Cracking Open Gran Turismo Spec II's Randomizer

Azullia / 0xFC963F18DC21

November 7, 2024

Abstract

Gran Turismo 4 Spec II is a modification for the NTSC-U release of Gran Turismo 4's Online Public Beta. One of its main defining features is a toggleable prize car randomizer, which randomizes what cars one receives as prizes for various actions, picked based on one's in-game username. We will discuss the inner workings of Gran Turismo 4's internal PRNG, and how it is used by Gran Turismo 4 Spec II to generate random prize cars.

Preface

Gran Turismo 4 Spec II is an ISO patch modification for the US soft-release of Gran Turismo 4's Online Public Beta created by TheAdmiester. It brings a lot of additions and changes to the main game, but one of its main draws / features is its prize car randomizer.

Debuting in 2023 as the initial public Randomizer build, the feature was brought into Spec II as a toggleable option. Both versions share the same apparent functionality: changing any prize car[1] into a randomly-selected option based on one's in-game username and the circumstances that one is winning a car in.

As with a lot of other randomizer-adjacent modifications for games, of course, there will be people who wish to know how it internally works, so as to:

- Find an optimal seed to route through.
- Have a really fun or overpowered seed to play through.
- Have an intentionally bad seed as a challenge.

And many more possible reasons. As a disclaimer however, the method described within this paper does not work with the original public Randomizer build from 2023.

[1]Prize cars are awarded by Gran Turismo 4 typically for completing all tests in a License grade with Bronze, Silver or Gold level times, winning all races in a single-race event, winning the overall championship of a championship events, completing sets of Driving Missions, etc..

Pseudorandom Number Generators (PRNGs)

With most if not all kinds of “randomness” that occurs in computer programs, they are never *truly* random, as that is antithetical to how computers work (i.e. by following instructions in a sequence). Without external inputs of data[2], computers cannot ever truly replicate something that is truly and irrefutably random[3].

Keeping that in mind, most if not all programs use *Pseudorandom Number Generators*. As their name suggests, they merely “fake” or “mimic” true randomness, and are not actually truly random. Typically, these take the form of functions of the form:

$$\text{PRNG}(\text{state}) := (\text{pseudorandom output}, \text{new state})$$

Random numbers are generated by repeatedly inputting the new state[4] as the next inputs to the PRNG algorithm. The pseudorandom output can then be used in any way the programmer sees fit in order to generate “random”-looking values.

Gran Turismo 4’s PRNG Algorithm

Gran Turismo 4’s main exposed interface for pseudorandom number generation is `MRandom`. This is a class exposed in Adhoc[5] that is used in all places where Gran Turismo 4 requires random-looking values[6].

`MRandom`

`MRandom` tracks a simple state: a single unsigned 32-bit integer (which represents a whole number between 0 and 4294967295 inclusive). This state is then fed into its PRNG function inside, which outputs and stores a new state, along with a value that is the “random-looking” value generated alongside. It performs this using a small portion of code inspired by the CRC32 algorithm.

CRC32

The CRC32 algorithm is a checksumming function. Its job is to take a piece of data, and perform some operations to mix groups of bytes in the data together

[2]Cloudflare does something cool for their secure random number generators involving some lava lamps (`https://www.cloudflare.com/en-gb/learning/ssl/lava-lamp-encryption/`), for example.

[3]By *truly* random, this means that any output is completely unaffected by any surrounding state or previous and future inputs or outputs.

[4]The *state* of a PRNG can be as simple or as complex as it needs to be.

[5]Adhoc is a scripting language used by Polyphony Digital for certain tasks in Gran Turismo 4 onwards. See `https://nenkai.github.io/gt-modding-hub/concepts/adhoc/adhoc/` for more information.

[6]The used car dealer uses `MRandom`, for example, to generate the mileages based on the used car cycle week number and car index.

in the order the data comes in. Normally, it can be used as a form of file integrity check (i.e. checking if a file is undamaged / unmodified), but the same property that makes it effective for doing so[7] also allows it to be a very simple PRNG algorithm.

