2010

IMPROVED SOFTWARE ACTIVATION USING MULTITHREADING

Jian Rui Zhang
San Jose State University

IMPROVED SOFTWARE ACTIVATION USING MULTITHREADING

A Thesis

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science

by

Jianrui Zhang

April, 2010

SAN JOSÉ STATE UNIVERSITY

The Undersigned Project Committee Approves the Project Titled

IMPROVED SOFTWARE ACTIVATION USING MULTITHREADING

by

Jianrui Zhang

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

_____

Dr. Mark Stamp, Department of Computer Science Date

_____

Dr. Robert Chun, Department of Computer Science Date

_____

Mr. Wai Leung Wong, Oracle Corporation Date

ABSTRACT

Software activation is an anti-piracy technology designed to verify that software products have been legitimately licensed [1]. It is supposed to be quick and simple while simultaneously protecting customer privacy. The most common form of software activation is through the entering of legitimate product serial numbers by users, which sometimes are also known as product keys. This technique is employed by various software, from small shareware programs to large commercial programs such as Microsoft Office. However, software activation based on a serial number appears to be weak, as various cracks for a majority of programs are available and can be found easily on the Internet. Users can use such cracks to bypass the software activation.

Generally, the verification logic for checking a serial number executes sequentially in a single thread. Such an approach is weak because attackers can effectively trace this thread from the beginning to the end to understand how the verification logic works. In this paper, we develop a practical multi-threaded verification design. We breakdown the checking logic into smaller pieces and run each piece within a separate thread. Our results show that the amount of traceable code in a debugger is reduced to a low percentage of the code -- especially when junk threads with deadlocks are used -- and the traceable code in each run can differ as well. This makes it more difficult for an attacker to reverse engineer the code and bypass any security check. Finally, we attempt to quantify the increased effort necessary to break out verification logic.

ACKNOWLEDGEMENT

I would like to sincerely thank my project advisor, Dr. Mark Stamp, for his expertise and thoughtful insights which guided my project. In particular, I would also like to thank him for introducing me to the topic of software security (in his CS265 lecture) and software reverse engineering (in his CS286 lecture in Fall 2009). The first class motivated me to do this project while the second class provided me the necessary knowledge to finish this project.

I would also like to thank my committee member, Dr. Robert Chun, for his expertise in parallel processing (in his CS159 lecture in Spring 2010), which provided the core of my project design. His hardware architecture class (CS247 in Fall 2008) motivated me to use redundancy as a critical part of my design.

I would also like to thank my committee member, Mr. Wai Leung William Wong, for his expertise in the field of software designs, which guided me to write code for my testing program.

In addition, I am thankful to my fellow colleagues in Dr. Stamp's CS286 in Fall 2009, who generously let me study their various anti-reversing techniques. These techniques – combined with my own research – provided the building blocks of my new design.

Table of Contents

List of Tables and Figures

Figures

Tables

1. Introduction

There are hundreds of millions of software products in the world for all kinds of needs. Among these, many are free or exist as open source, while others require users to pay for the use of the software. Many commercial software products provide trial versions – free of charge – so that users can try out their features before buying; some form of activation is required to obtain full use of the software. In this case inactivated (or trial) versions usually have reduced functionality and/or usage limits over time. Usually trial versions are identical to the full versions in terms of the binary code.

Most software products employ a serial number for protection. This method is the most popular. However, its effectiveness is in serious question since the trial version has the same binary code as the full version. Thus, it is possible to bypass or crack the software and remove the limitations of the trial version to obtain full use. In fact, many software products that use serial numbers for protection are cracked by bypassing or patching the activation mechanism. Since the activation mechanism is relatively weak. After breaking the activation mechanism, hackers can create key generators (commonly known as KeyGens) or patches and distribute them through the Internet so that other users can use the trial version software just like the full version, but without paying. One can find KeyGens or patches for many popular software products [20][21]. It is worth mentioning that many KeyGens or patches contain viruses; therefore, using such cracks can subject users to security problems. It is thought that a majority of computers used in China are running pirated versions of Microsoft Windows, which clearly shows the weakness in that software's activation mechanism. In fact, it is reported that Windows 7 was already cracked after the Lenovo OEM key leaked [2] a few months before its official release. Figure 1 shows the estimated level of software piracy in various countries.

Our research focuses on how serial number checking is performed. The goal here is to design an improved mechanism by using various anti-reversing techniques so that it is more difficult to break the checking mechanism.

The organization of this report is as follows. Chapter 2 discusses various categories of software activation while Chapter 3 focuses on various aspects of using serial numbers as activation mechanisms. In Chapter 4 we discuss the influence of programming languages and integrated development environments (IDEs) on software activation mechanisms. In Chapter 5 we discuss various ways to break software mechanism. Chapter 6 covers various anti-reversing techniques that can be used to strengthen activation mechanism and Chapter 7 covers our new design as well as our testing setup and results. Finally, Chapter 8 provides a conclusion and suggestions for future work.

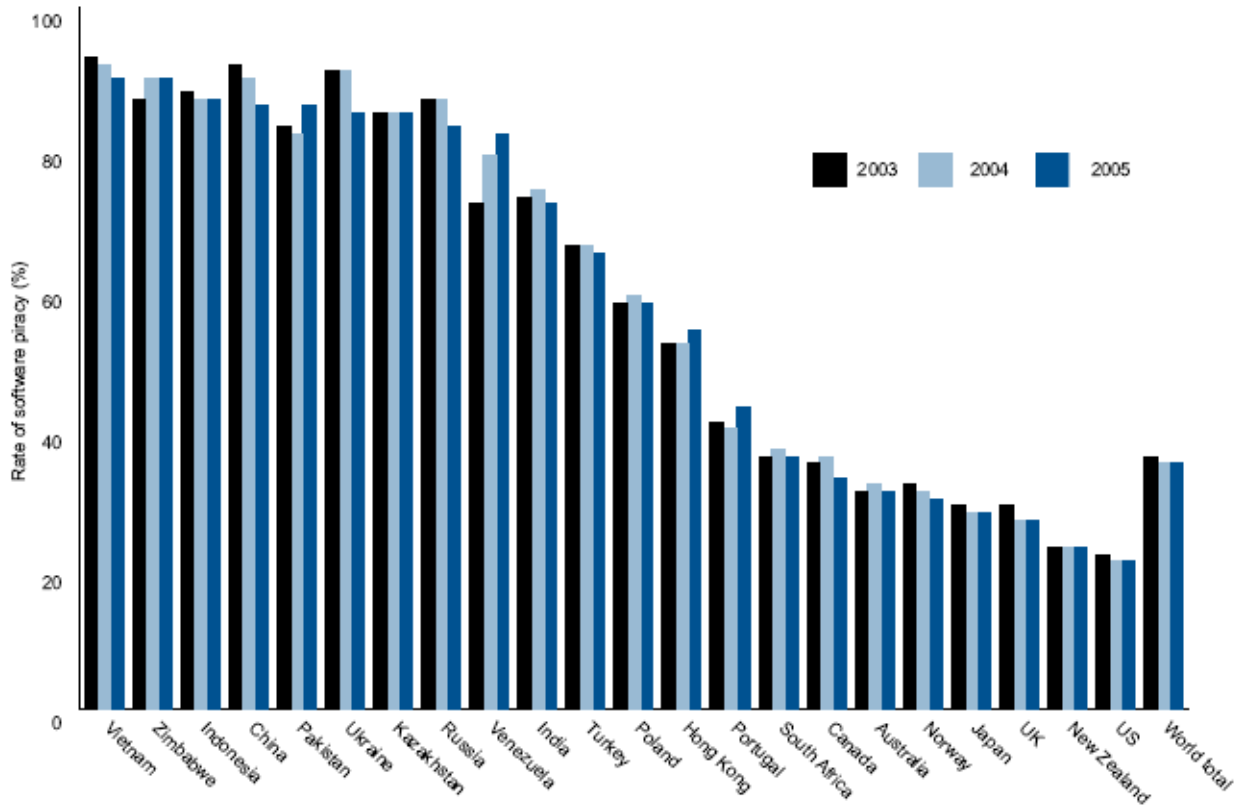

Figure 1. Level of Software Piracy in Different Countries
Source: Australian Institute of Criminology [23]

2. Software Activation

Software activation is used primarily as a way to make users pay for the software they use; this is how software companies make money for the continuance of their business. In the past (approximately a decade ago in the 1990s), software products were either free or the consumers had to purchase them. In the past one could not download or share software products via Internet, like we do today; as a result software privacy wasn't a big concern. In fact, Microsoft encouraged piracy as a way to market its Windows 95 operating system. This commercial practice has changed since Internet became popular. Today, consumers can try various software packages before they decide which to buy. In this process, software vendors must make the software attractive enough to the consumers while setting some limitations so that consumers will eventually pay for the software products. Through this change of practice came the concept of software activation.

The rest of the chapter will discuss various kinds of protection schemes and several activation mechanisms.

2.1 Categories of Protections:

2.1.1 Password

Password based protection usually requires users to enter their usernames and passwords for authentication in order to gain access. This is the easiest to crack [3]. This can be cracked by viewing "echoes" of the username and password by taking a memory snapshot. For more details, see section 5.2. While the logic for username and password checking is bulletproof, improper implementations for taking user input by programmers can make the checking mechanism weak. For example, using immutable strings to store data (username and password) instead of character arrays whose data one should clear after use. This may lead to revelation of such important information.

2.1.2 Base on How Many Times or Days One Can Use

One common practice employed by shareware today is to limit the number of days users can use the trial version of the software product. In this case, a trial version can be fully functional as opposed to reduced functionalities. The software would expire after the limit and become completely unusable. If a user likes that particular software and uses it for an important purpose, the user will have to purchase the software in order to continue to use it. According to [3], this category of protection is also fairly easy to break. We came across one software named "CD Key Generator" from Jedisware [24], which utilizes this kind of protection in its trial version and limits usage to only 5 days; we were able to break it in a matter of a few hours [4].

2.1.3 Base on SpecificEexpirationDdate

Currently marketed software products do not usually expire after a specific date; expiration is more or less an old practice. This practice is usually seen on beta products today, such as Microsoft's Windows beta versions. This kind of protection does not really protect the software products per se, but rather forces users to buy the full version when it is released. One simple way to break this kind of protection is to reset the clock to a time before the expiration date.

2.1.4 Having Functions Disabled

Another common practice in use today by many shareware is to provide users with a trial version whose functionalities are reduced. For example, Cyberlink's PowerDVD [25] lets a user play back DVD movies up to five minutes in the trial version. In many cases, the executable for the trial version is the same as the full version, which makes it possible to break the protection and turn the trial version into a fully functional version. In fact, hackers often break software based on such protection and distribute the cracked versions on the Internet.

2.1.5 Base on "Disk" or "CD-ROM" Access

There are software products that use presence of a disk (containing some critical information) in the CD-ROM to start the program. This method is mostly used by the computer game industry and is very easy to crack according to OCR [3]. One can easily find cracks online for nearly all popular game titles using this protection.

2.1.6 CryptographicAadd-ons

There are essentially two methods in this category: encryption and hashing. This category of protection is intended to make software temper-resistant from hackers. Encryption of code can make it extremely difficult for attackers to understand the underlying protection mechanism, whereas hashing is used to make code modification difficult. This kind of protection is usually used to protect important logics in software.

2.1.7 Others

There are also software products whose protections can fit into more than one category mentioned above in section 2.1. In this case, different protection methods are used in combination to reinforce each other to make cracking more difficult. Another method is to have different binaries for the trial and full versions. If the binary of the trial version does not contain all functionality of the full version and the full version cannot be obtained publicly, there is no point attacking the trial version.

2.2 Ways of Activating Software:

2.2.1 By Entering a Serial Number

The use of a serial number is the most popular choice for activating software. Most shareware and big name commercial products are activated this way, including Microsoft Windows. This method requires a user to type in a serial number obtained from the vendor after purchasing a legal copy of the software. In some cases, a username is also needed.

There are two common ways to distribute serial numbers. The first option is to distribute the serial number along with the media containing installation package. This is usually preferred by companies whose products are usually not downloadable (or not suitable for download due to large size); however, most software products can be distributed through download nowadays thanks to faster Internet. The second option is through email. This method is suitable for downloadable software like shareware whose sizes are typically small. After purchasing the product (usually online), vendors send an email confirmation to the user along with a serial number for the product.

In addition to requiring users to enter serial numbers, certain software vendors may require more information. For example, Adobe Photoshop requires users to go online (or by phone) to obtain a second activation code and use it to complete the activation process, see Figure 2. Microsoft Windows, on the other hand, repeatedly checks for activation information on user machines from its proprietary software, GenuineAdvantage, against its database (via the Internet) whenever critical updates for an operating system are downloaded and installed. As of this writing, there are only a few software vendors that require something more than a serial number.

2.2.2 By an Activation File

An activation file is sometimes used to activate software, although not common. This method usually works in conjunction with software distribution via download. A consumer purchases the software online at the vender's website, and the vendor sends an email to the user with an activation file attached. After receiving the activation file, the user must manually follow certain instructions (usually found from the same email) and save the file to some specified location. When the software launches, it checks for the existence of an activation file; and if file is found, the software turns into a full version mode if the contents of the activation file check out. Activation files usually contain contents such as username and other activation information, which is unique in each case. RarLab's popular WinRAR [27] uses this method for activation.

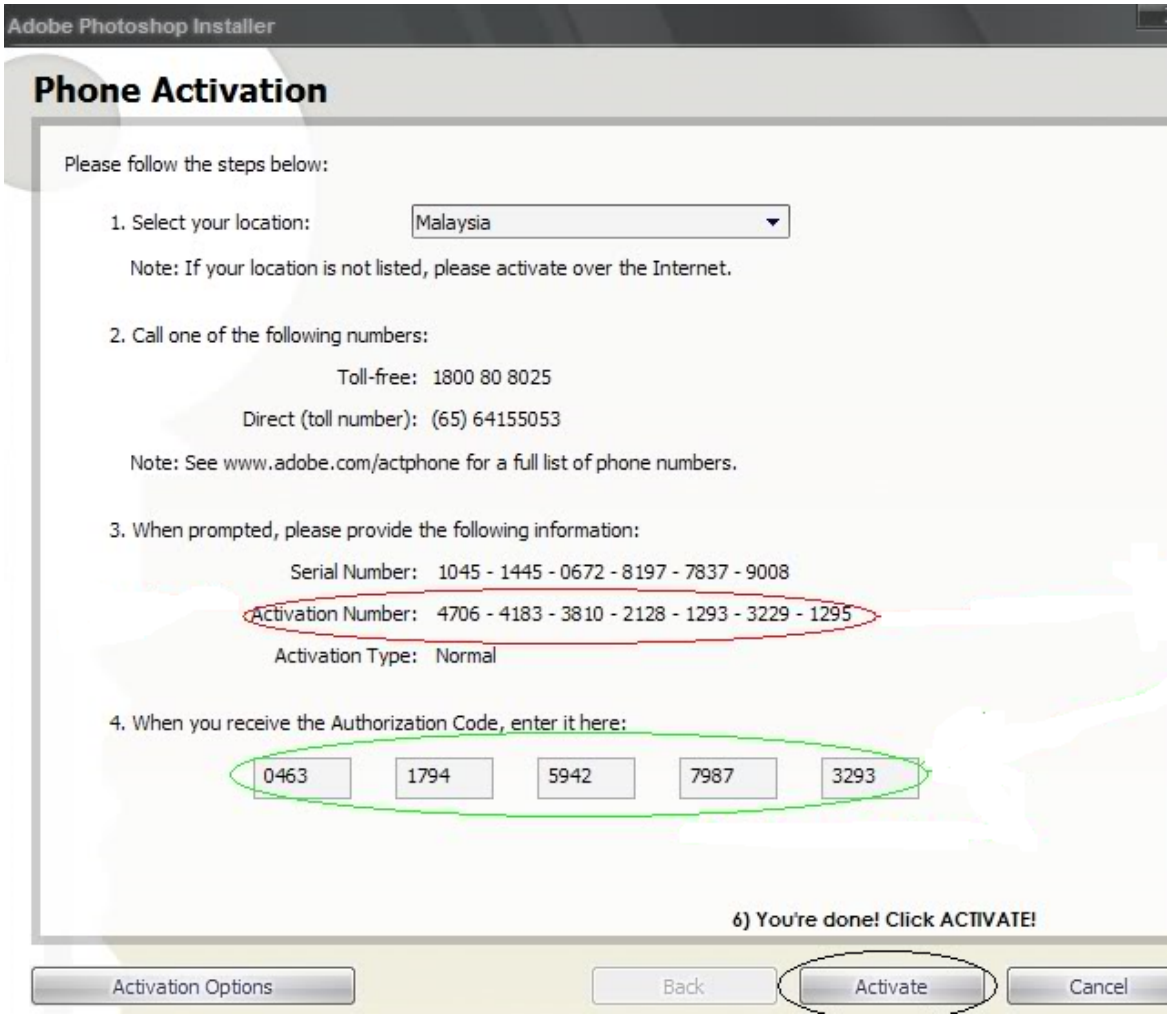

Figure 2. Adobe Photoshop's 2-layer Activation
Source: Chinmaan [26]

2.2.3 By Hardware Key

Activation by hardware key is perhaps the least common method in use today. This method requires software to work in the presence of some special hardware device [5]. This kind of activation is perhaps the most difficult to break since it is very time consuming to figure out what the hardware key does. For example, the hardware device may participate in some calculations performed by the software, and it would be difficult for hackers to figure out what calculations the hardware device does exactly. A secured hardware key can be a smart card. One advantage of using a smart card as a hardware key is that it has cryptography built in for protection to make the smart card tamper resistant. Any communication with the smart card is cryptographically secured and the smart card is able to lock or destroy itself if authentication is failed for 10 times consecutively [6]. Bank of China requires a USB drive to activate its online banking software [7].

2.2.4 Pre-activation at Vendor

Pre-activation at vendor is a method to activate many OEM software products on brand name computers. For example, Microsoft's Windows operating system (OS) is the most widely pre-activated software. When a user purchases a new brand name computer, the user may also get other bundled software from the vendor – pre-installed and pre-activated.

For Microsoft Windows, activation information may be stored in the BIOS on the motherboard and the OS would check the BIOS for activation information. This activation method for Microsoft Windows is often explored by hackers; see more details in Section 5.2

2.2.5 Comparison of Methods

Table 1 shows a comparison of different methods of activating software, including pros and cons of each method, summarized from ORC's lessons [3].

Table 1. Comparison of Different Methods for Software Activation

Method	Popularity	Convenience for the user	Effectiveness of protecting against privacy
Serial Number	Very popular for individual users, for many kinds of software	Convenient	Usually not effective, especially for shareware. Because one may be able to use the same serial number on multiple copies of the software.
Activation File	In use, but not common	Not very convenient	Not very effective, same copy of activation file may be used to activate multiple copies of the same software
Hardware Key	Barely in use today	Not convenient	Very effective
Pre-activation at vendor	Very popular for computer sellers, mostly for OS	Very convenient	Can be effective, but depends on exact implementation

3. Serial Number

Serial number is a string consisting of alphanumeric characters and is the most popular method of activating software products. It is sometimes known by other names, such as CD Key, product key, activation code, and so on. Each legal copy of the same software product should be distributed with a unique serial number; this unique serial number is used to indicate legitimate rights to use the corresponding software. Although the use of a serial number was intended to combat software privacy, it is not very successful.

Sections 3.1 below will discuss various aspects of using serial number as an activation mechanism.

3.1 Ways of Obtaining Serial Number:

The very basic problem of using a serial number to activate software is how users get the serial numbers from software vendors. In today's commercial practice, this problem is primarily solved in the 2 following ways: by email, and from retail package.

3.1.1 Send by Email

A common practice used by many shareware vendors to distribute serial numbers is to send out the serial number to users in emails after they pay for the software. This method is quite easy, but it requires the user to have an email account to receive such emails. A very similar way is to display the serial number on the webpage along with confirmation of purchase; this eliminates the need to have an email account. Another problem with this method is keeping the serial number after use. Users may need to reinstall the software for whatever reasons they have, which will require users to enter the serial number again. Usually, people are not good at keeping such important information well in order.

Sometimes free software requires activation too, such as Avast's anti-virus home edition [28]. Avast requires users to register online to obtain a serial number, which is good for 1 year. After registration, users then receive an email containing a new serial number. In essence, there is not much incentive for obtaining a serial number for freeware.

In summary, this is perhaps the most convenient and popular way to distribute serial numbers if software is purchased online by download.

3.1.2 From Software Retail Package

In contrast to obtaining serial numbers by emails when buying software products online, users would get serial numbers in printed media if they buy them from retail stores. This is the original way of distributing serial numbers. When users buy software products, they get an installation

disk, a user manual, and a serial number inside the box. But due to popularity of the Internet, fewer and fewer software products are distributed this way.

3.2 Ways of Checking Serial Number:

While serial number is used to fight against piracy, simply using it is not all that matters; how the serial number is checked has an impact on the effectiveness of software activation.

3.2.1 One Time Checking Upon Entering

Most software products only check the serial number once after it is entered, but this also make it easy to pirate software. In this scenario, after a serial number is deemed valid, it would be good forever (a practice used in many shareware packages). As long as a user can obtain a good serial number, their copy of given software will be activated and no tampering of the software itself is necessary. The most serious pitfall of this method is vendors are unable to keep track of the serial numbers being used.

3.2.2 Dual Checking

Dual checking is an attempted improvement over the one-time-checking mechanism. This method is mostly found from Adobe's products, such as Photoshop. After entering the serial number, users are required to either go online or call by phone to obtain a second activation code and use that to complete the activation process.

By contacting the vendor (Adobe in the case of above example), the vendor is able to validate whether the first serial number is good, how many times it is used, and if that particular serial number is actually distributed by Adobe. By doing this, Adobe is able to keep track of distributed serial numbers and, perhaps, take actions against those who spread them illegally. Of course, this is only an improvement over the previous method; it does not solve software piracy problems (if piracy can be solved at all).

3.2.3 Repeated Checking Over Time

This method would require software products to be validated multiple times over time, after successfully activating it in the first place. This makes piracy more difficult with a key management server. Microsoft employs this method in its Windows XP and later operating systems. When users download critical updates from Microsoft, Microsoft will try to check whether the running OS is a legal copy by using its GenuineAdvantage software. In this scenario, pirated copies will most likely be detected by Microsoft because the same serial number is used too many times. Figure 3 shows a screenshot of online detection.

Figure 3. Microsoft's Online Genuine Software Validation

The upside of this method is that it enables a vendor to detect piracy of its software with a relatively high success rate; the downside is that it must get users to repeatedly go back to the vendor. Distributing security updates is obviously a good reason to get users back to the vendor, but this reason only applies to a few software products because most other software does not provide routine updates like Microsoft. Therefore, while this method is considered to be a better improvement over dual checking, its use is severely limited.

3.3 Ways of Entering Serial Number:

There are a few ways to enter a serial number, how this is accomplished determines how difficult it is to break the protection.

