Reverse Engineering National Instruments' Multisim: Understanding the Offline Activation System

Jack Vega

March 8, 2025

Abstract

This paper presents an instructive exploration of reverse-engineering principles through a conceptual examination of Multisim's offline activation mechanism. The discussion emphasizes method, reasoning, and interpretation rather than circumvention. Each section demonstrates how technical observation aids understanding of class structures, license validation logic, and defensive design within legitimate research environments.

1 Introduction

Reverse engineering has long been an essential process for understanding the internal logic of compiled software and for testing the reliability of its security mechanisms. When performed ethically, it provides valuable insight into design weaknesses, license-management methodologies, and mechanisms used to authenticate licensed copies.

National Instruments' *Multisim* circuit-simulation suite utilizes an offline activation system representative of many contemporary desktop applications. Examining it conceptually reveals common strategies and shortcomings in software protection. Throughout this paper, each step of a reverse-engineering workflow, from identifying relevant binaries to analyzing class dependencies, is outlined to showcase analytic reasoning rather than to enable replication.

2 Overview of Reverse-Engineering Methodology

A structured approach begins by distinguishing two complementary modes of analysis:

- **Static Analysis:** inspection of code and metadata from binaries without executing them, revealing namespaces, classes, and strings.

- **Dynamic Analysis:** controlled observation of a program in operation, monitoring system calls, registry interactions, and file dependencies.

Combining both allows reconstruction of the interaction between user-input activation data and internal verification logic. Reverse engineering in this sense is a forensic exercise in comprehension. For this paper, Static analysis will solely be used.

3 Tools in This Paper

The tools listed here exemplify typical components of a reverse-engineering toolkit. Their descriptions are brief and conceptual, serving only to clarify their roles in analysis.

Detect-It-Easy (DIE). A file-type detector and metadata scanner used to identify whether a binary is native code or managed code (.NET, Java, etc.) and to expose compiler and packing information.

dnSpy. An open-source .NET decompiler that reconstructs readable source representation from compiled assemblies, enabling examination of classes, methods, and resources.

de4dot. A .NET de-obfuscation utility that restores logical structure and symbol names from obfuscated managed assemblies, aiding in comprehension of class relationships.

Process Dump. A memory-dumping tool capable of capturing the full image of a running process for later static analysis; in research, it is used to study program state.

Strings (Sysinternals). Extracts plain-text sequences from binaries to expose embedded messages such as error texts, valuable for locating routines referencing licensing messages.

4 Understanding Activation Mechanisms

Offline activation systems verify software entitlement without direct network communication. Typically, such schemes rely on three data sources: the machine's hardware fingerprint, the product's serial number, and the installed software version. These identifiers are combined through hashing or cryptographic encoding to form a license token or activation code.

A conceptual validation workflow includes:

1. The application collects identifying data from the host computer.
2. The data are processed into a cryptographic checksum or encoded string.
3. The checksum is compared with a stored or computed value that represents authorized access.

The mechanism's security depends on the irreversibility of that combination and the irreplicatability of the officially sanctioned codes outside of sanctioned environment. Weaknesses stem from insufficient obfuscation, encryption, exposed transformation logic, or predictable byte patterns.

Figure 1: Linear offline-activation workflow.

Through reverse analysis, investigators can understand where such transformations occur and evaluate whether the structure follows good cryptographic and software-engineering practice.

5 Preliminary Binary Inspection

Before diving into string extraction or deobfuscation, several reconnaissance steps established a clear overview of how Multisim loads and manages its components within memory. These steps relied on lightweight inspection tools used purely for static classification.

5.1 Detect-It-Easy (DIE) Analysis

The first inspection used **Detect-It-Easy (DIE)** to determine the form of the primary executable and its supporting modules. DIE reported that the binaries were compiled as managed .NET assemblies rather than native code, confirming that the subsequent investigation would take place within the Microsoft Intermediate Language (MSIL) environment. This finding justified the use of managed-code utilities such as **dnSpy** and **de4dot** for continued analysis.

