Deconstructing Dice and Destiny: Reverse Engineering for Deterministic Insights into a Probabilistic Game

William Mahoney[1], Hans-Jürgen Schäfer[2] and Øystein Schonning-Johansen[3]
[1]School of Interdisciplinary Informatics, University of Nebraska at Omaha, USA
[2]Maintainer, The Backgammon-Computer Rating List, Germany
[3]Department of Electrical Engineering and Computer Science, University of Stavanger, Norway

wmahoney@unomaha.edu
dankon87@tutanota.com
oysteijo@gmail.com

Abstract: All commercially produced works fall under the protection of copyright law, as governed by applicable intellectual property statutes. By default, copyright ownership is attributed to the original author of the work. For instance, if an individual creates and distributes a graphic design, they retain exclusive rights not only to the physical reproductions (e.g., prints or posters) as well as the original creative content. This standard equally applies within software development. A prominent example of a software copyright license is the GNU General Public License (GPL), a widely adopted free software license administered by the Free Software Foundation (FSF). If you modify a GPL-licensed program and distribute it, you must also make the modified source code available under the same GPL terms. This paper describes a reverse engineering effort to determine if a certain commercially released copy of a game contains GPL licensed software, in violation of the aims of the GPL. While the commercial software used in the project was sold roughly twenty years ago, the investigation techniques are applicable today and apply not just to the GPL but to intellectual property protection in general.

Keywords: Reverse engineering, Intellectual property protection, Open source, Backgammon!

1. Introduction

Long ago the lead author of this paper was inspired by an article about computer algorithms for Backgammon (Berliner 1980). As he wanted to learn a bit more about the C programming language, and was a frequent Backgammon player, he decided to write a C version of Backgammon to see if it could play a decent game. The program was named simply BKG. Imagine the surprise when out of the blue, decades later, this same author received an email. Rediscovering the code for BKG, along with some additional introductions, resulted in many back-and-forth emails about the shared interest in the game and in programs that play it. One of the authors maintains a ranking of skill levels in computer Backgammon games, but has not listed all available programs; some seem to play suspiciously like others and thus are not in the rankings (Schäfer 2025).

Soon our group of three had an unusual research question: the open-source GNU Project has a Backgammon game that has been in development for over 20 years, GNUBG. And a certain commercial Windows 2003 Backgammon game appeared to make gaming decisions that were identical to those of GNUBG. Is it possible to determine with reasonable confidence that Strong Gammon II (henceforth SG-II), the Windows game, actually contains software from GNUBG? Although this question, on the surface, does not seem very critical, the heart of the matter is whether we can use reverse engineering to determine if intellectual property theft has occurred. The fact that it is Backgammon is in this context fun but irrelevant; we present methods and results for deciding whether software has been stolen and incorporated in a competing commercial product.

First, consider copyrights. All commercially developed works are subject to copyright protection under applicable intellectual property laws. And by default, the copyright ownership is vested in the original creator of the work; if an individual designs and releases a graphic T-shirt design, they hold the copyright to both the tangible clothing, say, as well as the underlying design of the graphic. This same rule extends to software development. In this context, software, developers retain the copyright to their source code, architectural designs, the technical documentation, and many other project-related assets they might produce. As long as the authorship can be demonstrated, the developer is considered the legal owner of the intellectual property, unless rights have been assigned or transferred by some sort of contractual agreement. A developer may license the copyrighted material, specifying what exact rights the licensee has, or assign the copyright through a legal transfer of ownership. Only under this latter explicit assignment does a copyright (and the legal ownership) transfer to someone else.

One commonly encountered copyright agreement is the GNU General Public License (GPL), a widely used and free software license. GPL is maintained by the Free Software Foundation (FSF) and the purpose of GPL is to guarantee users the freedom to run, study, share, and modify software, commonly referred to slightly in jest as

"copyleft". The GPL enforces a robust copyleft mechanism, meaning that any software that is distributed with the GPL agreement must also be released under the GPL if it is distributed. This ensures the same freedoms are preserved in derivative works, but also has a significant ramification: if you use GPL-licensed code in your software and you distribute that software (*even commercially*), you are required to release your source code under the same license. This is distinct from some of the more permissive licenses like the MIT license.

Is software, licensed under the GPL, present in a commercial product with no acknowledgement of the source of the code? Of course, sometimes that is the case. This paper provides a narrative of one such case, and while the actual software systems in question are not "big money" commercial products, it is nonetheless interesting to detail how to go about making this determination. Thus, the paper discusses methods used to examine SG-II, with the suspicion is that it contains the intelligent move decision software from GNUBG. While this is an old program with small worth these days, the methodology used to make this determination is modern. The paper describes the methods used to show that this is, in fact, the case, that code from GNUBG is used in SG-II.

Section two of the paper outlines a few facts in general about intellectual property protection in the software industry. Section three presents a background in reverse engineering for those not familiar with the tools. Section four, the "meat" of the paper, shows that indeed, SG-II is based upon GNUBG, and evidence is presented that one can even determine the approximate version used. Lastly, section five presents conclusions and closing remarks.

2. Software Intellectual Property

Software intellectual property (IP) theft refers to the unauthorized use, reproduction, or distribution of software that is legally protected by copyright, patents, or trade secrets. IP theft might include outright copying of the source code, using proprietary algorithms without permission of the author, and distributing pirated versions of the code.