3.3.1 Only at Installation Time

A common way to enter a serial number is during installation. This method is usually used by software without trial versions. While cracking a serial number at installation time is more difficult, it is still doable. Usually a serial number entered at installation time is not checked

against the vendor's system, so once the software is cracked or a serial number is leaked, piracy is unstoppable.

3.3.2 After Installation

Many software products allow users to enter a serial number after installing the software. This makes it a little easier for hackers to attack the activation mechanism, as they do not have to investigate a large amount of codes compared to the installation package. This method is widely used by software products having trial versions or trial periods.

3.3.3 By Inserting Hardware Key During Use

Some software products have serial numbers built into hardware keys. In this case, users do not get to see the serial number at all. As discussed in Section 2.2.3, this makes it very difficult to break the software activation mechanism. This method is not widely in use today.

3.4 How serial numbers are generated

There are many ways to generate and store serial numbers. How this is done directly affects how easy or difficult it is to break the mechanism.

3.4.1 There are Third Party Software Activation Packages for Sale

If software developers do not have much experience in this field, they may be better off using third party products for protection. One company that provides such service is LogicProtect; it claims to provide "clever software activation, anti-piracy functionality and copy protection for your software" [8]. LogicProject's service description says its service is able to provide both activation and online verification [8]. This will make the overall process more robust. In essence, LogicProtect provides its service by letting developers integrate LogicProtect's DLL into their software. In its newest release (7.0), it even includes web service APIs, which make the online verification easier to implement.

3.4.2 Software Vendors Develop Their Own "Secret" Algorithms

In many cases, software companies prefer to develop their own secret algorithm for generating and checking serial numbers. The idea behind this practice is that the "secret algorithm" is supposed to be difficult to break because no one from the outside knows about it; however, this idea contradicts Kirchhoff's principle [9]. In fact, the majority of serial number generation and checking algorithms are broken by hackers. Once the part of the code responsible for serial number generation is identified, hackers can simply "rip" out such code and use it to create a KeyGen for that software product [10]. Among different secret algorithms, use of hash functions is one of the favorites.

3.4.3 Generate Keys From Other Software

Sometimes software developers use third party software products to generate serial numbers and develop their own code to verify the serial numbers. Jedisware's "CD Key Generator" is one software product that can generate serial numbers of various lengths and formats (such as use of hyphens, numbers only, and so on). The full version of "CD Key Generator" allows users to save the generated serial numbers in a file or in a few data structures such as array or arraylist. What was ironic is that "CD Key Generator" itself is not good at serial number checking. What it did essentially is to store all 5000 valid serial numbers as an array of strings in the software, see Figure 4, and does a comparison against all stored valid serial numbers to check for validity [4]. In this example, we can conclude that it is a terrible idea to store valid serial numbers in the software's source code.

3.5 Criticism of Software Activation

While use of software activation is popular, it is also criticized by many; some criticisms include the following [11]:

- If a computer is stolen or destroyed, the activation records on it may be completely lost. It is only by the good will of the company that products can be re-activated. This makes backing up to guarantee prevention of substantial loss impossible.
- It can cause inconvenience for the end-user, particularly if phone calls are necessary to complete activation or technical problems such as firewall blocks or activation server downtime, preventing the activation process from completing.
- It can enforce software license agreement restrictions that may be legally invalid. For example, a company may refuse to reactivate software on an upgraded or new PC, even if the user may have a legal right to use the product under such circumstances.
- If the company ceases to support a specific product (or declares bankruptcy), its purchased product may become unusable or incapable of being (re)installed unless an activation-free copy or final patch that removes or bypasses activation is released.
- Although many activation schemes are anonymous, some are accompanied by mandatory registration which requires providing the user's address, phone number, and other personal information before the product is activated.
- Many argue that product activation does not protect against piracy at all; pirates often find ways to circumvent product activation.
- Product activation has also resulted in many software vendors treating their customers with much more hostility than they did before they introduced it into their products. This can mean that all users, including those with no intention to illegally distribute their products or knowingly acquire or use bootleg copies, are suspected of being involved in activities related to piracy.
- Product activation where there is no straightforward way to transfer the license to another person to activate on their computer has been widely criticized as making second-hand sales of products, particularly games, very difficult. Some suspect

companies such as EA to be using product activation to reduce second-hand sales of their games in order to increase sales of new copies.

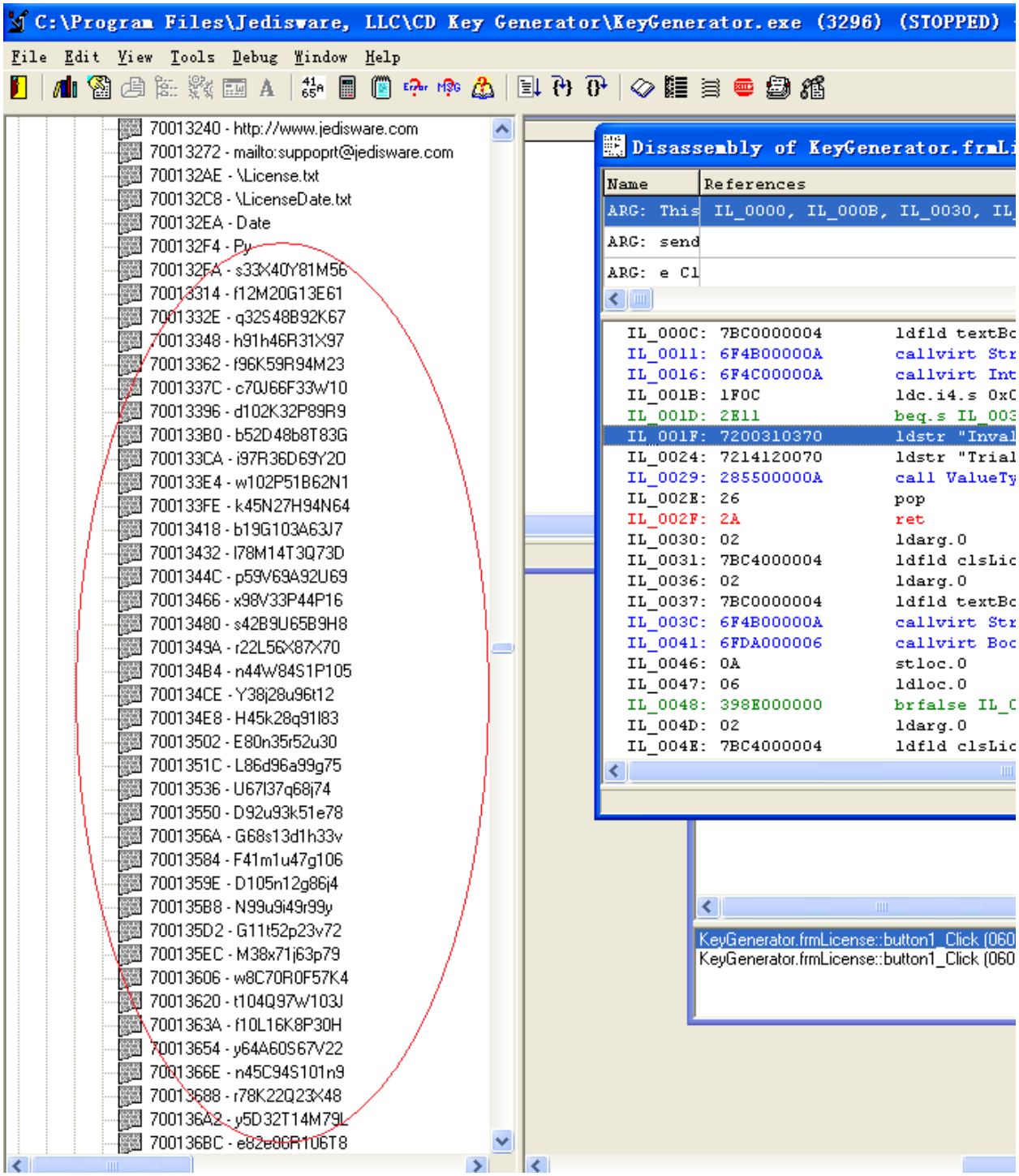

Figure 4. CD Key Generator's Valid Serial Numbers

4. Programming Languages and IDEs

While serial number generation and checking appears to be a problem in the domain of an algorithm, choice of programming language and IDE matters, too. In this project, we focus on the software reverse engineering aspect of programming languages.

4.1 Compiled Executable

Compiled executables are the most difficult ones to reverse engineer because reversers must have a good knowledge of assembly language. For this reason, C/C++, or other languages which compile code directly into binary machine code should be used to write code related serial number generation and checking.

Another advantage of compiled code is it can strip out all information regarding function and variable names, leaving hackers to interpret memory addresses.

We must emphasize that compile executables only makes reversing more difficult, but not impossible. Hackers can still use various tools (like IDA Pro [29]) and their expert knowledge to accomplish the job.

4.2 Byte Code

Unlike compiled executables, which are in the form of binary machine code, byte codes contain a considerable amount of information that is extremely helpful to hackers, and more importantly, they can be decompiled to obtain high level source code quite similar to the original source code. Developers need to add various obfuscation to the source code to effectively prevent reversing.

4.2.1 Java

Many programs written in Java are open source and, hence, free; therefore, there is not much incentive to attack such Java programs for our purpose. But if one needs to, one can use "Frontend Plus" to decompile the ".class" files containing the byte codes in order to obtain the corresponding "source code" [12].

To obfuscate source code written in Java, one can use "ProGuard" [13] to do the trick. It obfuscates symbolic information such as function names and variable names, even class names [13]. Once obfuscation is done, it would be very difficult to understand the original source code.

To add another level obfuscation, one may use "SandMark" to encrypt the ".jar" files [14]. But obfuscation done by this tool can be easily reversed, so it does not provide much value. Alternatively, users can use the Java Virtual Machine's (JVM) keytool to encrypt the ".class"

files and the corresponding certificate will be stored in the key store [15]. Users must know the password to the key store in order to retrieve the certificate and use it to decrypt the .class files.

There are tools currently available that can pack necessary libraries together with the user-written code and generate an executable [16]; this makes reversing considerably difficult at the expense of platform dependence.

4.2.2 .Net

.Net is essentially Microsoft's version of Java as they share many similarities, especially C#. Just like Java, after compiling the source code, byte codes known as Microsoft Intermediate Language (MSIL) are generated. Because of this, hackers can decompile such programs using tools to obtain codes quite similar to original source code.

To prevent reversing, developers can use tools like Xencode Postbuild [17] to add obfuscation, or even generate native binary machine code, which can run without the .Net framework [17].

4.3 Integrated Development Environment

If developers think IDEs do not matter for our purpose, they are wrong. First, different IDEs may compile source code differently. For example, for the same source code, Microsoft Visual Studio would generate binary code much smaller in size compared to Dev C++; this directly affects how the corresponding disassembly looks. In addition, different compilers may provide different compiling options, such as optimization and so on. Such options would also have an impact on the binary machine code. Sometimes, optimization may even undo or get rid of some the anti-reversing tricks used.

5. How to crack software activation

In this chapter, we will discuss breaking software activation from two perspectives: the regular user and the hacker. The difference in two perspectives will determine what needs to be done to break the activation.

5.1 KeyGen

KeyGen, short for key generator, is a tool used to generate serial numbers for software. Such tools are developed by hackers to activate software in order to eliminate all restrictions imposed in trial versions. Regular users can perform a Google search on the Internet to find the matching KeyGen for a particular software product, and then use it to generate serial numbers to active the software. That is all that's needed.

There are two ways to create a KeyGen:

1. Recreate the underlying algorithm after understanding the corresponding disassembly
2. Rip out the assembly code and use it directly.

Both of these methods require identifying the correct section of code responsible for checking serial number. After this, the first method requires extensive study of the code in order to reconstruct the algorithm used, whereas the second method only requires a copy and paste with some minor modification. Comparing the two methods, the second one obviously requires much less time. However, even if KeyGen is able to generate a serial number passing the algorithm's test, it still may not be good enough. Sometimes serial numbers are also checked against the vendors' databases and vendors may set special rules on valid serial numbers, such as not having certain alphanumeric characters. Such preset rules may not be present in the checking algorithm, and hence KeyGens wouldn't know about them.

5.2 Common Methods for Cracking Software Protection

Without going into much detail, hackers would generally do the following to find out and attack the activation mechanism, summarized from ORC's lessons [3]:

- Take snapshots of memory after inputting password or serial number and then try to identify the memory location by finding echoes of what you type in. Set breakpoints at them.
- Use some utilities to draw out the calling hierarchy of functions
- Use breakpoints wisely as "single-stepping" is expensive.
- Try to identify a section where you repeatedly find assembly codes because they could be precisely the protection as they are usually added at the end of development.

- Examine all referenced strings first, if possible; try to identify messages related to protection scheme (such as success or failure messages displayed on screen).
- Study virus as they are the best source for good “tight and tricky” assembly code. This is of course done prior to actual hacking.
- Sometimes it is good to use a sequence of working instructions than a series of “NOP”s as newer protection schemes “smells” them. For example, hashing of code can detect any change to it.

6. Anti-reversing Techniques

In this chapter, we discuss various anti-reversing techniques, including the purpose of their use and relative effectiveness in making cracking more difficult.

6.1 Detect Debugger

Hackers must use debuggers in order to successfully understand the design of the activation mechanism. Therefore, if protection is added to prevent code to be run by debuggers, one may force hackers to give up on reversing such software because they just lost their most valuable tool at their disposal.

6.1.1 IsDebuggerPresent()

IsDebuggerPresent() is a system function provided in Microsoft's library. If a process is started by a debugger, calling this function can try to detect the presence of the debugger; if a debugger is attached to a process after it is started by other means, calls to this function could return false.

One downside with this function is that this function can be easily identified by modern debuggers. As shown in Figure 5, OllyDbg is able to identify this system function call. Hackers can easily disable calls to this function to bypass the check, as shown in Figure 6. Our research found that this method is not particularly effective because they can be easily identified by debuggers.

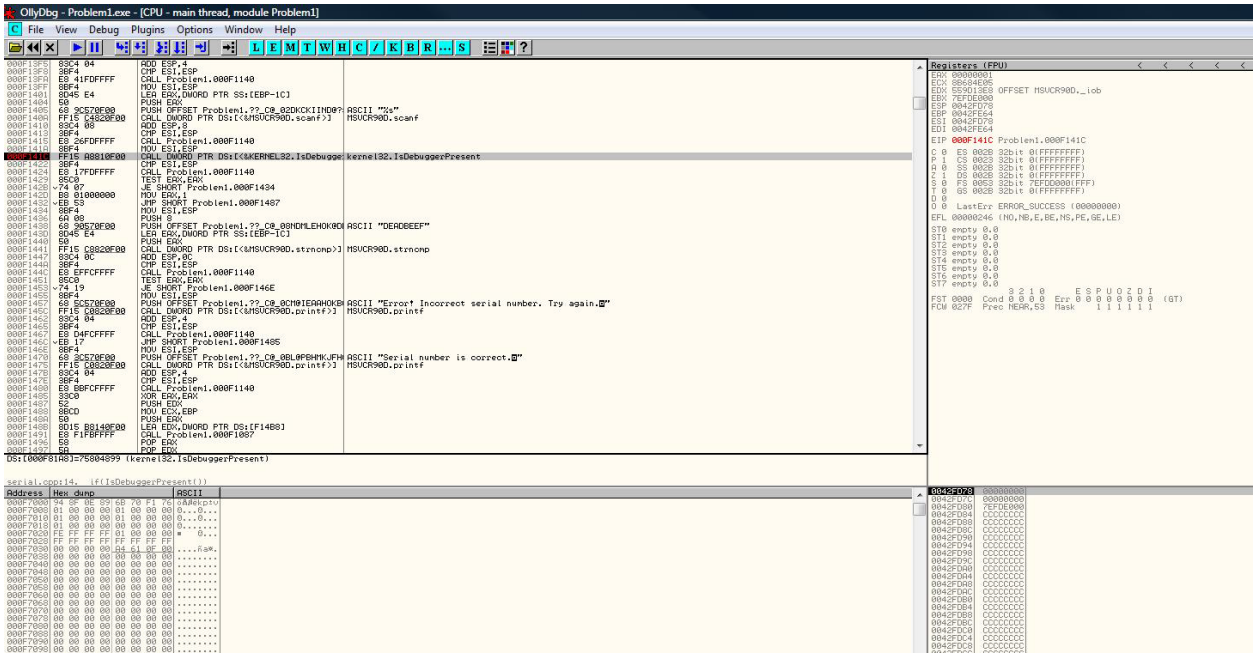

Figure 5. Identifying Function IsDebuggerPresent()

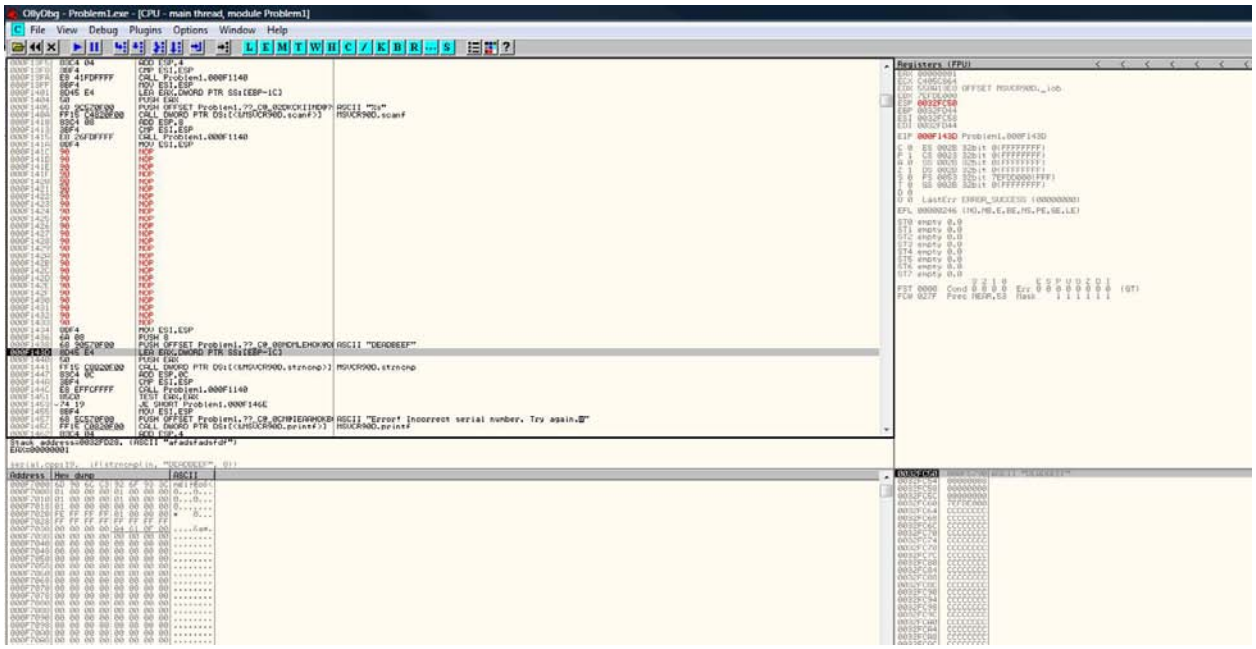

Figure 6. Bypassing Function IsDebuggerPresent()

6.1.2 Time Checking of Code

Developers can write their own code to try to detect presence of a debugger at runtime. Depending on how code is written, it may or may not be effective. One effective method is to check the run time of a segment of code. Developers can use trial and error to determine the normal run time of a block of code, and then set a limit based on the result. Developers can then use another thread to check the run time of that block of code and see if the limit has been exceeded; if so, it is probably because a debugger has stopped the execution.

6.2 Insertion of Assembly Code

Developers can add in various well-designed assembly codes to confuse disassemblers. However, our research found that modern disassemblers are smart enough to deal with this tactic. At best, only a few lines of disassembled code can be confused, hence proving this method of less value. In Figure 7, the boxed line of code in red shows the only line of assembly code that got messed up.

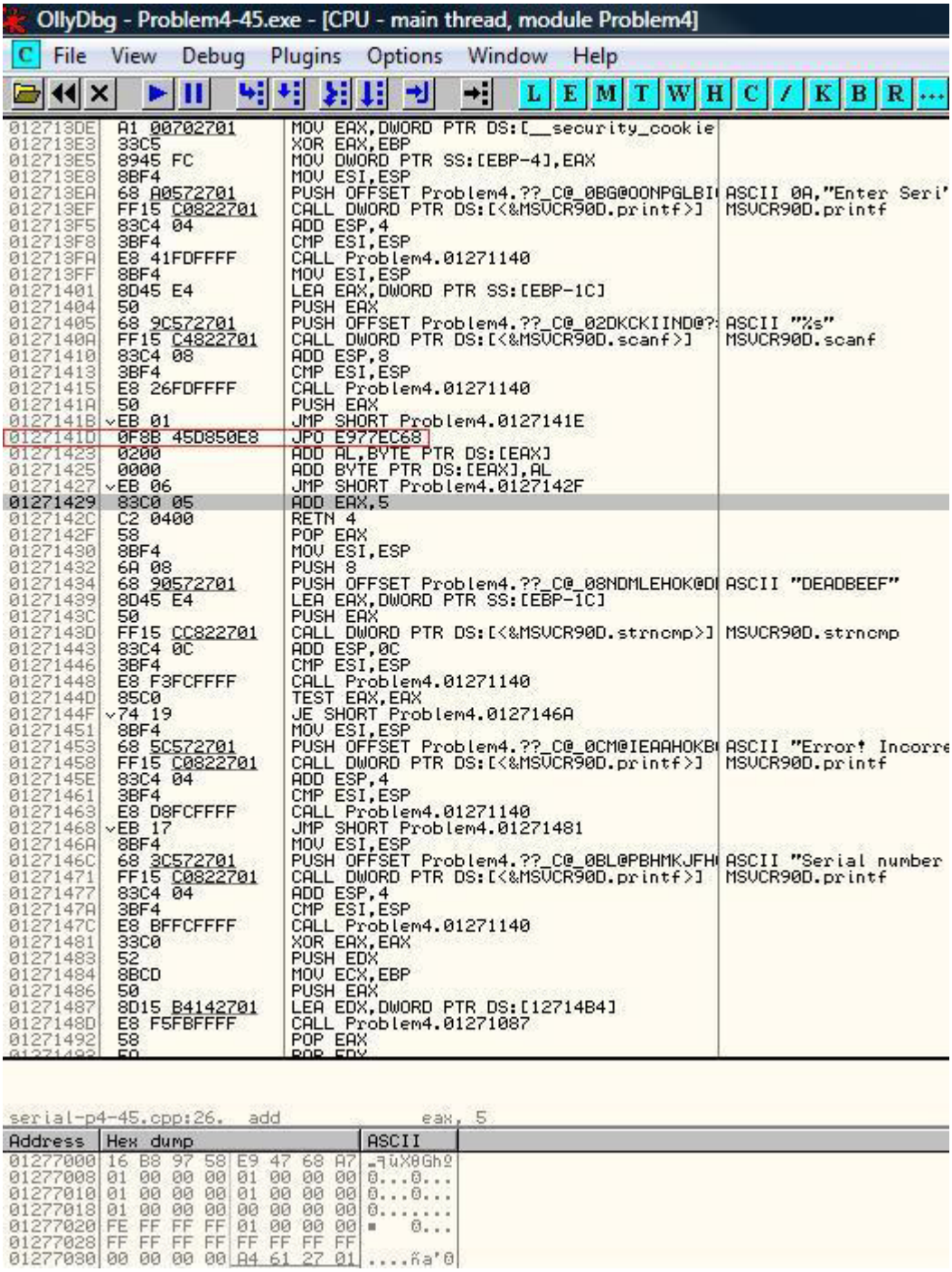

Figure 7. Effect of Using Assembly Code to Confuse Disassembler

6.3 Insertion of Junk Code

Insertion of junk code into meaningful code is intended to confuse hackers. Junk code works by causing hackers to spend more time studying useless code as well as divert their attention from good code. Our research found that when much junk code had been inserted, it may not be possible to identify the good code from the bad. It definitely took significantly much more time in hacking efforts. Overall, this technique can be very effective. In this section, we will discuss 3 kinds of junk code.