5.2 Memory Dump and Hidden Modules

Although the visible installation directory contained only a small set of assemblies, runtime observation revealed that additional dynamic libraries were being loaded into memory during execution. To confirm their identities, the program's process was captured using a **Process Dump** tool, producing a snapshot of every assembly resident in the application's address

space. Examination of the resulting dump exposed numerous hidden DLLs that did not exist as separate files on disk.

Below is a representative portion of the modules discovered during the dump, showing the naming convention used by hidden assemblies embedded in the running process:

```
multisim_exe_PID2768_hiddenmodule_2B10000_x86.dll
multisim_exe_PID2768_hiddenmodule_9F40000_x86.dll
multisim_exe_PID2768_hiddenmodule_6450000_x86.dll
multisim_exe_PID2768_hiddenmodule_9D50000_x86.dll
multisim_exe_PID2768_hiddenmodule_B000000_x86.dll
...
```

Each file corresponds to a memory-mapped assembly dynamically loaded at runtime. After obtaining the full set of hidden modules, a large-scale string extraction was run across all dumped DLLs using **Strings** to identify those containing activation-related text. This comparison isolated the assemblies most closely associated with licensing behaviour and revealed two in particular:

- **hiddenmodule_6450000_x86.dll** — contained the functional engine responsible for license validation and offline activation logic.
- **hiddenmodule_9D50000_x86.dll** — implemented portions of the activation user interface and status prompts.

These libraries became the primary focus of subsequent de-obfuscation and code inspection. The findings from the process-dump and string-analysis phase defined the set of assemblies examined throughout the remainder of the analysis.

6 String Analysis and Identification of Relevant Assemblies

With the two primary modules identified from the process dump (`hiddenmodule_6450000_x86.dll` and `hiddenmodule_9D50000_x86.dll`), the investigation next focused on interpreting their contents. Both assemblies appeared in the search results produced by the **Strings** analysis, as each contained text fragments such as *"Invalid activation code"* and other activation-related messages. Because they were managed .NET assemblies, they could be loaded directly into **dnSpy** for high-level inspection of their class structures and symbolic content.

Once examined in dnSpy, the distinction between the two became clear: `hiddenmodule_9D50000_x86.dl` contained mostly interface logic for dialog prompts and message handling, while `hiddenmodule_6450000_x8` implemented the actual licensing and activation logic that controlled validation. The latter became the focus of further analysis, as it contained the functional components governing offline activation.

Inside the decompiled view, two assemblies initially appeared relevant because they contained the activation-related strings identified earlier:

- `NationalInstruments.LicenseManagement.Activation`
- `NationalInstruments.LicenseManagement.Engine`

Closer inspection of the function names showed that the `Activation` assembly managed only the entry and exit points of the activation workflow, handling user requests and final status reporting, while the `Engine` assembly seemed to contain the core licensing logic itself, including activation code parsing, encoding, and the validation routines tied to offline activation.

This refined focus established the `Engine` assembly as the principal target for de-obfuscation and detailed reverse-engineering in the sections that follow.

7 Obfuscation and Structural Restoration

Many commercial .NET applications employ varying degrees of code obfuscation to discourage static analysis. Multisim's assemblies display the same behavior, featuring scrambled symbol names, redundant branching, and unnecessary loop padding. Such transformations complicate the correlation between readable source code and compiled intermediate language instructions.

Using **de4dot**, the identified assemblies were significantly cleaned to restore logical readability. De-obfuscation at this stage clarifies how modules interact without modifying program behavior. The result is a restructured representation of the software, highlighting namespaces and classes directly associated with licensing and activation logic.

This restored structure provided a coherent foundation for the deeper analysis of individual classes and methods.

7.1 Obfuscation Example: `SendActivationRequest`

To demonstrate the effect of automated obfuscation within these assemblies, a single method was selected for a before-and-after comparison. The `SendActivationRequest` function was

chosen arbitrarily; it is not part of the licensing workflow but provides a clear visual representation of how obfuscators introduce unnecessary instructions to disguise simple operations.

