1 Smart Array Copy by Shuffled Subsegments[1]

This challenge concerns array copy function(s) broadly inspired by real life implementations that aim to resist certain side-channel attacks. Their aim is to copy all array elements from one array to another array of the same length. This could be achieved with a simple loop through the array indices in order. However, to obfuscate the sequence of memory accesses performed, we aim to *shuffle* some of the operations performed on the arrays, as described below:

Smart Array Copy version 0

1. Assume that two pre-allocated arrays `SRC` and `DEST` are initially available; we wish to copy from the first to the second. Both are of length $n > 0$.

2. We consider the arrays to be divided into q non-overlapping, contiguous *segments* of some fixed length $m > 0$ (where also $m = kl$, as explained below). The first m array locations (e.g. `SRC[0]`, `SRC[1]`, ..., `SRC[m-1]`) make up the first segment, the next m elements are the second segment, and so on. Since m might not perfectly divide n, we assume we have $n = qm + r$ where q is the number of complete segments, and r is the number of remaining array locations (at the end of the array).

3. We will copy the array segment by segment in sequence. In the first version of our algorithm, the scheme is simple: segment 0 of `SRC` is copied to segment 0 of `DEST`, then segment 1 is copied to segment 1 and so on. After the last segment is copied, we will copy the remaining r array elements in simple sequential order (these last r elements are not treated as a segment).

4. Segments are not copied simply. Each segment is further decomposed into (exactly) $l > 1$ *subsegments* of length $k > 0$ (m must be chosen to be the product of these integers k and l). Again, these are contiguous, non-overlapping portions: the segment is simply split into equal parts.

5. We will copy the subsegments of a given segment according to the following *shuffled order* scheme. For each segment we first generate a random *permutation* σ of the integers $\{0, 1, 2, \ldots, l-1\}$: that is, a bijective function from and to this set of values. We then copy subsegments one by one in the order specified by the sequence $\sigma(0), \sigma(1), \ldots, \sigma(l-1)$. That is, we first copy subsegment $\sigma(0)$ to its corresponding position in `DEST`, then we copy subsegment $\sigma(1)$, and so on until the entire segment has been copied.

   For example, if $l = 3$ and for the first segment the generated permutation σ satisfies $\sigma(0)=0$, $\sigma(1)=1$, $\sigma(2)=2$, then the three subsegments of the first segment in `SRC` will be copied in simple order (as if we had simply looped through the whole segment). If, for the next segment the generated permutation σ' satisfies $\sigma'(0)=2$, $\sigma'(1)=1$, $\sigma'(2)=0$, its three subsegments will be copied in reverse order (first subsegment 2, then 1, then 0).

[1] We warmly thank Nikolai Kosmatov for contributing the idea for this challenge.

Tasks for version 0

(a) Specify a function (or other operation) `Permute` which, when invoked, returns a (random, unknown) permutation. You do not need to implement this function, but it's important to make clear how you represent permutations (or their properties) in your specification/modelling of `Permute`.

(b) Write an implementation of a function (or similar) `SmartCopy` which performs Smart Array Copy version 0, as described above.

(c) Verify that your implementation is memory-safe / crash free: in particular, that all array accesses performed are guaranteed to be within bounds.

(d) Verify that your implementation is guaranteed to terminate.

(e) Verify that, on termination, the elements of `DEST` will be exactly those stored originally in `SRC`, and that the elements of `SRC` will be unchanged. For this version 0, the elements should be in the same order in both arrays.

Notes

- In case it helps get started, you might try a simplification of the algorithm in which you build in the assumption that there is only a single segment (with several subsegments). That is, assuming $q = 1$ and $n/2 < m \leq n$. In this case there is only one "split" of the array (into subsegments, as well as some remainder r); the subsegments will be copied in shuffled order.
- If you cannot prove properties for arbitrary values of n, m etc. you may fix some of these parameters as pre-defined constants in your work. Please label these clearly as well as whether or not these specific values can be freely changed (without substantially affecting your verification results).

Extension: Smart Array Copy version 1 In some settings, preserving the order of elements may not be important. In this case, we can obfuscate yet further by also shuffling the order of elements as they are copied, as follows:

1-4. Identical to version 0.

5. We adapt our previous scheme for copying subsegments to incorporate a *second* permutation τ of the same values $\{0, 1, 2, \ldots, l-1\}$. For each segment, we generate a random pair of permutations σ and τ; as before, we use σ to choose the order of indices of subsegments to be copied from the segment of `SRC` (as before); we then further apply the permutation τ to this subsegment index to obtain the subsegment copied *to* in `DEST`. That is, the new scheme is to copy first subsegment $\sigma(0)$ in `SRC` to subsegment $\tau(\sigma(0))$ in `DEST`, then subsegment $\sigma(1)$ to $\tau(\sigma(0))$, and so on.

Tasks for version 1

(f) Implement a new version of the algorithm, incorporating this additional permutation.

(g) Verify the same properties (a)-(e) of your new version.

(h) Write a short textual comment labelled `MODULARITY:` explaining to what extent you are able to reuse parts of the *code* and *verification effort/results* between your two different versions.

Extension: Smart Array Copy version 2 Take either your version 0 or version 1 algorithm (or both!) and incorporate shuffling of the order in which *segments* are copied. That is, implement a similar scheme to the version 0 shuffling of subsegment handling, but this time for segments: instead of always handling segment 0, then segment 1 etc., the order of segments handled by the new algorithm is also randomly shuffled.

Tasks for version 2

(i) Implement a new version of the algorithm, incorporating this additional shuffle (make clear which previous version(s) you are extending).

(j) Verify the same properties (a)-(e) of your new version.

(k) Write a short textual comment labelled `MODULARITY:` explaining to what extent you are able to reuse parts of the *code* and *verification effort/results* between your different versions.