Big Deal

A new program for dealing bridge hands

Hans van Staveren

sater@xs4all.nl

September 8, 2000

Introduction

During the Olympiad 2000 in Maastricht a newly developed program, called ***Big Deal*** generated all deals. There were good reasons why existing programs were not considered adequate anymore, and in this abstract these reasons are summarised.

There have been too many incidents in the past where whole series of deals were recognised from previous occasions. There are also good reasons to assume the security of existing software is inadequate, although no successful security breaches are known. It is better to keep it that way.

In the rest of this article the most common flaws in existing dealing programs are highlighted, including the one that has almost certainly been responsible for many of the incidents where whole series of deals have reoccurred. A design for a new program is described, and details about the implementation is given.

The software, including the source code, is freely available. Bridge organisations world wide and at all levels, from the Bermuda Bowl to club events, are encouraged to use this program to generate their deals.

Authors and sponsors

The original thoughts that eventually led to the development of this software occurred to Hans van Staveren. Jeroen Kuipers designed and implemented most of the mathematical code in the program, under supervision of Professor of Mathematics Koos Vrieze, University of Maastricht. Hans implemented the various options and formats in the program, and maintains the current source code. The Dutch Bridge Federation encouraged the development, and paid for some costs. Ton Kooijman persuaded the WBF to use this program at the Olympiad. Marijke van der Pas suggested the name ***Big Deal***, since the size of the numbers is indeed the most significant feature of this software.

We are grateful for the feedback of various unnamed individuals on the Internet.

I dedicate this program to the memory of my Ineke.

Table of contents

Why a new program?

Given the fact that worldwide there must be many dozens of bridge dealing programs around, it could well be asked why we need yet another one. In this article we hope to explain why a program based on a totally new design stands a lot better chance of living up to the expectations people have for dealing software. And although it is probably too much to hope, our final goal is to have this program replace all others in use today. What reasons can we give?

During the course of many years various incidents happened around the world, where players recognised deals, usually even whole series of deals, that had been played at an earlier edition of the same tournament, or sometimes even at a different tournament altogether. Not all these occurrences could be attributed to human error, and the suspicion against the software rose. Only just recently the IBPA wrote a page-one article in their bulletin[1] about this phenomenon. Some years ago the author started to think about the issue of deal generating software from a theoretical point of view. He discovered some generic flaws in commonly used programs that were not easy to fix without a total redesign.

Requirements

Before we can design new software it should be clear what it should achieve. Dealing software is supposed to simulate correct shuffling by the players themselves, and should deviate from this goal as little as possible. Therefore dealing software should conform to certain minimum requirements:

1. The software should be able to generate every possible bridge deal, since that is also possible with manual dealing
2. The software should generate every deal with the same probability, without being influenced by the board number, previous hands or any other circumstance
3. It should be impossible to predict deals, even after having seen all other deals in the session

Clearly software has to pass the first requirement to even have a chance at the second.

Furthermore it should be possible to verify the software by anyone willing and capable of doing so. To make this realistic both the design and the source code of the software should be freely available, since that maximises the chance of catching inadvertent errors in design or coding. This will increase confidence in the correctness of the software.

There are those who argue that secrecy of the software can help increase the security, since attempts to break security will be hindered by the lack of knowledge. Historically this argument has always been proven wrong. Security through obscurity never works. Software should be secure even when knowledge about its internals is public.

To keep the software as simple as possible, which reduces the chance of errors, it should not implement too many extras. The minimum is to generate the deals for one session, without frills.

[1] In their IBPA Bulletin 425, dated June 5, 2000.

Flaws in current programs

This section obviously makes some broad statements since there are many dealing programs available in the world, and of course the author does not know the details about all of them[2].

The first fact about most current programs is that the authors have declined to make the design and the source code public. This makes it necessary to guess about how they work, and obviously makes it impossible for anybody but the author of a program to issue statements about in how far the program meets the requirements given above. Still it is possible to make some observations even about the programs that are kept secret.

Lack of randomness

Since programs are deterministic it is impossible for a program to generate randomness. This means all randomness has to be put into the program from an outside source. Dealing software needs enough randomness to be able to generate all possible hands, and not repeat series. This is a very fundamental point, take your time to understand this thoroughly.

Most programs in use today just ask the person behind the keyboard how many deals they should generate, and into which files the deals should be written. After that they just do it. This makes one wonder from where they get the randomness they need to put into the seed of their PRNG[3]. Computers are remarkably deterministic devices, and the amount of randomness that can be gathered without taking extraordinary measures is low. The author used to be a researcher in a computer science project where this same problem caused numerous headaches.