Software IP theft can have financial and reputational consequences for both individuals and organizations; the costs include lost revenue from pirated or counterfeit software, diminished brand value, and increased expenses related to legal action if necessary. Startups and small companies might be particularly vulnerable, as stolen code or ideas can be exploited by their larger competitors. American Superconductor, a Massachusetts-based infrastructure provider, had their largest customer illegally obtain the source code for wind turbine controllers, at an estimated cost of $100M (ShardSecure 2025). But large companies can also fall victim, as in the case of Oracle versus Rimini Street (Oracle 2023), where software specialists from Oracle used reverse engineering tools to demonstrate that the code within the Rimini Street suite was actually Oracle software. Methods to combat software IP theft include digitally watermarking the executable file (Mahoney 2018, Mullins 2020), which makes a unique executable file for each software purchase, thus enabling proof of ownership. In this scenario if the theft has occurred one can at least determine which customer provided the IP. Indirectly this makes the reverse engineering process more complex at the same time.

But has software IP theft occurred? Several reverse engineering techniques are useful to help make this determinization: disassembling or decompiling the suspect software to link its low-level instructions or reconstructed source code to the original; using tools to compare the code structure, control flow graphs, or specific function calls and logic configurations to detect similarities; running both the original and suspect software in a controlled environment. In this way one can make a good assumption as to the source of the internals of a particular executable program.

3. Background on Reverse Engineering of Binary Executables

Compiling a high-level language into machine code is a "one-way street". Similar to a hashing function, information is lost along the way. The names of variables are lost. The comments are lost. The original "look" of an arithmetic expression may change. A "for" loop and a "while" loop are identical, making it impossible to faithfully recreate an exact replica of the original code. In terms of reverse engineering software, what you get back from the process is more likely to "give you a pretty good idea" as to what the original source code might have looked like, but you will likely never get an exact picture. For example, a programmer may write:

```
int_variable = variable * 8 + 20;
float_variable = other / 2.0;
result = ( int_variable + other ) * 3;
```

But in the process of compiling, generating machine code, and then looking at the reverse engineered machine code, you may see:

```
int_variable = 20 + variable << 3;
float_variable = other * 0.5;
temp = int_variable + other;
result = temp + temp << 1;
```

Shifting a binary number left by three (<< 3) is the mathematical equivalent to multiplying by eight since $2^3 = 8$. But shifting is significantly faster than multiplying. Addition is commutative, so adding X to Y is the same as adding Y to X. Floating point division, such as multiplying by 2.0, is slow relative to multiplication, so multiplying by 0.5 is faster than dividing by 2.0. Multiplying a number by three is the same as adding the number plus two times the number, and of course, two times a number is the same as shifting the number left by one bit.

So! Information is lost in the process of compiling a high-level language into machine code.

And the challenge of reverse engineering binary code primarily involves disassembling and reconstructing the machine instructions into a form intelligible to humans. The complexity of this task varies depending on the underlying architecture. For instance, in architectures such as ARM or other typical Reduced Instruction Set Computers (RISC), disassembly is facilitated by the uniform instruction length and predictable alignment on specific memory boundaries. Conversely, architectures based on Complex Instruction Set Computing (CISC), such as the x86 family, present significant difficulties due to variable-length instructions and non-uniform encoding. Furthermore, the sheer number and variety of instructions, some of which may be undocumented or context-dependent, exacerbate the complexity of the reverse engineering process (Mahoney, 2021).

Reverse engineering normally uses a mixture of static and dynamic analysis. Static analysis involves examining program instructions without executing the binary, aiming to infer properties that are invariant across all possible execution paths. Techniques such as disassembly, decompilation, control flow graph (CFG) reconstruction, and data flow analysis are commonly employed to derive structural and behavioral insights from the code. Notable static analysis tools include Ghidra (2025), objdump (LinuxDevCenter, 2025), and udcli from the Udis86 project (2025). In contrast, dynamic analysis entails executing the binary to observe its runtime behavior and collect execution traces. However, dynamic analysis is inherently limited to the code paths exercised during execution, potentially missing unexecuted branches or conditional logic. Prominent tools for dynamic analysis include Binary Ninja (2025), Hopper (2025), OllyDbg (2025), Relyze (2025), and x64dbg (2025). IdaPro (Hex-Rays 2025, Pearce 2008) is a hybrid tool that performs static disassembly and analysis but also supports debugger integration, enabling partial dynamic analysis. Additionally, online platforms such as Defuse (2025) offer lightweight, web-based disassembly interfaces. These tools collectively employ a range of methodologies to interpret and understand binary behavior and structure.

Some tools are dedicated exclusively to static analysis, while others incorporate integrated debugging capabilities, supporting dynamic analysis workflows. Utilities such as udcli and objdump are lightweight, command-line–based static disassembly tools that provide raw instruction decoding without runtime context. In contrast, platforms like OllyDbg and x64dbg function primarily as dynamic analysis environments, offering full-featured debugging capabilities for real-time code inspection and execution tracing.

A key capability that has emerged over the past decade or so is the ability for the static analysis tools to make an intelligent guess as to what the original source code might have looked like. To do this, the software analyzes the control flow graph for a function, locating the conditional jumps, what exactly is tested by the condition, making an approximate representation of expressions, and so on. These tools, pseudocode generators, are in addition to the older process whereby an engineer would examine the flowchart of a function, for instance, and mentally create a picture of what the high-level software might have looked like. As discussed above, information is lost, so what comes out of a pseudocode generator is a “rough estimate” in many cases.

4. Static Analysis of SG-II

Using this background on the static analysis of executables, it ought to be possible to compare the binary aspects of SG-II with the known (and available) original source code from GNUBG. In this section we outline the tools necessary, and then the discoveries made using these tools. The authors present a convincing case that the SG-II executable program has utilized open-source in commercial software, where the original code was protected

by the GPL. The latter part of the section is in two parts: discovery of various Backgammon probability and move tables, followed by several comparisons of the pseudocode generated by the reverse engineering tools.

