chalmers
dave@cs.chalmers.se

Abstract

Recent interest in methods for certifying programs for secure information flow (noninterference) have failed to raise a key question: can efficient algorithms be written so as to satisfy the requirements of secure information flow?

In this paper we discuss how algorithms for searching and sorting can be adapted to work on collections of secret data without leaking any confidential information, either directly, indirectly, or through timing behaviour. We pay particular attention to the issue of timing channels caused by cache behaviour, and argue that it is necessary to disable the effect of the cache in order to construct algorithms manipulating pointers to objects in such a way that they satisfy the conditions of noninterference.

We also discuss how randomisation can be used to implement secure algorithms, and discuss how randomised hash tables might be made practically secure.

1 Introduction

The setting which motivates this work is that of confidentiality and privacy in mobile code. Assume that some user wants to run a program that originates from an untrusted source. For example, the program can have been downloaded from an untrusted site on the Internet. When the program is run, it has to be given access to some data that the user regards as confidential (i.e., high security data) in order to compute the desired results. While running, the program also needs to have access to the Internet in order to fetch various kinds of information from databases etc. Examples of such programs might be a financial advisor program or a home-doctor application.

*This work was performed at Department of Computing Science Chalmers University of Technology and Göteborg University, Sweden; a preliminary version appears in the first author's PhD thesis [3]

This setting has been the motivation behind a recent resurgence of interest in the analysis and certification of confidentiality properties of programs [12, 19, 18, 11, 17, 2]. These papers are all based on Denning and Denning's work [6], and focus on the development of semantic security conditions and program analyses based on a zero information flow property usually called *noninterference* [9].

A fundamental question is left open by these and other works: is it possible to write programs which are noninterfering? In this paper, we will discuss the programming constraints imposed by noninterference and their implications on the complexity of algorithms operating on secret data. It is important to investigate this in order to get an idea of whether the demands of absolute noninterference are realistic; can any interesting programs be written, given that they have to satisfy the demands of noninterference?

2 Noninterference

From the perspective of the analysis of *programs*, rather than more general and more abstract "systems", Cohen's work on information transmission [5] anticipates the essential ideas behind noninterference. Cohen formally defines (amongst other things) a notion of *strong dependency*. A variable `l` strongly depends on a variable `h` over the execution of a program `P`, if different initial values of `h` can result in different values of `l` when `P` has terminated. Noninterference is a very strong security criterion that specifies the absence of strong dependence on secret inputs. Informally formulated, it can be expressed more generally as follows:

> *Varying the high security inputs to a program should have no effect on the parts of its behaviour that are observable by the attacker.*

To begin to investigate the consequences of noninterference, we must first decide what parts of the program's behaviour are to be considered observable by an attacker (referred to as *observable behaviour* in the rest of the paper). This will influence how information can be leaked and thus also the requirements on secure programs. We assume that there are some kind of low security output actions available to the program. Such actions could be, for instance, internet communications observable by the attacker, and perhaps also direct communication by opening a socket to a machine that the attacker controls. We also assume that the attacker can observe the values communicated with low security output actions. Moreover, since the system executing the code has no control whatsoever over the attacker, we have to assume that he/she is able to observe the time when each output action is performed. Thus, the program can implement a *covert channel* (see Lampson [14]) to transmit information to the attacker by varying its timing behaviour depending on the high security data it wants to leak.

For secure programs we thus have to require that the values of the high security data should not affect the values communicated through low security output actions, nor should the high data affect the time at when these observable actions are performed.

2.1. High Inputs and Integrity

For multilevel security systems concerned with preserving the confidentiality of sensitive data, we take the standard approach regarding data integrity: information may flow upwards in the security lattice from "low" to "high" confidentiality levels, but not from high to low. This means that the attacker has the ability to add known (low confidentiality) elements to collections which contain secrets.

It is worth noting that without this possibility it would be much more difficult for an attacker to leak information. If the program was not allowed to construct input to algorithms consisting partly of secret data and partly of known data, and instead had to use input entirely consisting of the secret inputs to the program, only information about the relative relationships of the secret inputs could be leaked. [1]

2.2. Information Leakage

To achieve noninterference and be absolutely certain that the attacker cannot infer anything about the secret data manipulated by the program we have to restrict how high security data is used by the program. There are many programming language constructs that have the potential to leak secret information. We briefly describe how this can be done with a few examples. Assume that `h` stores some high security value.

[1]A more fine-grained security classification can be used for keeping track of *public* data, *possibly secret* data, and *definitely secret* data, where both *public* and *definitely secret* can be considered to be of type *possibly secret*, but not the other way around. This approach is used by Abadi in [1] to guarantee the integrity of cryptographic keys.

Output Actions The most obvious method to leak secret data would be to simply send it to the attacker with some kind of output action:

```
output(h);
```

To prevent such direct information flows, the values output in actions observable by the attacker must not be influenced by secret data.

Loops and Recursion These constructs affect the overall running time of the program and may also affect its termination behaviour. The following small code fragment leaks the least significant bit of `h` by inserting a delay between the output of `"start"` and `"stop"` when that bit is 1.

```
output("start");
l = 1000000;
while ( (h&1 == 1) && l>0 )
  l--;
output("stop");
```

Branching and Timing Testing and branching on high security data can cause information to leak in several ways. If any low outputs or other actions that have an observable effect are performed in any of the branches, the outcome of the test may be leaked. Two examples of this are:

```
if ( h>0 )                 if ( h>0 )
  output("yes");             l = "yes";
else                       else
  output("no");              l = "no";
                           output(l);
```

Any variance in execution time of the branches of a high security if-statement can also be used to leak information:

```
s = 1;
for (i=0; i<w; i++){
  if (k[i])
    C = (s*M) mod n;
  else
    C = s;
  s = C*C;
}
```

The example above is an implementation of the modular exponentiation algorithm that computes $\mathtt{M}^{\mathtt{k}} \mod \mathtt{n}$ into the variable `C`, where `k` is represented as an array of `w` bits, with the most significant bit at index 0. The modular exponentiation algorithm can be used in RSA encryption, where `M` is the clear-text message, `k` is the encryption key, and `C` is the cipher-text. This implementation is not secure, since the two branches of the if-statement have different timing behaviours, and as shown by Kocher [13], the entire encryption key `k` can thus be learned by an attacker capable of measuring the execution time of the algorithm.