Obfuscated Version

```
protected override string SendActivationRequest(ActivationRequest
    activationRequest)
{
    int num = 13733;
    int num2 = num;
    num = 13733;
    switch (num2 == num)
    {
    }
    num = 0;
    if (num != 0)
    {
    }
    num = 1;
    if (num != 0)
    {
    }
    num = 0;
    return NetworkInteraction.SendRequestToServer(activationRequest);
}
```

Listing 1: Obfuscated version of SendActivationRequest

In its obfuscated state, this method contains redundant variable assignments, empty switch statements, and empty conditional chains that serve no computational purpose. Such operations exist solely to distort control flow and make the decompiled method appear more complex than it truly is.

De-obfuscated Version

After the assembly was processed with **de4dot**, the same method was restored to a concise and fully legible form:

```
protected override string SendActivationRequest(ActivationRequest
    activationRequest)
{
    return NetworkInteraction.SendRequestToServer(activationRequest);
}
```

Listing 2: De-obfuscated version of SendActivationRequest

The cleaned version reveals that `SendActivationRequest` simply forwards an activation request to the network communication routine. The preceding comparison illustrates how obfuscators inflate code volume without adding cryptographic protection or meaningful logical complexity. While this noise can temporarily hinder superficial inspection, automated de-obfuscation quickly restores readability, reaffirming that effective software security relies on strong validation design rather than cosmetic code transformation.

8 Activation Code Class

After identifying the three classes central to the licensing engine, the next step was to examine their internal structure in detail. The `ActivationCode` class was chosen as the starting point because it seemed the most closely related to the goal of this reversing.

This class converts the encoded text entered by the user into structured data that later passes through the validation routines. Understanding its constructor reveals how the software decomposes an activation key into individual fields that other components, such as the `Encoder` and `ActivationCodeTransformer`, subsequently process.

```
1  public ActivationCode(string encodedString)
2  {
3      byte[] array = Encoder.Decode(encodedString);
4      if (array.Length != 17)
5      {
6          throw new FormatException(StringUtils.FormatInternal("Invalid
               activation code length [{0}]", new object[] { encodedString
               }));
7      }
8      if (array[0] == 28)
9      {
10         if (array[1] == 0)
11         {
12             this.Type = ActivationCode.ActivationCodeType.
                   EvaluationRespectSentry;
13         }
14         else
15         {
```

```
            if (array[1] != 1)
            {
                throw new FormatException(StringUtils.FormatInternal("
                    Invalid V2 activation code [{0}] of type [{1}]", new
                    object[]
                {
                    encodedString,
                    array[1]
                }));
            }
            this.Type = ActivationCode.ActivationCodeType.
                EvaluationIgnoreSentry;
        }
        this.HostIdType = ActivationCode.a((int)array[2]);
        this.Signature = ActivationCode.a(Encoder.ConvertFromBase(29,
            array, 3, 10));
        int num = (int)Encoder.ConvertFromBase(29, array, 13, 4);
        this.Expiration = ((num > 0) ? TimeUtils.
            GetExpirationDateFromUnixEpoch(num) : TimeUtils.
            PermanentValue);
        this.Count = 0;
    }
    else
    {
        if (array[0] / 14 > 0)
        {
            this.Type = ActivationCode.ActivationCodeType.Lease;
            int num2 = (int)Encoder.ConvertFromBase(29, array, 13, 4);
            this.Expiration = ((num2 > 0) ? TimeUtils.
                GetExpirationDateFromUnixEpoch(num2) : TimeUtils.
                PermanentValue);
            this.Count = 0;
        }
        else
        {
            this.Type = ActivationCode.ActivationCodeType.Standard;
            this.Expiration = TimeUtils.PermanentValue;
            this.Count = (int)Encoder.ConvertFromBase(10, array, 13, 4);
        }
        this.HostIdType = ActivationCode.a((int)(array[0] % 14));
        this.Signature = ActivationCode.a(Encoder.ConvertFromBase(16,
            array, 1, 12));
    }
```

```
50      if (string.IsNullOrEmpty(this.Signature))
51      {
52          throw new FormatException("Signature cannot be empty");
53      }
54  }
```