4.1 Description of Tools

The reverse engineer has several tools at his/her disposal, some very basic and some very complex. The authors present here a list of the items used in the investigation.

First, a "hex editor" is critical. What is this tool? It allows one to examine files at a binary level and to search for specific binary sequences, change values at a binary level, and save these changes. Files are shown as a series of hexadecimal (base-16) numbers, usually in a grid format. Every file on a computer, and here we will specifically refer to executable programs, is just a series of bytes; a hex editor shows you those bytes, regardless of what kind of file it is, be it text, an image, a database, etc. Many advanced hex editors include templates, which allow for the user to dissect the header information in a JPG file, for example, or show the sections in a Windows executable file.

A reverse engineering tool is essential, and two were used in this investigation. The first is a commercial product, IdaPro (HexRays 2025), which is quite advanced and can be used for a very large variety of different CPUs and architectures. This product has existed for quite some time and has numerous "plug-in" modules and significant documentation. Second, Ghidra (2025) is a somewhat newer reversing tool originally made within the National Security Agency but now publicly available for use. In both cases the tools can pick apart Windows executable files with great detail, allowing one to see the machine instructions in a variety of ways, such as flowcharts, "flat" disassembly lists of the instructions, and so on. Both contain a pseudocode generator as described previously, which attempts to provide a textual C program representing what the original source might have looked like. These tools assume that the binary software is not heavily obfuscated by inserting false instructions (Shinn 2011); but the typical developer does not go out of his or her way to thwart the reverse engineering process.

A Windows executable file contains quite a bit of information that is hidden. These "portable executable" (PE) files, contain several "headers" that can be indicators about how the PE was compiled. If the binary was made with Windows Visual Studio (VS), for example, it might contain a "Rich Header" (Webster 2017) that contains the version numbers for the tools contained within the VS framework. Tools such as "PE-Bear" (Hasherezade 2025) will break apart the Rich information into these versions.

Lastly there is one additional tool that is critical to the investigation, and that is the fact that the source code for GNUBG is publicly available in their repository (GNUBG 2025). As described in the next sections, using this knowledge one can show, to a reasonable degree of certainty, the version of GNUBG used by SG-II.

4.2 Initial Discoveries

The investigation starts with an examination of the executable program for SG-II using the various tools described above; let's just take a look at it! While it is certainly possible to edit many of the fields in the various headers for a PE file, typical software developers do not see a reason unless there are special circumstances. Assuming there were no special circumstances here, we can put a date on SG-II of August 13, 2003 for this program, as shown in the output from PE-Bear in the following figure:

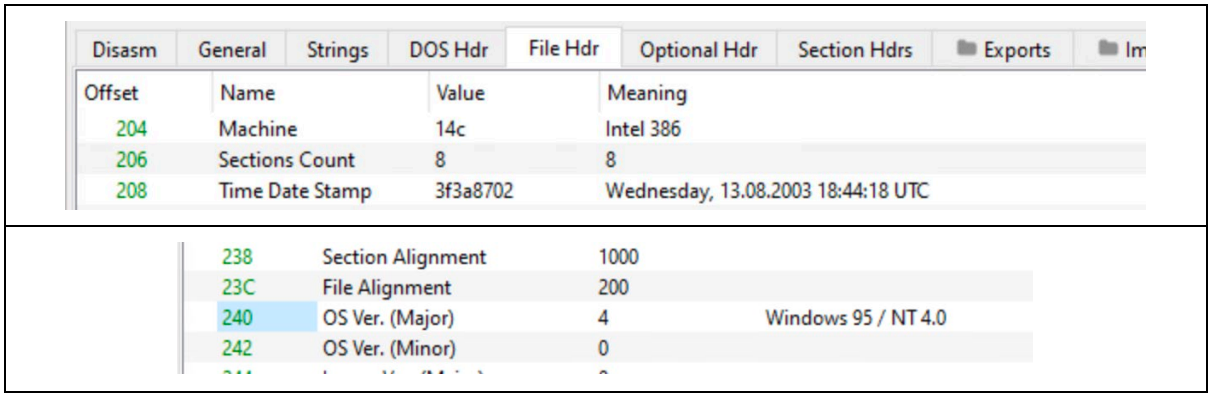

Figure 1: SG-II File Header information with (likely) timestamp above, operating system version ID below

Later the reader will see that this timestamp is likely correct, based upon available versions of GNUBG available around mid-2003. An additional clue that this is correct is provided by the "Operating System Version" number

in the "Optional Header" of the file. This number is four, but the current Microsoft reference for this field starts at five (Microsoft 2025). While possibly not as reliable, Wikipedia (2025) indicates that a value of four says the program can run on Windows 95, likely still popular by the time of August of 2003.

Also, almost all reverse engineering tasks start with using a hex editor to just make cursory observations about the internals of the file. For example, the program was written using the Borland C++ compiler, as shown here:

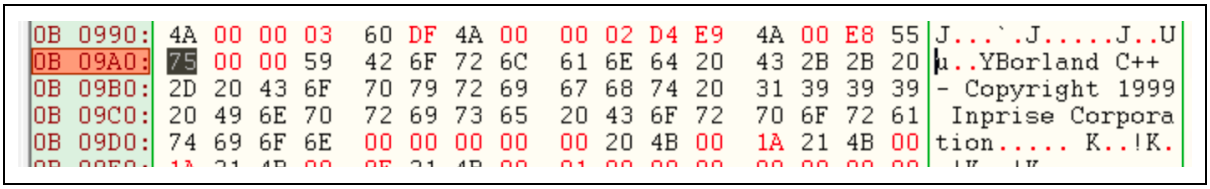

Figure 2: Partial contents of SG-II executable, showing the compiler used

Putting the program into the Ghidra tool, which attempts to determine the compiler environment based on certain characteristics of the program, also says it is Borland C++.