So how does `MRandom` work?

Essentially, `MRandom`'s PRNG function looks like the following:

```
def mRandomNext(seed: UInt32): (UInt32, UInt32) = {
  var temp: UInt32 = (17 * seed) + 17
  var result: UInt32 = 0

  for (_ in 0 until 4) {
    val index: UInt8 = result ^ temp
    temp = temp >> 8

    result = (result >> 8) ^ CRC32TABLE(index)
  }

  return (result, (17 * seed) + 17)
}
```

It simply operates on 8 bits of the 32-bit integer at a time, calculating a lookup index (that wraps around 0 through 255), that points to a value in a precalculated CRC32 lookup table of values[8], both using the Exclusive Or[9] operation.

Although, this doesn't explain how we get useable numbers within a range from these generated pseudorandom values, since without any extra operations, we are stuck with a number anywhere between 0 and 4294967295. This is where the next two functions come in.

`GetRange` and `RandomInt32ToFloat`

`GetRange` wraps around the raw PRNG function to allow it to generate random whole numbers within a range.

```
def getRange(incMin: Int32, excMax: Int32, seed: UInt32): Int32 = {
  val (rval, _) = mRandomNext(seed)
  val multiplier = randomInt32ToFloat(rval)

  // The decimal of the result of the multiplication is chopped off.
  return ((excMax - incMin) * multiplier + incMin)
}
```

`RandomInt32ToFloat` translates an unsigned 32-bit integer into a decimal value between 0 and 1, where each number between 0 and 4294967295 is

[7]A large cascade effect, where a small change in input makes a large change in output.
[8]The first table in `https://web.mit.edu/freebsd/head/sys/libkern/crc32.c`
[9]`https://en.wikipedia.org/wiki/Exclusive_or`

translated evenly into that range (with the caveat that 1 is never returned, hence the maximum given in `GetRange` is exclusive).

Spec II Prize Car Randomizer

In Gran Turismo 4 Spec II, if the prize car randomizer is enabled, when one would normally earn a prize, instead of awarding the prize associated with that instance of a prize, it will randomly select a car from a list of all cars available in Spec II.

How it randomly selects a car is based on the username, the current event that is giving the prize car, and what type of trigger it is:

- Username is self-explanatory. It is the up to 30-long string one inputs when making a save on a memory card for the first time[10].
- Events are a list labels of events that you can win a prize car in, they represent things like event hall events (Sunday Cup, etc.), License grade tiers (B Bronze, etc.) or Mission Set completions (e.g. One-Lap Magic 30-34).
- Types of triggers denote what kind of condition it was activated (an event win, mission set completion, license grade completion, etc.).

These parts are then put together into one big string, and that is then used as the seed whenever one wins a prize (e.g. Username “Foo”, event Sunday Cup (“am_sunday_0000”) and type “win” is put together to become “Fooam_sunday_0000win”). The question is then:

How does it turn that text into a number to feed into `GetRange`?

Fowler-Noll-Vo Hashes

A hash function is a way to generate a “summary” of a piece of data, similar to a checksum. They also ideally exhibit the same property as a checksum, where a small change in input data creates a massive change in the final hash output.

Gran Turismo 4 Spec II uses a variant of the Fowler-Noll-Vo algorithm, a very simple to implement hashing algorithm, especially for text / strings. Its implementation in Spec II is specifically the FNV-1a variant, which looks like:

[10] If entering usernames there, **do not** use the HD HUD / UI texture pack, as that incorrectly blanks out some of the keys on the improved keyboard.

```
def fnv1a(str: String): UInt32 = {
  // These starting values can usually be substituted for other
  // suitable values in other implementations of FNV-1a.
  val prime: UInt32 = 16777619
  var result: UInt32 = 0x811C9DC5

  for (chr in str) {
    result ^= chr.toUInt8
    result *= prime
  }

  return result
}
```

So the example string such as “Fooam_sunday_0000win” is then passed into `fnv1a` (which in this case becomes 3244945079), and then the result is passed into `GetRange` as the seed.