6.3.1 Junk Logic

Junk logic is junk code added in the code section. Common examples include adding useless instructions and mixing them together with useful code. This provides protection at the expense of run time. Depending on how much junk code is inserted, run time overhead can be significant.

6.3.2 Junk Data

Junk data refers to useless variables in source code. Its purpose and use is more or less like junk logic, except it may not have considerable overhead in run time.

6.3.3 Polymorphism

Polymorphic code was originally and commonly used in writing viruses. It uses a polymorphic engine to mutate the code while keeping the original algorithm intact [18]. This technique is built on top of encrypted code. It mutates the encryption/decryption code so that each copy looks different. While this technique was invented by virus writers, it can be applied to normal code to increase protection.

6.4 Recursion

Recursive function calls are good for significantly increasing the stack size because many parameters and return addresses will be placed onto the stack in the process. This can effectively disrupt a hacker's view of information stored on the stack. One downside with this technique is recursive functions are usually short in length of code and hence can be easily spotted and understood. If the recursion does not do anything useful, hackers can simply disable them.

6.5 Hash Function

When used as protection, hash functions can detect change in code effectively with certainty. Developers can choose to do one or both of the following: hashing the entire executable, or just part of the code.

6.5.1 Entire Image of Binary Executable

Hashing the entire image of binary executable can detect code patching created by hackers. One problem with this is that one must store the hash value external to the executable. A possible improvement using this method may be using cryptographic hash with keys and applying it several times with different keys. This is to prevent making permanent changes to the code.

6.5.2 In-memory Checksum

Hashing can also be applied to code loaded into memory. The purpose of doing this is the same as before – to prevent hacking from making changes, except that such changes are made at runtime. This can effectively prevent hackers from using debuggers to modify code at runtime in order to change execution flow. But this is very difficult to implement in practice.

6.6 String Obfuscation

String obfuscation can be used to hide certain types of important information; this can be done by using encryption. Simple encryption techniques, such as simple XOR or one time padding, can accomplish this purpose. One problem with simple encryption is that a hacker can get information out of the cipher text based on its length. To make string obfuscation more effective, developers should to use a different length for the encrypted strings compared to the original ones. Another problem with this technique is that hackers are not usually interested in the strings themselves; rather, they want to know how and where the strings are used. Checking mechanisms often display messages to users after they input serial numbers to indicate success or failure; these messages often give out the location of checking mechanism. Given that hackers are more interested in identifying locations of checking mechanism, they can trace system function calls related to outputting messages, such as “print” or “Messagebox.show()” instead of focusing on trying to work out the obfuscation method. In this regard, string obfuscation may not provide much benefit for our purpose. Figure 8 gives an example of a debugger identifying system function called “fopen” and using it to find out string “readme.txt” as file name from EAX register.

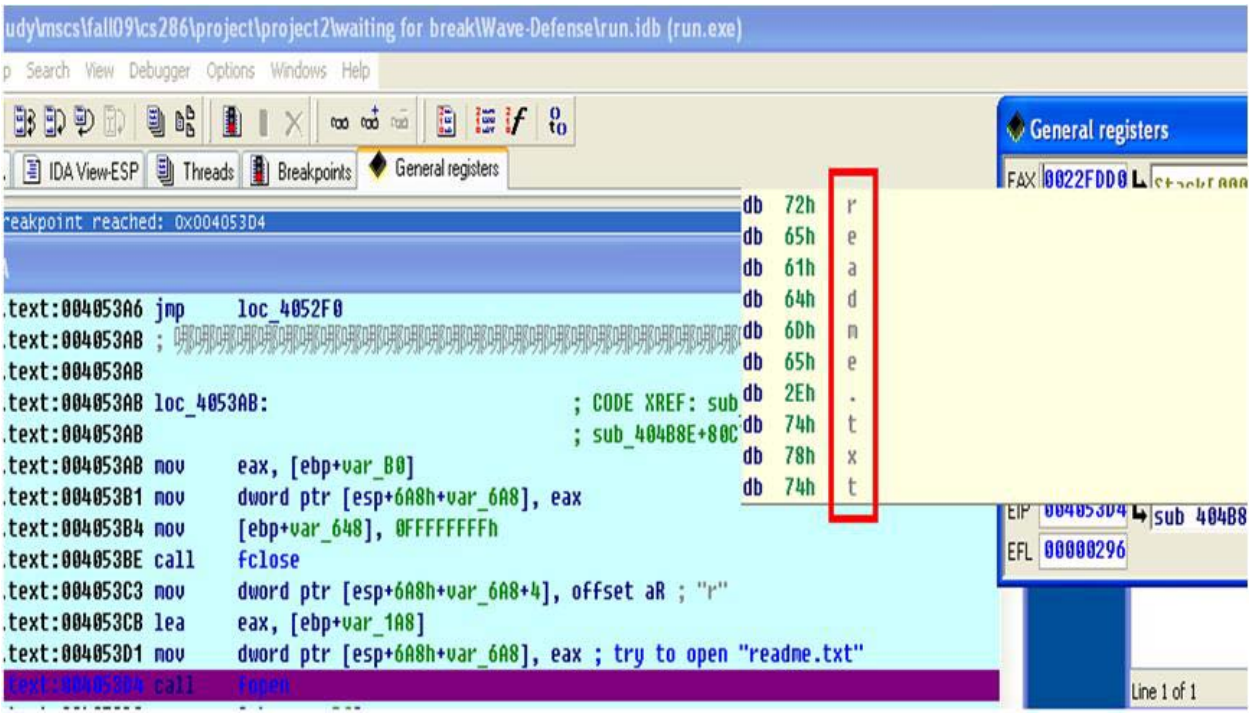

Figure 8. Getting Obfuscated String in Clear Text

6.7 Opaque Predicate

Opaque predicate refers to comparisons whose outcomes are either always true or always false. Using them increases the number of branches of code hackers need to trace, which can be very time consuming. Sometimes opaque predicates may actually be useless as they can be easy to spot; for example, if opaque predicates make use of floating point calculation in an algorithm that only uses integer calculation (or non-floating point calculation in general, as often is the case for serial number checking), hackers would know what code to skip. In contrast, using opaque predicates in places where they shouldn't be found may lead to a revelation of important logic. After tracing code a few times, hackers can realize their existence base on execution flow as well.

6.8 Control Flow Obfuscation

Control flow obfuscation refers to code executing in strange order or, at least, appears as a strange order. This is usually accomplished by using many "jumps." In essence, this is used to break locality of code. Psychologically, people would think code blocks next or close to each other are related and often are executed sequentially. Once locality is broken, hackers can feel lost when they have to jump through different places in order to trace code. Figure 9 shows how complex control flow can be by adding a considerable amount of junk code into one subroutine. Figure 10 shows the effect of breaking up all of a program's code into smaller functions making code harder to trace. Figure 11 offers a zoom-in view of a port of Figure 10. In Figure, each box

indicates a function, black being user written and pink being system functions, and each line indicates a function call from one function to another.

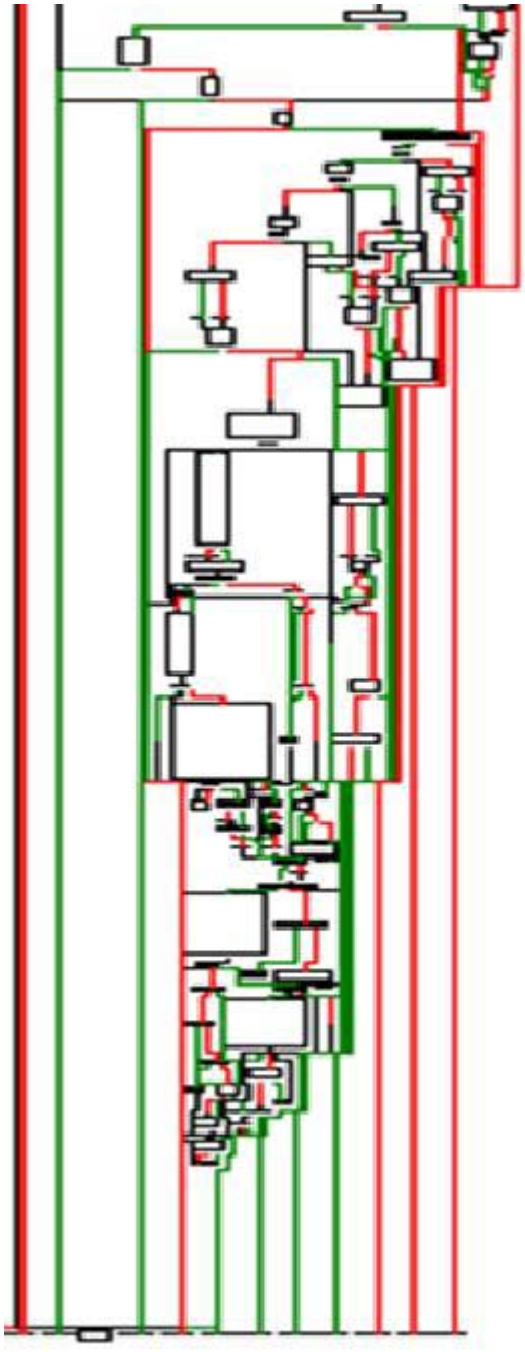

Figure 9. Flowchart of a Subroutine

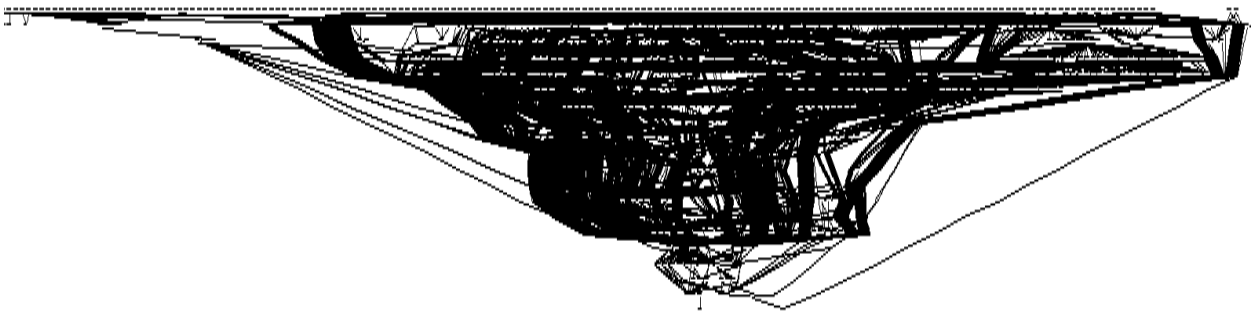

Figure 10. Function Call Hierarchy

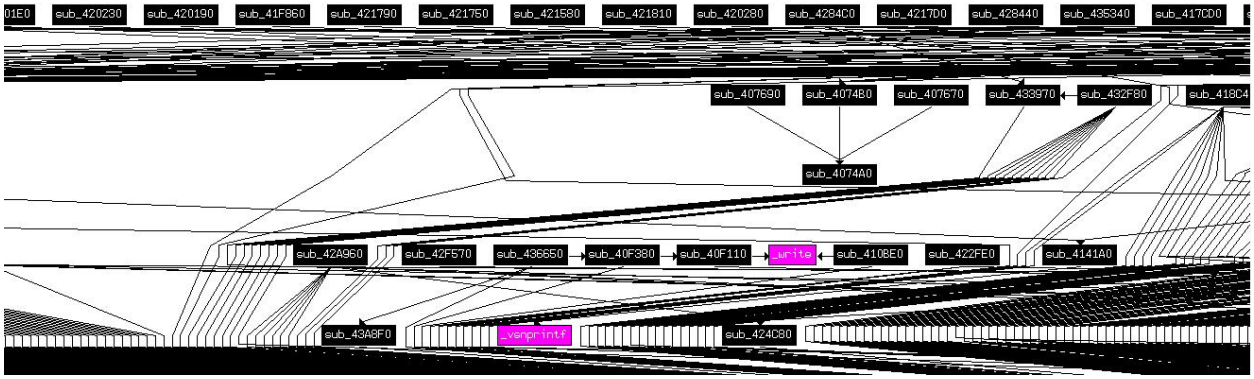

Figure 11. A Zoom-In View of Figure 10

6.9 Multiple Validation

Multiple validation logics can be used to prevent single point failure. For example, if a hacker just patches one of several such logics, the overall activation mechanism cannot be broken since the outcomes of validation from different logics are inconsistent. In this case, software can detect tampering of its code. It is up to software developers to decide what actions to take when such inconsistency is encountered. One advanced technique is to have the logics correct each other on-the-fly. If hackers are not aware of the presence of multiple validation logics, this technique can be very effective, as it would generally be difficult for hackers to figure out why their patches are not working.

6.10 Multithreading

The original purpose of having a multithreaded application is to parallelize some of the logic and have the threads execute concurrently to increase overall efficiency. Here we use multithreading to increase difficulty of debugging.

It is inherently difficult to debug a multithreaded program even if its developers have the source code due to a variety of reasons, such as data synchronization and so on. The difficulty arises from the fact that only an operating system has control over when and which thread runs, but not the application itself and hence not the developers either. In addition, debug mode and release mode may yield different results for the same piece of code. For example, if the developers didn't initially synchronize data correctly, the release mode may yield incorrect results whereas nothing may seem wrong in debug mode because the debug mode may force synchronization as it has to display the result to the viewer.

For our purpose, we can use multiple threads to do the work concurrently so that hackers cannot easily single step through code to find out how the logic works, since validation may have been completed elsewhere.

6.11Windows Events

Windows events are directly related to graphical user interface, commonly known as GUI. Here we use windows events to obfuscate the execution flow, more or less like using multithreading. Windows events are raised by users through interaction with a GUI and processed by an interface thread (sometimes known as an event thread). Developers can take advantage of this by handling multiple events in the code so that execution will jump from one place to another sporadically making hackers feel lost. Events, such as mouse movements, will be triggered many times, which can certainly annoy hackers.

6.12 Comparison of Effectiveness

Table 2 below compares relative effectiveness of various anti-reversing techniques discussed above base on our research of them.

Table 2. Comparison of Effectiveness of Different Anti-Reversing Techniques

Method	Relative Effectiveness	Pro	Con
Junk code	Very good	Can make code very hard to trace, force hacker to distinguish between useful and useless code, very time consuming	Over head in run time, can be significant if junk code is too much
Recursion	Weak	Makes stack huge in size	Recursive functions are usually short and easy to understand
Hashing	Good	Can detect change in code	A little overhead in run time
String obfuscation	Weak	Makes hacker hard to find the correct logic to investigate	Can trace system functions for output to trace the strings
Opaque predicate	Weak	Makes code have more branches	Human may be able to identify them easily, a little run time overhead
Control flow obfuscation	Very good	Makes code very difficult to trace by breaking proximity of locality	Not much
Multiple validation logic	Good	Can prevent single point failure	Not much, other than having to write more code
Multi-threading	Very good	Makes it difficult to trace code	May be difficult to implement
Window events	Good	Can break sequential execution of code, events may occur many times (such as Idle or MouseMove)	Not much

7. New Design

This section will propose a new design, along with testing results of the new design. Section 7.2 will go over the techniques used in the proposed new design; Section 7.3 will discuss testing results of the new design.

7.1 Consideration of New Design

In this section, we will discuss several techniques considered but excluded from the new design.

7.1.1 Shift Workload Online

One way to use hardware keys is to use the hardware device to perform part of the computation; similarly, we can do part of computation online, such as using web services. In this approach, the installed local copy does not have full functionality. The server side can check for proper licensing before completing requested computation. This way, activation mechanism is nearly hack-proof; however, such activation mechanism is way too complicated to implement, not to mention significant overhead and slowness, which renders this method impractical in most applications.

7.1.2 Encrypting Executable

Encrypting executable is a strong anti-disassembling method. But this is extremely difficult to implement in practice, especially with new security features built into current operating systems (OS). Storing an encryption key safely is another issue.

7.1.3 Disabling Debugger

It is impossible to do reverse engineering work without a debugger, so disabling them (in one way or another) seems to be an attractive choice. But in practice, it is very difficult, if possible at all, to disable use of a debugger. The core issue here revolves around the inability to determine presence of a debugger effectively and accurately, partially due to new hardware architecture and new OS security features.

7.2 New Design

7.2.1 License File

Instead of requiring a user type in a program serial number from a keyboard, a license file will be used. The license file should be generated by the software vendor, and distributed to users via email; users should then save the license file in a proper place on their hard drives.

The license file should be encrypted using a strong encryption algorithm, such as Advanced Encryption Standard (AES), with an/a encryption/decryption key derived from a password, one that is only known to the vendor and user (each user will decide their own password during

registration process). In this design, the format of the license file is XML, and contains information such as username, the hash value of program's binary, a serial number, and necessary validation information. Other information, such as trial expiration date, can be also included if necessary.

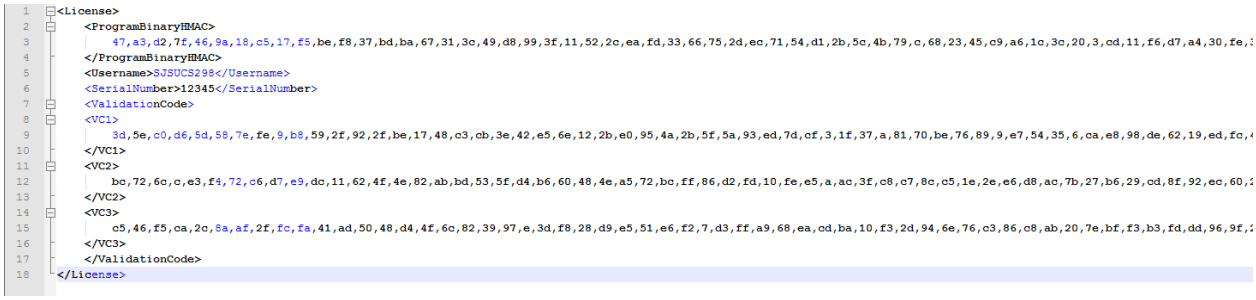

Figure 12. A Sample License File in XML Format

The hash value of the program's binary is intended to deter modification of the program by attackers. A Hashed Message Authentication Code (HMAC) algorithm is used to calculate the hash value with a key derived from the user's password.

The reason for using a license file, instead of manual user input, is to make it more difficult to locate the corresponding code responsible for validation. With breakpoints smartly set in a debugger, an attacker may be able to quickly find out roughly the beginning and end of a code region of interest, and then concentrate on that particular area. This is possible if the debugger is able to jump to that section of the code in question when it executes. In contrast, it is difficult to discover when the code of interest executes if it does not require user interaction; additionally, hackers would have to trace code from the very beginning to find out where code of interest locates.

7.2.2 Multi-threading

Using multiple threads to do work for serial number checking is the core idea in this design. The entire serial number verification process is divided into many small pieces (functions), and each piece will be run using a separate thread. Any dependency among threads can be resolved by "WaitHandle." On a high level design, the verification can be divided into 4 parts: verifying the program binary's hash value, and 3 verification logics for checking the serial number. See more details on multiple validations in Section 7.1.3 below. Each of these 4 logic blocks is further divided.

There are a few reasons why a multithreaded processing is chosen here over a single threaded version. First, it breaks the sequential execution flow. Even if code is broken into many pieces, the execution flow is not changed (disrupted); a hacker can still easily trace the execution to understand in which order the code is run. Once the order is known, code can be analyzed more effectively. In essence, breaking-up code and running it in a sequential order, at most, makes

code tracing a bit annoying, having to jump from one place to another. Having many jumps can break attacker's sense of locality, but with analytic tools, code can be easily understood by drawing a flow chart. In contrast, using multiple threads running concurrently will fundamentally change the execution order, which makes code much more difficult to trace.

Second, multithreading is very debugger-unfriendly. Even with source code, a multi-threaded application can be very difficult to debug [19]. Timing is absolutely one of the most important factors when debugging a multi-threaded application. A bug observed in normal run may not be reproducible in debug run simply because the timing is different. Also, a debugger is not able to trace two threads at the same time, in the sense that one cannot single step through more than one section of disassembly at the same time, even if the debugger is aware of existence of other threads.

Third, it is out of anyone's control when and which thread runs; this is only determined by the operating system's (OS) task scheduler. Because of this, different runs of the same code on the same debugger may yield different execution sequence, depending on which thread the debugger is able to gain control over.

7.2.3 Multiple Validation

Using multiple validation logics has the obvious advantage over a single logic because it may prevent a single point failure. This design employs 4 validation logics with 2 of them being able to correct each if an inconsistent result is detected. While this method is not foolproof, it certainly should work against attackers, as attackers will have to spend much more time locating existence of these logics and then breaking them. At the beginning of program, only 2 of the 4 logics are executed, and the other 2 are delayed according to our new design. This way, attackers may not discover the other logics even if they follow the execution flow from the start.

7.2.4 GUI and GUI Events

In this new design, certain GUIs are disabled by default, and their corresponding event handlers are not registered with the event. This is used to prevent unauthorized use of some special functions, such as full functions not found in trial versions. GUIs are properly enabled and event handlers properly registered if, and only if, all validation logics determine the program is a legitimate full version (not a hacked version). They are routinely turned off and on again to prevent an attacker from enabling them at a program's start by modifying the binary code.

7.2.5 OnIdle Event

OnIdle is an event issued by the OS when a program is in an idle state; it allows for idle time processing of low priority tasks. When a program needs user interaction, this event will be issued very frequently, as the user is very slow compared to the hardware. When the program does not have a user focus (not being the topmost application), this event may not be issued since this

entire program may not receive any CPU time. This new design utilizes the abovementioned feature of OnIdle as an anti-debugger technique and will use this event to process certain important tasks, such as synchronizing encryption/decryption keys and serial number checking.

Serial number checking takes advantage of an idle event being run very frequently, whereas key synchronization takes advantage of an idle event and can only run when a program has user focus. In the latter case, crypto keys may not be synchronized if the OnIdle function does not run, such as when the debugger windows are on top of the program's window.