It is highly likely that the amount of randomness that can be extracted without special effort from a standard DOS environment (where almost all programs run) will not exceed 32 bits[4], and even this is optimistic. Furthermore it is highly likely, given a view at the few programs that allow a peek at the source and discussions with authors, that a PRNG with a seed of at most 32 bits is almost universally used. In all cases this leads to a number of values for the seed of at most 2^{32}, which according to the software basics means a maximum of 2^{32} possible series of PRNG values. This, according to the birthday paradox[5] will lead to a likely duplicate series after at most the square root of 2^{32}, which is at most 2^{16} or about 65000 runs of the program. And remember that the same series of numbers from the PRNG will lead to the same set of bridge deals. If the amount of randomness would be a less optimistic and more realistic 20 bits, this would lead to likely duplicates after about 1000 runs.

We believe that the information in the last paragraph is the explanation for many, if not all of the instances of repeated series of hands that are not caused by human error.

Even if the birthday paradox would not be there to lead to duplicate series, let us look at what a seed of 32 bits would mean for the first deal of the generated set of bridge deals. There would be a maximum of 2^{32} possible different deals, while we know[6] that the amount of bridge deals is more than ten times a million times a million times a million that number. So

[2] No programs are mentioned by name here. Interested readers should contact the authors of the software they normally use for details.
[3] PRNG: Pseudo Random Number Generator, a piece of software responsible for simulating random behaviour in a program. See reference section on page 12.
[4] The bit is the unit of information in a computer. See reference section on page 12.
[5] The name for the phenomenon that when selecting random items from a set, the chance of duplicates is much higher than most people would expect, see reference section on page 12.
[6] For calculation of the total number of possible bridge deals see reference section on page 12.

clearly any program that starts with less than at least 96 bits of randomness fails the requirement of being able to generate all deals.

All the above arguments assume the rest of the program is absolutely perfect, if not it can obviously make the probability of duplicates only even higher.

Lack of security

Now let us think a moment about the requirement that it should be impossible to guess deals in a session after having seen other deals. Let us assume that some bridge players, or even a less scrupulous National Organisation, knowing which dealing program will be used at the next important championships, want to increase their chances to win. They hire some software specialists and some mathematicians with experience in the science of cryptography[7]. They look at the source code if they can get it, and if not they perform some reverse engineering[8]. Whichever way they do it, they now know how the program works. In particular they know how the program maps a series of PRNG values to bridge deals. It is extremely likely that the first deals in a session allow them to deduce the first numbers in the series generated by the PRNG. Now given a usual implementation of a PRNG the first couple of numbers in the series make it possible to work out what the seed was, which means the rest of the series is known, which means that the rest of the deals are known. This makes playing them suddenly a lot easier.

As far as I know nobody has as yet performed this last feat, but then again if someone had they would not make it public. Developments in computer hardware, getting smaller and cheaper every day, increase the likelihood that this could happen during a session. Programs had better be designed to make this impossible, and current knowledge in cryptography is sufficient to achieve this.

Design principles of the new software

To avoid all these flaws, the new software was designed according to a couple of principles:

1. Gather more than 96 bits of randomness to stand a chance of satisfying the requirement to be able to generate every possible bridge deal
2. Use a PRNG with good statistical properties to satisfy the requirement of generating all hands with equal probability
3. Use a PRNG with strong cryptographic properties to satisfy the requirement of not being able to predict deals
4. Implement many different output formats to make it easy to use this software instead of as many others as possible
5. Make it as difficult as possible for the operator of the program to make the sort of mistake that would lead to a duplicated series of deals

It will be shown that the first three principles will lead to the use of a so-called cryptographic hash[9].

The fourth principle will lead to an extensible design, where adding an output format is a relatively simple task, and the operator has the choice of which format or formats to use.

---

[7] For a very short explanation about what cryptography is see the reference section on page 13.

[8] Reverse engineering is taking the executable of a program and translating it back to source code. This is a complex operation, but history has shown that it can always done by determined and competent people.

[9] A cryptographic hash is a sort of mixing function for bits, which cannot be run backwards. See the reference section at page 13.

The fifth principle will be taken to extremes: as far as possible the software should view the operator as an enemy in the struggle to generate unique sequences. Whatever the operator tries, he should not be able to get the program to generate the same sequence twice.