Listing 3: ActivationCode class constructor reconstructed from a managed assembly

Analysis of this constructor reveals a predictable and highly structured approach to handling activation data. When the `ActivationCode` class decodes a provided string, it first verifies a fixed seventeen-byte length, indicating that the format is based on a predefined schema rather than dynamically generated at runtime. Each byte within the array has an assigned function:

- **Byte 0 - HostIdType:** Represents the host identifier type. In observed cases, this value is typically `0x14`, which indicates a permanent activation mode.
- **Bytes 1-12 - Signature:** Form the signature segment that links the activation code to a specific licensed machine or configuration. These bytes appear to hold the key portion later used during signature validation.
- **Bytes 13-17 - Expiration or Metadata:** Reserved for auxiliary data, most notably expiration timestamps or usage-count information.

The constructor logic implements these distinctions through layered `if`-statements that map particular byte values to activation types and expiration rules. This segmented structure demonstrates a standard data-serialization strategy common in managed-code licensing frameworks—each field carries isolated meaning and can be verified independently. The consistent seventeen-byte layout provides reliable parsing across product versions and exposes a transparent internal schema that, once the assembly is readable, can be interpreted and reasoned about with relative ease.

9 Encoder Class

The `Encoder` class is responsible for all encoding operations within the licensing framework—including the generation of activation keys and the construction of host identifiers. Rather than reconstructing the entire algorithm line by line, the de-obfuscated class can be reused directly by providing custom inputs. This approach removes unnecessary complexity and concentrates on how the system formats and transforms data rather than on the internal mathematics of each operation.

To analyze its behaviour in isolation, the `Encoder` class was copied from the cleaned assembly and placed into a separate `.NET Framework 4.8` console application. This reconstruction allowed the encoding process to be tested safely with controlled input values. The simplified file structure of the project is shown below:

```
MultisimEncoder
|- Properties
|- References
|- App.config
|- Encoder.cs
|- Program.cs
```

With this setup, arbitrary byte sequences can be supplied to the encoder, and the resulting formatted strings examined. A representative example uses randomised raw values to demonstrate how the class transforms binary data into its activation-style string output.

```
static void Main(string[] args)
{
    // Generate a sample raw code using known valid byte positions
    string raw_code = "\x14\x01\x10\x13\x04\x10\x13\x02\x12\x11\x16\x17\
       x13\x00\x00\x00\x00";

    // Encode the byte data into an activation-style key
    string code = Encoder.Encode(Encoding.UTF8.GetBytes(raw_code));
    Console.WriteLine(code);
    Console.ReadLine();
}
```

Listing 4: Example test program using the reconstructed Encoder class

When the program executes, it outputs a newly generated, formatted string that resembles an activation key. This experiment confirms that the `Encoder` class performs deterministic, format-enforcing translation rather than cryptographic transformation: bytes are mapped to characters within a base-29 range and grouped with hyphen separators for readability.

Before these controlled tests were run, an internal constraint in the constructor was verified, byte values passed to the encoder must not exceed the base limit set by the routine, ensuring all inputs remain valid for translation.

```
public static string Encode(byte[] data)
{
```

```
    int num = (int)(data.Aggregate(23U, (uint seed, byte item) => seed *
        37U + (uint)item + ((item > 9) ? 55U : 48U)) % 29U);
    StringBuilder stringBuilder = new StringBuilder();
    stringBuilder.Append(Encoder.smethod_3(0, num));
    stringBuilder.Append(Encoder.smethod_3(0, 0));
    stringBuilder.Append(Encoder.smethod_3(num, 0));
    foreach (byte b in data)
    {
        if (stringBuilder.Length % 5 == 4)
            stringBuilder.Append('-');
        if (b >= 29)
            throw new FormatException(
                "All values must be less than the encoding base");
        stringBuilder.Append(Encoder.smethod_3(num, (int)b));
        num = (num + 1) % 29;
    }
    stringBuilder[1] =
        Encoder.smethod_3(0, Encoder.smethod_1(stringBuilder.ToString())
           );
    return stringBuilder.ToString();
}
```

Listing 5: Simplified logic showing enforcement of encoding constraints

The deterministic nature of this routine demonstrates that the encoding used by Multisim is table-based and reversible. While such a method ensures platform consistency for managed applications, it lacks the entropy and asymmetry characteristic of modern cryptographic systems. Predictability simplifies legitimate generation for authorised tools but also underscores the advantage of introducing non-reversible signatures to strengthen overall license integrity.