An additional interesting discovered characteristic is in the following figure:

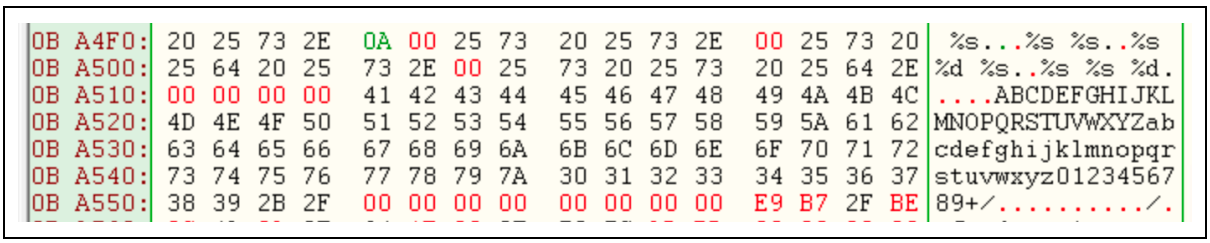

Figure 3: Partial contents of SG-II executable, including a Base-64 encode/decode string

In this figure, two items stand out. Although the program was compiled with C++ there is C code in the source, as the various strings like "%s" and "%d" refer to formatting strings used by the C function group "printf". Secondly, it might be considered unusual to have the long alphabetic string in the executable; this string is used for Base-64 encoding and decoding, used for (for example) sending binary data in text-only emails. What might it be doing in a Backgammon game? And yet one can easily find the same encoding string in the source code for GNUBG. This is not any kind of "smoking gun" but interesting.

```
extern char *PositionIDFromKey( unsigned char auchKey[ 10 ] ) {

    unsigned char *puch = auchKey;
    static char szID[ 15 ];
    char *pch = szID;
    static char aszBase64[ 64 ] =
        "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";
```

Figure 4: Portion of GNUBG 0.02 file positionid.c

Another odd clue is that GNUBG handles several optional Backgammon rules. Most commercial Backgammon games only handle the standard rules of the game, but GNUBG also supports an additional rule, "Beavers". Players can agree to these slight rule modifications and play a marginally different game relative to standard Backgammon. Immediately it is clear that SG-II contains the same "Beavers" rule variation, as shown from data contained in strings within the executable, but also variants named "Jacoby" and "Crawford". And interestingly, GNUBG allows these variations as well.

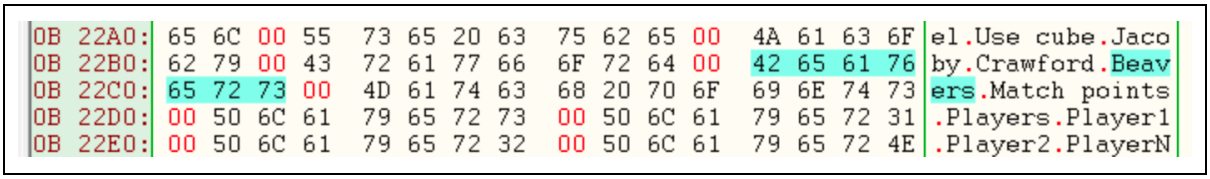

Figure 5: Partial contents of SG-II executable Windows file

Let us move on to additional static analysis of SG-II.

4.3 Analysis of Calculated Moves and Other Tables

GNUBG, as an open-source project, has all the code available to examine (GNUBG 2025). Each of the releases, going back to version 0.12, has a date stamp, with 0.12 listed as "2003-12-13 08:43". Unfortunately, versions prior to 0.12 are not on this site, but a much older version, 0.02, can be found (SourceForge 2025). Our next step is to compare source code from these early versions versus the binary artifacts to be found in SG-II. Initially however it is necessary to have a short tutorial on floating point numbers.

Most (all?) modern CPUs use a standard format for floating-point (as opposed to integer) numbers. Floating point numbers are generally an approximation of the actual value, since there are infinitely many float numbers and not infinitely many bits to store them in! The standard format is IEEE-794 (IEEE 2025) and consists of 32-bits split into a sign, exponent, and mantissa. (The latter, this author has always said, would make a great name for a heavy metal band.) Since it is an approximation, it is problematic to look for floating-point values, expressed in base 10, in the source code, and expect the exact value in the binary. In the process of converting them to the IEEE format, the result may only be a very close approximation of the value you see when examining the executable artifact. One may match three out of four bytes exactly, with the fourth byte of the fractional part of the number off by a few bits. But we will see that in this case the binary has exact, not approximate, matches. Lucky.