Exceptions Since the exception mechanism can cause the flow of control to abruptly jump, it is very suitable for implementing information leaks. As an example of this, we consider devision by zero:

```
try {
  h = 10/h;
  output("h != 0");
} catch (ArithmeticException e) {
  output("h == 0");
}
```

In a secure program, confidential data cannot be used in such a way that it causes exceptions to be thrown. This means that, e.g., array indexing cannot be performed with a high security index value unless it is guaranteed to be within the bounds of the array. Other partial operations, such as pointer dereferencing, for instance, must be subjected to similar constraints.

A consequence of this restriction is that:

> the size of secret data manipulated by a secure program cannot be kept entirely secret.

Since any program only can consume a finite amount of memory, information leaks can be implemented by allocating all of the available memory and thereby causing the program to crash.

Primitive Operators The application of some primitive operators can actually reveal information about the values of their arguments. For instance, on most computer architectures, multiplication is performed much faster if one of the operands is zero. To get noninterference, primitive operators must have implementations that do not reveal anything about the values of the operands through their execution time. Due to subtle things like cache behaviour, this includes also reading from, and writing to, the memory.

Requiring that the timing of observable actions be independent of the high security data manipulated by the program puts considerable constraints on how secure programs can be constructed. In fact, the implications of these requirements on program development are quite harsh even on the parts of the program that do not perform any observable output actions. Consider a procedure that operates on high security data without performing any output actions. To be both secure and reusable, that procedure must be implemented in such a way that its execution time is independent of the values of the high data it manipulates. If this requirement is not fulfilled, the procedure cannot be called in an arbitrary context and thus its re-usability is very limited.

3 Writing Secure Procedures

In the rest of this paper, we will discuss how secure procedures can be constructed, considering in particular how some algorithms for searching and sorting can be made noninterfering. We will concentrate on the effects that loops, recursion and branching have on execution time, and only discuss algorithms and procedures that are total in their high security arguments and which do not perform any observable outputs.

3.1. Loops and Recursion: the Worst-Case Principle

An algorithm with a termination condition that depends on high security data can be rewritten such that this dependence is removed, provided that there is an upper bound on the algorithm's execution time. The idea is to modify the algorithm so that it always exhibits its worst-case behaviour with respect to its high security inputs. This can be done in two ways: by adding a dummy loop or by using another termination condition that is only dependent on low security data. Assume that h and l are the high and low-security arguments to the algorithm and that the normal implementation would execute in time $t(h,l)$. Furthermore, assume that the worst-case execution time with respect to h is bounded by $t_{\text{wc}}(l)$.

Dummy loop. A total execution time of $t_{wc}(l)$ can be achieved by sequencing the normal implementation of the algorithm with some piece of code that executes in exactly time $t_{\text{pad}} = t_{\text{wc}}(l) - t(h,l)$. While theoretically possible, this is probably very tricky to do correctly in practice.

Low termination condition. A simpler approach to rewrite the algorithm is to find a low security termination condition that corresponds to the worst-case execution. Given that the body of the loop or recursive procedure can be rewritten in such that its execution time is independent of `h`, this will result in a secure algorithm. The body will have to perform useful work for the algorithm as long as there is something to do and keep on doing redundant operations otherwise.

Whichever method is used, with this approach the lower bound complexity of the algorithm be pushed up to the upper bound complexity. In other words, for a noninterfering algorithm operating on secret data of a given size, the worst-case and best-case executions have to be exactly equal. This technique is an effective design principle for noninterfering programs which we will refer to as the *worst-case principle*.

3.2. Branching

One approach to preventing timing leaks is to forbid branching on high data [18]. If branching on high data is not allowed, there are essentially no interesting algorithms that can be implemented. By forbidding decisions to be made on high data, only algorithms that are essentially parametric in their high security arguments are allowed. High data can be copied, moved about and passed to total primitive operators but not take part in any other computations.

Fortunately, there is nothing that prevents branching on high data as long as it does not affect the observable low behaviour of the algorithm. Again, some dummy code could be added after the branching structure to compensate with time depending on which branch was taken, but again, it is probably very difficult do this correctly in practice. The simplest method of achieving secure branching on high data is to make sure that all branches have the same execution time and observable effect on the execution state. This is the approach taken by the first author in [2, 3].

3.3. Searching and Sorting

In Sections 4 and 5, we describe different data structures and algorithms for searching and sorting a collection of high security objects. Both searching and sorting are fundamental building blocks of many other algorithms, so finding secure versions of these should provide a hint on how other algorithms can be implemented securely.

All algorithm implementations assume that the size of the collection, i.e., the number of objects operated on, is of low security. This is a necessary assumption since attempting to keep the size of a collection secret is futile. The running time of any algorithm that inspects more than a constant number of objects in the collection must depend on its size. If the collection has some maximum size which is bounded by a low value m, we can apply the worst-case principle and pad the collection with dummy elements to that is has size exactly m.

3.3.1 Two Different Assumptions on What Is Observable

Even very small differences in the execution state, like the state of the cache, for instance, can be observed in the execution time of a program. Consider the following code fragment:

```
if ( h>0 )
  z = x;
else
  z = y;
z = x;
```

Under the assumption that neither `x` nor `y` are cached before this code is run, the execution time of the last assignment will reveal the outcome of the test `h>0`. The assignment `z = x` , and thereby the whole code fragment, will execute slightly faster if the test is true than if it is false. Even though the two branches of the if-statement perform the same kind of computations, they reference different variables and thus their effects on the cache are different. If the test was true, `x` will (almost certainly) be cached at the end of the if-statement and thus referencing `x` again will be slightly faster than if `y` was cached instead.

We will say that a program which exhibits variances in execution time due to cache behaviour has a *cache-leak*. The capacity of covert timing channels implemented through cache-leaks is not likely to be very high, but it is certainly high enough that they should not be neglected. To reinforce this point, a simple experiment we have made (Appendix A) suggests that covert timing channels based on cache-leaks can have a capacity of at least 1 bit per second, based on an experiment on a 300 MHz SUN UltraSPARC-IIi.