Final Algorithm

Putting it all together, we have the following algorithm:

```
extern val allCars: List[Car]

def randomPrize(username: String, event: String, trig: String): Car = {
  val seed = fnv1a(username + event + trig)
  val index = getRange(0, allCars.length, seed)

  return allCars(index)
}
```

A very simple algorithm overall, but very effective at its objective of delivering the player random prize cars based on username[11], while being able to choose the same car again should a player decide to (e.g.) repeat an event they’ve won a prize car before.

Appendices

Spec II Adhoc Sources

They can be found on TheAdmiester’s GitHub here: `https://github.com/TheAdmiester/OpenAdhoc-GT4SpecII`.

Please be warned that programming knowledge is needed to understand the code, and a primer can be found on Nenkai’s GT Modding hub at `https://nenkai.github.io/gt-modding-hub/concepts/adhoc/adhoc/`

[11]For cars with multiple possible colours, the seed fed to `GetRange` is based on the PS2’s system clock.

Analyzer Tool Sources

They can be found at `https://github.com/MF42-DZH/GT4S2RC`.

Seed Bruteforcing

With the possibility to now check the cars found in a seed for Spec II's Randomizer, it becomes possible to look though seeds (many, many seeds) in order to try and find a seed optimised for some purpose.

TeaKanji was and is a major help in this process, providing both approximate "viability" values for all cars in Spec II, and a list of cars that are essential for a 100% run using just prize cars supplied by the randomizer. We've both gone through likely millions of usernames, checking their prizes to see if we can either:

- Find a seed with consistently overpowered car drops.
- Find a seed where it is possible to 100% the game in with just prize cars.

The first goal was achieved relatively comfortably by trying to find seeds with a high average car viability, some examples include:

- `GatewomanBoundaries` (with its **five** 787Bs!)
- `ZaniestDisconcert` (with its **B License Bronze** Formula Gran Turismo!)
- **[REDACTED AT THE REQUEST[12] OF CERO, PEGI, THE ACB, AND THE ESRB]**
- `std::reinterpret_cast<float>` (with its **five** BMW V12 LMRs!)

The closest we have gotten to a username that can 100% the game with only prize cars from the randomizer has four cars needing to be bought for a candidate seed, one such example being "AcritanRawest" (case-sensitive). It is theoretically possible that a username that has three cars missing can be found, but we only know its FNV-1a hash of 1369703188.

The 100% Impossibility (as of 2024-11-03)

It turns out that there is no username out of the many, *many* possible usernames[13] that can complete the game to 100% completion with just randomized prize cars. This property can be found by exploiting the implementation of the randomizer. Recall FNV-1a's implementation:

[12](at gunpoint)

[13]There are $93^{30} + 93^{29} + \cdots + 93^{1}$ possible usernames

```
def fnv1a(str: String): UInt32 = {
  // These starting values can usually be substituted for other
  // suitable values in other implementations of FNV-1a.
  val prime: UInt32 = 16777619
  var result: UInt32 = 0x811C9DC5

  for (chr in str) {
    result ^= chr.toUInt8
    result *= prime
  }

  return result
}
```

There are only 2^{32} possible hashes that can be output from this function and used as a seed for `MRandom::GetRange` for selecting cars. If one looks at the algorithm's definition, the `result` variable has no post-processing being applied to it after the main hasing loop finishes. This means one can derive the following property: the FNV-1a hash of any arbitrary string is equal to the hash of its immediate prefix, XOR-ed by the character code of the remaining character, then multiplied by 16777619 (modulo 2^{32}), formally:

$$\begin{aligned}
\forall s \in \text{Non-Empty Strings} \; . \; s &\equiv \text{All-But-Last Character} \diamond \text{Last Character} \\
\implies s &\equiv \text{init} \diamond \text{last} \\
\implies \texttt{FNV-1a}(s) &\equiv (\texttt{FNV-1a}(\text{init}) \oplus \text{last}) \times 16777619 \mod 2^{32}
\end{aligned}$$