7.2.6 Encrypting Calculated Results

With certain functions that require paying for a full version license, their results will be encrypted and then decrypted with key pairs. One key is calculated in advance at license issue time and stored in the license, while the other is derived from a serial number checking process. If everything goes right, these two keys are identical; therefore, encrypting the result then decrypting it should not change the result. If keys do not match, the correct result will be altered in the decryption process, yielding an incorrect final result for output.

This method adds protection against unauthorized use of a full version feature when not properly licensed, but it may carry significant overhead due to crypto-operations.

7.2.7 Junk Thread and Deadlock

Since multi-threaded checking is more effective than a single threaded version in terms of anti-reversing in theory, this design will run extra threads to complicate the situation more. And these extra threads will be used in combination with deadlocks.

Deadlock refers to a situation in which 2 or more threads each holding some resources while waiting to acquire more which are held by other threads; because no thread is able to obtain all required resources to proceed, all of them will sit idle and blocked. A classic example of deadlock caused by cycle is shown in Figure 13.

Deadlocks can work well against stepping through code in a debugger. When stepping through instructions in a debugger, one cannot move to the next instruction until the current one finishes. For example, if one tries to step over a function call that takes a long time to finish, the instruction right after the function call cannot be executed until the call returns. In this case, execution is temporarily blocked. If that function never returns, such as running an infinite loop, then the next instruction will be blocked indefinitely. In this new design, we will purposely create a deadlock situation with extra junk threads (threads that do not execute any useful work). When a debugger picks such a thread for a user to step through, it is expected that the progress will be blocked indefinitely. This technique attempts to divert an attacker from stepping through those threads that do work of real interest.

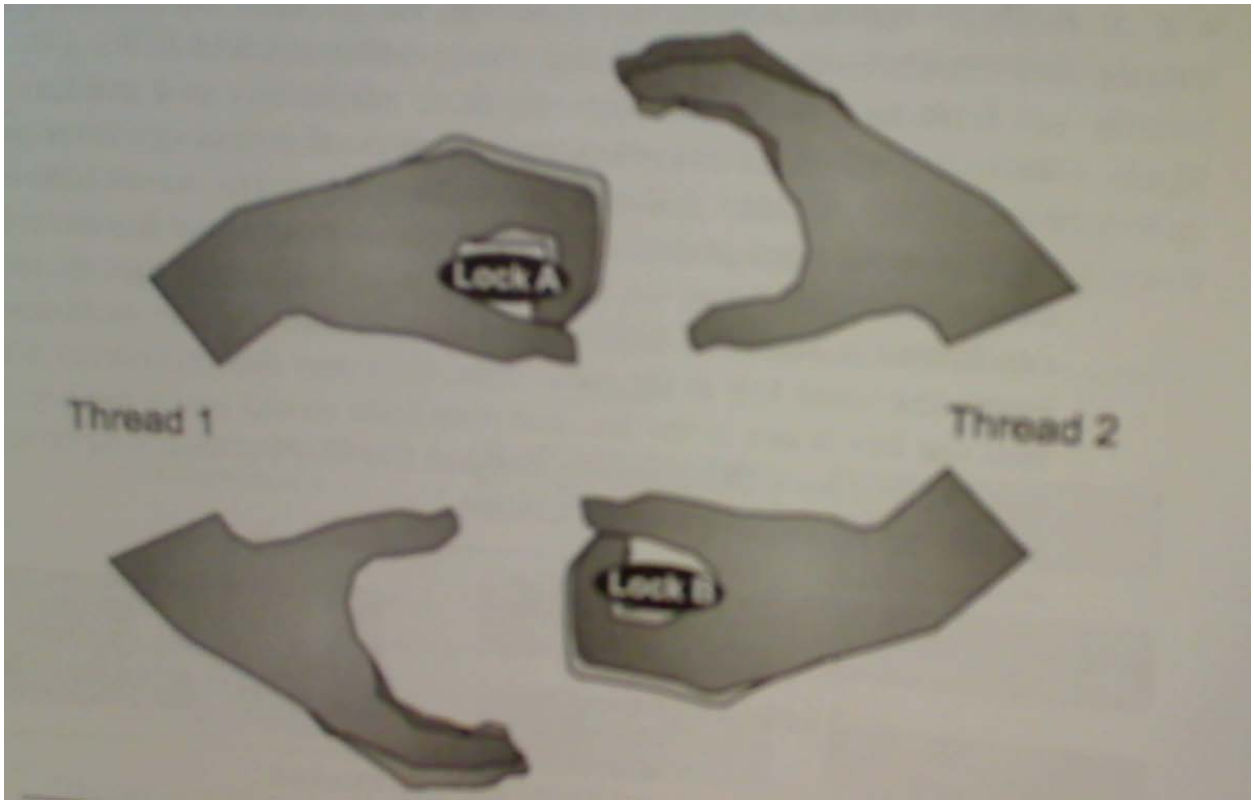

Figure 13. Deadlock Caused by Cycle
Source: Shameem Akhter and Jason Roberts, Multi-Core Programming [19]

7.2.8 Delayed Execution

In this new design, certain operations are delayed to hide its relationship with other operations. For example, one important use of this is exiting the program when checking fails to pass. Certain system calls can be easily identified by debuggers, by tracing these backwards sequentially, an attacker may discover where checking is performed. By delaying a certain execution and running it in another thread, we can effectively break an attacker's sense of code locality, making backwards tracing pointless. Using this technique, we can shift comparisons away from checking logic, forcing an attacker to trace more code.

7.2.9 Code Obfuscation

Obfuscated code is more difficult to understand, because one has to distinguish between the useful and useless code. This is often accomplished by inserting junk code and shifting code blocks around it. In this project, we hope to apply this technique to scramble code, but it is not easy to find a good polymorphic engine to accomplish this task. Xenocode's PostBuild [17] has built-in code obfuscator; we will use it without analysis of its effectiveness.

7.2.10 Putting the Blocks Together

Figure 14 below shows the flow and dependency of blocks responsible for verifying the integrity of a program's binary by hashing. If modification is detected, the program will terminate itself.

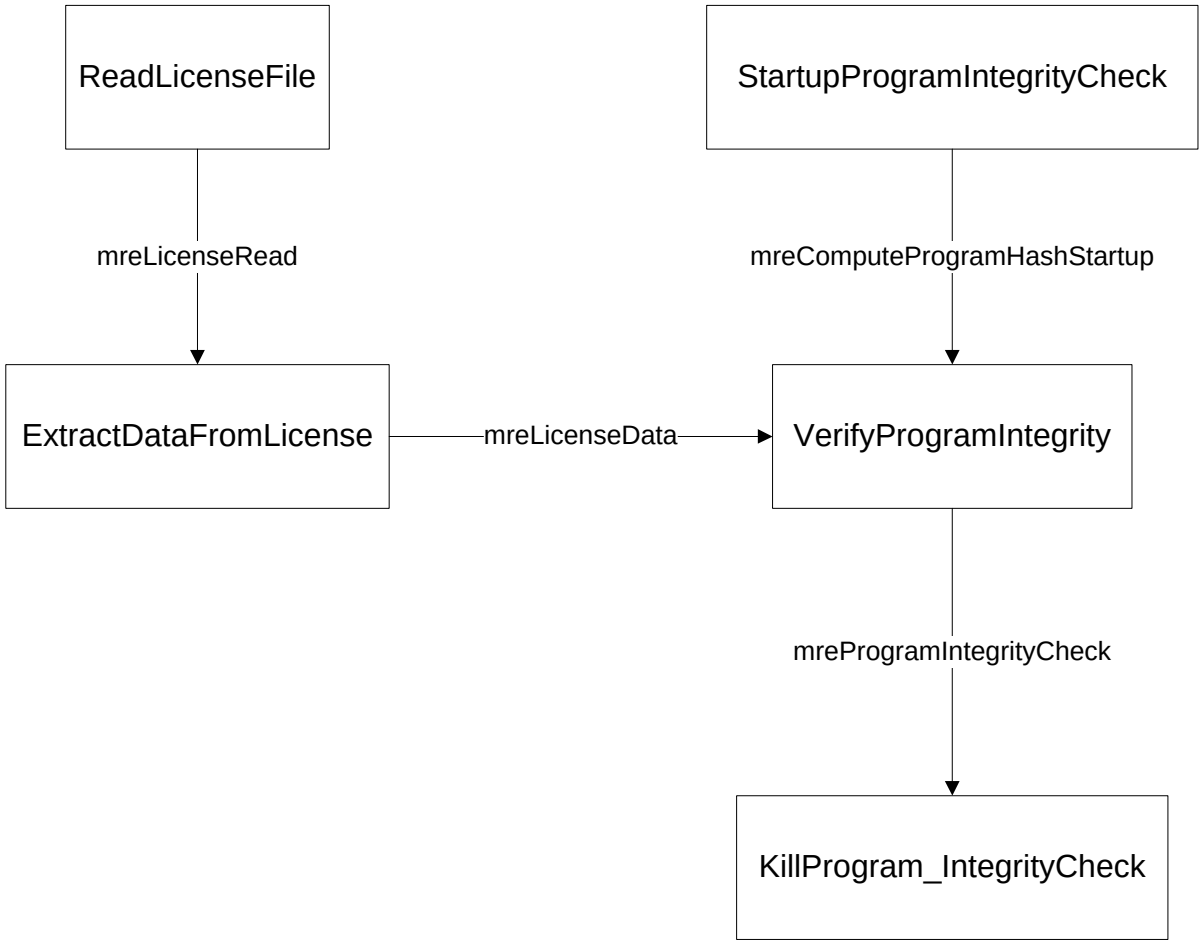

Figure 14. Block Diagram for Checking Program Binary Hash

Figure 15 shows the flow and dependency of blocks responsible for verifying the serial number at program startup. If verification is passed, GUI and corresponding handlers are enabled.

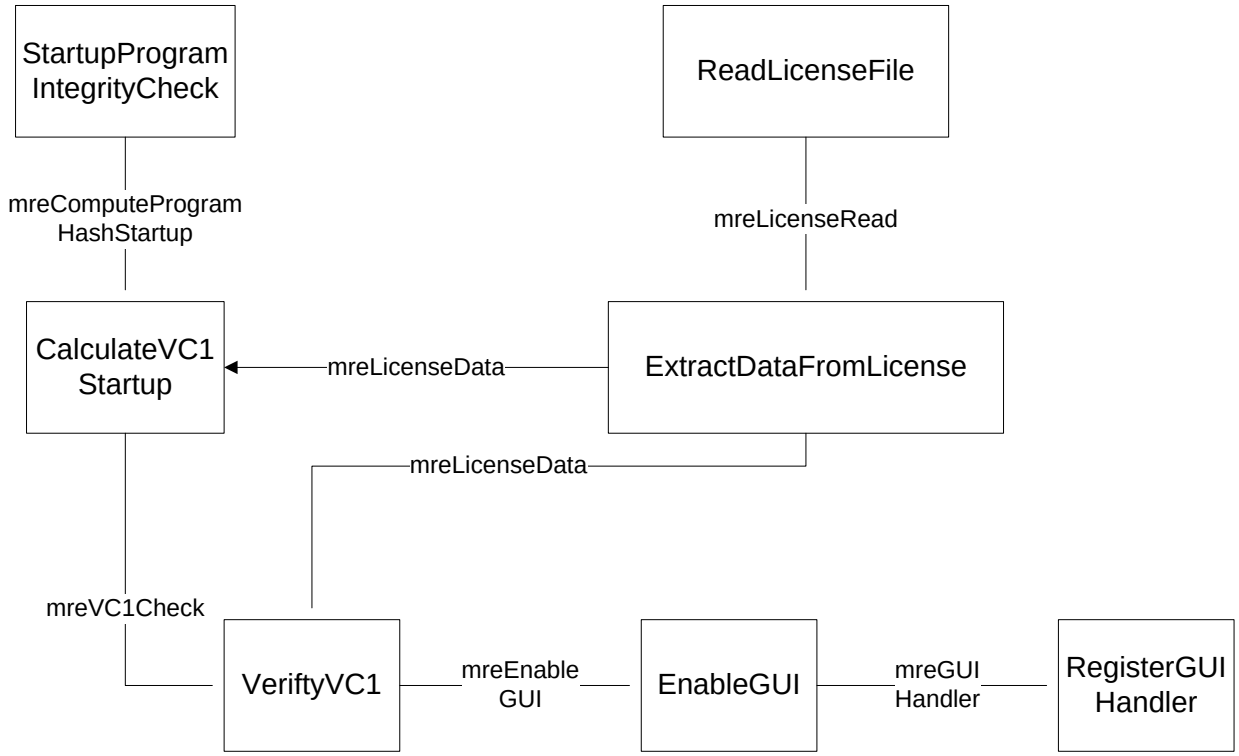

Figure 15. Block Diagram for Checking Serial Number Using First Checking Module

Figure 16 shows the flow and dependency of using a secondary module to verify a checking result obtained at program startup. If secondary checking demonstrates a different result than the startup checking, overall verification is deemed failed. In this case, the program will terminate.

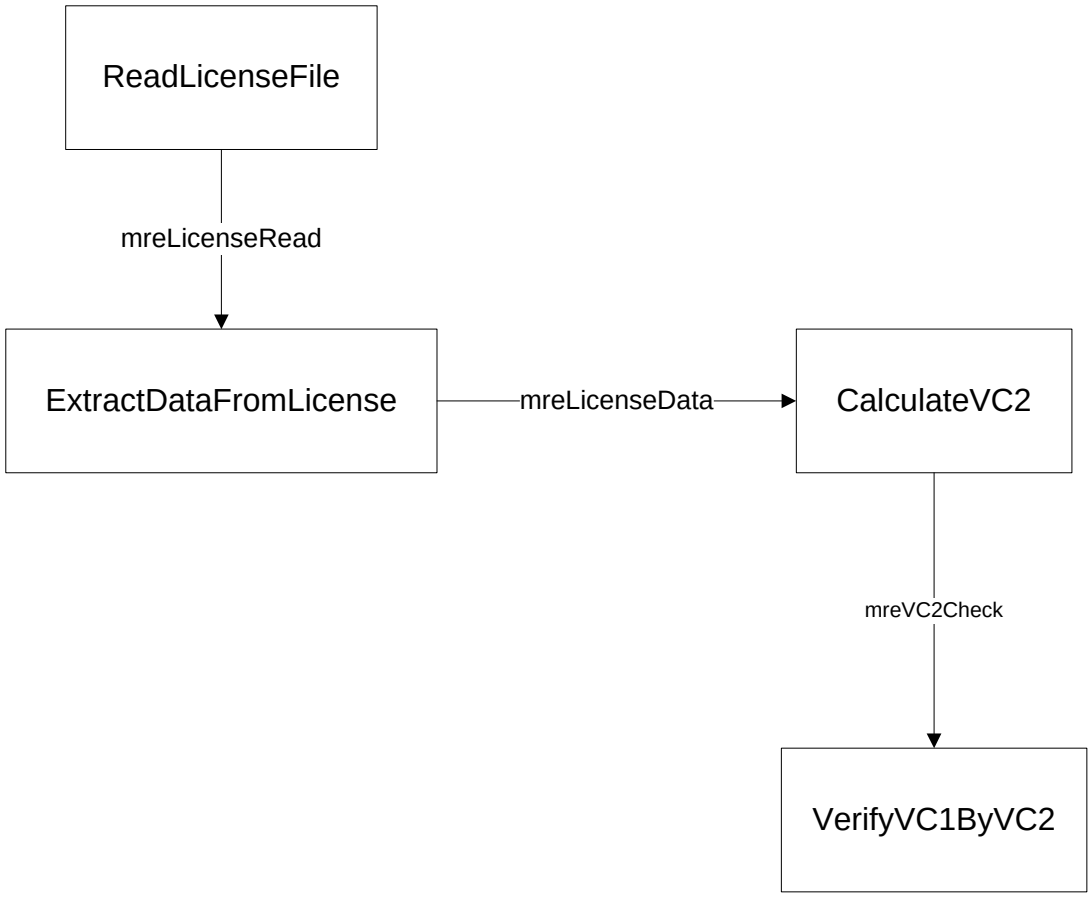

Figure 16. Blocking Diagram of Using 2nd Module to Verify 1st Module

Figure 17 shows a flowchart of utilizing a timer to activate the 3rd verification module, whose result will be compared to that of the secondary module. If a difference is detected, overall verification is deemed failed, GUI will be disabled and handlers will be deregistered, and the program will terminate.

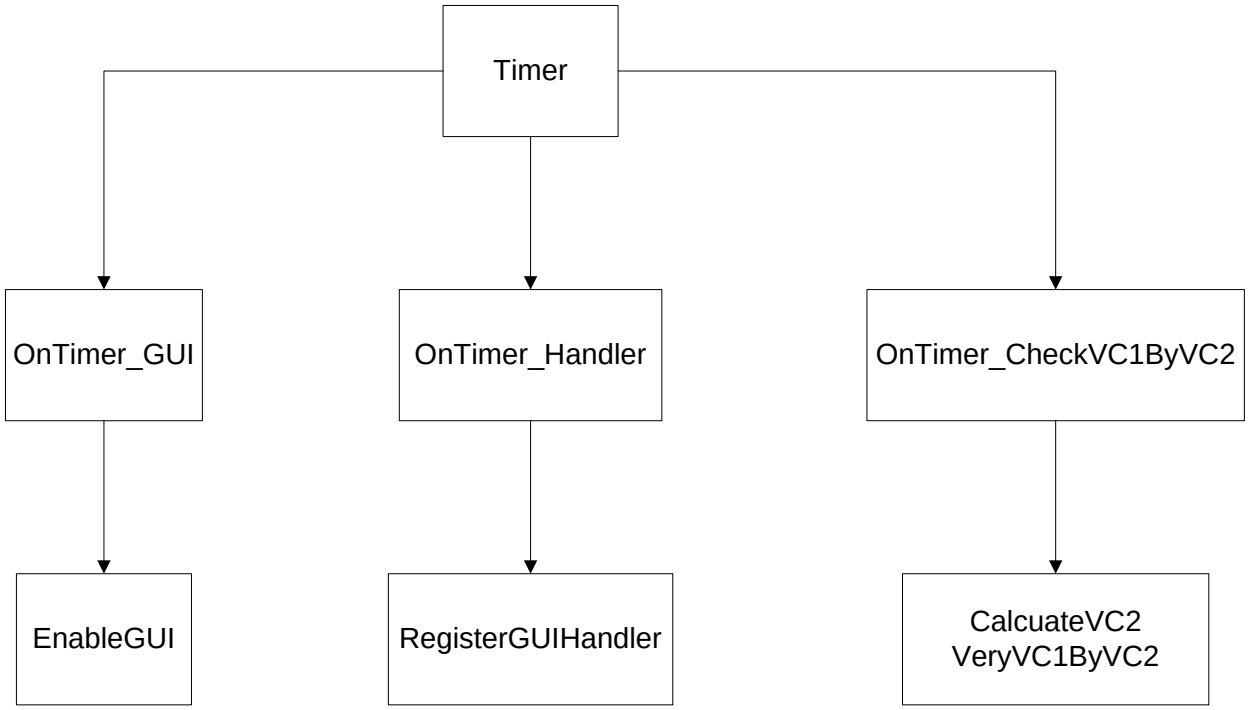

Figure 17. A 3rd Module Checking 2nd Module Periodically

7.3 Test Setup and Metric

A demo program will be written in C#, then convert to native x86 binary using XenoCode's PostBuild, without any obfuscation applied. Microsoft Visual Studio (MSVS)'s built-in debugger will be used alongside a source code to set expectations; this will not be the real world scenario. Tests will be repeated using OllyDbg and IDA Pro. These tests will be the main testing. A d program can be set to run in a specific mode (single threaded versus multi-threaded), and a number of junk threads can be specified.

Tests will be divided into 3 parts. Part one will be the correctness of implementation. Tests in part one will include testing for correct thread count, as well as the correct behavior of some functions. Part two will be comparing a single threaded version against a multi-threaded version. Testing in part two will determine whether using multiple threads for checking has advantages over a single threaded version. Part three will examine whether junk threads will make attacking more difficult.

In our testing, we will use number of lines of disassembly that can be stepped through as the main metric. In a single threaded version, one should be able to step through all relevant code in order to analyze it, whereas in a multi-threaded version we expect only some of the code can be traced. If an attacker cannot trace and analyze all the relevant code, there is little chance the attacker can successfully break the software's security.

Also, extra effort needed to implement the multi-threaded version will be considered and compared to the single threaded version.

Finally, XenoCode's obfuscator will be simply evaluated, by comparing how much of the disassembled code are different.

7.4 Testing and Results

7.4.1 Correctness of Implementation

Here, we will examine correctness of implementation of a few key features.

First, we will examine the license file. When a correct password is entered, the program stays stable; when a wrong password is entered, the program terminated after 10 seconds, as designed.

Second, we will look at the GUI and GUI handler. As show in Figure 18 below, menu "C1" is enabled after the program is launched with the correct password.

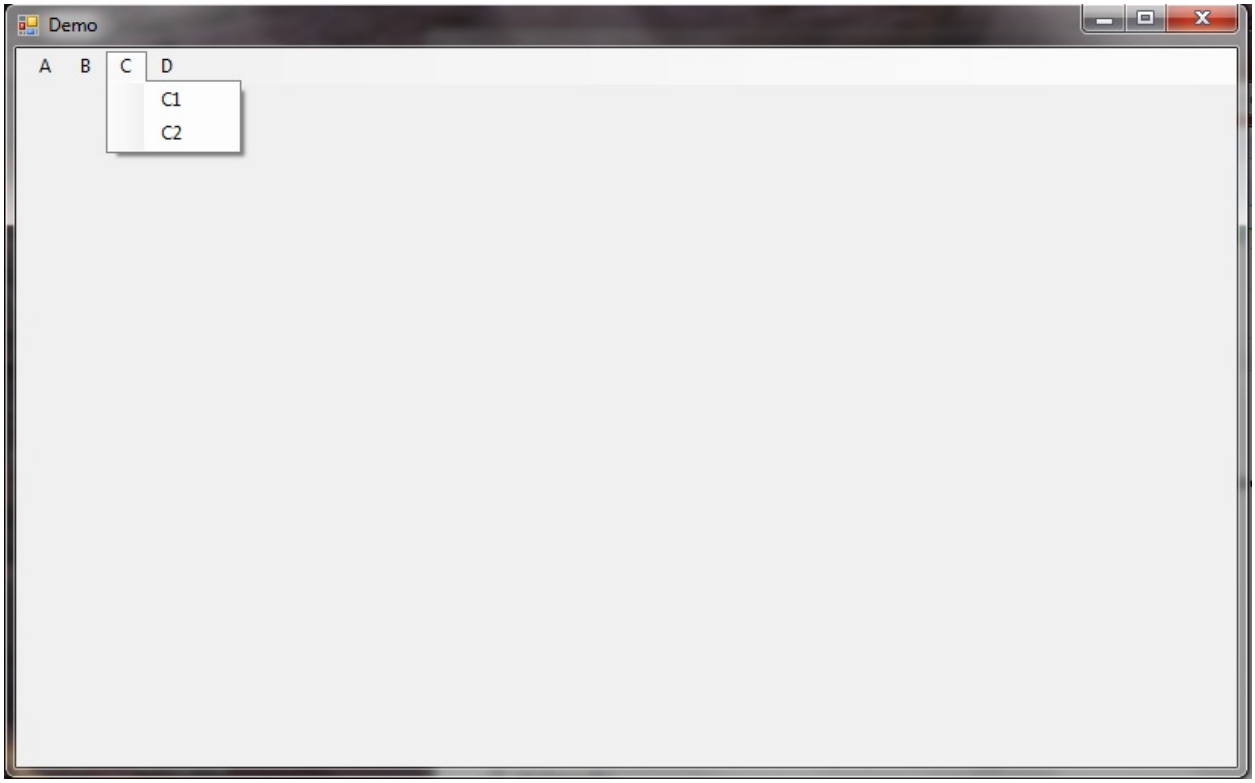

Figure 18. Menu "C1" is enabled

After menu "C1" is clicked, result is correctly displayed as shown in Figure 19. The displayed string is encrypted then decrypted using the AES128 algorithm, as described in Section 7.2.6

above. When the string is correctly displayed, Idle event handler must be functioning correctly since it is s responsible for updating the encryption and decryption keys.