In the rest of this paper we will consider two disjoint assumptions as to whether cache-leaks can be observed or not, and discuss their effect on how secure programs must be written. The two assumptions are:

Cache-leaks can be hidden. There are a number of methods that can be used to effectively close cache-leaks. For instance, the compiler can be instructed to emit extra memory reference instructions before each reference to a variable, such that it is evicted from the cache. A simpler, but perhaps not as efficient approach, would be to have a parallel process running that continuously garbles the cache. Thereby, any cache-leak based covert timing channel is drowned in noise, which further reduces its capacity. The ultimate protection from cache-leaks would of course be to turn off the cache, but this might not always be practical or even possible.

Cache-leaks cannot be hidden. To be absolutely certain that a branch on high data does not introduce any cache-leaks, all branches have to modify the cache in the same way. This essentially means that the algorithms memory reference pattern must be independent of high data. As we shall see, this requirement essentially prevents the writing of programs that manipulate pointers or references.

4 Searching

Since algorithms for searching are very tightly coupled to the data structure that is used to store the elements, we consider a few such data structures in their entirety. Data structures for searching normally support three operations:

Insert, Delete and Find. Naturally, all these operation must be implemented securely in order to achieve noninterference. Neither the values already stored in the data structure, nor the values used as arguments to the operations can be allowed to affect their execution time.

4.1. Hash-Tables

The often most efficient and perhaps most commonly used data structure for searching is the Hash-table. It consists of two components: an array A of size s, and a hash-function that maps objects to indices of A, i.e., $\{0, \dots, s-1\}$. Since the size of the set from which objects are drawn is normally many times larger than s, there are several objects that the hash-function will map to the same index. When such collisions occur for some objects at an index i, $A(i)$ can be made to point to a secondary search structure which is used to store the colliding objects. If the size of the hash-table is properly dimensioned with respect to the number of objects stored in it, and if the hash-function has good spreading properties, all of the operations Insert, Delete and Find can be made with an average complexity of $\mathrm{O}(1)$. To maintain these nice properties when the hash-table starts to fill up and collisions become more frequent, the array A can be replaced with a larger one and all the elements must then be rehashed into the new array. Done properly, this will not affect the average cost of Insert, Delete and Find operations by more than a constant factor.

The $\mathrm{O}(1)$ average complexity of deterministic hash-table operations also assumes that the objects inserted have some random distribution. In the worst-case, all objects will hash to the same index and the complexity of all operations will be that of the search structure used for collisions. Following the worst-case principle, this means that a secure hash-table, where n elements have been stored, would have to provide Insert, Delete and Find operations with execution times that mimic the corresponding operations on a secure n-element structure of the kind used for collisions. Implementing and using such hash-tables is thus pointless since they will only impose extra slow-downs compared to the just using the secondary search structure.

4.2. Search Trees

The idea of modifying Insert, Delete and Find so that they mimic the worst-case behaviour, works fine for all kinds of search trees. All operations on a search tree must know the depth of the tree and, e.g., an invocation of Find must always search to the bottom of the tree and then perhaps continue to perform dummy operations if the deepest branch was not taken. Similar behaviours are required from the Insert and Delete operations. To get good performance, a search structure with a worst-case complexity of $\mathrm{O}(\log n)$ must be used, for example AVL-trees or 2-3-trees.

```
if ( h>0 ) {
   Insert($S, o_1$);
   Insert($S, o_2$);
}
else {
   Insert($S, o_2$);
   Insert($S, o_1$);
}
Find($S, o$);    // $o$ can be any object
```

Figure 1. Variable h influences the state of the cache.

4.3. Linear Search

None of the search structures described above above are cache-leak free. It is obvious that any deterministic operation with a better than linear complexity cannot be cache-leak free since it will not reference all objects stored in the structure. Such an operation will thus have different effects on the cache depending on the search structure and the object operated on.

The only way to get a cache-leak free search structure is to ensure that all operations always reference every object stored in the structure. In addition, this must be performed in an order which is independent of the values of the objects stored in the structure. Moreover, when pointers to objects rather than actual objects are stored in the search structure, the order in which the objects are referenced must be independent of their values, and the order in which they where inserted. This last requirement can be motivated by the following example. Let S be a search structure that stores pointers to objects and let o_1 and o_2 be two objects located at memory addresses such that they occupy the same cache block. If the order that objects in S are referenced by a Find operation depend on the order in which they are inserted, e.g., if the last inserted object is referenced first, then the code given in Figure 1 will generate a cache-leak. Assuming a directly mapped cache, this code will, depending on h, cache either o_1 or o_2 but not both. This example generalises easily to other types of caches by using more objects that occupy the same cache block.

It is unclear to us how to implement a deterministic search structure, operating on pointers to objects, that is noninterfering and does not have cache-leaks. We conjecture that it is not possible. Any deterministic implementation of Find will traverse the objects in a fixed order and thus how the pointers to the objects are stored in the structure will affect the order in which they are referenced. With any Insert operation that is also deterministic, the internal arrangement of the objects stored in the search structure have to depend on the values of the objects it stores and/or the sequence of operations performed on it. In Section 6, we discuss how randomisation can be used to achieve security and freedom from cache-leaks.

5 Sorting

In this section, we will discuss three algorithms for sorting arrays: Quicksort, Heapsort and Mergesort. Obviously, a sorting algorithm can be used to introduce a timing variance if its running time depends on the distribution or order of the elements in the collection it sorts. However, since all general sorting algorithms have to examine every element in the collection – or more precisely, every *sort key* – they might seem less suited for introducing cache-leaks than algorithms for searching. Nonetheless, if the order in which the objects are referenced can vary with the values or initial order of those objects, then the algorithm might leak high information into the state of the cache.

For algorithms that sort arrays where the sort key is a literal "unboxed" value, like integers for instance, this is only a problem if the array is larger than the cache, so that there are two indices in the array that will occupy the same cache block. Even then, the timing variances introduced are very small since data larger that the cache – usually at least several kilobytes – must be sorted in order to get a timing variance of a few cache misses.

When it comes to sorting arrays where the sort key is accessed by dereferencing a pointer, then the problem of cache-leaks is much more severe. To see why, we observe that every deterministic sorting algorithm must inevitably reference the keys in an order that depends on the values of the keys. Interestingly, it does not matter if the sequence of compares and swaps of elements in the array that the algorithm performs are completely independent of the values of the objects pointed to by the array. The reason is that, since the sorting algorithm is deterministic, the last pair of objects that are compared is completely determined by the input array. Even if the last two indices compared by the sorting algorithm are always i and j, the objects that are referenced from these indices in the finally sorted array are inevitably determined by the *values* of the objects sorted.