Implementation

So how is this new software implemented? During the running of this program various phases can be distinguished:

1. Getting the right amount of randomness.
2. Generating a PRNG sequence with enough security.
3. Converting the PRNG numbers to bridge deals.
4. Writing the bridge deals to a file in the right format(s).

We will describe the details of all these phases, and some implementation details in the next subsections.

Acquiring randomness

As we have stated before, computers do not easily supply large amounts of randomness, and our design goal was to start with at least 96 bits. We decided to get 160 bits, since that is both comfortably larger than the absolute minimum, and a convenient size for a strong cryptographic hash function that is available in the public domain.

So where do we get randomness? Actually randomness can never be found in a standard computer[10], it has to come from the outside world. Things like the timing of keystrokes or mouse movements or revolutions of hard disks are typical sources. Because we did not want to place too many requirements on the environment[11] we decided to go mainly for the timing of keystrokes. To do this the program asks the operator to start typing until instructed to stop, and then gathers both the keystrokes and the timing between keystrokes.

For each piece of information, the key struck and the timing, we guess the amount of randomness in the information, and are pessimistic in our guesses. For example, the operator might have a certain rhythm in his or her typing, so we discount the part of the timing greater than 1/8th of a second. Also when keys are struck timed very closely apart, or when the same key is struck twice we are extremely pessimistic about the amount of randomness in the information. All in all, using a timer accurate to a millionth of a second, we generally guess only some five bits of randomness. We count at most two bits of randomness for the keystroke itself.

We continue to gather information until we have reached the amount of randomness we are aiming at (160 bits), plus some reserve (currently 33%) because of our general conservative attitude. All this pessimism in estimating randomness could lead to the operator having to type slightly longer, but it usually still only requires the operator to type some dozens of keys, which takes somewhere between five and ten seconds, which should be no trouble at all.

Now we have a whole lot of information, which according to our pessimistic estimate contains 160 bits of randomness. So how do we squeeze the non-randomness out, and convert

[10] It would be nice if computers had hardware random generators. These are actually easy to make for a couple of dollars extra, using resistors and quantum mechanic principles. Someday in the future that might be standard, but not today.

[11] The computer might not have a mouse, or a hard disk. Also we want the program to be portable to DOS, Windows, Unix, Linux, MacOS, etc. Currently it runs on DOS and Unix. The program is easily extendable to make use of other sources of randomness if present.

all this information to a seed of 160 bits? We use the method recommended in RFC1750[12], and run all information gathered through our cryptographic hash called RIPEMD-160[13]. So eventually we have a truly random 160-bit seed to work with.

Does all this mean that the new program is not vulnerable to the birthday paradox? No, the birthday paradox is always there. Remember though that the birthday paradox comes into play when we are close to having generated a number of series of deals, equal to the square root of the number of available seeds. In this case the number of available seeds is the number of values in a 160-bit number, which is 2^{160}. The square root of 2^{160} is 2^{80}, and 2^{80} is equal to about 10^{24}. So after a million times a million times a million times a million series of deals we run into a serious risk of duplicates. To illustrate the low risk of this let us assume that all humans on Earth deal series of hands using this software, day and night, with one series per second per human. It would then still take over six million years to get into the danger zone. Somehow I do not believe that ***Big Deal*** is still used by then, if only because the game of bridge might have changed.

One thing should be mentioned here: even if the estimates for randomness turn out to less pessimistic than thought, and the amount of randomness bits gathered turn out to be less than 160, that will only have an effect on the number of series that the program can produce. If the number of random bits is a substantial part of 160 the requirement of not reproducing series should still be met. Furthermore, this does not influence the statistical property of any series of hands, as will be seen further on.

A secure PRNG

Starting with our 160-bit seed we need to implement a secure PRNG. To do this we use our secure hash algorithm RIPEMD-160 again in the following way[14]:

- We use a 160-bit hash from information about the operator of the program, gathered at first use
- We use our 160-bit seed
- We use a 32-bit counter
- For each new pseudo-random number we increase the counter by one, and run the operator-info, the seed and the counter through the secure hash, resulting in a 160-bit number
- This 160-bit number is our next pseudo-random number

The series of resulting numbers is guaranteed to be pseudo-random, since if it wasn't it would mean that the structure in the input of the secure hash (two 160-bit constants and one increasing counter) would be visible in the output, which would make the secure hash breakable. Many renowned cryptographers worldwide have tried to break these hashes[15] and could not, so we are going to take their word for it.

