Development of Uncrackable Software

Saptarshi Laha
Department of Computer Science & Engineering
SRM University, Delhi-NCR
Sonepat, India
saptarshi.laha@gmail.com

Surbhi
Department of Computer Science & Engineering
SRM University, Delhi-NCR
Sonepat, India
surbhi.s@srmuniversity.ac.in

***Abstract*—One of the major challenges faced by the industries today is about the development of general-purpose software and the proneness of the same being cracked. A glimpse at the current implementation methods reveals that they serve no good as a permanent solution which leads us to a discussion on a variety of viable solutions to completely eliminate cracking from the computing world by either making it completely infeasible to crack or moving the source of operation to a third party provider out of the reach of the user and other such simple yet sophisticated mechanisms.**

Keywords—cracking, virtual machine, software protection, anti-cracking solutions

I. Introduction

Cracking refers to the changing of routines and/or subroutines within a program which allows a user to gain unauthorized access to functionalities within a program not intended for that user such as bypassing copy protection or converting trial software to full version without purchasing it. In the recent years, cracking is at its peak with hobbyist crackers and other professionals seeking monetary benefits from their "illegal" acts investing time and resources into breaking the underlying code related to the protection system enforced by the piece of software.

Many of the modern-day programs are created in languages like Java, which depend on a virtual machine lying underneath specifically crafted to run bytecode generated by the partial compilation process. The virtual machine simulates an original or fictitious CPU depending upon the designers of the virtual machine. The problem arises when a virtual machine is so commercially successful that one can figure out the underlying mechanism of its working and hence can exploit the program by modifying the code to perform initially unintended actions. The positive side to commercially unsuccessful virtual machines is the fact that not many people are aware or are interested in learning the inner workings of the same and hence are unable to exploit programs targeted to such environments to benefit their needs. The issue is, deploying to such virtual machines leads to equivalently unsuccessful commercial software.

II. General Model Of The System

A. Hardware vs Software

We wish to analyze both virtual machines and actual hardware to combat the above-mentioned situation. While virtual machines decrease the performance of the system by manifolds, it ensures the protection of the content running on top of it. This is because the hardware manufacturer is likely to provide a manual for the functionalities of the system. The manual to any system is a double-edged sword, it acts as good reference to both developers and crackers. While, a developer will think of useful strategies to enforce protection of the application, the cracker armed with the knowledge of the working of the system will nevertheless find a way to crack it. Hence the question boils down to "When will the patch to remove a certain protection mechanism be released?" and not "Whether the patch to remove a certain protection will be released?".

B. Performance Issues

Performance of a virtual machine is an important factor when it comes to performance of the applications getting emulated by it. To keep the performance close to ideal, the developer should focus on keeping the functionalities of every opcode as close to simple as possible. This means that every opcode should be composed of 3-5 simpler operations. This boosts the performance of the emulation system (virtual machine) and in turn also boosts the performance of the code running on top of it. Apart from this, when the virtual machine is tailor made to meet the requirements of a specific application, the instruction set is rather limited making the emulation system smaller and more efficient, in terms of both memory usage and code execution.

III. Encryption Approach

Encryption can be both Symmetric and Asymmetric in nature. There are two major areas where encryption can be applied:

1. The opcode list supported by the virtual machine.
2. The byte-code generated by a program.

Each of these areas have their own disadvantages. If the opcode list is encrypted, it eventually needs to be decrypted to the original form to run the program, or the program byte-code needs to be encrypted as well to make the encrypted versions match, to test for the opcodes in order to run the program and vice versa.

A. Applying Symmetric Encryption

Before the development of an application, the opcode list generated from an application should be run through a symmetric encryption algorithm. The deciphering should be an intrinsic property of the virtual machine that runs the program. In this manner, one can produce encrypted programs that run on a virtual machine which firstly deciphers opcodes and then processes it.

Another way of implementing symmetric encryption would be to use the same encryption key and encrypt the opcodes that are needed by the virtual machine to run the program before running an encrypted program.

B. *Applying Asymmetric Encryption*

A two key pair can be generated and referred to as K_1 and K_2 using algorithms such as RSA. The opcodes list generated from an application in this case can be encrypted by using K_1 before deployment and decrypted by using K_2 before or during the execution by the virtual machine. The deciphering process should be an intrinsic property of the virtual machine in this case as well.

The other way of implementing this would be using K_1 to initially encrypt the opcodes supported by a virtual machine, and then decrypting the opcode using K_2 while running an application in it.

IV. Problems With Encryption

Problems with encryption can be categorized into 2 categories:

1. Underlying opcode remaining the same.
2. Comparisons being performed in lower level code remaining the same.

A. *Underlying Opcode Remaining The Same*

In case of encryption of the opcode list generated by an application, the program is encrypted initially, but is eventually deciphered by the virtual machine while running it. Hence a cracker can use a general-purpose debugger to step through the code of the virtual machine and figure out the decrypted values of the opcodes of the program running on the virtual machine. This gives him an upper hand in exploiting the application. The exploitation in this case consists of modifying the virtual machine by adding functionalities or altering inbuilt functions to break the protection mechanism employed by the system.

Apart from this, a person won't have to invest much time understanding the functionality of a custom-built opcode in this case as the opcode list remains static and since the virtual machine is made in a low-level language, there will be a compare statement in the form of:

CMP [XX], [YY]

followed by a jump or call statement leading to a location/routine with an intended or unintended outcome. If this jump or call statement of the form:

JMP [Label/Location]

CALL [Label/Function_Label]

is modified, then the resulting output can be modified, and a series of such modifications can completely nullify the protection mechanism in place.