To describe how this can be exploited, we consider the following scenario. Assume that `sort` is a deterministic procedure that sorts an array where keys are accessed via pointers in the elements of the array. Keys are compared and objects are moved. Each comparison will dereference two pointers and thereby cache the two corresponding objects. Suppose that for the instance of sorting a four-element array, the last comparison that `sort` performs is between the last two elements in the array. A secret boolean h can

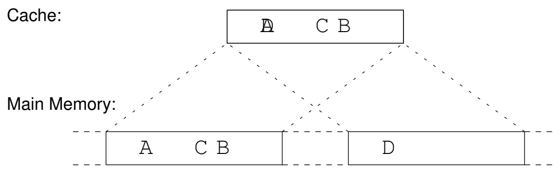

Figure 2. Illustration of objects A and D colliding in the cache.

then be leaked by modifying the values of, and then sorting four objects A, B, C and D, such that A and D collide in the cache, as illustrated by Figure 2. Assuming a directly mapped cache is used, only one of A and D can be cached simultaneously. To leak whether h is larger than 3, the four objects can be assigned values as follows:

$$A = 3 \qquad B = h - 1 \qquad D = h \qquad C = h + 1$$

If h is larger than 3, the last two elements in the sorted array, and thus the last two compared, will be D and C. Thus, information about the value of h is leaked by whether A is cached or not after the sorting has finished.

5.1. Quicksort

The most commonly used algorithm for sorting is probably Quicksort. It can sort an array in-situ, i.e., without using any extra space, and it has a low constant factor overhead. The average-case performance of Quicksort is $O(n \log n)$, but the worst-case performance is $O(n^2)$. The average-case complexity relies on that choosing a pivot element and partitioning the array in two parts of roughly equal size can be done in $O(n)$.

To adapt the deterministic version of Quicksort to securely sort high data is probably impossible without either loosing the $O(n \log n)$ complexity or introducing impractical constant factor slow-downs. The reason for this is that the pivot element controls the recursive calls and thereby the termination of Quicksort. To keep the $O(n \log n)$ complexity, an $O(n)$ method of partitioning an array into two halves of roughly the same size must be used. To achieve noninterference, this method must be such that the sizes of the two partitions only depend on the size of the array and not the values it stores. The recursive calls can then be made on parts of the array solely determined by its length, n. Performing such a partitioning corresponds to finding the median in linear time. The most well known and straight forward algorithm of finding the median is itself based on Quicksort and thus has a worst-case complexity of $O(n^2)$. With a considerably more involved[2] algorithm developed by

[2]Dor and Zwick themselves describe the *green factories*, which are the

Dor and Zwick [8], the median can be found using less than $3n$ comparisons in the worst-case. This algorithm seems to require an extra $\mathrm{O}(n)$ amount of memory however, so using it as the basis of the partitioning step of Quicksort would, in addition to introduce a large constant factor overhead, also ruin its in-situ properties. As the speed of Quicksort to a large extent relies on fast partitioning, other sorting algorithms, like for example Heapsort, will probably be faster if the partitioning is slowed down too much. How the median selection algorithm developed by Dor and Zwick can be adapted to noninterference is unclear to the authors but it ought to be possible by always running it according to the worst-case.

As we shall see in section 6.3, randomisation offers a much simpler approach.

5.2. Heapsort

Heapsort is an in-situ algorithm with worst-case complexity of $\mathrm{O}(n \log n)$. It is not as commonly used as Quicksort however, since it has a higher constant factor overhead and often runs slower in practice. Heapsort has two phases:

1. The elements to be sorted are first arranged to form a heap, which is an almost complete binary tree where each node is no smaller than any of its children. This phase can be done in $\mathrm{O}(n)$. The heap can be represented within the array which is to be sorted, which gives Heapsort its in-situ properties.

2. Once the heap has been built, the largest element is at its top. The sorted array is then constructed by removing the top element and reestablishing the heap property n times. The heap property can be reestablished in $\mathrm{O}(\log n)$, thereby giving Heapsort its $\mathrm{O}(n \log n)$ complexity.

Both of these phases can be implemented securely without changing their complexity. The method used to do so is very similar to that used to implement noninterfering search-trees. All operations that move elements down (or up) in the heap should always perform the same number, $\Theta(\log n)$, of compare-and-swap operations. In Appendix B, Java source code for a noninterfering version of Heapsort is provided.

5.3. Mergesort

Merge sort performs a logarithmic number of linear time merges, which gives it a worst- and average-case complexity of $\mathrm{O}(n \log n)$. The standard merge sort algorithm can be implemented in-situ for linked lists but requires an extra $\mathrm{O}(\log n)$ space to sort arrays. The extra space is needed to perform the merging. In [4], Batcher describes a technique called odd-even merging, that happens to be very suitable for meeting the demands of noninterference. This method of merging was developed for hardware sorting networks and has a complexity of $\mathrm{O}(n \log n)$. The sequence of compare-and-swap operation performed by the odd-even merge is determined *solely* by the lengths of the arrays that are merged. Thus, by using this technique, a variant of Mergesort can be implemented which has a complexity of $\Theta(n \log^2 n)$ and is cache-leak free for arrays of objects. Moreover, this algorithm can actually be implemented in-situ for arrays! Of course, the discussion in the beginning of Section 5 still applies, so even this variant of Mergesort can give rise to a cache-leak when it sorts arrays of pointers to objects. Appendix C contains Java source code for an implementation of Mergesort based on Batcher's odd-even merging.

central part of the selection algorithm, as being “extremely complicated” and that fully describing them would be “too long for a journal paper”.

6 Randomised Algorithms

In this section, we will discuss randomised algorithms that operate on high security data. Such algorithms might not satisfy the conditions of noninterference, due to the high security data somehow being involved in computations that affect the program's observable behaviour, but they can still be secure in practice.

Randomised algorithms suggests that probabilistic noninterference is the most suitable security criterion to use. Semantic models for probabilistic noninterference has been investigated by, e.g., McLean [15], Gray [10], and in the programming language setting by Volpano and Smith [20], and Sabelfeld and Sands [17]. The basic idea of probabilistic noninterference is that the distribution of a secure program's observable behaviour should not vary with the distribution of high security inputs to the program. Probabilistic noninterference is, like standard noninterference, a zero insecure information flow criterion.