The following figure contains three items: first, in upper left, source code from GNUBG "neuralnet.c", which includes a long table of floating-point numbers, used for Backgammon move selection. Second, upper right, the output from a program the first author used to print the floating-point along with it's (reversed) four-byte raw value. Note that the numbers here do not exactly match, for the exact reason described, floating-point is approximate. Finally, under these sections, a portion of a hex dump from SG-II which clearly shows these values are in the executable file.

```
/* e[k] = exp(k/10) / 10 */
static float e[100] = {
0.10000000000000001,
0.11051709180756478,
0.12214027581601698,
0.13498588075760032,
0.14918246976412702,
0.16487212707001281,
0.18221188003905089,
```

```
0.10000000149011612 - cd cc cc 3d
0.11051709204912186 - c9 56 e2 3d
0.12214027345180511 - ae 24 fa 3d
0.13498587906360626 - bd 39 0a 3e
0.14918246865272522 - 4a c3 18 3e
0.16487212479114532 - 3d d4 28 3e
0.18221187591552734 - c0 95 3a 3e
```

```
0B A310: CD CC CC 3D  C9 56 E2 3D  AE 24 FA 3D  BD 39 0A 3E  ...=.V.=.$.=.9.>
0B A320: 4A C3 18 3E  3D D4 28 3E  C0 95 3A 3E  52 35 4E 3E  J..>=.(>..:>R5N>
0B A330: 38 E5 63 3E  05 DD 7B 3E  10 2D 8B 3E  34 D0 99 3E  8.c>..{>.-.>4..>
0B A340: 70 FD A9 3E  35 DE BB 3E  51 A0 CF 3E  65 76 E5 3E  p..>5..>Q..>ev.>
0B A350: 63 98 FD 3E  10 22 0C 3F  F8 DE 1A 3F  AE 28 2B 3F  c..>."?...?.(+?
0B A360: EB 28 3D 3F  D0 0D 51 3F  54 0A 67 3F  CD 56 7F 3F  .(=?..Q?T.g?.V.?
0B A370: BF 18 8D 3F  99 EF 9B 3F  FA 55 AC 3F  E8 75 BE 3F  ...?...?.U.?.u.?
0B A380: D2 7D D2 3F  0A A1 E8 3F  25 8C 00 40  20 11 0E 40  .}.?...?%..@ ..@
0B A390: 19 02 1D 40  59 85 2D 40  2E C5 3F 40  5B F0 53 40  ...@Y.-@..?@[.S@
```

Figure 6: GNUBG 0.12 neuralnet.c floating-point constants, upper left, converted to hex in upper right, and are present in the binary for SG-II below

We can continue this with other integer tables from the GNUBG source from both the file "eval.c" and "pub_eval.c", as shown below. These are from the IdaPro tool and not all integers were converted from hex to decimal, but the table is clear.

```
static int aanCombination[ 24 ][ 5 ] = {
  {  0, -1, -1, -1, -1 }, /*  1 */
  {  1,  2, -1, -1, -1 }, /*  2 */
  {  3,  4,  5, -1, -1 }, /*  3 */
  {  6,  7,  8,  9, -1 }, /*  4 */
  { 10, 11, 12, -1, -1 }, /*  5 */
  { 13, 14, 15, 16, 17 }, /*  6 */
  { 18, 19, 20, -1, -1 }, /*  7 */
  { 21, 22, 23, 24, -1 }, /*  8 */
  { 25, 26, 27, -1, -1 }, /*  9 */
```

```
} *pi, aIntermediate[ 39 ] = {
    { 1, { 0, 0, 0 }, 1, 1 }, /*  0: 1x hits 1 */
    { 1, { 0, 0, 0 }, 1, 2 }, /*  1: 2x hits 2 */
    { 1, { 1, 0, 0 }, 2, 2 }, /*  2: 11 hits 2 */
    { 1, { 0, 0, 0 }, 1, 3 }, /*  3: 3x hits 3 */
    { 0, { 1, 2, 0 }, 2, 3 }, /*  4: 21 hits 3 */
    { 1, { 1, 2, 0 }, 3, 3 }, /*  5: 11 hits 3 */
    { 1, { 0, 0, 0 }, 1, 4 }, /*  6: 4x hits 4 */
    { 0, { 1, 3, 0 }, 2, 4 }, /*  7: 31 hits 4 */
    { 1, { 2, 0, 0 }, 2, 4 }, /*  8: 22 hits 4 */
```

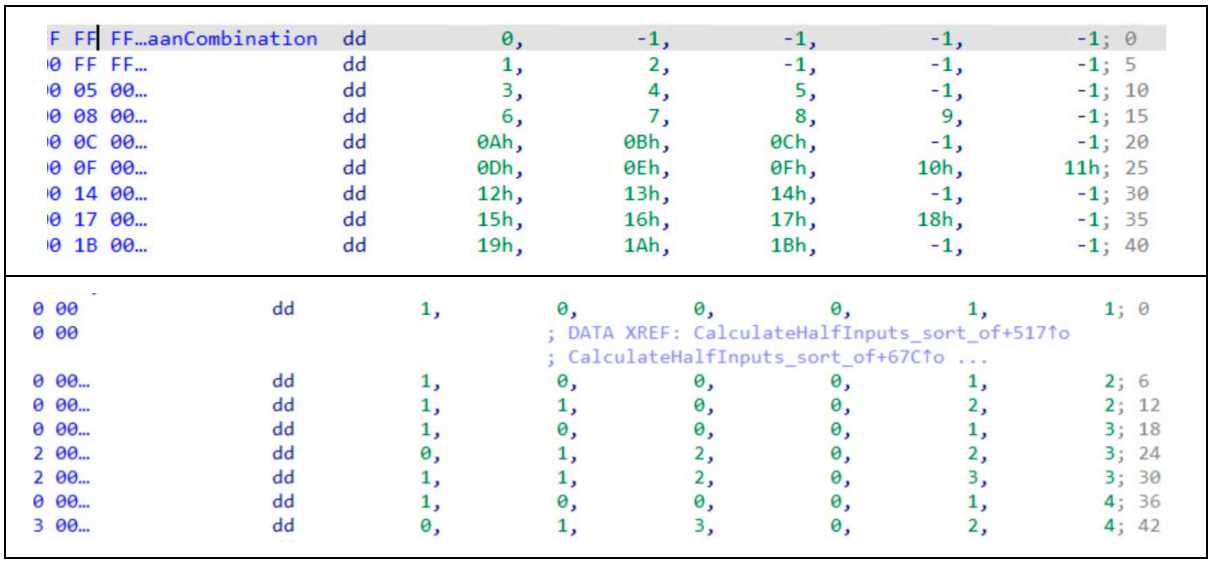

Figure 7: GNUBG 0.02 eval.c constants are present in the binary for SG-II; top left "aaCombination" is in the center screen shot and top right "aIntermediate" is in the bottom screen shot. Numbers with a trailing 'h' are hex but the decimal values match the original code table values