---

[12] RFC1750: Randomness Recommendations for Security, by Eastlake, Crocker and Schiller. This is a document from the Internet defining documents called "Request For Comments", and deals with the subject of randomness for cryptographic purposes. We followed their recommendations.

[13] RIPEMD-160 is a 160-bit cryptographic hash function, designed by Hans Dobbertin, Antoon Bosselaers, and Bart Preneel. It is intended to be used as a secure replacement for the 128-bit hash functions MD4, MD5, and RIPEMD. MD4 and MD5 were developed by Ron Rivest for RSA Data Security, while RIPEMD was developed in the framework of the EU project RIPE (RACE Integrity Primitives Evaluation, 1988-1992). RIPEMD-160 is a strengthened version of RIPEMD with a 160-bit hash result, and is expected to be secure for the next ten years or more.

[14] As recommended by Carl Ellison for the IEEE P1363 draft standard.

[15] These same hashes are used for digital signatures in financial transactions, so breaking these hashes would have a major effect on the financial world. This of course is the reason for selecting standard hashes: it gives one confidence that they have been very thoroughly analysed.

The series of numbers can only repeat if all the information going into the hash is the same. The counters are always the same (1,2,3,…), so the only way to have a repeat of the series is to have the operator-info the same and the 160-bit random seed the same. If the operator identified himself uniquely to the program, no one with a different identification could accidentally generate a same set of deals, even if he managed to get the same 160-bit random seed.

The series of numbers is secure, since there are only three ways to predict the next number, even if one is to assume that the information about the operator is well-known.

- One is to guess the 160-bit seed, which cannot be done, since the randomness used for the seed is unpredictable even to the operator.
- Another way would be to manage to find the original of the hash from the output. This would necessitate running the hash backwards, which for a cryptographic hash cannot be done.
- A last way would be to just try all the possible 160-bit seeds and hope to get lucky. The universe will not exist anymore when this process finishes.

All this together gives us a secure, non repeating, pseudo-random stream of 160-bit numbers.

Making bridge deals

So the next task is converting these 160-bit pseudo-random numbers into bridge deals. It is more or less conventional for bridge dealing programs to implement shuffle algorithms, where rows of 52 cards are mixed by swapping cards determined by pseudo-random numbers. If we would do it this way the correctness of our program would depend both on the pseudo-random numbers and the implementation of the shuffle algorithm. Because our pseudo-random numbers are very large we can do something more elegant instead.

The number of possible bridge deals is just a bit under 2^{96}, as shown in the reference section. This means that it should be possible, at least in theory, to design a pair of functions, one of which can convert any bridge deal into a 96-bit number, and the other can convert that 96-bit number back into the same bridge deal. Jeroen Kuipers has indeed designed and implemented this pair of functions, and he proved the correctness of his implementation in a document provided with the software[16]. We use the second function of the pair in the program.

So the algorithm to convert the stream of 160-bit pseudo-random numbers into bridge deals runs as follows:

1. First reduce the 160-bit number to a 96-bit number by just throwing away 64 bits. Since the original 160-bit number is pseudo-random this 96-bit number will also be pseudo-random.
2. The number of bridge deals is somewhat less than 2^{96}, so if this 96-bit number is too big throw away the whole number and go back to step 1 with the next 160-bit pseudo-random number.
3. Convert the 96-bit number into a bridge deal. Given the pseudo-randomness of the original numbers this should be able to generate each possible bridge deal, with equal probability.

This in effect finishes the innovative parts of the program.

[16] In the files Godel.tex, Godel.ps and Godel.pdf. Kurt Gödel was a German mathematician whose works had a profound effect on mathematics. In his most famous theorem he used the technique of creating an equivalence relation between symbol strings and numbers. Although we do not use his famous theorem we honour him by using his name, because we use an equivalence relation between bridge deals and numbers.

Generating the output

Much more mundane than the previous parts of the program is the output phase. The only interesting part here is the fact that the operator can choose which output format, or formats he or she desires. Currently the program implements the BRI, DGE, DUP, BER, BHG and PBN formats. It is the intention of the authors to add any commonly occurring other format, since the goal of this project is to have this program replace existing programs with as little pain as possible to the operator.

The program is structured internally in a way that makes it easy to add output formats without having to touch the code that actually generates the deals. This will lessen the risk of modifying the program for output purposes. The output code is table driven, meaning adding a format will just need another line in a table in the program and extra code for that format.