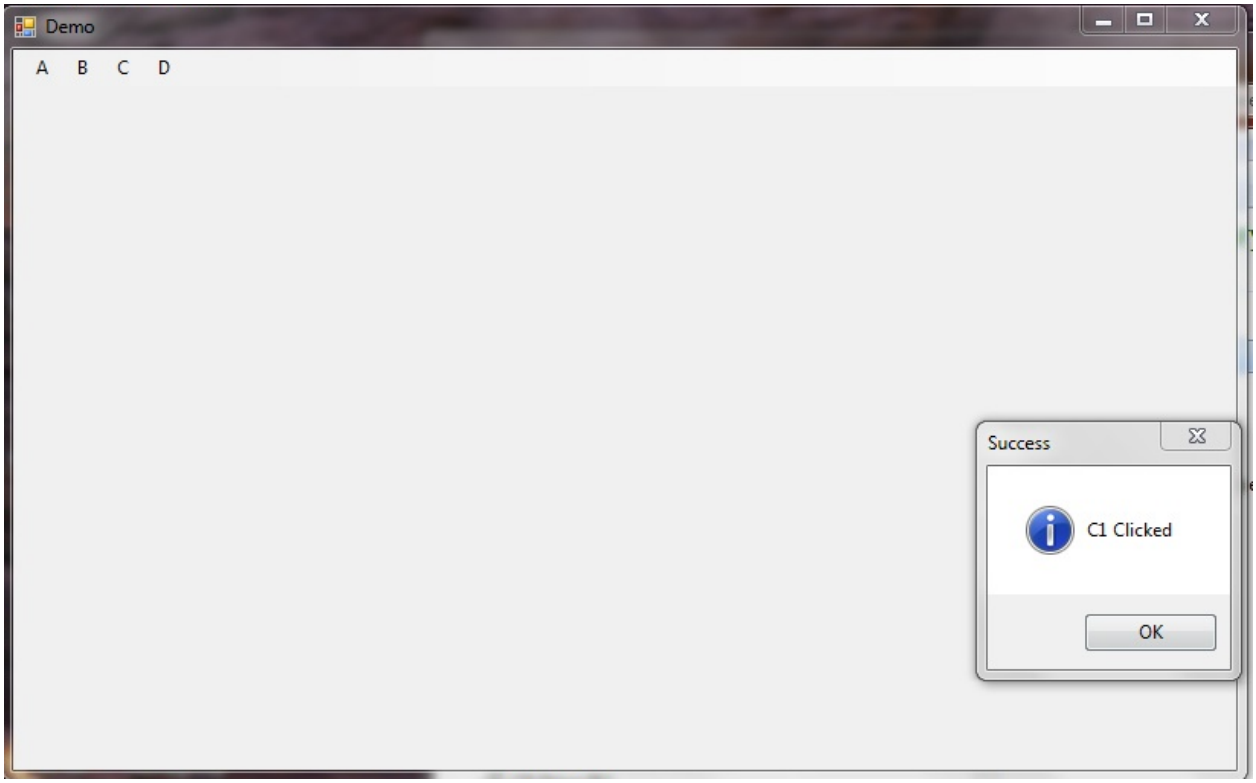

Figure 19. Menu “C1” Clicked

Lastly, we will examine correctness of threads. Table 3 below summarizes results of different test runs.

Table 3. Demo Program's Thread Count in Various Running Modes

Thread Mode	Number of Junk Threads	Observation
Single threaded	N/A	Program runs on 9 threads minimum (GC + GUI + Timers + asynchronous event firing, and so on). Max was 12 as reported by WTM.
Multi threaded	0	WTM reported a max of 12 threads running at the same time. Thread count gradually falls to 9 according WTM, which is the similar to single threaded mode. This makes sense too since when checking is done, most extra threads are terminated. Theoretically, the program should launch 10 individual threads, but it appears that they do not all run at the same time.
Multithreaded	5	WTM reported a max of 17 threads running at the same time; it falls to 14 after a while. Total count is 17 because of 5 junk threads.
Multi threaded	10	WTM reported a max of 22 threads running at the same time; it falls to 19 after a while. Total count is 22 because of 10 junk threads.
Multithreaded	15	WTM reported a max of 27 threads running at the same time; it falls to 24 after a while. Total count is 27 because of 15 junk threads.

The thread counts in the Table 3 are consistent, assuming 9 threads are needed to run the application on average after checking is completed. Running code in debuggers has the same count as running it without debuggers; therefore, implementation of threading is correct.

7.4.2 Testing in Development Environment

The demo was tested with Microsoft Visual Studio. The tests shown in Table 4 are done with MSVS's debugger with source code. The reason for using this testing environment is so that we can set breakpoints correctly and track which function is being executed. In other words, this is for the purpose of convenience and to set our expectation when debugging in other environments; without such convenience, debugging can only be substantially more difficult (this should be the best testing scenario possible). Table 4 summarizes testing results in various scenarios.

Table 4. Observation of Various Testing Scenarios Using MSVS's Debugger

Test Case Number	Observation
1	Single Threaded, no junk thread, break on all relevant functions. Unable to proceed to other functions because Idle function runs continuously and this is the function captured debugger's attention all the time. GUI is launched, but unable to interactive with it because Idle is constantly running.
2	Single Threaded, no junk thread, break on all relevant functions except Idle. Without Idle interfering, GUI is launched, and can be interacted with normally. Checking is done sequentially in the right order as specified. All parts of code can be traced. A frequent timer event can severely disrupt debugger process, as relevant functions run all the time. All handlers of timer event can be debugged, as long as breakpoints are set for them. Present of timer didn't affect debugging code relevant checking functions, because they do not take effect until initial checking is done.
3	Single Threaded, no junk thread, break on all relevant function, but Idle added in later. As long as the first breakpoint for checking is reached before setting the breakpoint of Idle, checking can be traced as in case 2 above, but timer events cannot be trace due to constant running of Idle.
4	Multi Threaded, 0 junk threads, break on all relevant functions. Observation is identical to case 1 above, which matches expectation
5	Multi Threaded, 0 junk threads, break on all relevant functions except Idle. The first function can be partially traced, a few other functions can be traced partially at random (indeterministic about how much of a function can be traced). Timer events can interfere with normal tracing
6	Multi Threaded, 0 junk threads, break on all relevant function, but Idle added in later. Result similar to case 5, except Idle will disrupt code execution more severely compared to case 5 above. In essence, checking is executed interleavingly with Idle.
7	Multi Threaded, 2 junk threads (minimum needed to create deadlock), break on all relevant functions. Deadlock situation is successfully created. After deadlock, only Idle can be traced, no checking can be traced.
8	Multi Threaded, 2 junk threads, break on all relevant functions except Idle. Deadlock situation created, but checking can be traced like in case 5 above.
9	Multi Threaded, 2 junk threads, break on all relevant function, but Idle added in later. If added too soon, then like case 7 above; if added late enough, then like case 6 above.

Additional tests were performed with more junk threads (with numbers being 5, 10, and 15) in the same setup as test case 7, 8, and 9 in Table 4, and the same results were obtained correspondingly. In this case, more threads being deadlocked added no extra benefits. In fact, the presence of a deadlock added no more difficulty compared to just being multithreaded in this particular testing environment. This is due to the fact that Microsoft's debugger (with source code) can smartly execute code in an interleaving manner, allowing the execution to change from one thread to another, although it is out of the user's control which thread is executed and when.

Single breakpoint was also tried out in testing. In multithreaded case, it is definitely worse than setting breakpoints on all relevant functions (functions cannot be traced without setting breakpoints at them in this case). In a single threaded case, depending on where the single breakpoint is set, it is possible to trace all code relevant to checking.

With MSVS in single threaded mode, line counts are the same for setting a breakpoint at only the start and at all functions except Idle; but if a breakpoint was set at Idle, line count dropped significantly, see Figure 20 below. The reason is an Idle event was issued many times by the OS to the application, hence triggering the Idle event handler to run many times.

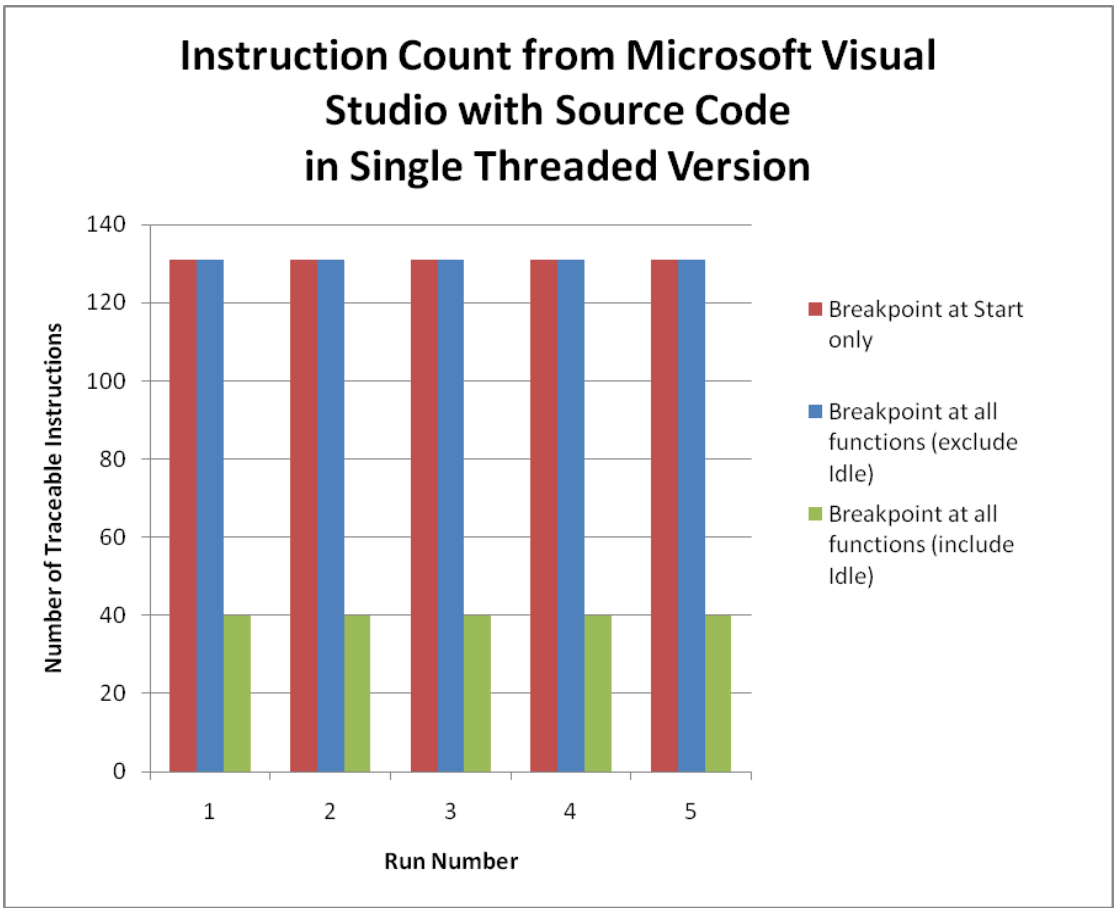

Figure 20. Line Counts of Different Runs from MSVS in Single Threaded Mode

When running the same program in multi-threaded mode, the line counts stay the same across all runs at 40 and 30, for setting breakpoints at all functions including Idle and at start only respectively. When breakpoints are set at all functions excluding Idle, line counts varies significantly across runs, ranging from 40 to over 140, with an average being 73.35, see Figure 21.

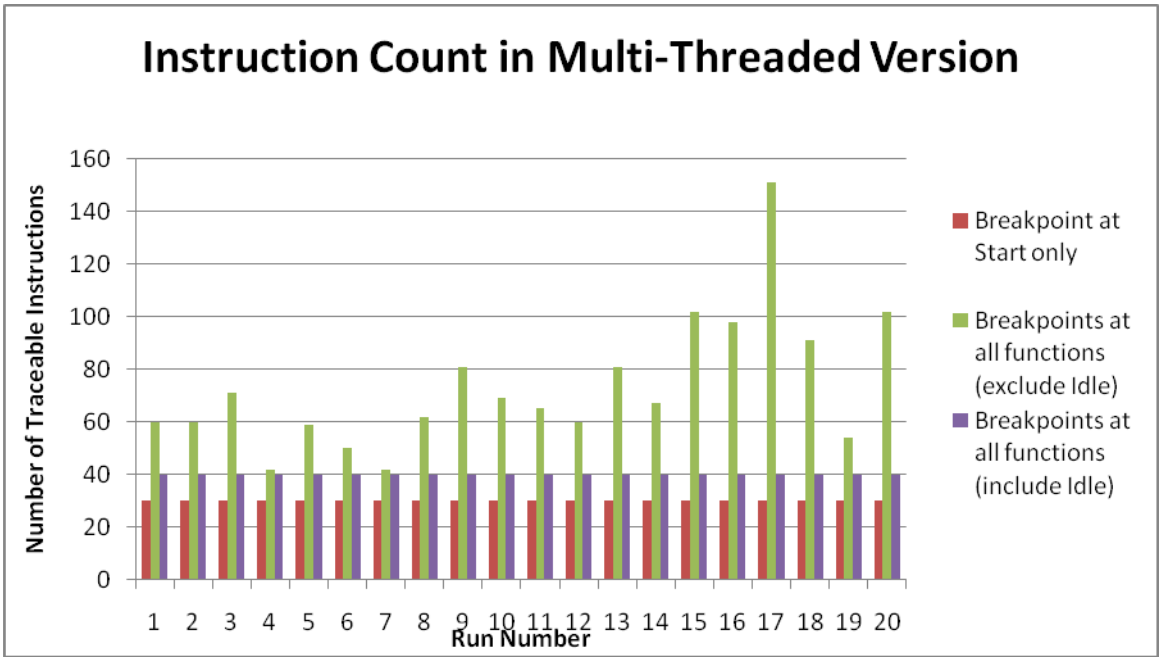

Figure 21. Line Counts of Different Runs from MSVS in Multi-Threaded Mode

When a breakpoint is set at the start and additional threads (junk threads) are introduced, line counts stay the same across several runs, suggesting the number of junk threads does not seem to matter, see Figure 22.

When junk threads are used and breakpoints set at all functions except Idle, line counts vary significant. Here we distinguish between useful lines (lines of code of threads doing useful work) and junk lines (lines of code from junk threads doing nothing useful). Numbers of junk threads tested were 2, 5, 10, 15, 20, and 25. As shown in Figure 23 to 28, when number of junk threads increases, lines of junk code increase and lines of useful code decreases overall.

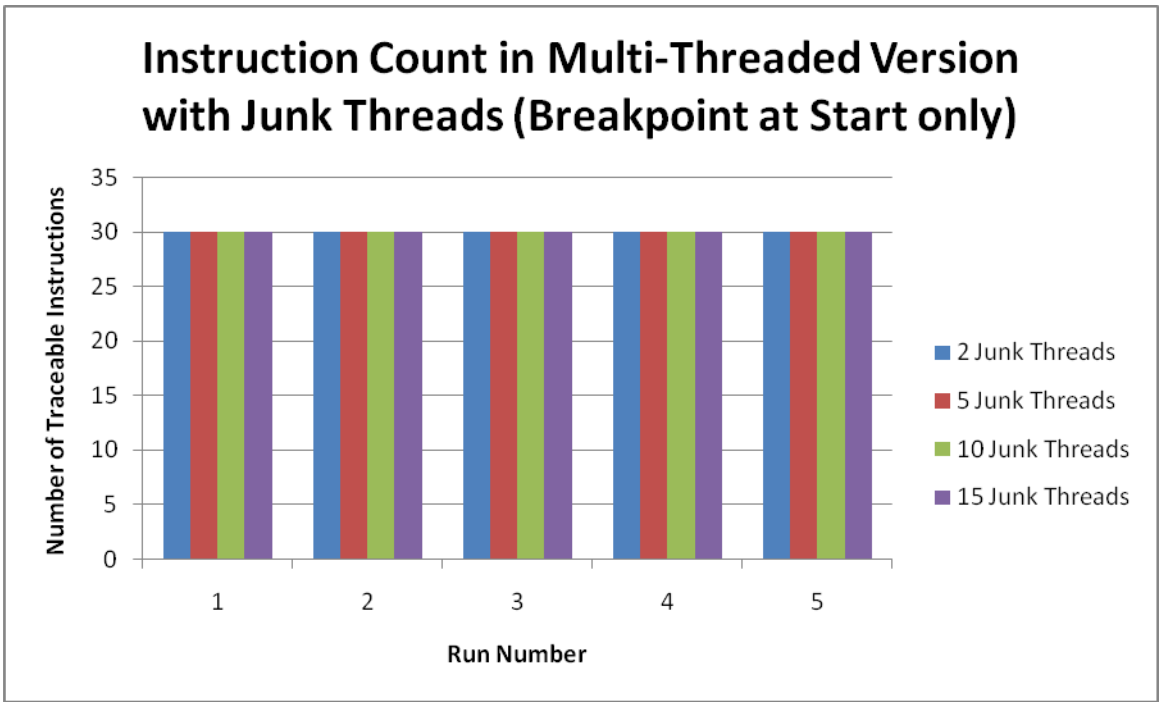

Figure 22. Line Counts of Different Runs from MSVS in Multi-Threaded Mode with Junk Threads used