Additionally, data from "pub_eval.c" in version 0.02 is included, along with the floating-point representation in the IdaPro tool:

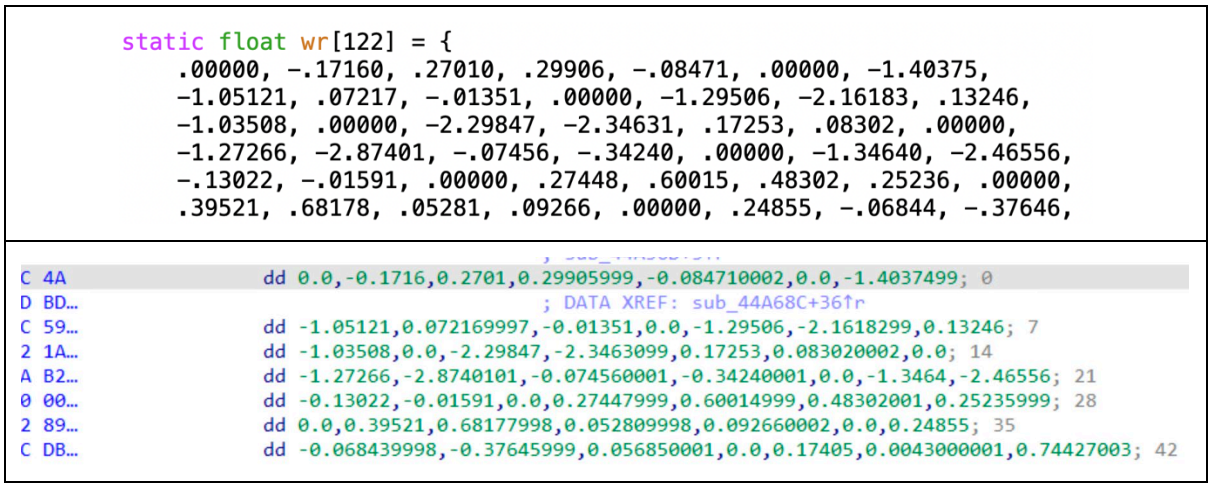

Figure 8: GNUBG 0.02 and 0.12 pub_eval.c floating-point constants are present in the binary for SG-II

Other integer tables are present as well but will not be shown here. These include a variable named "aaRoll" and various others.

Lastly, we can now draw one conclusion from this portion of the investigation. GNUBG version 0.02 does ***not*** have the array of floating-point values shown above ("e[100]" in figure 6), but version 0.12 ***does***. Conversely, version 0.12 has several arrays of integers named "third1_d1" for example, whose constant values are not to be found in SG-II. Presumably these were added to the code at some point. Thus, the conclusion is that the GNUBG code used in SG-II is ***between*** version 0.02 and the first (old) version available via the GNU repository, version 0.12. This also matches the timeframe for the timestamp on the file header, August 13, 2003, and the release date for 0.12, December 13, 2003. The GNU software used in SG-II appears to pre-date GNUBG 0.12 by possibly a year or so.

4.4 Analysis of Evaluation Functions

A feature of the reverse engineering tools such as IdaPro or Ghidra is that once an area of memory has been identified, a certain array of floating-point numbers used in move selection for example, the tool will then identify where in the instruction stream the array is referred to. This is a "cross reference" and a few of them

are already indicated in the above figures as "DATA XREF sub_...", indicating that there is a subroutine (function) that uses this table.

The authors repeat what was stated above, that compilation is a "lossy" function. Nonetheless the various reversing tool pseudocode generators will attempt to show the user a representation of roughly what the function source code might have looked like. Now that it is evident where the tables are in the program, one can see if the pseudocode using these tables resembles the source code for GNUBG. The following is the first example of this. The authors have already named the function to match the original source, and renamed a few of the variables to match as well, and the resemblance is somewhat striking:

```
int __cdecl ClassifyPosition(int *anBoard)
{
  int nBack; // [esp+0h] [ebp-Ch]
  int nOppBack; // [esp+4h] [ebp-8h]
  int i; // [esp+8h] [ebp-4h]

  nOppBack = -1;
  nBack = -1;
  for ( i = 0; i < 25; ++i )
  {
    if ( anBoard[i] )
      nOppBack = i;
    if ( anBoard[i + 25] )
      nBack = i;
  }
  if ( nBack < 0 || nOppBack < 0 )
    return 0;
  if ( nOppBack + nBack > 22 )
    return 4;
  if ( nBack > 5 || nOppBack > 5 || !pBearoff1 )
    return 3;
  if ( (unsigned __int16)PositionBearoff(anBoard) <= 923u
    && (unsigned __int16)PositionBearoff(anBoard + 25) <= 923u
    && dword_4C876C )
  {
    return 1;
  }
  return 2;
}
```

```
static positionclass ClassifyPosition( int anBoard[ 2 ][ 25 ] ) {

    int i, nOppBack = -1, nBack = -1;

    for( i = 0; i < 25; i++ ) {
        if( anBoard[ 0 ][ i ] )
            nOppBack = i;
        if( anBoard[ 1 ][ i ] )
            nBack = i;
    }

    if( nBack < 0 || nOppBack < 0 )
        return CLASS_OVER;

    if( nBack + nOppBack > 22 )
        return CLASS_CONTACT;
    else if( nBack > 5 || nOppBack > 5 )
        return CLASS_RACE;

    if( PositionBearoff( anBoard[ 0 ] ) > 923 ||
        PositionBearoff( anBoard[ 1 ] ) > 923 )
        return CLASS_BEAROFF1;

    return CLASS_BEAROFF2;
}
```