10 ActivationCodeTransformer Class

After analyzing the `ActivationCode` and `Encoder` classes, the next step was to trace how the program links decoded activation data with the license records used internally by the application. This responsibility lies with the `ActivationCodeTransformer` class, the component that connects user-supplied activation codes to validated license objects. Tracing method calls within **dnSpy** revealed three key routines, `smethod_1`, `smethod_6`, and `smethod_7`, that work sequentially to transform, compare, and confirm license information.

10.1 smethod_1 Function

Within the `ActivationCode` constructor a call was found to `smethod_1`, marking the first interaction between a decoded activation string and the internal license catalogue. This routine converts the input string into an `ActivationCode` object and iterates through existing licenses, pairing each with the activation data by invoking `smethod_6`. Based on the results returned from that call, `smethod_1` assigns an appropriate status, such as *invalid code*, *no matching license*, *date expired*, or *success*, and builds a corresponding `CodeApplicationData` object.

```
1  private static ActivationCodeTransformer.CodeApplicationData smethod_1(
       string string_1)
2  {
3      if (string_1 == "NOCHANGE")
4      {
5          return new ActivationCodeTransformer.CodeApplicationData(
               string_1,  ApplyActivationCodeStatus.NoChange, null, null,
               null);
6      }
7      ActivationCode code = new ActivationCode(string_1);
8      var <>f__AnonymousType = LicenseData.MainLicenses.Select((License
           mainLicense) => new
9      {
10         OriginalLicense = mainLicense,
11         TransformedLicense = ActivationCodeTransformer.smethod_6(code,
               mainLicense)
12     }).FirstOrDefault(item => item.TransformedLicense != null);
13
14     if (<>f__AnonymousType == null)
15         return new ActivationCodeTransformer.CodeApplicationData(
               string_1,  ApplyActivationCodeStatus.NoMatchingLicense, null,
                null, null);
16
17     if (<>f__AnonymousType.TransformedLicense.ExpirationDate.IsInPast())
18         return new ActivationCodeTransformer.CodeApplicationData(
               string_1,  ApplyActivationCodeStatus.DateExpired, <>
               f__AnonymousType.OriginalLicense, null, null);
19
20     if (ActivationCodeTransformer.smethod_5(<>f__AnonymousType.
           OriginalLicense <>f__AnonymousType.TransformedLicense))
21         return new ActivationCodeTransformer.CodeApplicationData(
               string_1,  ApplyActivationCodeStatus.
               DateWorseThanCurrentLicense, <>f__AnonymousType.
```

```
            OriginalLicense, null, null);

    ProtectedDatastore.LicenseSentry licenseSentry =
        (code.IsEvaluation ?
         new ProtectedDatastore.LicenseSentry(<>f__AnonymousType.
            OriginalLicense, DateTime.UtcNow,  <>f__AnonymousType.
            TransformedLicense.ExpirationDate.UtcDateTime) : null);

    return new ActivationCodeTransformer.CodeApplicationData(string_1,
             ApplyActivationCodeStatus.Success,
             <>f__AnonymousType.OriginalLicense,
             <>f__AnonymousType.TransformedLicense,
             licenseSentry);
}
```

Listing 6: Reference chain connecting activation data with license objects

The structure of this method shows a controlled hand-off between activation data and license records. Each possible outcome is represented as a defined status, providing a consistent mechanism for the application to record validation results before deeper analysis proceeds in `smethod_6`.