One could then see that this can be applied inductively, where you can take successive prefixes of a target string to hash under FNV-1a to "precalculate", so to speak. Given a modified version of FNV-1a that looks as follows:

```
def fnv1aSuffix(initialValue: UInt32, stringSuffix: String): UInt32 = {
  // These starting values can usually be substituted for other
  // suitable values in other implementations of FNV-1a.
  val prime: UInt32 = 16777619
  var result: UInt32 = initialValue

  for (chr in str) {
    result ^= chr.toUInt8
    result *= prime
  }

  return result
}
```

We can then effectively use this property along with our modified FNV-1a to factor out the username's hash to be a "precalculated value" that we then hash along with the concatenated event label and trigger types to generate the prize car list for a particular username hash[14], using something like the following to test a single event's prize for a username hash and then applying it to all prizes and all hashes:

[14]There are many more usernames than hashes, so some different usernames will hash into the same value

```
extern val allCars: List[Car]

def randomPrizeHash(hash: UInt32, event: String, trig: String): Car = {
  val seed = fnv1aSuffix(usernameHash, event + trig)
  val index = getRange(0, allCars.length, seed)

  return allCars(index)
}
```

Out of the possible 2^{32} hashes / prize lists of cars, the following statistics were observed after bruteforcing the data for all those hashes:

- 215 hashes result in a prize lineup missing 5 cars for a 100% playthrough. There are too many to list here.
- 16 hashes result in a prize lineup missing 4 cars for a 100% playthrough.
  - 585265398, 634454505, 752339779, 1392599962, 1520546429, 2103087292, 2258225527, 2311524843, 2776772991, 3019592698, 3194864652, 3250908732, 3289153551, 3597215756, 3605929611, 3634232672
- **1** hash results in a prize lineup missing 3 cars for a 100% playthrough.
  - 1369703188

No possible username hash generate a prize car list with two cars, one car or even zero cars missing for a 100% playthrough with random prize cars only.

However, for the sole 3 cars missing hash, we have ran a brute-force hash reversal to find that one of the possible usernames for that is “`yED)x`” (without quotation marks).

Reversing the FNV-1a Hash

The simple implementation of the FNV-1a hash also lets us perform a brute-force hash reversal for short inputs. It is impossible to, with 100% certainty, obtain the exact input string used to generate a hash, but we can at least make do with having a result string that hashes back into the hash we wanted to brute-force.

We simply need to run its steps in reverse:

1. Reverse its multiplication by 16777619 under modulo 2^{32}.
2. XOR our result from Step 1 by the numeric value of a character.

Repeating these steps for every character we want to prepend to the username being constructed, until we reach back to the original offset of the algorithm of `0x811c9dc5`. XOR is thankfully its own inverse, but inverting a multiplication under modulo is a little more complex. We will essentially need to find a number **In** such that:

$$\begin{aligned} 16777619x &\equiv x' \mod 2^{32} \\ \mathbf{In} \cdot x' &\equiv x \mod 2^{32} \end{aligned}$$

Or more formally, find the *modular multiplicative inverse* of 16777619 under modulo 2^{32}. There thankfully is an optimized solution to this, taking into account that 16777619 and 2^{32} are co-prime (the only number that can divide both of them with no remainder is 1).