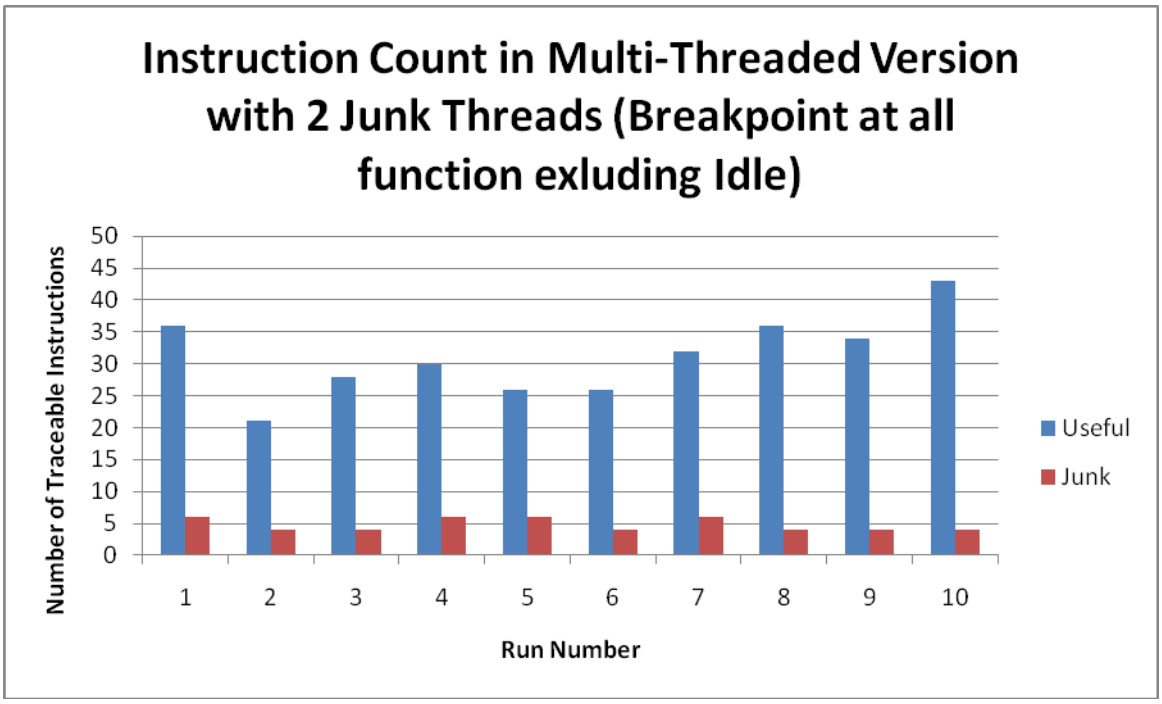

Figure 23. Instruction Counts of Useful and Junk with 2 Junk Threads

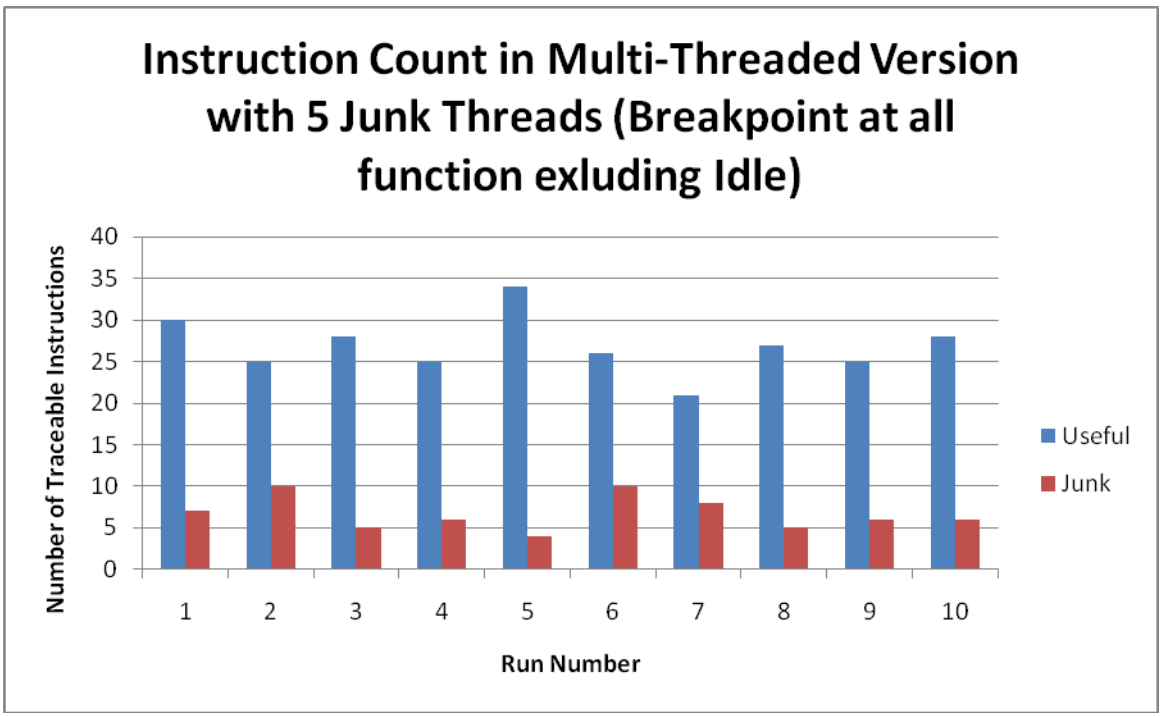

Figure 24. Instruction Counts of Useful and Junk with 5 Junk Threads

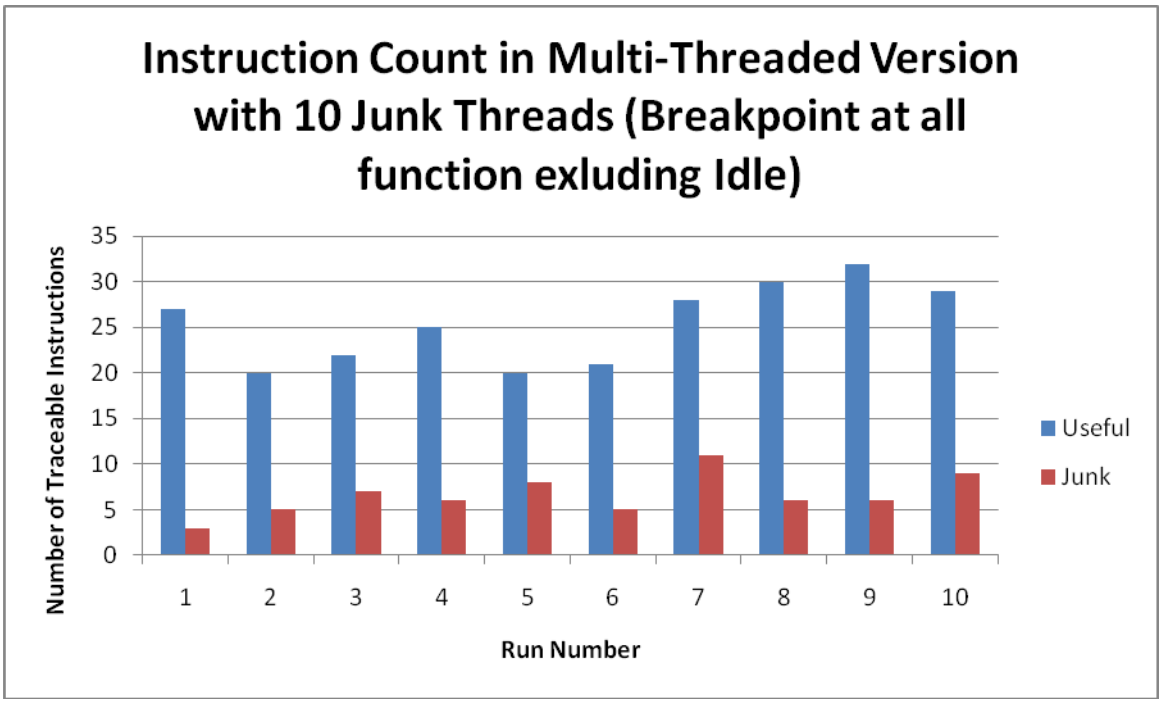

Figure 25. Instruction Counts of Useful and Junk with 10 Junk Threads

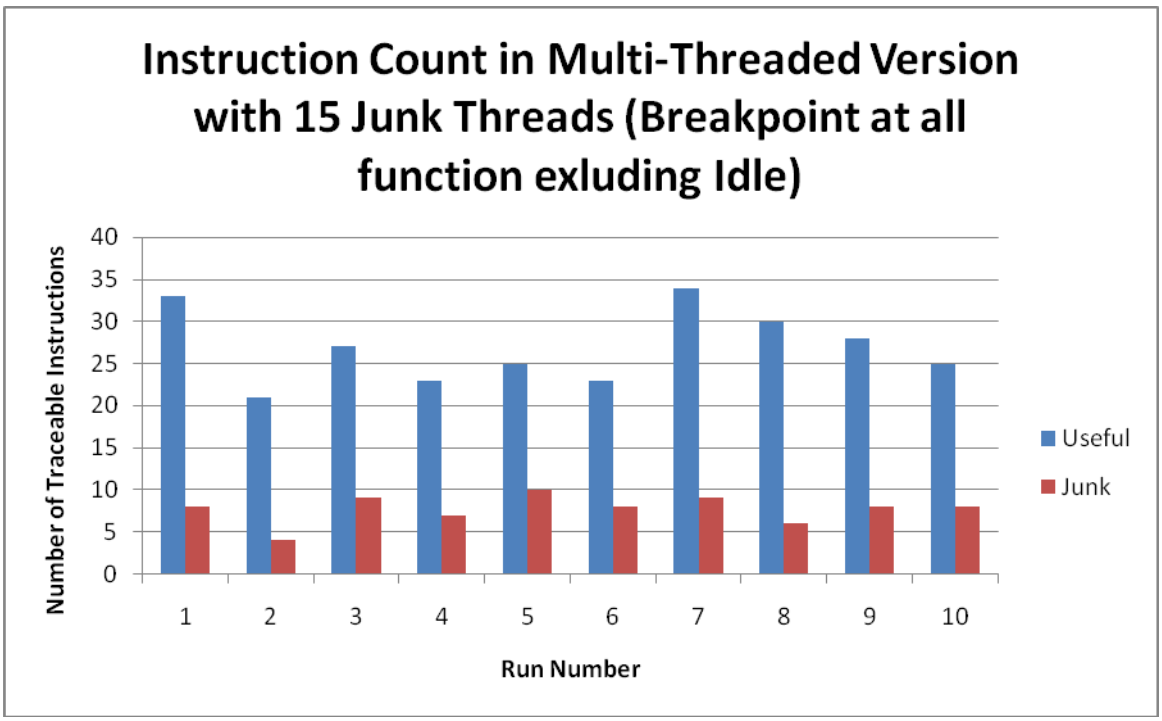

Figure 26. Instruction Counts of Useful and Junk with 15 Junk Threads

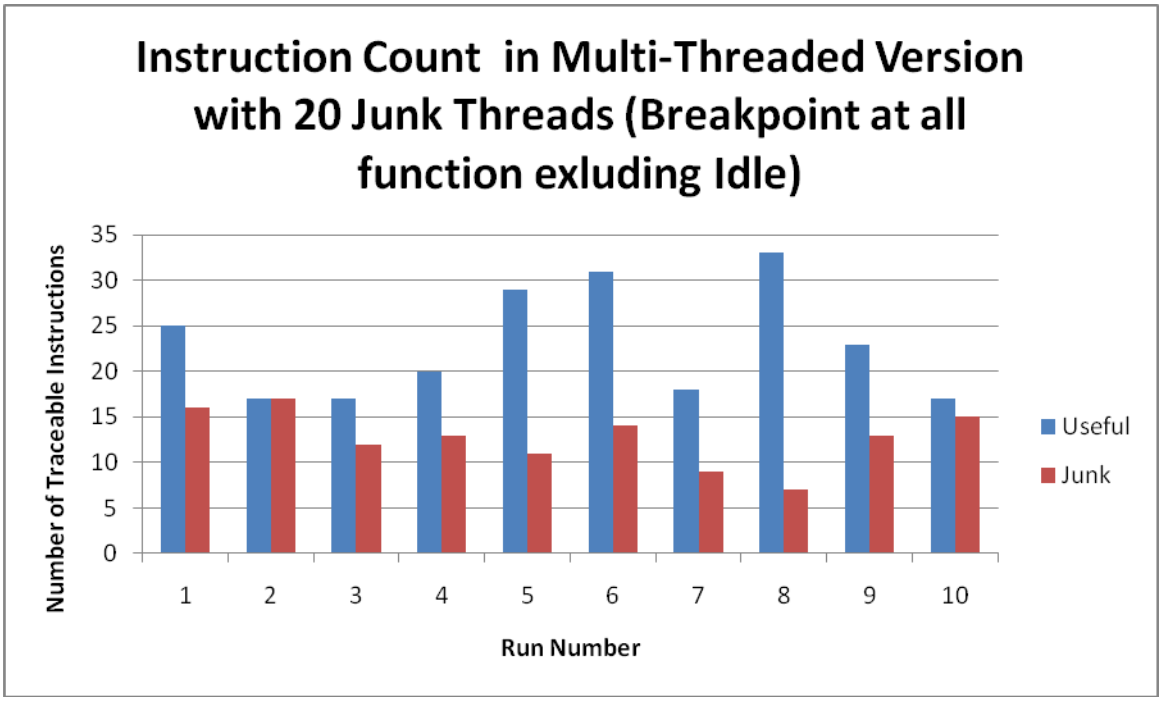

Figure 27. Instruction Counts of Useful and Junk with 20 Junk Threads

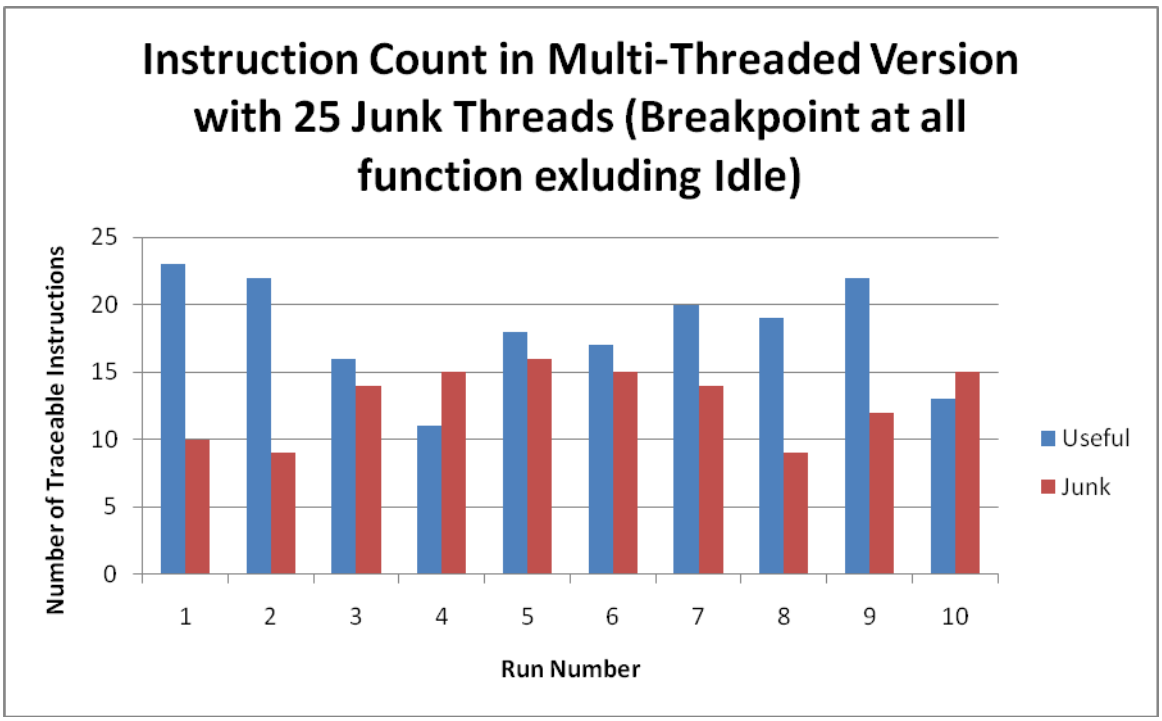

Figure 28. Instruction Counts of Useful and Junk with 25 Junk Threads

The average of each scenario show in Figures 23 through 28 is also plotted on the same graph, shown in Figure 29 below. According to our test, when using multi-threading mode without junk threads, an average line count is about 73, compared to 113 in the single threaded mode. When junk threads are used, line counts for useful lines drop significantly even if only 2 junk threads were used. As more junk threads are used, useful line counts drop even more. In contrast, junk line counts increase steadily at a slower pace as more junk threads are introduced.

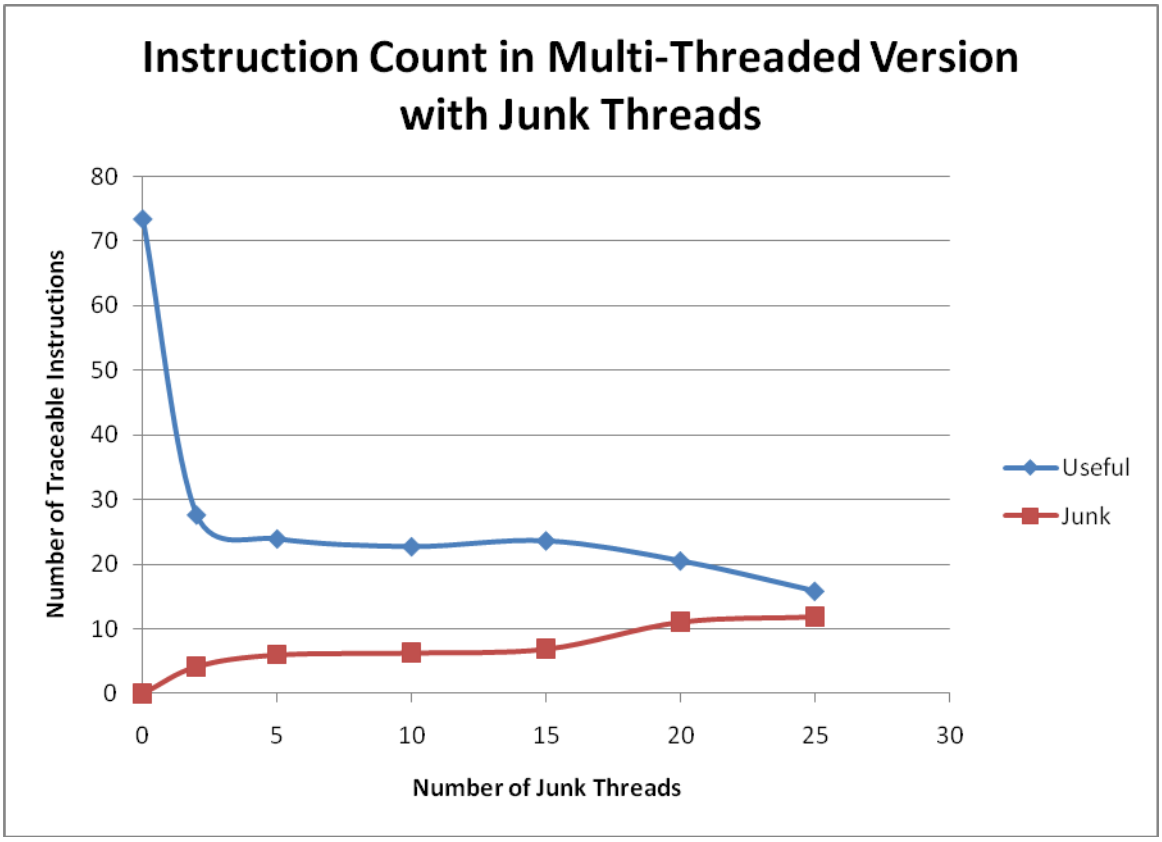

Figure 29. Average Line Counts of Useful and Junk Instructions When a Different Number of Junk Threads are introduced

Figure 30 demonstrates a percentage count for the average number of traceable, useful instruction. As shown, only about 20% to 30% of useful instructions can be traced when junk threads are used, as opposed to about 75% when none are used. When 25 junk threads are used, traceable useful code drops to about 14%. From an attacker's perspective, the lower the percentage, the less useful code he can trace, which in turn means more difficult for the attacker to understand the code when it comes to reverse engineering.

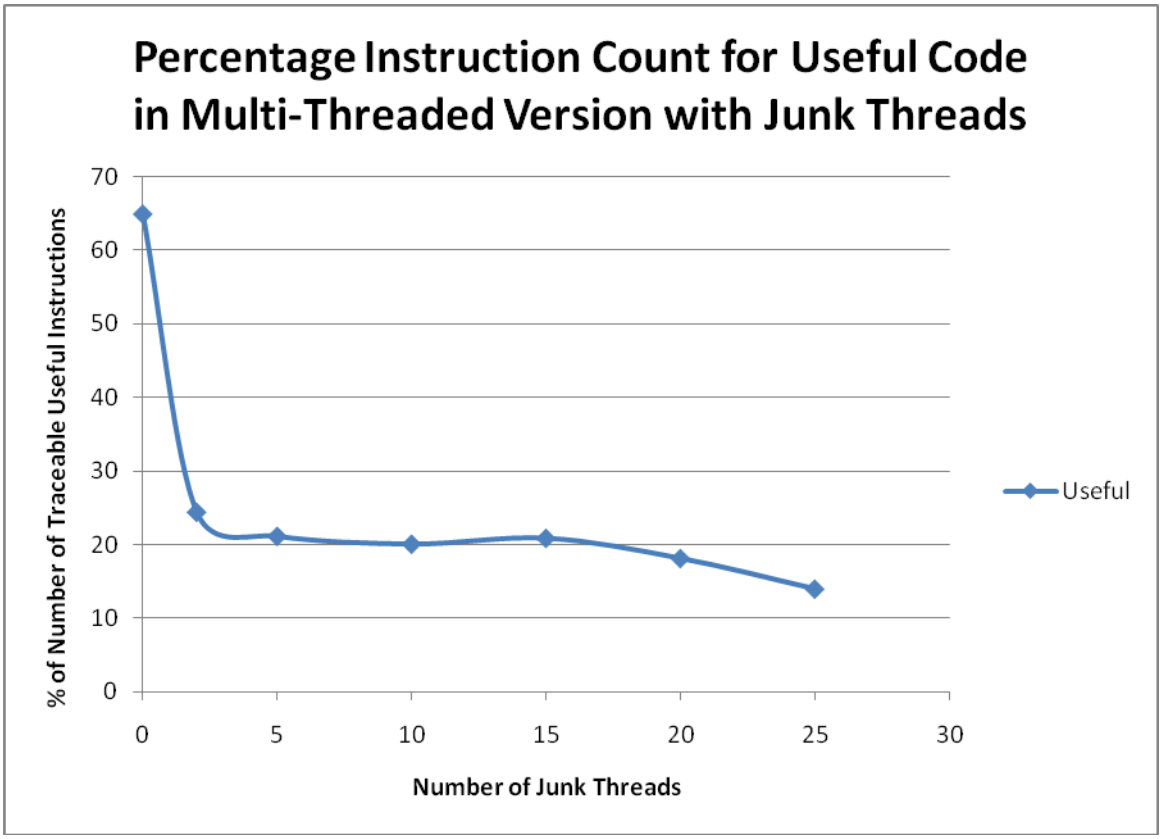

Figure 30. Percentage of Average Number of Traceable Useful Instructions

7.4.3 Single Threaded vs. Multi-Threaded

In this section, we will perform tests using both OllyDbg [30] and IDA Pro [29].

First, tests were run with OllyDbg using the same pattern as with MSVS, with the exception that breakpoints were not set in the same way. Also, a different counting scheme is used. All of instruction counts were based on the calculation of addresses in blocks selected as relevant. Some of the codes were included in counts not executed by the debugger. They were only considered a rough estimate. Counts are likely to include a large number of instructions that are not relevant to checking; but rather, they are part of windows API libraries, such as the code executed initially to start the program or GUI libraries. High number of line counts is due to the inability to clearly identify relevant code correctly from disassembly. Because of the inability to identify code, no breakpoint is set in testing. An average result from the single threaded case will be called “total”..When counting an instruction in a multi-threaded mode, a different approach is used. We will try to identify code that cannot be traced based on the thread table provided by OllyDbg, and subtract them away from the “total”; the resulting number will be regarded as the count for that particular test run. In theory, this number also represents the maximum amount of code an attacker can trace.

In single threaded mode, it seems like code can run normally; therefore, it should be theoretically possible to trace the code execution as long as breakpoints are smartly and properly set after correctly identifying relevant code sections in disassembly. Even though checking is done in a single thread, system still has other threads running in the background, such as the Idle event. Figure 31 below shows the instruction counts (in terms of bytes) across 5 runs. Result shows that all 5 runs give the same count, which is consistent with our assumption above.

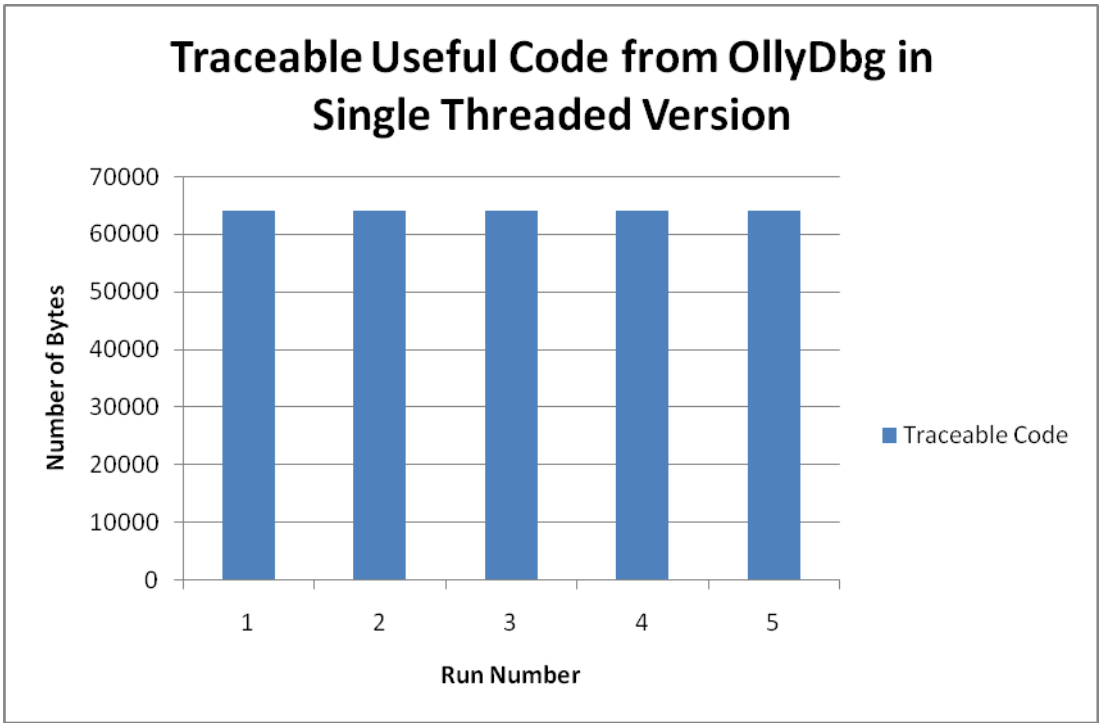

Figure 31. Numbers of Traceable Instructions from OllyDbg in Single Threaded Mode

In a multithread mode, things get much more complicated:

- OllyDbg seems to capture the first available thread and executes that one in the foreground (making it available to step through). In this case, it appears to be always the same thread in our tests. Also, it appears like the thread captured by debugger is the runtime's GUI thread, which launches other checking threads. Once checking threads are launched, this captured thread pauses. Depending on how fast we step through this captured GUI thread, we may or may not see other threads because they can finish. In cases where we can jump into other threads, we cannot tell which checking thread (or even the Idle thread) we jump into.

