CSCI 432, Group Project Part 3

Group 10

October 20, 2015

In most modern networks their is a stable, generally deterministic, data structure in place to handle requests, which is, by definition, very predictable. As with many systems, this was designed with usability, rather than security, in mind. However, users with malicious intent, such as those trying to gain an advantage in online gaming or those wanting breach security for other reasons, have shown that this predictability is also a vulnerability. They use the predictability of the data structure to send overwhelming requests to the network in attacks known as denial-of-service, or D.O.S. attacks. If this predictability is taken away and the ability to know how a network will respond to every request is eliminated, a D.O.S. attack would be difficult or impossible to mount. This is where our particular algorithm comes in. The purpose of the algorithm is to create a data structure that has unpredictable time for any input, while still performing at near optimal speed. Such a data structure must be able to perform the same operations as the original system, while also non-deterministically, i.e. randomly, altering itself. This new system should be able to resist all timing- dependent attacks, even from attackers who know the algorithm and the previous i/o values used. In their paper, Darrell Bethea and Michael K. Reiter discuss a data structure for set operations with these desirable characteristics[**?**].

The problem of creating a data structure with unpredictable timing has been attempted previously and in general there are two types of structures. There are structures whose goal is to make the worst case unpredictable and those whose goal is to make the worst case the least detrimental. Both types have their limitations and can still be abused by a cleaver enough foe. A algorithm creating truly unpredictable timing must change its internal structure often, if not every iteration, so that its timing cannot be predicted, particularly based upon previous iterations. To accomplish these goals this algorithm uses a unique implementation of skip lists along with a dynamic origin to create a data structure with unpredictable timing.

At the core of this algorithm are skip lists, which are specialized linked lists. Skip lists are a series of linked lists L_0 through L_m, where m is not constant, meaning the number of linked lists can change depending on the input. The base list L_0 contains every value entered into the set, L_1 will contain half the values of L_0, and so on until L_m only contains a single value. What's important about these lists is that for any list L_k every value on list L_k is also on list L_{k-1}. This fact, along with the lists being sorted, makes traversing through the lists extremely efficient. To traverse the lists we start at list L_m and move across our values until the next value is greater than our search value. Then we go down a list to list L_{m-1} and search that tree starting at the value we stopped with on tree L_m.

The next step of the algorithm is to take the normal implementation of skip lists and alter it slightly. In the initial implementation of skip lists if you get to the greatest value without having found your search value, then your search value must not be in the list. This method assumes that we start at with the smallest value in the lists and move to the largest but this is predictable. This algorithm needs to be able to start at any point, including the origin, and check all values in the lists regardless of whether they are larger or smaller than the starting value. To do this the creators of this algorithm made the skip lists circular, meaning that rather than the list having a specific head or tail the largest value points to the smallest value. Only when traversing past the starting point a second time do we stop.

The last critical step to making this algorithm truly timing unpredictable is giving it a dynamic origin. Ultimately the goal is to choose a new, semi randomly chosen origin after every function call, loopup(), insert(), or delete(), is made. This makes it so no matter how many trials or repetitions of the same value an attacker attempted he or she could never guess where the origin was.

To summarize, predictability in request handling, although an efficient solution, makes the network vulnerable to certain attacks. In an effort to prevent such attacks, this algorithm creates a data structure with unpredictable timing which eliminates the effectiveness of D.O.S. attacks while maintaining an optimal speed and original functionality.

The set ADT supports three basic operations, lookup, insert, and delete. We provide pseudocode for those operations here.

Algorithm 1 TUFL-LOOKUP

Input: search element v
Output: v if v is found, null otherwise

```
1: cur ← head
2: level ← head.height
3: done ← false
4: while !done do
5:     if cur.next < v  or  v < head < cur.next then
6:         cur ← cur.next
7:     else if cur.next = v then
8:         head ← CHOOSE-NEW-HEAD(head)
9:         return v
10:    else if level > 1 then
11:        level ← level − 1
12:    else
13:        done ← true
14:    end if
15: end while
16: head ← CHOOSE-NEW-HEAD(head)
17: return null
```

Algorithm 2 TUFL-INSERT

Input: new element v
Output: True if inserted, false if exists

```
1: height ← CHOOSE-RANDOM-HEIGHT
2: location ← FIND-INSERT-LOCATION(v)
3: if v = location then
4:     head ← CHOOSE-NEW-HEAD(height)
5:     return false
6: else
7:     head ← CHOOSE-NEW-HEAD
8:     ADD-ELEMENT(location, v, height)
9:     return True
10: end if
```

Algorithm 3 TUFL Delete

Input: element to delete v
Output: true if v was removed, false if it did not exist

```
1: location ← FIND-INSERT-LOCATION(v)
2: if v = location then
3:     DELETE-ELEMENT(location)
4:     head ← CHOOSE-NEW-HEAD
5:     return True
6: else
7:     head ← CHOOSE-NEW-HEAD
8:     return False
9: end if
```

References

[1] Bethea, Darrell, and Michael K. Reiter, *Data Structures with Unpredictable Timing* ESORICS, 2009.