Figure 9: GNUBG 0.02 and 0.12 eval.c ClassifyPosition function, pseudocode left, original right

Note in the figure that textual replacement ("#define" in C) removes any way for the binary to map integer constants such as 4 back to their original name of "CLASS_CONTACT", but that the values do match. This example was selected for this paper because it is reasonably short unlike other functions; but similarly, the function "EvalBearoff1Full" is also nearly identical, as are many more. Also indicative is that moving to a machine code address "above" the function, and examining the function there, matches one to functions that are above this in "eval.c". And moving down in the machine code after this function takes one to functions that are after this in "eval.c". The conclusion is that, at least for this file, nothing in the original code was rearranged before it was integrated into SG-II.

This section concludes with two more examples for illustrative purposes.

```
int *__cdecl setx(int *pos)
{
  int *result; // eax
  int n; // [esp+4h] [ebp-Ch]
  int jm1; // [esp+8h] [ebp-8h]
  int i; // [esp+Ch] [ebp-4h]
  int j; // [esp+Ch] [ebp-4h]

  for ( i = 0; i < 122; ++i )
    x[i] = 0.0;
  for ( j = 1; j <= 24; ++j )
  {
    jm1 = j - 1;
    n = pos[25 - j];
    if ( n )
    {
      if ( n == -1 )
        x[5 * jm1] = 1.0;
      if ( n == 1 )
        x_plus_1[5 * jm1] = 1.0;
      if ( n >= 2 )
        x_plus_2[5 * jm1] = 1.0;
      if ( n == 3 )
        x_plus_3[5 * jm1] = 1.0;
      if ( n >= 4 )
        x_plus_4[5 * jm1] = (long double)(n - 3) * 0.5;
    }
  }
  x_sub_120 = -(long double)*pos * 0.5;
  result = pos;
  x_sub_121 = (long double)pos[26] * 0.06666666666666666667;
  return result;
}
```

```
static void setx( int pos[] ) {
    /* sets input vector x[] given board position pos[] */
    int j, jm1, n;

    /* initialize */
    for(j=0;j<122;++j) x[j] = 0.0;

    /* first encode board locations 24-1 */
    for(j=1;j<=24;++j) {
        jm1 = j - 1;
        n = pos[25-j];
        if(n!=0) {
            if(n==-1) x[5*jm1+0] = 1.0;
            if(n==1) x[5*jm1+1] = 1.0;
            if(n>=2) x[5*jm1+2] = 1.0;
            if(n==3) x[5*jm1+3] = 1.0;
            if(n>=4) x[5*jm1+4] = (float)(n-3)/2.0;
        }
    }
    /* encode opponent barmen */
    x[120] = -(float)(pos[0])/2.0;
    /* encode computer's menoff */
    x[121] = (float)(pos[26])/15.0;
}
```

Figure 10: GNUBG 0.02 and 0.12 pub_eval.c setx function, pseudocode left, original right

Note that as was outlined above, multiplication is faster than division, and the compiler has avoided dividing by 2.0 or 15.0 by using their reciprocals. Lastly:

```
  sub_445AE8();
  if ( a3 != *(_DWORD *)(a1 + 4) )
  {
    v4 = realloc(*(void **)(a1 + 32), 4 * a3);
    *(_DWORD *)(a1 + 32) = v4;
    if ( !v4 )
      return -1;
    for ( i = *(_DWORD *)(a1 + 4); i < a3; ++i )
    {
      if ( dword_4D6980-- )
      {
        v7 = dword_4D6984[dword_4D6980];
      }
      else
      {
        sub_4412F8(&dword_4D6980);
        dword_4D6980 = 15;
        v7 = dword_4D6984[15];
      }
      *(float *)(*(_DWORD *)(a1 + 32) + 4 * i) = (long double)((unsigned int)(unsigned __int16)v7 - 0x8000)
                                                 * 0.0000076293945;
    }
  }
  if ( a3 != *(_DWORD *)(a1 + 4) || a2 != *(_DWORD *)a1 )
  {
    v8 = malloc(4 * a2 * a3);
    v23 = v8;
    if ( !v8 )
```

```
CheckRC();

if( cHidden != pnn->cHidden ) {
    if( !( pnn->arHiddenThreshold = realloc( pnn->arHiddenThreshold,
            cHidden * sizeof( float ) ) ) )
        return -1;

    for( i = pnn->cHidden; i < cHidden; i++ )
        pnn->arHiddenThreshold[ i ] = ( ( irand( &rc ) & 0xFFFF ) -
                                        0x8000 ) / 131072.0;
}

if( cHidden != pnn->cHidden || cInput != pnn->cInput ) {
    if( !( pr = malloc( cHidden * cInput * sizeof( float ) ) ) )
        return -1;
```

Figure 11: GNUBG 0.12 Portions from neuralnet.c NeuralNetResize function, pseudocode top, original bottom

A few notes about the top pseudocode. Things like "(a1 + 4)" arrive because the original variable reference was "pnn -> cHidden", and in C this refers to a portion of a structure. The "cHidden" field of the structure is four bytes into the data, thus the strange pseudocode – there is no way to know the original names of the data or the fact that it was originally a structure. Also note again the avoidance of division, and the call to the standard "malloc" procedure, which the reversing tool has mapped correctly back to the standard library.

