COMS W4995 Parallel Functional Programming: SeedCracker Project Proposal

A Parallel Minecraft Seed Reverse Engineering Tool

Federick Gonzalez (fag2113), Justin Chen (jbc2186)

22 November 2021

1 Background

Minecraft is the best-selling video game of all time. The original game was written in Java in 2009 by Mojang, with development continuing through its official release in 2011 and up to the present day. The game is set in a blocky, procedurally generated 3D world, in which the player is free to explore, extract and farm resources, build, and (in multiplayer servers) interact with other players.

The procedural generation of this world is governed by a 64-bit "world seed." If the player does not manually enter in a seed, it is randomly generated. Although the seed is normally accessible to the player, the seed may be inaccessible for various reasons, such as the world file being lost, the world being on a multiplayer server, or the world belonging to a streamer or Mojang developer and as a result only accessible through video (or even just screenshots). Thus, whether out of nostalgia for an old now-lost world, a desire to cheat, or as a love project from a fanbase, there is significant motivation to reverse engineer world seeds.

Brute forcing the 64-bit seed would take thousands of years, and would require knowledge of the game logic, so at first it would seem that reverse engineering the seed is an impossible task. However, Minecraft is written in Java. As a result, it has been fully decompiled, allowing for a complete look at the game code. From this, we know that Minecraft uses Java's Random class, which is not very secure. Java Random uses a linear congruential generator (LCG) with a modulus of 2^{48}:

$$seed_{n+1} = (25214903917 \cdot seed_n + 11) \bmod 2^{48}$$

Thus, although it takes a 64-bit seed, only the lower 48 bits are used for most generation. We therefore can attack only these 48 bits independently of the other 16. Furthermore, because randomly generated seeds also use Java Random (by generating two 32-bit integers and combining them together), for these seeds the 48-bit seed actually corresponds to only a single world seed on average, and can be extended to the 64-bit seed nearly instantly. If using a manually entered seed, the search space is larger, but can be brute-forced using other generation features which do use the upper 16 bits.

Although a brute-force attack on a 48-bit integer is feasible (and would take on the order of hours or days), we can further narrow down the search space by taking advantage of Java Random's poor parity. Current state of the art solutions involve combining various structure generation features to find the world seed, including an in-game client mod created by KaptainWutax in 2019 which calculates the seed in real time. One early technique involves the use of slime chunks; this algorithm will be the focus of this project.

2 Explanation of Slime Chunk-Based Algorithm

The slime-based approach to seed reverse engineering was an early proof-of-concept, developed in 2014 by Tim Goddard and presented at New Zealand security conference Kiwicon. An equivalent algorithm was also independently developed by Badel2 and released in 2017.

Slimes are hostile creatures which spawn underground. Their spawning locations are determined by a grid of “chunks” in the world; chunks where slimes can spawn are called “slime chunks.” When generating a chunk, the game generates a seed by adding the world seed to a number generated from the chunk coordinates; if the random number generated from this seed is divisible by 10, then the chunk is a slime chunk:

```
    Random rnd = new Random(seed + chunkVal) ^ 0x3ad8025f);
    return rnd.nextInt() % 10 == 0;
```

Our general approach involves collecting slime chunk coordinates, and filtering all possible seeds by those which which generate a number which is divisible by 10 in said slime chunks. Because $10^{15} > 2^{48}$, 15 slime chunks should be theoretically sufficient to identify a seed (though with such a low number, the possibility of collisions is high). Furthermore, we can optimize this solution by noting that all multiples of 10 are even. We can therefore filter only those seeds which will generate an even number. It turns out that when using Java Random’s LCG, the lower 18 bits determine whether the final bit in the resulting number is 0 or 1, and therefore we can first reduce the search space by filtering the lower 18 bits. Each slime chunk essentially cuts the search space in half; with approximately 18 slime chunks, it becomes possible to find a single 18-bit suffix.

In that case, we are left with only $48 - 18 = 30$ bits left to brute-force, which will provide us with anywhere from a handful to several hundred seeds. This takes seconds to complete. If we provide fewer slime chunks, this leaves us with a larger search space and thus a longer running time, allowing us to therefore adjust the computational difficulty of the problem as necessary.

3 Parallelization

The primary limiting factor of this seed cracking algorithm is the brute force attack on the remaining possible seeds after the filtering on the lower 18 bits. This can be parallelized by partitioning the search space. In Haskell, we can accomplish dynamic partitioning using Strategies, as with Marlow’s Sudoku solver; this should translate nicely to our search problem. We can also similarly parallelize the pre-filtering step, but the execution time on this step should already be negligible compared to the main search even when executing sequentially. An alternative parallelization strategy would be to parallelize the process of checking an individual seed, by partitioning on the chunk coordinates which need to be checked. This may also lead to some speedup, but partitioning the seeds is likely simpler and may likely be more fruitful due to the large size of the search space.