10.2 smethod_6 Function

The second stage of the process, `ActivationCodeTransformer.smethod_6`, handles the transformation of activation values into modified license objects. This function confirms that a license is valid, not operating under beta conditions, and compatible with the incoming activation type. When those checks pass, it requests host information from `smethod_7` to link the activation with a specific hardware context. It then constructs a new `LicenseOverrideInfo` object, appending relevant state keywords such as *BetaActive*, *LeaseActive*, *ExtendedDemoActive*, or *FullDemo* based on current parameters, and produces a temporary license override.

```
private static License smethod_6(ActivationCode activationCode_0,
   License license_0)
{
    if (!license_0.IsValid || license_0.VendorString.IsBetaActive)
        return null;

    if (activationCode_0.IsEvaluation &&
       !license_0.VendorString.IsExtendedDemoSupported &&
       !license_0.VendorString.IsExtendedDemoActive)
```

```
        return null;

    HostId hostId = ActivationCodeTransformer.smethod_7(
                        activationCode_0, license_0);
    if (hostId == null)
        return null;

    DateTimeOffset expiration = activationCode_0.Expiration;
    if (activationCode_0.Type ==
        ActivationCode.ActivationCodeType.EvaluationRespectSentry &&
        LicensingEngine.ProtectedDatastore
            .ExtendedDemoSentries[license_0].EndDateUtc < expiration.
               UtcDateTime)
    {
        expiration = new DateTimeOffset(
            LicensingEngine.ProtectedDatastore.ExtendedDemoSentries[
               license_0]
                .EndDateUtc.ToLocalTime());
    }

    List<LicenseOverrideInfo.Keyword> list =
        new List<LicenseOverrideInfo.Keyword>();
    if (license_0.VendorString.BetaExpiration != null)
        list.Add(LicenseOverrideInfo.Keyword.BetaActive);
    else if (activationCode_0.Type ==
             ActivationCode.ActivationCodeType.Lease)
        list.Add(LicenseOverrideInfo.Keyword.LeaseActive);
    else if (activationCode_0.IsEvaluation)
    {
        list.Add(LicenseOverrideInfo.Keyword.ExtendedDemoActive);
        if (license_0.VendorString.FullDemoEnd != null)
        {
            DateTimeOffset dateTimeOffset =
                license_0.VendorString.FullDemoEnd.ExpirationDate ??
                TimeUtils.GetExpirationDateFromToday(
                    license_0.VendorString.FullDemoEnd
                        .PotentialDays.GetValueOrDefault());
            list.Add(LicenseOverrideInfo.Keyword.FullDemo(dateTimeOffset
               ));
        }
    }

    Logger.AssertTrue(
```

```
        license_0.LicenseFile.FileType == LicenseFileType.Main,
        "Should only apply activation codes to main license [{0} ({1})]"
            ,
        new object[] { license_0, license_0.LicenseFile.FileType });

    return new LicenseOverrideInfo(
                hostId,
                license_0.VendorString.BetaExpiration ?? expiration,
                list).CreateLicenseFrom(license_0);
}
```

Listing 7: Transformation of activation and license objects within smethod_6

This portion of the code acts as both a filter and a transformer. It validates environment conditions, applies contextual tags, and assembles a license suitable for verification. Once this construction is complete, the method transitions control to `smethod_7` for the final signature comparison.