Requests to add another format can be directed at the authors, and if documented clearly[17] it should be expected that within a month at most the next version of the program, including this format, should be finished and published. Only very rarely used formats might not get implemented.

Options

Since the main goal of the program is to make it as easy as possible to uniquely generate the deals for one session, the program should have as little options as possible. But since during development and testing some options were useful a choice was made to deliver the program in two versions, simple and complex.

The simple version, *bigdeal,* should be used by anyone that just wants to generate deals to play. It has no options that can influence the gathering of randomness, does not log any of the internal numbers, reducing the chance of a clerical error that could publish it, and is limited to dealing 100 boards, enough for the longest session.

The complex version, *bigdealx*, allows various ways to add randomness at the command line, the generation of debugging, limited statistics and more. It is also not limited to 100 boards. This version is not recommended for common use. For details please read the documentation about use of the two versions that is included with the program.

Implementation details

The program is written in the language C, and is currently portable between DOS and Unix. It should port to MacOS without any trouble, but we do not have that ourselves. Volunteers are welcome. The line count for the source code is in the order of 2500, but that includes 800 lines for all the output formats, 400 lines of code for the complex version only and 400 lines of unmodified RIPEMD-160 code. That leaves about 900 lines for the core code. The program is actually pretty easy to understand, and we encourage everybody with programming skills to have a look.

Given the fact that the program uses multiple precision arithmetic at various places, and cryptographic hashes, it is relatively slow. But even on the slowest computer we tried it on it still generated some ten deals per second, with current day hardware usually doing more than 200. Since the goal is just to deal the hands for one session that is fast enough.

[17] Documentation can be text, source code, or as a last resort a file in the format accompanied by some text format showing the hands.

Peer review

To increase the chances of finding any design or implementation flaws in the software before use there has been discussion about the design and implementation in two forums, bridge-dealing@egroups.com and developers@cirl.uoregon.edu, and the software has been published on the Internet for almost a year now. Various suggestions have been received from participants in these forums, and some of these have been implemented. Some small implementation deficiencies were found, but none of them seriously effecting the goal. It is believed the software is stable and without significant errors.

Peer review never ends, so the authors of the software remain available to answer any questions. If anyone has a reason to suspect that this program does not live up to its promises, for whatever reason, the authors would appreciate feedback. Documentation included with the program state contact addresses and details to include in such feedback in order to increase effective communication.

Testing

Now it is all well and good to state that the program should be great and dealing bridge hands exactly as theory predicts it should, but a healthy dose of suspicion will lead people to ask for results of testing.

Now testing can only ever show the presence of errors, and never their absence. Remember that all existing programs with too low randomness at the start pass a lot of tests too. If a lot of testing has not shown anything suspicious there is some reason to feel confident, but the most confidence should always be reached through continuous peer review.

Also, the only really good test is a test whether the program generates all deals with equal probability, but given the enormous number of deals this is not at all practical. So the only realistic possibility is to compute some set of statistics, and hope that any errors in design or implementation will show up using those tests.

Having said all that a Fin called Kaj Backas[18] has heavily tested the program statistically. The authors themselves have also generated millions of deals in tens of thousands of sets, running various statistics relevant to bridge players, such as distribution frequencies, point counts, K being behind the A, etc, etc. All tests conformed to theoretical expectations. So both design and testing suggest the program works as advertised.

Furthermore the program has already been in use with the Dutch Bridge Federation for a complete season without complaints from the players.

How to get it?

The program is downloadable from www.xs4all.nl/~sater. Use of the program to deal hands for any tournament is without restrictions. Commercial use of the algorithms, for example in bridge playing software, can be arranged for a nominal fee. The fee will be transferred to a fund that will promote something in bridge, as yet to be determined.

[18] Kaj G. Backas <Kaj.Backas@systecon.fi>

How to use it?

The first time the software is used it will ask the operator to identify him self and some basic questions about the preferred output formats. For the rest it needs the number of boards, the filename prefix (the part before the dot), and some input to gather randomness.

There is however an extremely important point, which is actually the same for this program as for all others. If you trust the software you are using, and we hope we have convinced you that this software is to be trusted, you should never look at the deals generated. Even if you deal 16 boards, of which in the first three North has an eight-card spade suit, so be it. If you would deal by hand it could happen too. If you ever decide to manipulate the output, for example by dealing the session again if you do not like the hands, you are interfering in the interaction between players and deals. No player has the right to assume North cannot have an eight-card spade suit on board three because he had it on boards one and two.