Probabilistic noninterference is still a very strong a security condition because it concerns information dependencies and does not allow any insecure information flow at all. However, even some programs that have insecure information flow might actually be secure in practice, given that this flow is sufficiently small from an information-theoretic perspective.

We will now take a second look at some of the algorithms and data-structures discussed in Sections 4 and 5, to see how randomisation can be used to achieve probabilistic noninterference or to limit the probability of information leakage so much that security is achieved in practice.

6.1. Hash-Tables

The reasoning in Section 4.1 assumes that the hash-function is fixed and that each of the operations on the hash-table thus have to protect against an attacker who knows which objects will collide. Instead of using a fixed hash-function, one can be chosen uniformly at random from a *2-universal hash family* each time a hash-table is created (see, e.g., [16]). Security can be achieved in practice for all operations without losing the $\mathrm{O}(1)$ complexity. The defining property of 2-universal hash-functions is that the probability for collision between two distinct objects is less or equal to $1/s$, where s is the size of the hash-table's internal array. The operations on the hash-table still have the worst-case complexity of the search structure used for collisions, but the expected time for any operation is bounded. Even though the time to perform an Insert, Delete or Find depends on the objects previously inserted into the hash-table, this cannot be utilised in practise to produce a covert timing channel. Due to the randomisation, the worst-case is just as (un)likely for any sequence of operations.

Although the hash-function is chosen randomly, it is fixed once the hash-table is created. Thus a program that uses a randomised hash-table might test it with some cunning sequence of operations and thereby learn which objects collide. Once colliding object have been discovered, the attacker can use this to create timing differences in order to leak information. The operations on the hash-table are thus not probabilistically-noninterfering, but fortunately, the success of such testing in practice can be made arbitrarily hard. To resolve a collision, it is likely that rather few instructions have to be executed. Given that a collision can be resolved with less than a hundred instructions, the extra time associated will be less than 10^{-6} seconds on a reasonably fast computer. It is thus safe to assume that an external attacker, only observing the Internet communications of the program, cannot observe the timing variance resulting from one single collision. To observe a collision, it then has to be repeated some thousands of times, for instance by repeatedly inserting and deleting the same element. To make it harder for the program to learn about the hash-function, the hash-table operations can be implemented to always behave as if some constant number, $k-1$, of collisions occur. Thus k objects must collide at the same index in the hash-tables internal array before any timing variance in the operations can be observed. If we also pick a hash-function uniformly at random from a k-universal hash-family (see [7] for details), the probability of any k elements colliding at the same index is $1/s^k$. For a hash-table of size s that holds at most n elements before it increases its size and rehashes, the probability that k objects collide at the same index is:

$$\frac{1}{s^k}\binom{n}{k} \quad = \quad \frac{n!}{s^k k!(n-k)!}$$

Assuming a load factor[3] of 1, i.e., no more than s elements are stored in the hash-table before its capacity is increased and rehashing occurs, we get:

$$\frac{1}{s^k}\binom{n}{k} \quad \leq \quad \frac{n!}{n^k k!(n-k)!} \quad \leq \quad \frac{n!}{k!n!} \quad = \quad \frac{1}{k!}$$

Learning about the hash-function through testing can thus be made arbitrarily difficult by choosing a larger k. This reasoning gives us an upper bound on the collision probability of any given Insert operation.

To prevent the attacker from learning about the hash-function by repeatedly inserting and deleting elements, the hash-table can protect itself by continuously replacing the hash-function with a new randomly generated one. This requires rehashing of all elements in the hash-table but the $\mathrm{O}(1)$ expected complexity of the hash-table operations can be kept if the hash-function is replaced after every $c \cdot s$ operations, where c is a constant.

Remark Note that a hash-table implemented in this way will still not be noninterfering, but it is most likely to be secure enough for all practical purposes. Returning to the assumptions above, we try to estimate the difficulty of using this hash-table for implementing timing leaks. Assume that an observable timing variance does not occur until k objects have collided at the same index. With the assumption that the kth collision takes 10^{-6} seconds longer than the first $k-1$ collisions and that the attacker can only observe deviations with a granularity of 10^{-3} seconds, the attacker needs to perform 10^3 operations on the kth colliding object to transmit any information (e.g., repeated Inserts and Deletes or one Insert followed by repeated Finds). Since the hash-table rehashes after $c \cdot s$ operations, an upper bound on the probability for succeeding with leaking one bit becomes:

$$\frac{1}{k!} \cdot \frac{c \cdot s}{10^3}$$

If we choose $k = 20$, then for a hash-table with a size of 1MB and $c = 10$, this probability is less than $4.4 \cdot 10^{-15}$.

6.2. Linear Search

With randomisation, we can implement search structures that support Find, Insert and Delete operations that are secure and cache-leak free. To avoid cache-leaks, all elements are referenced by the Find and Delete operations, but in a randomly chosen order. Thereby, the operations reference the objects stored in the search structure in an order that is independent of all previous operations performed on it.

[3]The load factor of a hash-table is the ratio of the maximal number of elements stored in the hash-table and its size. In this case $n/s = 1$.

The randomisation does not necessarily have to be made in the Find and Delete operations. With a linked list implementation, for instance, Insert can add an element at a random place in the list. The Find and Delete can then simply traverse the entire list since that will reference all elements and do so in a random order.

6.3. Quicksort

Randomisation provides the possibility of a noninterfering and cache-leak free version of Quicksort. As is well-known, by choosing the pivot element at each step by a uniformly random choice, Quicksort has an expected complexity of $O(n \log n)$ for every input [16].

To consider the noninterference properties of this standard randomised variant, let us begin with the usual simplifying assumption that all elements in the array are distinct. At each recursive step of the algorithm, the selection of a random pivot implies that the size of the two subproblems (those elements smaller than or larger than the pivot element, respectively) is dependent purely on the random choice. Each possible split being equally likely, it is easy to see that the sizes of the subproblems, and hence the *distribution of running times* of the algorithm, for any input consisting of distinct elements, is not dependent on the values or the ordering of the keys in the original array. In other words, the algorithm is probabilistically noninterfering.