10.3 smethod_7 Function

The final stage, `smethod_7`, is responsible for the direct comparison between the activation code's stored signature and a new signature derived from the license data. The function loops through every available host identifier of the type specified in the activation code, builds a temporary `LicenseOverrideInfo`, and calculates the expected signature using `Core.GetIncrementSignature`. If the computed value matches the activation code's signature, the host is accepted and returned as valid.

```
private static HostId smethod_7(ActivationCode activationCode_0, License
    license_0)
{
    using (IEnumerator<HostId> enumerator =
            HostId.GetHostIdsOfKind(
                activationCode_0.HostIdType).GetEnumerator())
    {
        while (enumerator.MoveNext())
        {
            HostId hostId = enumerator.Current;
            LicenseOverrideInfo overrideInfo =
                new LicenseOverrideInfo(
                    hostId,
                    license_0.VendorString.BetaExpiration ??
                        activationCode_0.Expiration,
```

```
                    null);

            object obj = ActivationCodeTransformer.object_0;
            string orCreateValue;
            lock (obj)
            {
                orCreateValue =
                    ActivationCodeTransformer.dictionary_0.
                        GetOrCreateValue(
                         new ActivationCodeTransformer.LicenseAndOverride
                            (
                             license_0, overrideInfo),
                         () => Core.GetIncrementSignature(
                                  overrideInfo.CreateLicenseFrom(
                                     license_0)));
            }

            // Primary signature comparison
            if (StringComparer.OrdinalIgnoreCase.Equals(
                    activationCode_0.Signature, orCreateValue))
            {
                return hostId;
            }
        }
    }
    return null;
}
```

Listing 8: Signature comparison logic from ActivationCodeTransformer

This equality check completes the license validation sequence. Matching signatures confirm that the activation string corresponds to a legitimate license record tied to the machine, completing the offline verification process.

10.4 Tracing the Validation Flow

Together, these three methods define a layered verification structure:

- `smethod_1` links the activation string and license entries and assigns validation outcomes.
- `smethod_6` filters ineligible licenses and constructs temporary overrides for active ones.

- `smethod_7` performs the signature comparison that determines activation legitimacy on the host system.

The progression from input parsing to host-specific signature matching provides a clear view of where credibility is established within Multisim's licensing engine. Understanding this relationship forms the basis for the deeper investigation of how these components interact with the greater license-management framework.

11 Application of Findings

After tracing the complete validation pathway within the `ActivationCodeTransformer` class, the final stage of analysis examined how small structural changes could theoretically influence the program's license validation decisions. Two areas of interest emerged, both located within the previously reviewed functions `smethod_7` and `smethod_6`. These observations illustrate how single conditional statements govern the entire outcome of activation verification and license scope.

11.1 Validation Logic in `smethod_7`

Within `smethod_7`, the verification loop contains the conditional that determines whether the generated signature from license data matches the signature retrieved from the activation code. Conceptually, this condition forms the software's principal defence gate: the program accepts the license only when both signatures correspond. If that conditional were altered to always return a positive result, for instance by unconditionally returning the active `HostId`, the routine would effectively bypass verification and instruct the system to treat all inputs as valid. This underscores how critical that single equality check is to maintaining the integrity of activation validation.

```
1 // Token: 0x06000687 RID: 1671 RVA: 0x00019B70 File Offset: 0x00017D70
2 private static HostId smethod_7(ActivationCode activationCode_0, License
      , license_0)
3 {
4     using (IEnumerator<HostId> enumerator =
5     HostId.GetHostIdsOfKind(activationCode_0.HostIdType).GetEnumerator()
          )
6     {
7         while (enumerator.MoveNext()) {
8             HostId hostId = enumerator.Current;
```

```
            LicenseOverrideInfo overrideInfo = new LicenseOverrideInfo(
                hostId, license_0.VendorString.BetaExpiration ??
                activationCode_0.Expiration, null);
            object obj = ActivationCodeTransformer.object_0;
            string orCreateValue;
            lock (obj)
            {
                orCreateValue =
                ActivationCodeTransformer.dictionary_0.GetOrCreateValue(
                    new
                ActivationCodeTransformer.LicenseAndOverride(license_0,
                    overrideInfo),
                () =>
                Core.GetIncrementSignature(overrideInfo.
                    CreateLicenseFrom(license_0)));
            }
            return hostId;
        }
        ...
    }
    ...
}
```

Listing 9: Section of smethod_7 highlighting the central equality condition

Analysis of this loop demonstrates that the correctness of license validation depends entirely on the comparison above. The investigation thus identifies precisely where the program's trust boundary occurs.