- Repeated runs yield different results in our test runs. This can get even worse if we randomize the start order of checking threads.

- From running code in OllyDbg in multithreaded mode, we were unable to (or cannot easily) determine relationships among various threads (such as which depends on which).
- Setting breakpoints is an extremely difficult task in multithreaded mode, because one thread may block another. If we want to trace one thread, we must set a breakpoint for it. But if that thread runs in the background and it blocks the one running in the foreground (the one we are currently stepping through), then we will be in a deadlock like situation since the foreground thread cannot proceed until the blocking thread finishes, which it cannot because of the breakpoint. If breakpoints are not properly used in a particular run, we cannot even bring up the GUI of the program (which happened quite often as we cannot set the breakpoints right). With no breakpoints set, we can get to the GUI of the program.
- While OllyDbg may take us to the code representing the thread (by double click on the thread), it is only possible when the thread hasn't finished execution. This may require one to work very fast.

Base on the results of test runs, as shown in Figure 31 below, it is clear that different execution paths are taken at different runs; therefore, resulting in a different count each time. Due to this fact, it is more difficult for an attacker to reverse engineer from the disassembly because he would get a different view of code each time he tries.

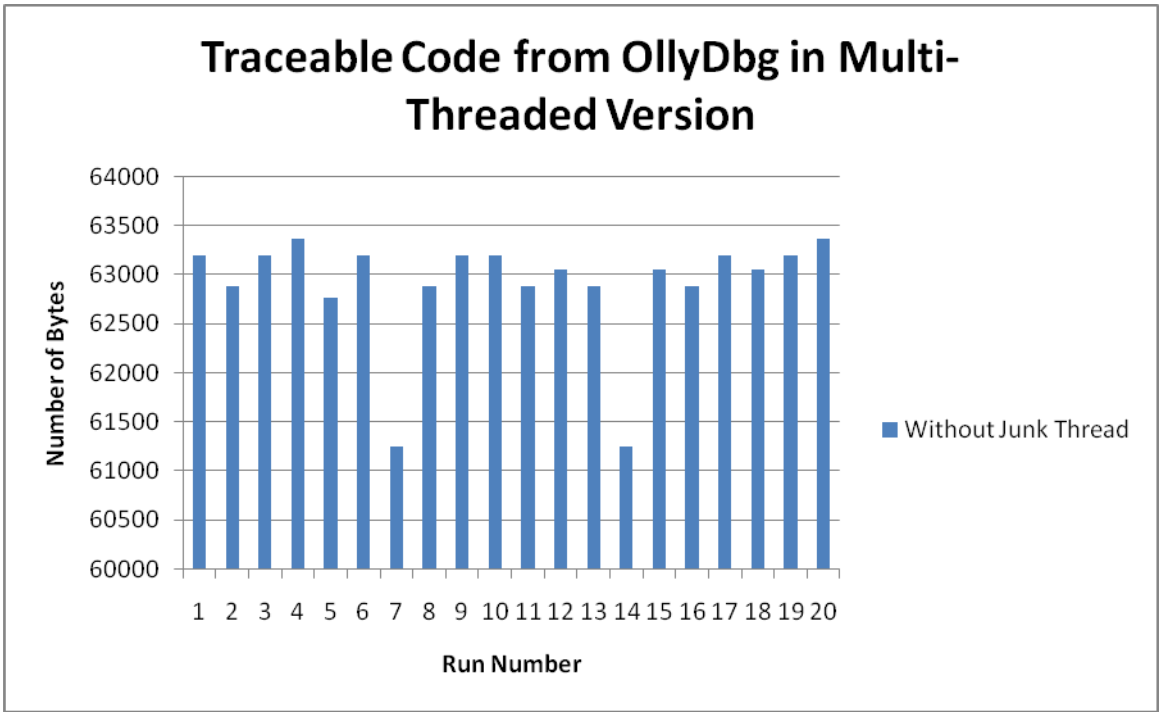

Figure 32. Numbers of Traceable Instructions from OllyDbg in Multi-Threaded Mode

Testing with IDA Pro yielded similar results in single threaded mode as OllyDbg, although result is slightly different from that obtained from OllyDbg. This is due to inability to clearly identify code in the counting process. Overall, code seems to run OK in debugger, meaning it can be effectively traced and analyzed in theory.

In a multithread mode, things get much more complicated (even worse than OllyDbg):

- This time, we cannot even enter the password into the program, since it is launched in another thread. This is very devastating because without it, nothing else will run properly. IDA Pro clearly didn't capture this thread in the foreground. This is going to be the end of it even if other code can run. We didn't notice this before in OllyDbg since it got into a deadlock trap.

- IDA Pro, like OllyDbg seems to capture the first available thread it can and executes that one in the foreground (making it available to step through). In this case, it is appears to be always the same thread in our tests. But this thread it captured seems to run in an endless loop; it is perhaps the message processing thread from the runtime, or the Idle event thread. Even though it is able to show the different threads in a thread window, it cannot jump to any of them, not even to their location in disassembly.

- Again, we cannot determine relationships among various threads running code in IDA Pro in multithreaded mode, just like in OllyDbg, because we cannot step through them.

In the case of IDA Pro, we are not able to obtain a meaningful count of instructions in multi-threaded mode, because we cannot identify code corresponding to different threads.

7.4.4 Multi-Threaded With Use of Junk Threads

Testing with OllyDbg, these observations are obtained with junk threads added on top of other checking threads:

If junk threads are launched before checking threads, we were never able to get the program run correctly, as we got into the deadlock trap. No matter how many junk threads we used, the result was always the time, see Figure 33; therefore, it is unclear whether number of junk threads matter because timing is another important factor. This is likely due to junk threads are launched before the useful checking threads (in a sequential order). When junk threads are launched first, they are the only threads around (in addition to system threads), and OllyDbg seems to capture one such thread (probably because checking threads are not even launched yet) and shows it in the foreground, and then wait indefinitely. In this case, maybe 2 junks are sufficient for our purpose.

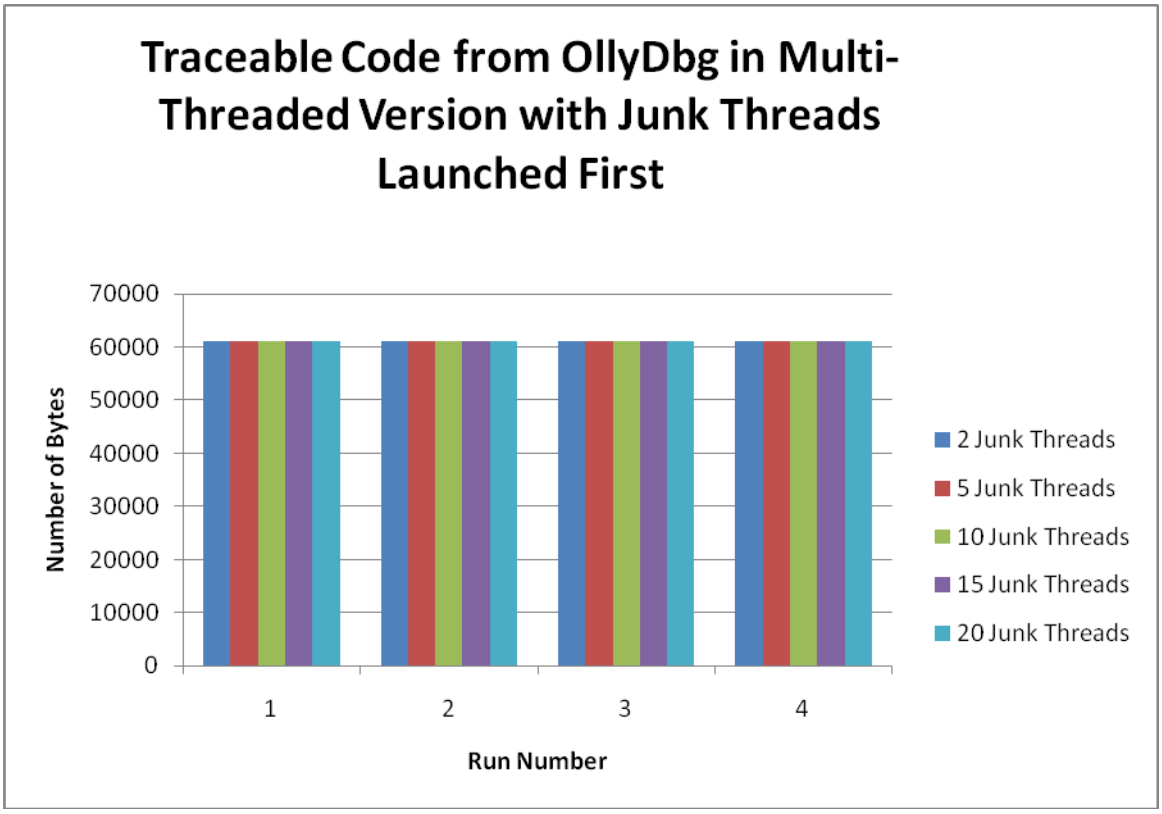

Figure 33. Numbers of Traceable Instructions from OllyDbg in Multi-Threaded Mode

If junk threads are launched after checking threads, the situation becomes more or less like the regular multithreaded case discussed earlier in Section 7.4.3, except it is deadlock causing trouble instead of breakpoints (or actually can be both of them at the same time if breakpoints are set). Figures 34 through 39 show the results of using different number of junk threads; they definitely reduce attacker's chance of getting into the right places compared to just multi-threading with no junk threads.