There has been a deliberate decision not to generate statistics about the hands generated in the simple version of the program. The statistics of one session are meaningless anyway. The danger of generating statistics is that a tournament organiser would see the statistics of hands he generated, and would decide to deal again, because of the fear that the players would complain. By not generating statistics we help him not to notice until the boards are played.

The software is not designed to prevent human error. Care should still be taken not to use old files lying around on the same computer. One way of doing this, needing strong discipline, is to move all files with deal information to an archive directory after use. This way they are still available for reference purposes, but will not be used again accidentally.

Conclusions

New dealing software has been created. It is good, it is free and it is easy to use. Whatever software you are using now, please consider very seriously to switch. If you need another output format you know where to find us. If you hesitate for another reason, please contact the authors of your current software, show them this article, and ask for comments. Should you be persuaded to keep using your current software, please let us know why, because it means we failed.

Reference information

In this section some background material is supplied for reference purposes.

Number of bridge deals

The number of possible bridge deals is equal to $52!/13!^4$, where ! is the mathematical factorial symbol(4! = 1x2x3x4 = 24, 13! = 1x2x…x12x13 = 6227020800). The number of possible bridge deals computes to 53,644,737,765,488,792,839,237,440,000 which is about 0.7×2^{96}. This last way of writing the number is very relevant to this article.

What is a bit?

Computers store all numbers they work with in memory cells with a certain size. This size is measured in bits (the word *bit* comes from *B*inary dig*IT*). A memory cell of *n* bits wide can store all numbers from zero to 2^n-1, for example an 8-bit cell can store all numbers from 0 to 255. Common cell sizes in computers are 1, 8, 16 and 32 bits with maximum values of 1, 255, 65535 and 4,294,967,295 respectively. The size of all information around computers is likewise measured in bits, with information measuring *m* bits if it needs a memory cell of at least *m* bits to store it. The unit byte is also often used, a byte measuring 8 bits, but we will not use the unit byte in this document.

Randomness and PRNG's

Computers of the sort we use today are totally deterministic machines, which will do exactly the same every time they are started with the same input. This means that any program that is supposed to behave differently every time, like dealing software definitely should, has to gather from the environment some randomness, and use that to modify its behaviour. It is very common for software to use standard functions called pseudo random number generators, or PRNG's, that return a different pseudo-random value each call. This series of values will pass certain statistical tests. The way to make a PRNG return a different series of values is to give it a seed value, usually a number of a certain fixed size, again usually at most 32 bits. The randomness gathered from the environment is usually used as a seed. A PRNG is usually implemented by very simple mathematical operations, rarely more than a multiply and an addition per call to the PRNG. After the seed is given to the PRNG the whole series follows from it, so if the same PRNG is given the same seed twice, it will generate the same series twice. The corollary is that no PRNG can ever generate more different series than its seed can have values.

The Birthday Paradox

Assuming some persons are randomly gathered in one room, there will be a probability that two of them have the same birthday. Assuming there are 365 possible birthdays how many persons should be in the room before this probability exceeds 50%? Not many people when asked this question for the first time guess the correct answer, which is an astonishingly low 23! This is the so-called birthday paradox, and its relevance should be clear in this article. The same question can of course be asked of randomly drawing, with replacement, some values from any finite set of values: how many values should be drawn before there is a substantial chance of drawing duplicates? At this point it is not necessary to go into details about the computation: suffice it to say that when the numbers get large there is a substantial probability of a duplicate when the size of the draw reaches the square root of the size of the original set.

Cryptography and its relevance to dealing

Cryptography can be defined as the art of communicating while keeping secrets. It could be asked what this could possibly have to do with dealing hands for a bridge tournament. Looking at it in a certain way, you could see that the organisation of a tournament communicates a bridge deal to the players when they take it out of the board. The organisation wants to keep the rest of the deals a secret until the players have to play them. Given this way of looking at it, it is not surprising that cryptographic principles play a role in the design of the new software.

Cryptographic hashes

What is a hash? A hash is a piece of software that takes an arbitrary amount of information and computes from that information a fixed length checksum. What is a cryptographic hash? The important difference between a hash and a cryptographic hash is that in the case of the former, given a checksum, it could be possible to find an amount of information with that checksum, while in the case of the latter that is computationally infeasible[19]. So you cannot run a cryptographic hash backward: you cannot find the original information from the hash. A necessary, but not sufficient, condition for a hash to be cryptographic is that the checksum is of a large size, typically at least 128 bits.

[19] In plain language this means you can try till the end of time before you find it.