11.2 Conditional Scope in `smethod_6`

A second inspection point was found in `smethod_6`, where the license type is filtered before the verification routine executes. The method checks each incoming license name and only proceeds with those matching the expected product identifier. For instance, a typical segment appears as:

```
private static License smethod_6(ActivationCode activationCode_0,
    License license_0) {
    ...
    // Added this if statement
    if (license_0.Name != "Multisim_ProPower_PKG") {
        return null;
    }
```

```
7       HostId hostId =
8       ActivationCodeTransformer.smethod_7(activationCode_0, license_0);
9       ...
10  }
```

Listing 10: Excerpt showing license-type filtering in smethod_6

This discriminates among license variants and ensures that activations apply only to the correct package type. By analysing this portion of code, it becomes clear how product tiers are differentiated within the licensing logic, and why even minor conditional changes could broaden or restrict accessible features.

11.3 Implications

Together, these observations illustrate how tightly activation enforcement is bound to a few specific logical checkpoints. The equality comparison in `smethod_7` and the license filtering in `smethod_6` serve as critical control nodes for determining both the legitimacy and the level of a user's activation. From a defensive programming perspective, they highlight the importance of isolating trust boundaries and securing comparison logic through strong cryptographic constructs and signature verification. For researchers, recognising these narrow decision points clarifies where licensing frameworks assert integrity and how future implementations can better protect such core validation routines.

12 Security Implications and Broader Analysis

The preceding sections revealed how a managed activation workflow operates at the level of class structure and logical flow. Observing how specific byte fields, conditional statements, and verification chains determine license authenticity provides direct insight into the strengths and weaknesses of software-based protection. From a defensive–engineering perspective, several lessons stand out.

12.1 Integrity of Validation Boundaries

A program's ability to enforce legitimate activation rests on the reliability of a few conditional gates, most often equality checks or product–tier filters. In the examined implementation, the outcome of validation is decided primarily at two points:

- The signature comparison in `smethod_7`, which confirms whether an activation code originates from a genuine license source.

- The product–type constraint in `smethod_6`, which determines the permissible scope of activation within the product line.

Hardening these gates through non-reversible cryptographic checks and secure storage of intermediate results prevents deterministic reuse and guards against tampering. When internal decisions are made on plain logical comparisons, the entire activation process depends on the secrecy of the surrounding code rather than on robust mathematics which this program failed to complete properly.

12.2 Design Lessons for Managed Licensing Systems

Because the analysed assemblies operate entirely within the .NET ecosystem, they benefit from managed security features such as runtime verification and structured exception handling, but they also inherit the transparency of intermediate language code. To improve resilience against analysis:

- Activation logic should employ asymmetric cryptography so public keys verify signatures without exposing the generation process.
- Core decision routines can be isolated into native components or remote services when feasible.
- Obfuscation and integrity checks should occur at multiple points to detect modified assemblies before execution.

Adopting these measures transforms validation from a deterministic comparison task into a cryptographically anchored verification process.

12.3 Ethical Perspective

Reverse engineering occupies an essential position in both security research and educational environments. Conducted responsibly, it demonstrates where trust assumptions reside in complex systems and how easily they can be altered when unsupported by strong cryptography. Such inquiry advances defensive knowledge by revealing the delicate points where licensing logic may fail and where future improvements should focus.

13 Conclusion

Through static inspection of managed code and structural analysis of the license workflow, this analysis reconstructed the key pathways used by Multisim's offline activation system.

Each segment, the `ActivationCode`, `Encoder`, and `ActivationCodeTransformer` classes, demonstrates how simple, modular functions interact to determine product authenticity. While effective within its environment, the design relies primarily on easily traceable comparison logic instead of irreversible cryptographic verification.

Reverse-engineering this mechanism provided clarity on how the licensed software verifies its integrity, and, more importantly, why relying solely on logical conditions introduces risk. By viewing activation verification as a chain of narrowly defined trust boundaries, developers and security researchers can identify where their own systems may be most vulnerable. Ultimately, understanding these mechanisms strengthens both ethical research and practical software-protection strategies.

Acknowledgements

All exploration described here was performed for educational purposes to promote awareness of secure development practices.