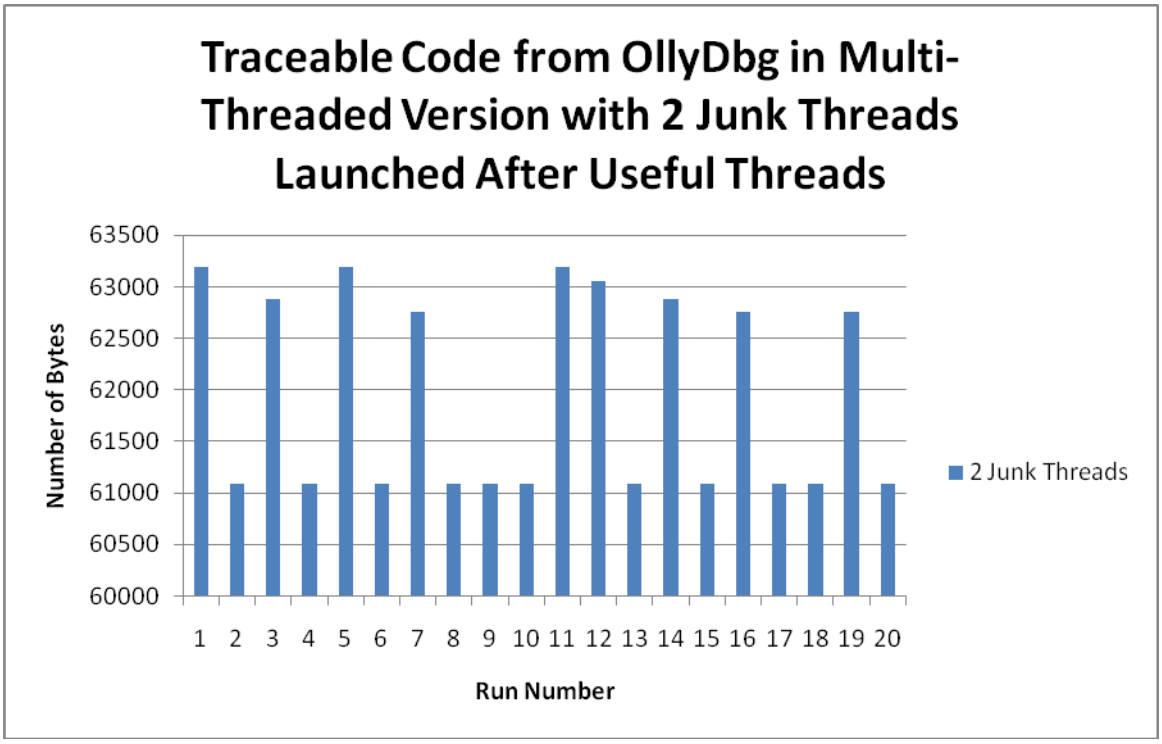

Figure 34. Instruction Counts of Useful and Junk with 2 Junk Threads

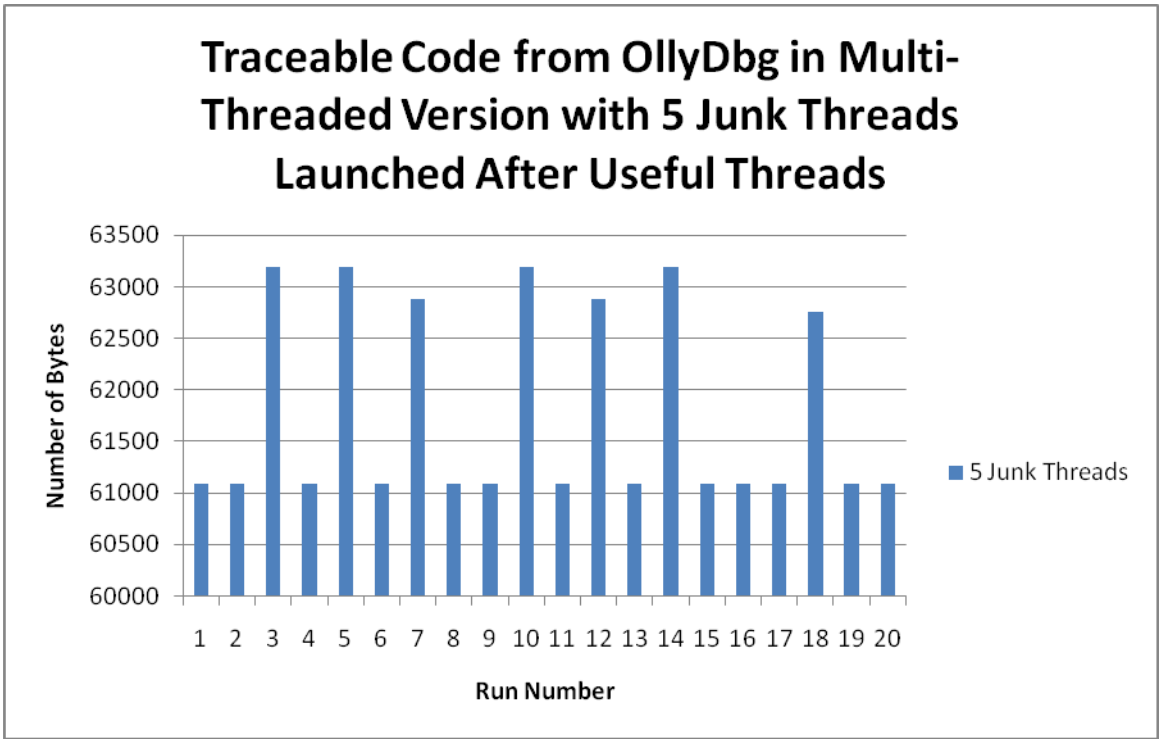

Figure 35. Instruction Counts of Useful and Junk with 5 Junk Threads

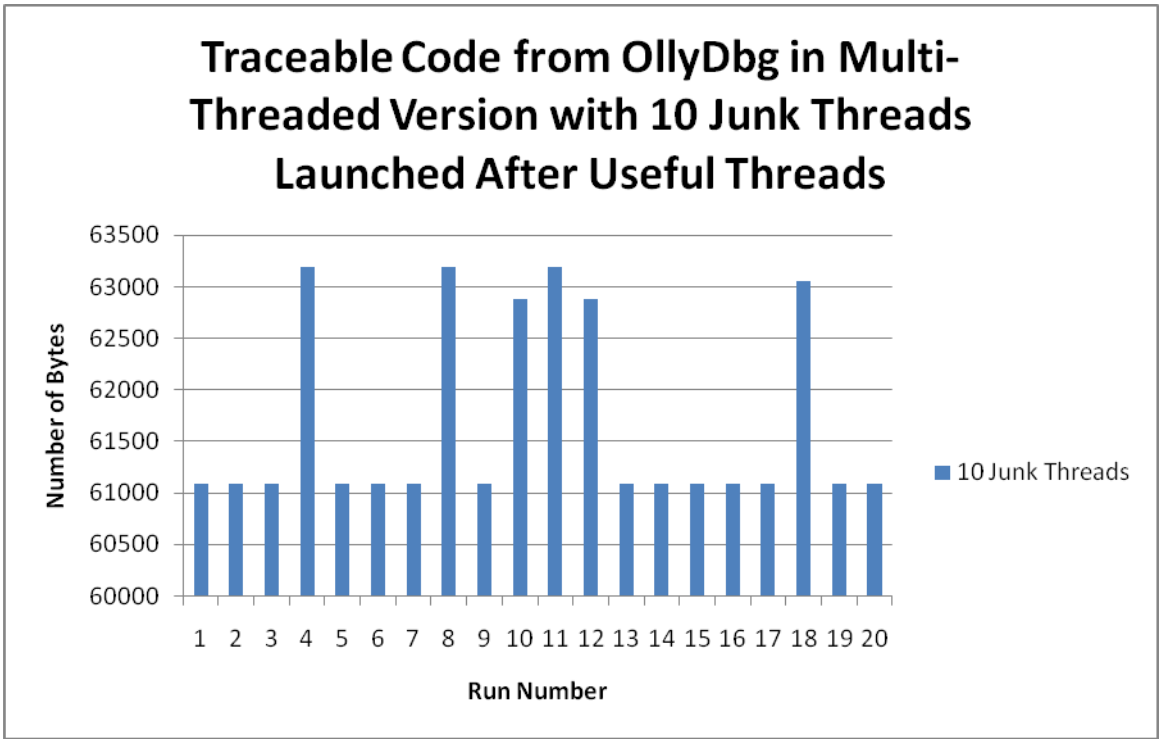

Figure 36. Instruction Counts of Useful and Junk with 10 Junk Threads

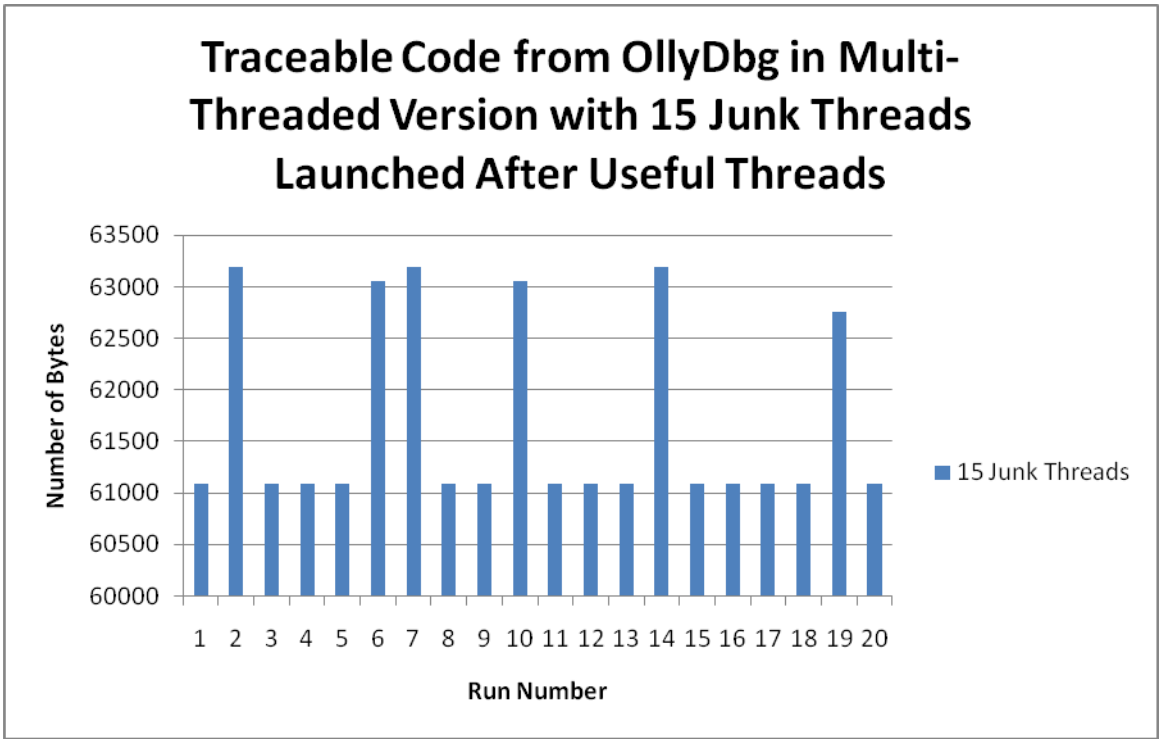

Figure 37. Instruction Counts of Useful and Junk with 15 Junk Threads

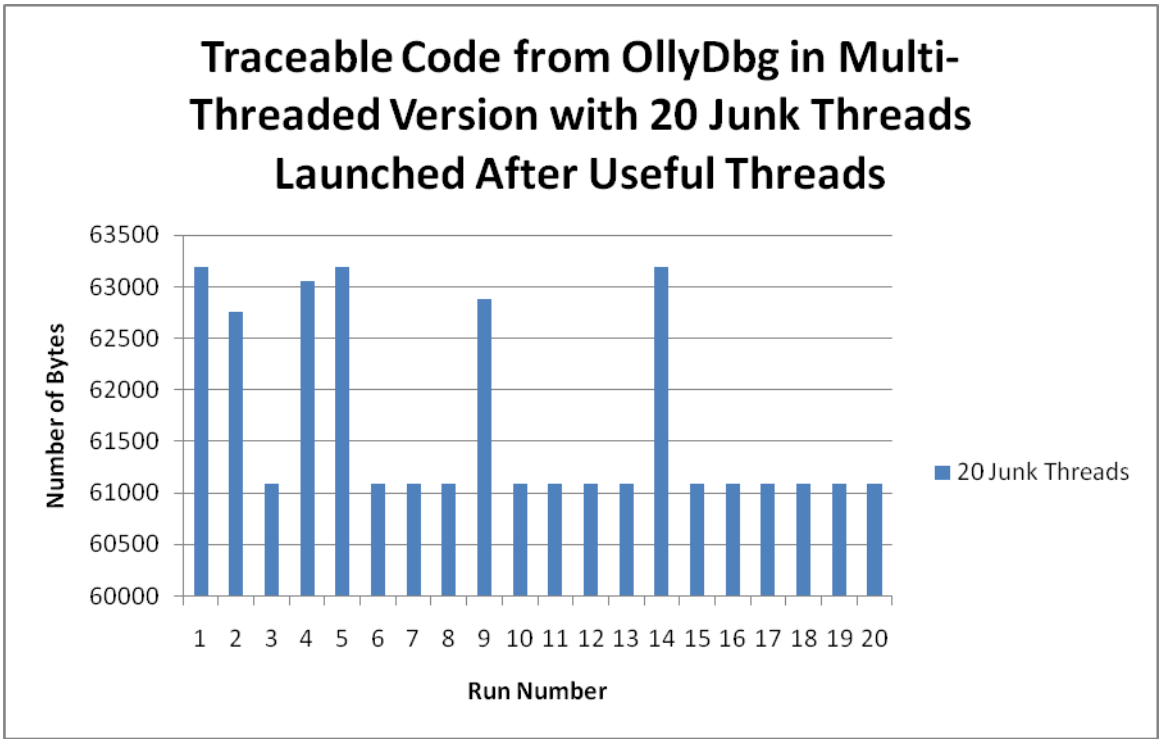

Figure 38. Instruction Counts of Useful and Junk with 20 Junk Threads

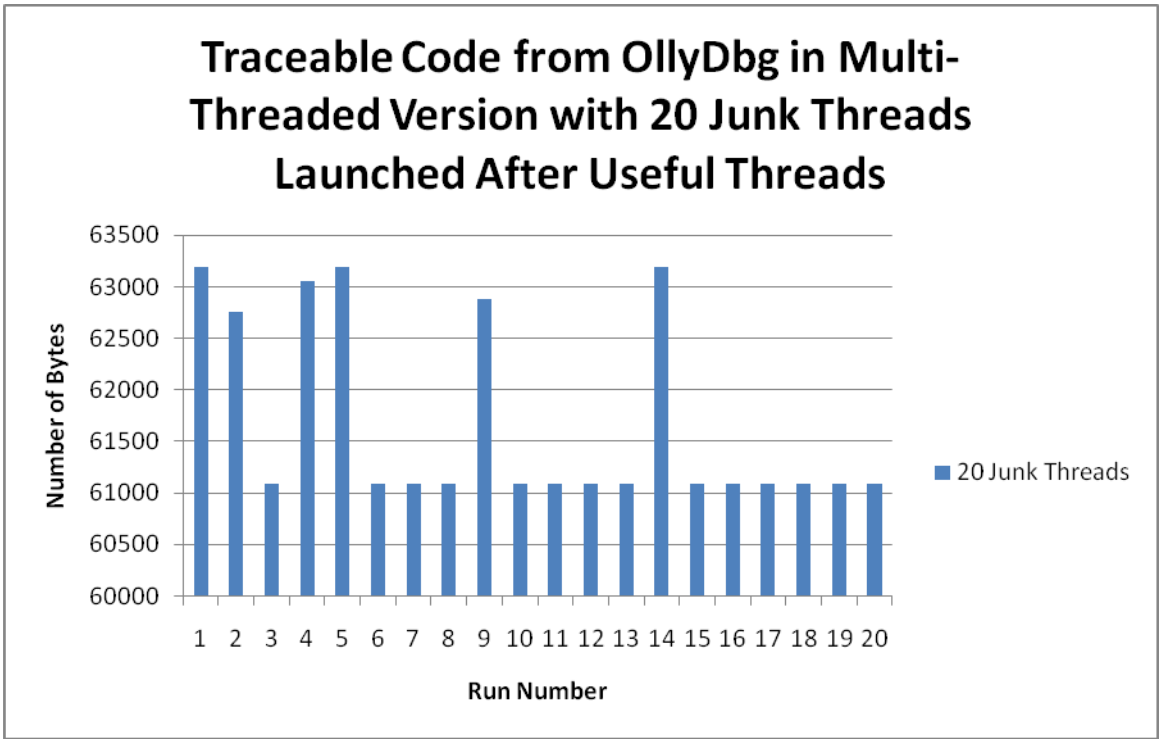

Figure 39. Instruction Counts of Useful and Junk with 25 Junk Threads

Assuming each thread, either junk or useful, has equal chance of being captured by OllyDbg and put to foreground, more junk thread should work to our benefits statistically in theory. While result indicated that one can get into deadlock with higher chance using more junk threads, it does not completely agree with probability provided by statistics, see Figure 40. In this case, timing matters too in addition to number. Timing comes in many factors, including how fast we step through code, when OS really launches and executes a thread, and so on. In short, simple testing result on this can be generalized as the more junk threads the better.

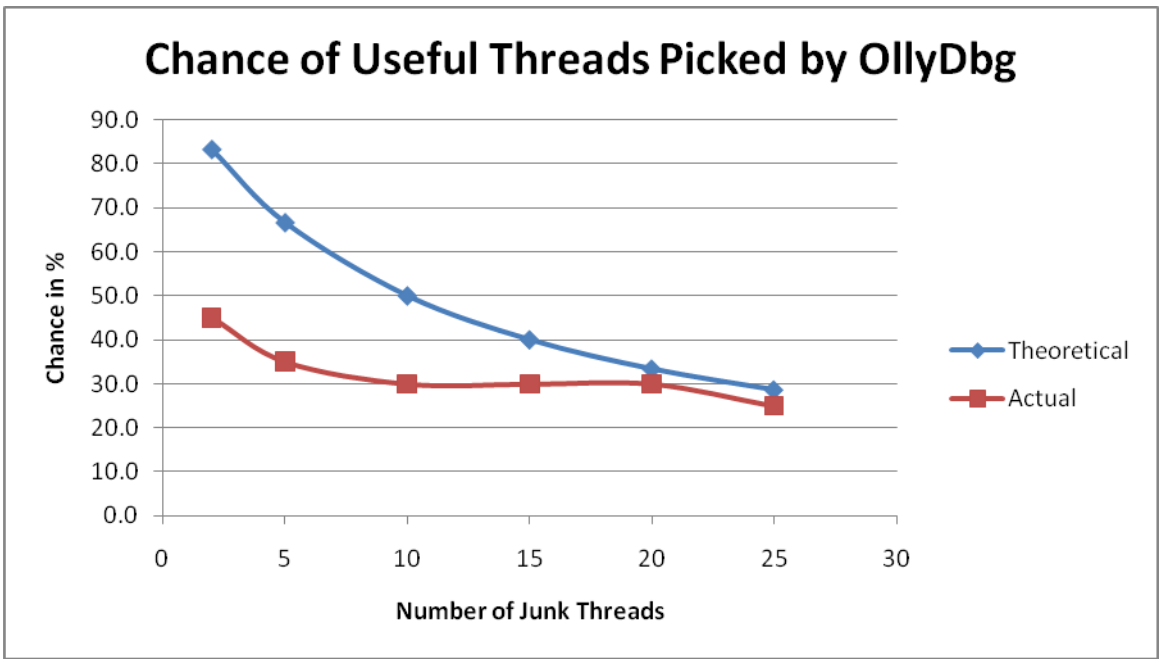

Figure 40. Chance of Useful Threads Picked Out by OllyDbg When Junk Threads Used

We are unable to repeat tests with IDA Pro, because we cannot identify code corresponding to threads. Because of this, we would tend to say from an attacker's prospective OllyDbg appears better than IDA Pro for purpose of reversing code.

7.4.5 Effort Needed to Implement Multi-threaded Version

To implement this new design using multiple threads, extra effort is needed. Extra efforts are summarized in Table 5 below:

Table 5. Extra Efforts Needed to Implement New Design

Work	Extra Effort
Dividing workload from single function into multiple smaller functions	This requires minimum effort, only a little extra time is required (about 30 minutes for this demo). This step is simple overall. Time is mostly spent on coding than analysis.
Ensuring dependencies among multiple threads are not changed	This requires some significant effort; about 2 extra hours are used. Time is mostly spent on analysis. In C#, about 40 lines of extra code are added for this purpose.
Coding the multiple threads	This requires minimum effort assuming one is familiar with the threading library in use. In this demo, about 10 minutes were needed for this part of coding. In C#, about 30 lines of extra code are added for this purpose.
Coding junk threads and deadlocks	This requires minimum effort. In this demo, about 5 minutes were needed for coding, and about 20 lines of code are written.
Other work related to multithreading	Coding timer function requires minimum effort, properly launching application in multithreaded mode also requires only little effort.

In summary, the extra effort in coding is not too difficult assuming one is already familiar with the library related to multithreading. On the other hand, making sure the design works properly requires more work in the analysis phase. In the demo, total effort is not more than 4 hours and approximately 100 extra lines of code; this is not much overall given the positive outcome.

7.4.6 XenoCode's Obfuscator

Code obfuscation is also part of the new design; incorporation of it would make disassembled code more difficult to trace and force attackers to waste time by studying junk codes. In this project, XenoCode's built-in obfuscator was used primarily for this purpose. Obfuscation is achieved by inserting junk code into binary code. Without detailed analysis of its effects, the result seems good if the highest level of obfuscation is used. We plotted the effects of obfuscation of all 4 levels against the original source code, as shown in Figure 41. In Figure 41, each vertical bar is a comparison between the obfuscated code and the unobfuscated code. Areas colored in red represents a difference in code, whereas areas colored in white represents the same

code. There are 4 levels of obfuscation provided by XenoCode, level 1 being the lightest obfuscation and level 4 being the heaviest obfuscation. The results in Figure 41 from left to right are corresponding to level 1 to level 4. As shown by figure, there are only a little white areas overall in level 4, suggesting good obfuscation.

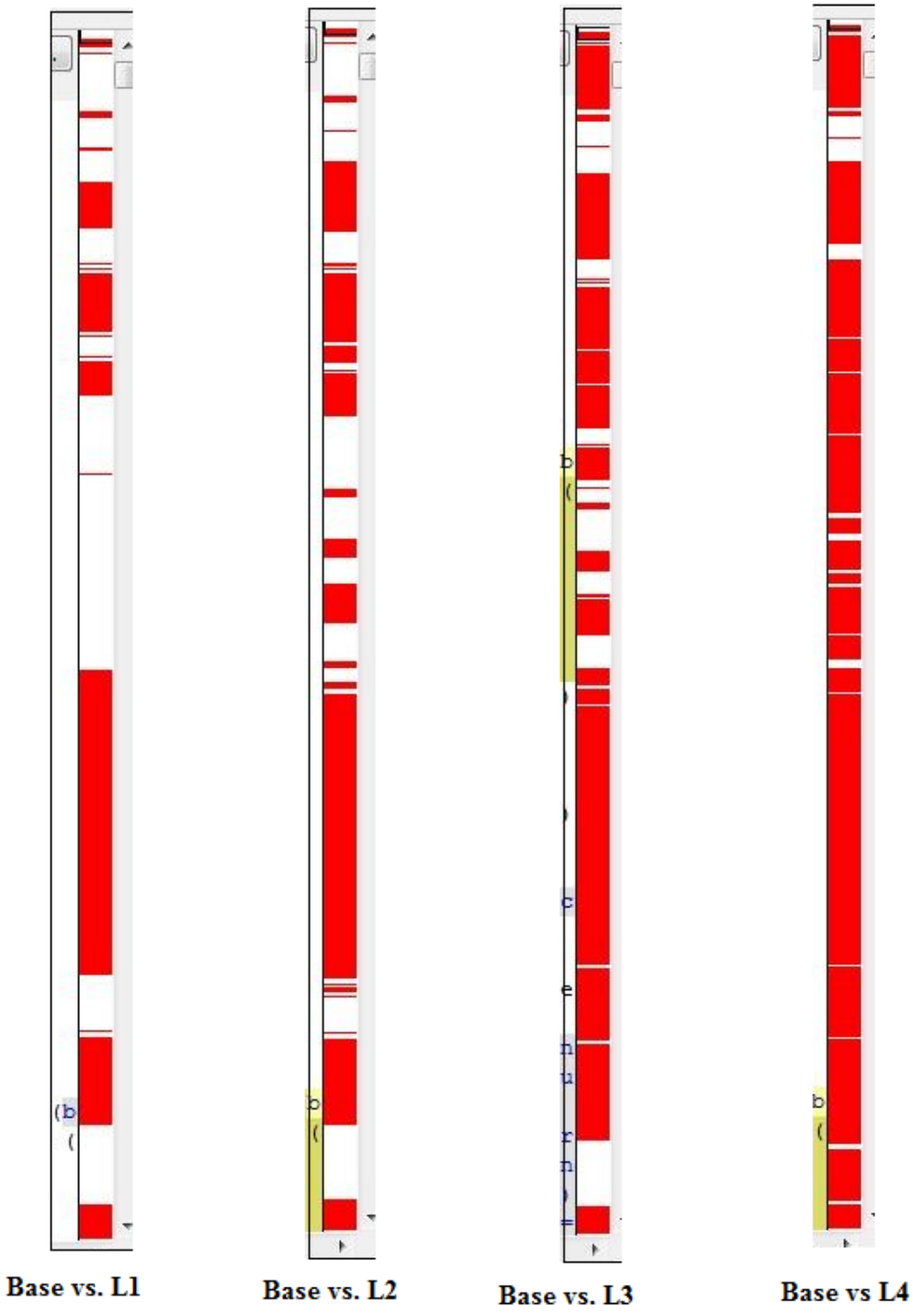

Figure 41. Quality of XenoCode's Obfuscator

8. Conclusion and Future Work

Our new design uses multiple threads and multiple validation modules for verifying serial numbers. After careful analysis of test results, running code in a multithreaded manner for checking serial numbers has clear advantages over the single threaded option. In particular, the following tricks appear to be quite effective for our purpose:

1. Accepting user input in a thread other than the checking threads.
2. Running Idle event handler.
3. Use of junk thread and deadlock, especially launching them before useful ones.
4. Checking serial number in multiple threads.

Our method achieved the primary goal of this work. It proves cracking a serial number validation can be made more difficult if multiple threads are used instead of a single thread since it reduces the amount of traceable code. Also, overall extra efforts needed to implement the new design are small compared to that of the entire software development cycle, making this method practical to use.

We studied how multiple threads can make dynamic analysis of disassembly in debuggers more difficult to perform. Future research can be expanded to include how difficult it can be to extract code to create KeyGens from a multithreaded checking mechanism, especially when code is obfuscated by third party tools. Also, the effects of a running timer (especially those with short time intervals) could be studied further to understand its impact on debugging code. In addition, one could use third party tools to try to analyze interaction between threads to see if thread dependency can be found; and if so, can the dependency be understood. One could also try to use threads purposely running in an infinite loop instead of deadlocks to find out which method is better for our purpose. Finally, one can try to implement our new design in another programming language to see if our method still holds against attack.

9. References

[1] activatesoft.net, “Product Activation Overview”, http://www.activatesoft.net/activation_overview.asp, retrieved on 08/24/2009

[2] Chris Davies, “Windows 7 cracked after Lenovo OEM key leaks”, http://www.slashgear.com/windows-7-cracked-after-lenovo-oem-key-leaks-2950684/, retrieved on 08/24/2009

[3] ORC, “How to Crack”, http://www.mindspring.com/~win32ch/howtocrk.zip, retrieved on 09/21/2009

[4] Jianrui Zhang & Shengyu Li, “CS265 Project 2 Report”, 05/11/2009

[5] MLC Technologies, “Hardware Key Activation”, http://www.mcl-collection.com/support/licensing/hardware_key.php, retrieved on 09/02/2009

[6] Schlumberger, “Cyberflex Access Cards Programmer’s Guide”, Jan 2004

[7] Bank of China, “Security Mechanism (Cooperate Service)”, http://www.bankofchina.com/en/custserv/bocnet/200812/t20081212_144526.html, retrieved on 12/05/2009

[8] Logic Protect, http://www.logicprotect.com/index.asp, retrieved on 10/23/2009

[9] Mark Stamp, "Information Security: Principles and Practices", 2006

[10] Mark Stamp, lecture notes on “Software Breaking”, Fall 2009

[11] Wikipedia, http://en.wikipedia.org/wiki/Product_activation, retrieved on 10/25/2009

[12] Martin Cowley, “Frontend Plush”, http://frontend-plus.software.informer.com/, retrieved on 12/23/2009

[13] Eric Lafortune, “ProGuard”, http://proguard.sourceforge.net/, retrieved on 11/10/2009

[14] Christian Collberg, “SandMark”, http://sandmark.cs.arizona.edu/, retrieved on 11/25/2209

[15] Scott Oaks, “Java Security”, Published by O’Reilly, 2001

[16] Borland, JBuilder 2007 Documentation

[17] XenoCode, http://www.xenocode.com/ , retrieved on 11/29/2009

[18] Wikipedia, http://en.wikipedia.org/wiki/Polymorphic_code , retrieved on 12/01/2009

[19] Shameen Akhter & Jason Roberts, “Multi-Core Programming: Increasing Performance through Software Multi-threading”

[20] BestSerials, http://www.bestserials.com/ , retrieved on 04/20/2010

[21] CrackLoader, http://www.crackloader.com/ , retrieved on 04/10/2010

[23] Australian Institute of Criminology, http://www.aic.gov.au/, retrieved on 04/10/2010

[24] Jedisware, http://www.jedisware.com/ , retrieved on 04/10/2010

[25] Cyberlink, http://www.cyberlink.com/products/powerdvd/overview_en_US.html, retrieved on 04/10/2010

[26] Chinmaan, http://i179.photobucket.com/albums/w306/chinmaan/activation.jpg , retrieved on 04/10/2010

[27] RabLab, http://www.rarlab.com/ , retrieved on 04/10/2010

[28] Avast, http://www.avast.com/free-antivirus-download, retrieved on 04/10/2010

[29] IDA Pro, http://www.hex-rays.com/idapro/, retrieved on 04/10/2010

[30] OllyDbg, http://www.ollydbg.de/, retrieved on 04/10/2010

Appendix A: Data

Table 6. Instruction Count from MSVS with Source Code in Single Threaded Version

Run #	Instruction Count (Breakpoint at Start only)	Instruction Count (Breakpoint at all function excluding Idle)	Instruction Count (Breakpoint at all functions including Idle)
1	131	131	40
2	131	131	40
3	131	131	40
4	131	131	40
5	131	131	40

Table 7. Instruction Count from MSVS with Source Code in Multi-Threaded Version

Run #	Instruction Count (Breakpoint at Start only)	Instruction Count (Breakpoint at all function excluding Idle)	Instruction Count (Breakpoint at all functions including Idle)
1	30	60	40
2	30	60	40
3	30	71	40
4	30	42	40
5	30	59	40
6	30	50	40
7	30	42	40
8	30	62	40
9	30	81	40
10	30	69	40
11	30	65	40
12	30	60	40
13	30	81	40
14	30	67	40
15	30	102	40
16	30	98	40
17	30	151	40
18	30	91	40
19	30	54	40
20	30	102	40
Average	30	73.35	40

Table 8. Instruction Count from MSVS with Source Code in Multi-Threaded Version with Different Number of Junk Threads with Breakpoint Set at Start Only

Run #	Instruction Count (2 Junk Threads)	Instruction Count (5 Junk Threads)	Instruction Count (10 Junk Threads)	Instruction Count (15 Junk Threads)
1	30	30	30	30
2	30	30	30	30
3	30	30	30	30
4	30	30	30	30
5	30	30	30	30

Table 9. Instruction Count from MSVS with Source Code in Multi-Threaded Version with Different Number of Junk Threads with Breakpoint Set at All Functions Excluding Idle

Run #	Instruction Count (2 Junk Threads)		Instruction Count (5 Junk Threads)		Instruction Count (10 Junk Threads)		Instruction Count (15 Junk Threads)		Instruction Count (20 Junk Threads)		Instruction Count (25 Junk Threads)	
	Useful	Junk	Useful	Junk	Useful	Junk	Useful	Junk	Useful	Junk	Useful	Junk
1	36	6	30	7	27	3	33	8	25	16	23	10
2	21	4	25	10	20	5	21	4	17	17	22	9
3	28	4	28	5	22	7	27	9	17	12	16	14
4	30	6	25	6	25	6	23	7	20	13	11	15
5	26	6	34	4	20	8	25	10	29	11	18	16
6	26	4	26	10	21	5	23	8	31	14	17	15
7	32	6	21	8	28	11	34	9	18	9	20	14
8	36	4	27	5	30	6	30	6	33	7	19	9
9	34	4	25	6	32	6	28	8	23	13	22	12
10	43	4	28	6	29	9	25	8	17	15	13	15
Average	31.2	4.8	26.9	6.7	25.4	6.6	26.9	7.7	23	12.7	18.1	12.9

Table 10. Average Instruction Count from MSVS with Source Code in Multi-Threaded Version with Junk Threads

# Junk Threads	Useful	Junk	Ratio of Useful to Total
0	73.35	0	0.649
2	27.6	4.2	0.244
5	23.9	6	0.212
10	22.7	6.3	0.201
15	23.6	6.9	0.209
20	20.5	11.1	0.181
25	15.8	11.9	0.14

Table 11. Traceable Useful Instruction Count from OllyDbg in Single-Threaded Version

Run #	Instruction Count
1	64136
2	64136
3	64136
4	64136
5	64136

Table 12. Traceable Useful Instruction Count from OllyDbg in Multi-Threaded Version

Run #	Instruction Count (bytes)
1	63190
2	62884
3	63190
4	63366
5	62762
6	63190
7	61252
8	62884
9	63190
10	63190
11	62884
12	63056
13	62884
14	61252
15	63056
16	62884
17	63190
18	63056
19	63190
20	63366

Table 13. Traceable Code from OllyDbg in Multi-Threaded Version with Junk Threads Launched First

Run #	Instruction Count (bytes)	# Junk Threads
1	61094	2
2	61094	2
3	61094	2
4	61094	2
5	61094	5
6	61094	5
7	61094	5
8	61094	5
9	61094	10
10	61094	10
11	61094	10
12	61094	10
13	61094	15
14	61094	15
15	61094	15
16	61094	15
17	61094	20
18	61094	20
19	61094	20
20	61094	20

Table 14. Traceable Code from OllyDbg in Multi-Threaded Version with Junk Threads Launched After Useful Threads

Run #	2 Junk Threads	5 Junk Threads	10 Junk Threads	15 Junk Threads	20 Junk Threads	25 Junk Threads
1	63190	61094	61094	61094	63190	61094
2	61094	61094	61094	63190	62762	62884
3	62884	63190	61094	61094	61094	61094
4	61094	61094	63190	61094	63056	61094
5	63190	63190	61094	61094	63190	63366
6	61094	61094	61094	63056	61094	61094
7	62762	62884	61094	63190	61094	61094
8	61094	61094	63190	61094	61094	63056
9	61094	61094	61094	61094	62884	61094
10	61094	63190	62884	63056	61094	61094
11	63190	61094	63190	61094	61094	63190
12	63056	62884	62884	61094	61094	61094
13	61094	61094	61094	61094	61094	61094
14	62884	63190	61094	63190	63190	61094
15	61094	61094	61094	61094	61094	61094
16	62762	61094	61094	61094	61094	61094
17	61094	61094	61094	61094	61094	61094
18	61094	62762	63056	61094	61094	62762
19	62762	61094	61094	62762	61094	61094
20	61094	61094	61094	61094	61094	61094

Table 15. Average Instruction Count from MSVS with Source Code in Multi-Threaded Version with Junk Threads

# of Junk Threads	# of Useful Threads	Total Thread Count	Theoretical	Actual
2	10	12	83.3	45.0
5	10	15	66.7	35.0
10	10	20	50.0	30.0
15	10	25	40.0	30.0
20	10	30	33.3	30.0
25	10	35	28.6	25.0