This last figure reinforces the conclusion from the previous section, that the code is newer than version 0.02. Not shown here, but the earlier version, 0.02, of the source does not have the call to "CheckCRC" (above figure, right), but it is visible in the pseudocode as "sub_445AE8". So again, SG-II uses GNUBG open-source from after version 0.02, as evidenced here, but before version 0.12, as pointed out above via arrays of integers whose constants are not to be found in the binary. And again, this matches the timestamp on the file versus the timestamps on the versions of GNUBG.

5. Conclusions

Utilizing reverse engineering tools is a useful method to make determinations regarding intellectual property theft because the tools plus knowledge helps uncover hidden evidence of copying or misappropriation. Illustrative of this are the software packages here which play Backgammon. But the methods are identical regardless of the software. If the resulting code structure, algorithms, or even variable names (if present) are substantially similar to proprietary code, that's a strong indicator of IP theft, especially if those similarities can't be explained by coincidence or the use of common libraries. Also, reverse engineering tools can reveal unique implementations, algorithms, or logic flows (or errors) that are highly specific to the original software creator. If these proprietary techniques are found in another's software, it strengthens the claim that copying occurred.

This research focuses not on high-dollar software from makers such as Oracle, but on low-dollar (or in the case of GNU, zero-dollar) software. But the methods used are the same, and here the results seem clear.

Strong Gammon II has utilized some of the source code from GNUBG, and the source code copied was a version after 0.02 but before 0.12.

References

AMD, AMD64 Architecture Programmer's Manual - Volume 3: General-Purpose and System Instructions, Available: https://www.amd.com/system/files/TechDocs/24594.pdf [Accessed April 2025].

Berliner, Hans J. "Computer Backgammon", Scientific American, Vol. 242 No. 6, June 1980, 64-72.

Binary Ninja, Available: https://binary.ninja [Accessed April 2025].

Defuse, Available: https://defuse.ca/online-x86-assembler.htm [Accessed April 2025].

Ghidra, Available: https://ghidra-sre.org [Accessed April 2025].

GNUBG, "Index of /gnu/backgammon", Available: https://alpha.gnu.org/gnu/backgammon/ [Accessed April 2025].

HexRays IdaPro, Available: http://www.hex-rays.com/products/ida/index.shtml [Accessed November 2024].

Hasherezade, "PE Bear", Available: https://github.com/hasherezade/pe-bear [Accessed April 2025].

Hopper, Available: https://www.hopperapp.com [Accessed April 2025].

IEEE, "IEEE Standard for Floating-Point Arithmetic", at Available: https://standards.ieee.org/ieee/754/6210/ [Accessed April 2025].

Intel, "Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference, A-M", 2006.

Linn, C. and Debray, C., "Obfuscation of Executable Code to Improve Resistance to Static Disassembly", Proceedings of the ACM Conference on Computer and Communications Security (CCS 2003), Washington, DC, USA, Oct. 27-30, 2003.

LinuxDevCenter, Available: http://www.linuxdevcenter.com/cmd/cmd.csp?path=o/objdump [Accessed April 2025].

Mahoney, W., McDonald, J. T., "Enumerating x86-64 – It's Not as Easy as Counting", Available: https://www.unomaha.edu/college-of-information-science-and-technology/research-labs/_files/enumerating-x86-64-instructions.pdf [Accessed April 2025].

Mahoney, W., Franco, J., Hoff, G. and McDonald, J. T., "Leave it to Weaver", 8th Software Security, Protection, and Reverse Engineering Workshop, Puerto Rico, December 3-4, 2018.

Mullins, J. A., McDonald, J. T., Mahoney, W. and Andel, T., "Evaluating Security of Executable Steganography for Digital Software Watermarking", 2020 Cybersecurity Symposium, Moscow, Idaho.

Microsoft, Available: https://learn.microsoft.com/en-us/windows/win32/sysinfo/operating-system-version [Accessed: April 2025].

OllyDbg, Available: http://www.ollydbg.de/ [Accessed November 2024].

Oracle, Available: "Oracle Wins Copyright Case Against Repeat Violator Rimini Street", press release July 25, 2023, https://www.oracle.com/news/announcement/oracle-wins-copyright-case-against-repeat-violator-rimini-street-2023-07-25/ [Accessed April 2025].

Pearce, W., "Reverse Engineering Code with IDA Pro", Elsevier Science, 2008.

Relyze, Available: https://www.relyze.com/overview.html [Accessed April 2025].

Schäfer, Hans-Jürgen, "Blot-Backgammon", Available: https://www.mustrum.de/blot.html [Accessed April, 2025].
ShardSecure, "What's the Real Cost of IP Theft?", August 29, 2023, Available: https://shardsecure.com/blog/real-cost-ip-theft [Accessed April 2025].
Shinn, S. and Mahoney, W., "Optimal Values for Disrupting x86-64 Reverse Assemblers", International Journal of Computer Science and Network Security, Volume 11, Number 11, 2011.
SourceForge, "GNU Backgammon", Available: https://sourceforge.net/projects/gnubg/files/gnubg/0.02/gnubg-0.02.tar.gz/download [Accessed April 2025].
Udis86, Available: http://udis86.sourceforge.net/ [Accessed April 2025].
Webster, George, Bojan Kolosnjaji, Christian von Pentz, Zachary Hanif, Julian Kirsch, Apostolis Zarras, and Claudia Eckert, "Finding the Needle: A Study of the PE32 Rich Header and Respective Malware Triage", 14th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), July 2017.
Wikipedia, Available: https://en.wikipedia.org/wiki/List_of_Microsoft_Windows_versions [Accessed April 2025].
x64dbg, Available: https://x64dbg.com [Accessed April 2025].