The question of what to do when there are repeated elements in the original array is potentially more difficult. But a simple brute-force solution is adequate in theory: extend each key with a unique index, thus forcing them to be distinct. Comparison must compare the elements and the additional indices in all cases. The practical drawback of this approach are the additional constant-factor costs in time and space.

Regarding cache-leaks: by flipping a coin to determine the order in which each pair of recursive subproblems are solved, cache-leaks are also eliminated. This is because the objects are then referenced in a completely random order. Thus all elements are equally likely to be resident in the cache after sorting.

6.4. Cache-Leak Free Sorting

Randomisation can be used to make any secure sorting algorithm that operates on pointers to objects cache-leak free. The algorithm can first be used to sort the array, and then all elements are referenced a second time, in a random order so that neither the values of the objects sorted nor the sorting algorithm will affect which objects are cached.

7 Discussion and Conclusions

Under the constraints of noninterference we have shown that sorting can still be made with a $\Omega(n \log n)$ complexity, but even with runtime/compiler support available, hashtables cannot be implemented to support noninterfering operations. Thus, searching a collection of secret objects cannot be made faster than $\Omega(\log n)$ if strict noninterference is required.

We conclude this paper with two conjectures on the restrictiveness of noninterference and its implication on the complexity for searching and sorting. Our first conjecture regards the most conservative case when the memory reference pattern must be independent of high security data to avoid cache-leaks.

Conjecture 1
(Noninterference, Pointers and Cache-Leaks)
If a program's cache behaviour cannot be prevented from affecting its running time in an observable way, then searching and sorting a collection where keys are accessed via pointers cannot be achieved in a noninterfering way.

This conjecture is based on the fundamental difficulty with pointers and cache-leaks discussed in Sections 4.3 and 5. To simply disallow the use of pointers and references to keys in order to deal with this problem is not a very realistic solution. Normal programs need to pass references and dereference pointers. For languages like Lisp and Java, passing references is even inherent in their implementations. Without references and pointers, programs that manipulate linked data structures of any kind cannot be implemented. Note that array indices behave just like pointers and references where cache-leaks are concerned.

As discussed Section 3.3.1, it is realistic to assume that cache-leaks can be effectively eliminated with the help of the runtime environment, the compiler, or by algorithm-specific measures as suggested e.g. for sorting in section 6.4. (The practical costs of eliminating cache leaks could, of course, be prohibitively high in some situations.)

Given that cache-leaks can be eliminated we are lead to our second conjecture:

Conjecture 2 (Noninterference and Pointers)
Provided that a program's cache behaviour can be prevented from affecting its running time in an observable way, searching a collection of secret objects still needs an $\Theta(\log n)$ asymptotic complexity to guarantee noninterference. Sorting a collection of secret objects in a way that satisfies noninterference, can be made with a complexity of $\Theta(n \log n)$.

This would have consequences on the complexity of many other algorithms that must satisfy noninterference. Since the lower bound for searching is raised to $\Omega(\log n)$, a slowdown will be imposed on very many algorithms that are adapted to be noninterfering. For most programs, however, this slowdown will be bounded in practice. The reason being that, for algorithms that store all objects in main memory, the depth of the balanced tree used as search structure will be limited by the size of the main memory. For a computer with at most one gigabyte of memory, the factor will be less than 30.

For practical purposes, these slowdowns would need to be weighed against the the randomisation-based techniques to implement practically-secure hash-tables suggested in Section 6.1, which carry a number of constant-factor overheads in creating random hash-functions, rehashing and hiding possible collisions.

To conclude, since the lower $\Omega(\log n)$ bound for searching will often be bounded by a reasonable constant factor, the slow-downs imposed by noninterference on most algorithms will be limited in practice. We thus consider it likely that feasibly efficient noninterfering versions of common algorithms relating to searching and sorting can be constructed.

8 Acknowledgements

The material presented in this paper has benefited very much from discussions with our colleagues Andrei Sabelfeld, Josef Svenningsson and Devdatt Dubhashi.Thanks also to the anonymous reviewers for many insightful and constructive comments.

References

[1] M. Abadi. Secrecy by typing in security protocols. In *Proceedings of Theoretical Aspects of Computer Software, Third International Symposioum*, volume 1281 of *LNCS*, pages 611–638. Springer-Verlag, 1997.

[2] J. Agat. Transforming out timing leaks. In *POPL'00: The 27:th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 40–53. ACM, ACM Press, January 2000.

[3] J. Agat. *Type Based Techniques for Covert Channel Elimination and Register Allocation*. PhD thesis, Department of Computing Science, Chalmers University of Technology, Sweden., January 2001.

[4] K. E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computer Conference*, volume 32, pages 307–314, 1968.

[5] Ellis S. Cohen. Information transmission in sequential programs. In Richard A. DeMillo, David P. Dobkin, Anita K. Jones, and Richard J. Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.

[6] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[7] M. Dietzfelbinger. Universal hashing and k-wise independent random variables via integer arithmetic without primes. In R. Reischuk and C. Puech, editors, *STACS 96, 13th Annual Symposium on Theoretical Aspects of Computer Science*, volume 1046 of *Lecture Notes in Computer Science*, pages 569 – 580. Springer Verlag, Feb 1996.

[8] D. Dor and U. Zwick. Selecting the median. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 28–37, 1995.

[9] J. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, April 1982.

[10] J.W. Gray III. Probabilistic interference. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 170–179, Oakland, California, May 1990. IEEE Computer Society Press.

[11] N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Conference Record of the Twenty-Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 365–377. ACM, 1998.

[12] D. Le Métayer J.-P. Banâtre, C. Bryce. Compile-time detection of information flow in sequential programs. In *Proc. European Symposium on Research in Computer Security*, volume 875 of *LNCS*. Springer Verlag, 1994.

[13] P. C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Neal Koblitz, editor, *Advances in Cryptology – CRYPTO'96*, volume 1109 of *LNCS*, pages 104–113. Springer-Verlag, 1996.

[14] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, Oct 1973.

[15] John McLean. Security models and information flow. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 180–187, Oakland, California, May. IEEE Computer Society Press.

[16] R. Motwani and P. Raghavan. *Randomised Algorithms*. Cambridge University Press, 1997. ISBN 0 521 47465 5.

[17] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, Cambridge, England, July 2000. IEEE Computer Society Press.

[18] D. Volpano and G. Smith. Eliminating covert flows with minimum typings. *Proc. 10th IEEE Computer Security Foundations Workshop*, pages 156–168, June 1997.