The fact that they are co-prime means that we do have such an inverse that we can calculate, via use of the **Extended Euclidean Algorithm**, which we can use to connect 16777619, our mystery number **In**, and their greatest common divisor (1, since they're co-prime):

$$\begin{aligned} 16777619 \cdot \mathbf{In} + 2^{32} \cdot y &= 1 \\ 16777619 \cdot \mathbf{In} + 2^{32} \cdot y &\equiv 1 \mod 2^{32} \\ 16777619 \cdot \mathbf{In} &\equiv 1 \mod 2^{32} \text{ (Multiples of our modulus decay to 0.)} \end{aligned}$$

The Extended Euclidean Algorithm (EEA) takes in two integers, a and b, and return another two integers, x and y, that satisfy the following identity:

$$ax + by = \text{gcd}(a, b)$$

We have already seen that because our two numbers are co-prime, we have a greatest common divisor of 1. If one squints long enough, you can see that **In** fits where x is in that identity, so the EEA can be used here to find our inverse. The EEA can be defined recursively as follows:

$$\text{EEA}(a,b) \triangleq \begin{cases} (1,0) & b = 0 \\ \begin{array}{lrcl} \textbf{Let} & q & = & a \text{ div } b \\ & r & = & a \text{ mod } b \\ & (u,v) & = & \text{EEA}(b,r) \\ \textbf{In} & & & (v, u - (q \times v)) \end{array} & \text{otherwise} \end{cases}$$

Evaluating $\text{EEA}(16777619, 2^{32})$ gives $(899433627, -3513497)$. This means our modular multiplicative inverse **(In)** is 899433627, finally satisfying our equation from before:

$$\begin{aligned} 16777619x &\equiv x' \mod 2^{32} \\ 899433627x' &\equiv x \mod 2^{32} \end{aligned}$$

Taking all of that together, we can derive a way to brute-force a username that is hashed into a desired input.

```
// Consult S2RA/Bruteforce.hs to see the character set.
extern charset: String

type Intermediate = (String, Int, UInt32)

def reverseHash(hash: UInt32, maxLength: Int = 30): Option[String] = {
  val potentials: Queue[Intermediate] = new Queue[Intermediate]()
  potentials.enqueue(("", 0, hash))

  while (potentials.nonEmpty) {
    val (soFar, len, chs) = potentials.dequeue()
    val prepended = charset.map { c =>
      (c +: soFar, len + 1, (899433627 * chs) ^ c.toUInt8)
    }

    prepended.find { case (_, _, h) => h == 0x811c9dc5 } match {
      case Some((n, _, _)) => return n
      case None => prepended.foreach(potentials.enqueue)
    }
  }

  return None
}
```

There is the caveat that this algorithm is very memory inefficient, and is extremely slow. There may not also be a username returned, since the character set typeable into Spec II's username input keyboard is limited.
The username that we have found earlier follows this path to be reversed:

$$
\begin{aligned}
&(1369703188 \times 899433627) \oplus 120 \text{ (`x')} \equiv 55520612 \mod 2^{32} \\
\implies &(55520612 \times 899433627) \oplus 41 \text{ (`)')} \equiv 1710164901 \mod 2^{32} \\
\implies &(1710164901 \times 899433627) \oplus 68 \text{ (`D')} \equiv 2019549347 \mod 2^{32} \\
\implies &(2019549347 \times 899433627) \oplus 69 \text{ (`E')} \equiv 4228665076 \mod 2^{32} \\
\implies &(4228665076 \times 899433627) \oplus 121 \text{ (`y')} \equiv 2166136261 \mod 2^{32}
\end{aligned}
$$

2166136261 is the base 10 / decimal representation of `0x811c9dc5`, so we finish there. Unwinding the calculation backwards, the username formed is "`yED)x`", just as mentioned before.

Acknowledgements

- **Nenkai**: Disassembly Help and GTHD Debug Executables Archive
- **TeaKanji**: Car Viability Info, 100% Necessities Info, Bruteforcing Script Help
- **clienthax**: The `PS3GhidraScripts` package for Ghidra
- **beardypig, astrelsky**: The `ghidra-emotionengine` scripts for Ghidra
- **mrxbas, Yellowbird**: General Support
- **TheAdmiester**: Creating Gran Turismo 4 Spec II