[19] D. Volpano and G. Smith. A type-based approach to program security. In *TAPSOFT'97*, volume 1214 of *LNCS*, pages 607–621. Springer-Verlag, April 1997.

[20] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *Proc. 11th IEEE Computer Security Foundations Workshop*, pages 34–43, June 1998.

A. A Cache-Leak Based Covert Timing Channel

This section presents a small C-program that uses cache behaviour to encode the value of a bit in its execution time. The program is tailored to use the internal data cache of a SUN UltraSPARC-IIi CPU. As it is provided, the program is merely intended to serve as a proof of concept that cache behaviour can be used to implement covert timing channels. However, if the calls to `ftime()` in the function `covert_send()`, are replaced by function calls that send a network packet to a receiving host, the program would implement a true covert channel.

The source code of the C-program is given below. On a SUN Ultra 10 with a 300 MHz SUN UltraSPARC-IIi processor with a normal load, the program can leak more than 0.9 bits per second. In a 30 minute test run, when the machine was used for editing and reading mail etc., but no other continuous computations than the covert channel program was run concurrently, 990 out of 1100 bits were successfully transmitted. This gives a failure rate of 0.1% and a bit rate of 0.94 bits correctly leaked per second.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <time.h>
#include <sys/timeb.h>

/**************************************************
  From:
  http://www.sun.com/microelectronics/
           whitepapers/UltraSPARC-IIi/03.html#3.7

  The UltraSPARC-IIi data cache is a 16 KB direct
  mapped, software selectable write-through
  non-allocating cache that is used on load or
  store accesses from the CPU to cacheable pages
  of main memory. It is a virtually-indexed and
  virtually-tagged cache. The D-cache is
  organized as 512 lines, with two 16-byte
  sub-blocks of data per line. Each line has a
  cache tag associated with it. On a data cache
  miss to a cacheable location, 16 bytes of data
  are written into the cache from main memory.
  ************************************************/

#define Garb_N  (1024*4+128) /* size-
of(int) = 4 */
#define N  (1024*4)
static int garbArr[Garb_N];
static int xs[N];
static int ys[N];

/*
 * This function flushes data out of the cache by
 * filling it with garbage.  Since 16 bytes are
 * read into the cache at each cache miss, it is
 * enough to read every 2 elements of an int
 * array to fill the cache.
 */
```

```
void garbleCache(void) {
  int i;
  register int garb;
  for(i=0; i<Garb_N; i += 2)
    garb = garbArr[i];
}


/*
 * This function introduces delays.  By first
 * reading either xs or ys and then reading xs,
 * the value of cache_xs is encoded in the
 * execution time.  A call to garbleCache() is
 * used to make sure neither xs nor ys is cached
 * before the encoding starts.  The whole
 * procedure is repeated n times to produce a
 * longer delay.
 */
void delay(int cache_xs, int n) {
  register int garb;
  int *arr, i;
   while ( n-- > 0 ) {
    garbleCache();
    if ( cache_xs )
      arr = xs;
    else
      arr = ys;
    for(i=0; i<N; i += 2)
      garb = arr[i];
    for(i=0; i<N; i += 2)
      garb = xs[i];
  }
}

/*
 * This function "sends" a secret bit h.  It does
 * so by producing three delays.  First, the
 * delay of a bit set to 1 and then the delay of
 * a bit set to 0 and finally the delay of h.
 */
void covert_send(int h, unsigned timers[]) {
  struct timeb t;
  int n = 1000;
  ftime(&t);
  timers[0] = t.time *1000 + t.millitm;
  delay( 1, n);
  ftime(&t);
  timers[1] = t.time *1000 + t.millitm;
  delay( 0, n);
  ftime(&t);
  timers[2] = t.time *1000 + t.millitm;
  delay( h, n);
  ftime(&t);
  timers[3] = t.time *1000 + t.millitm;
}
/*
 * This function is the "receiver".  It measures
 * the three delays introduced by covert_send()
 * and then guesses the value of h depending on
 * if it is closer to the delay of a 1-bit or the
 * delay of a 0-bit.
 */
int covert_recieve(int h) {
  unsigned timers[4];
  int d1, d0, dh;
  do {
    covert_send(h, timers);
    d1 = timers[1] - timers[0];
    d0 = timers[2] - timers[1];
    dh = timers[3] - timers[2];
  } while ( d1 >= d0 ); /* Redo noisy samples. */
  return abs(d1-dh) < abs(d0-dh);
}

/*
 * Simple main function to test and measure the
 * timing channel.
 */
int main(int argc, char *argv[]) {
  int g, h=1;
  int ok, nOk=0, nFailed=0;
  unsigned start, tot_time;
  struct timeb t;
  ftime(&t);
  start = t.time*1000 + t.millitm;
  while(1) {
    g = covert_recieve(h);
    ftime(&t);
    tot_time = t.time*1000 + t.millitm - start;
    ok = g == h;
    if (ok) nOk++; else nFailed++;
    printf("Expecting %d, got %d.   %s "
           "\tTotal: %5d tot %5d ok.  "
           "Failrate: %.4f  Time: %10d ms"
           "  Ok bitrate: %1.3f b/s\n",
           h, g, ok? "OK    " : "FAILED",
           nOk+nFailed, nOk,
           ((float)nFailed)/((float)nOk+nFailed),
           tot_time,
           ((float)nOk)/((float)(tot_time/1000))
           );
    h = !0;
  }
  return 0;
}
```

B. Noninterfering Heapsort

The Java code for a secure version of Heapsort is presented in this section. It consists of a class `Heapsort` with one public method `sort()`, that sorts an array of `int`s. The sorting method meets the demands of noninterference in the sense that the number of compares and swaps made are independent of the values in the array it sorts.

```
class Heapsort {
public void sort(int arr[]) {
  int len = arr.length;
  sort(arr,len);
}
private void sort(int arr[], int len) {
  int arrlen = arr.length;

  //  1, Build the heap by joining
  //     smaller heaps into larger.
  int depth=1; //The maximal depth of any heap.
  int modDepthAt = (((arrlen-1)>>1)-1)>>1;
  for ( int hp=arrlen>>1; hp>=0; hp-- ) {
    // arr[hp] should be joined.
    if ( hp == modDepthAt ) {
        depth++;
        modDepthAt = modDepthAt << 1;
    }
    swiftDown(arr, hp, arrlen, depth);
  }
  depth = 0;
  modDepthAt = 0;
  for( int i=0; i<arrlen; i=(i<<1)+1 ) {
    depth++;
    modDepthAt = (modDepthAt<<1)+2;
  }
  modDepthAt = (modDepthAt-1)>>1;
  // 2, Do the sorting.
  //    Heap part of arr shrinks to left.
  for ( int hp=arrlen-1; hp>0; hp-- ) {
    swap(arr, 0, hp);
    // Now, the Heap ends at hp.
    if ( hp == modDepthAt ) {
        depth--;
        modDepthAt = (modDepthAt-1)>>1;
    }
    swiftDown(arr, 0, hp, depth);
  }
}
/* Assumes that all elements "below" elemIx form
 * a heap in arr, but arr[elemIx] might not be
 * larger than its children.  Creates a heap
 * staring at elemIx by performing depth
 * compare-and-swap operations.  The array, or
 * the heap part of the array ends at hpLimIx.
 */
private void
  swiftDown(int arr[], int elemIx,
            int hpLimIx, int depth) {
  for ( int i=0; i<depth; i++ ) {
    // chL is the index of the left child.
    int chL = (elemIx<<1) +1;
    int chR = (elemIx<<1) +2;
    int ch;
    chL = chL < hpLimIx ? chL : elemIx;
    chR = chR < hpLimIx ? chR : elemIx;
    // Decide where to move arr[elemIx]
    if ( arr[chL] <= arr[chR] )
        ch = chR;
    else
        ch = chL;
    if ( arr[ch] > arr[elemIx] )
        swap(arr, ch, elemIx);
    else
        dont_swap(arr, ch, elemIx);
    elemIx = ch;
  }
}
private void
  swap(int arr[], int ix1, int ix2) {
  int tmp1 = arr[ix1];
  int tmp2 = arr[ix2];
  arr[ix1] = tmp2;
  arr[ix2] = tmp1;
}
private void
  dont_swap(int arr[], int ix1, int ix2) {
  int tmp1 = arr[ix1];
  int tmp2 = arr[ix2];
  arr[ix1] = tmp1;
  arr[ix2] = tmp2; } }
```

C. Noninterfering, in-situ Mergesort

This section presents Java code that implements an in-situ version of Mergesort, operating on an array of `int`s. The merging-method used is the odd-even merge described by Batcher [4], for constructing hardware sorting networks. This sorting algorithm performs a sequence of compare and swap operation that is completely determined by the length of the array that it sorts.

```
class MergeSort {
public void sort(int arr[]) {
  insituOddEvenMergeSort(arr,0,arr.length,1);
}
private void insituOddEvenMergeSort
  (int arr[], int startIx, int len, int step) {
  // Nothing to do if arr
  // contains 1 element or less
  if ( len <= 1 )
    return;
  // 1, Split arr in the middle.
  int len1 = len >> 1;
  int len2 = (len >> 1) + (len&1);
  int startIx2 = startIx + len1*step;
  // 2, Sort the two halfs recursively
  //    and merge.
  insituOddEvenMergeSort(arr, startIx,
                         len1, step);
  insituOddEvenMergeSort(arr, startIx2,
                         len2, step);
  insituOddEvenMerge(arr, startIx, startIx2,
                     len1, len2, step);
  return;
}
/**
 * Merges two arrays embedded in arr.  The first
 * array starts at arr[startIx], has len1
 * elements separated by step.  The second array
 * starts at arr[startIx2] and has len2
 * elements, also separated by step.
 *
 * The merge results in one array embedded in
 * arr, starting at arr[startIx1], with len1
 * elements separated by step and continuing from
 * arr[startIx2] with len2 elements separated by
 * step.
 *
 * "Odd"  indexes are startIx + 0,2,4,... *step
 * "Even" indexes are startIx + 1,3,5,... *step
 */
private void
  insituOddEvenMerge(int arr[], int startIx1,
                     int startIx2, int len1,
                     int len2, int step ) {
  if ( len1 == 0 || len2 == 0 ) return;
  if ( len1 == 1 && len2 == 1 ) {
    cmpSwap(arr, startIx1, startIx2);
    return;
  }
  int odd1_startIx   = startIx1;
  int even1_startIx  = startIx1 + step;
  int odd2_startIx   = startIx2;
  int even2_startIx  = startIx2 + step;
  int merge_step    = step <<1;
  int odd1_len      = (len1 +1) >>1;
  int even1_len     = len1 >>1;
  int odd2_len      = (len2 +1) >>1;
  int even2_len     = len2 >>1;

  insituOddEvenMerge(arr, odd1_startIx,
                     odd2_startIx, odd1_len,
                     odd2_len, merge_step);
  insituOddEvenMerge(arr, even1_startIx,
                     even2_startIx, even1_len,
                     even2_len, merge_step);
  int oddMlen = odd1_len + odd2_len;
  int evenMlen = even1_len + even2_len;

  int i;
  for (i=0; (i+1)<oddMlen && i<evenMlen; i++) {
    // The 2i'th and 2i+1'th elem in the merged
    // array should be the cmpSwap of odd
    // element i+1 and even element i.
    int oddIx = (i+1) < odd1_len ?
      odd1_startIx + (i+1)*merge_step :
      odd2_startIx + (i+1-odd1_len)*merge_step;
    int evenIx = i < even1_len ?
      even1_startIx + i*merge_step :
      even2_startIx + (i-even1_len)*merge_step;
    if ( evenIx < oddIx )
        cmpSwap(arr, evenIx, oddIx);
    else
        cmpSwap(arr, oddIx, evenIx);
  }
  return;
}
/**
 * Compares and possibly swaps two values of arr.
 * After the call, arr[ix1] <= arr[ix2].  The
 * same number of reads and writes to arr are
 * made regardless of the values of arr[ix1] and
 * arr[ix2].  */
private void
  cmpSwap(int arr[], int ix1, int ix2) {
  int elem1 = arr[ix1];
  int elem2 = arr[ix2];
  if ( elem1 > elem2) {
    arr[ix1] = elem2;
    arr[ix2] = elem1;
  }
  else {
    arr[ix1] = elem1;
    arr[ix2] = elem2;
  }
}
}
```