B. *Comparisons Performed In Lower Level Code*

Whenever the virtual machine has encrypted values of opcodes and a key is used to encrypt the resulting opcodes of a program before comparing opcodes to check which operation is to be performed, or in cases where the virtual machine's opcodes are decrypted before the comparison process, the virtual machine can be stepped through to find out the key being used in the encryption process of the opcodes in the program or the decryption process of the opcodes in the virtual machine and hence the functionalities of the virtual machine can be studied. After investing a considerable amount of time and resources, one can figure out what each opcode is supposed to do and one can easily modify either the opcodes generated in the program or modify the behavior of the virtual machine by changing the operations performed on encountering a particular opcode or adding new opcodes to the list to perform different functionalities. When these series of changes are made, the protection mechanism within a system can be completely bypassed.

The issue with all such encryption mechanisms used not only increase the overhead cost of execution of the program but also slows down the application's execution time at the cost of barely any protection. The real problem boils down to the storage of the key in the local user system in case of encryption and/or decryption where the user has access to the final output being generated by either of the two processes and has access to the key. This makes the software more vulnerable to cracking because it compromises its security during runtime. The main problem of such an approach is the need to decrypt encrypted opcodes or encrypt unencrypted opcodes with a pre-set or dynamically changing key to provide a layer of protection from potential crackers. This strategy isn't successful as the key needs to be stored and used in the encryption or decryption process, which is visible to the user and thus can be exploited. This type of an approach would only provide robust protection if there was a mechanism set in place to run encrypted opcodes at a hardware level where it couldn't be analyzed and if there was an option to reprogram the instruction set of the computer completely or partially to provide additional and/or similar functionalities to the current instruction set.

V. Multiple Virtual Machines

A. *Virtual Machine Sandwich Approach*

This approach uses more than one virtual machine to run a program. This is done by crafting a basic virtual machine in low level languages and then crafting more virtual machines on top of the initial virtual machine. Sometimes these are layered as in a sandwich of virtual machines on which the software deployed runs.

Multiple virtual machines add to the overall code that is present in the software system which acts as a great way to prevent reverse engineering of such software as one line of code in a high level language can translate to multiple lines of opcodes, but it consumes resources exponentially per layer of virtual machine that is incorporated in running a piece of software, thereby increasing the overhead cost of running a program on top of the layers of virtual machines while providing close to zero security in terms of software protection because it can easily be cracked by any knowledgeable cracker.

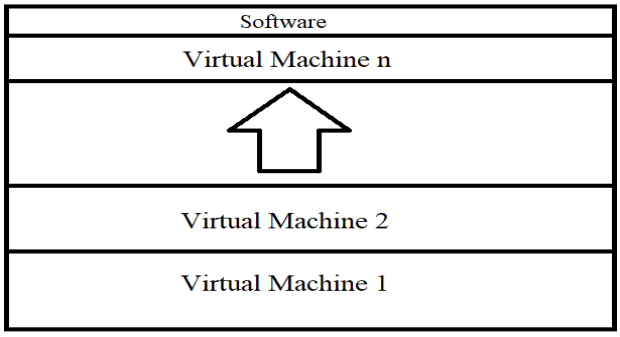

Fig. 1. Virtual Machine Sandwich

B. Sandwiched Virtual Machines Using Encryption

In this technique, multiple virtual machines can be layered one over another as in the previous case, but also adding encryption to each of the virtual machines. Some of these approaches are used in industrial level protection of software, such as many triple A titles use VMProtect and Denuvo(which is an application that provides encryption on top of the virtual machine that it runs on) which is an enormous virtual machine on its own, to run games. Not only does this affect the performance of the game by slowing it down from its actual speed, but also doesn't suffice to provide enough protection as most of these are cracked eventually by knowledgeable crackers.

Although, such layered architectures cannot be cracked by a beginner, but a knowledgeable cracker with enough time to invest into such applications can eventually crack it.

The process of going about cracking such applications is cracking one layer of virtual machine at a time. Whenever needed, the cracker is free to exploit the issues in simple encryption approach to crack the encrypted opcodes.

Hence, not only does all the extra layering and deciphering adds to the overheads at the cost of reasonable protection, but ultimately it leaves the user of such software extremely dissatisfied as the amount of system resources used will skyrocket with each layering needed apart from the deciphering process in each layer of encryption.

VI. Server Checks

A. The Need For Server Checks

So far, all the protection mechanisms barely solve the issue of crafting uncrackable software while adding to the time complexity of the application. The next logical approach is to present a server check for the application each time it is started. This means that the server confirms if the user is registered and can use the functionalities that s/he is using. In case of a failure, the software notifies the user about the software being unregistered and quits.

How this combats the situation of the previously mentioned condition is by not using the local system for checks, but rather confirming the validity of the application from an external server which is fed with legitimate data by the software developer. This implementation is present in a lot of applications designed for multiple operating systems.

Some applications apart from having a server check also checks for hash values of the application and digital watermarks placed within the program or virtual machine.

B. Why Server Checks Fail

While server checks are powerful, one cannot deny the fact that the software runs on the local machine where the user of that machine has complete control on the behavior of the application. What this means is that the user can simply change the comparison, jump and call statements like earlier to completely bypass the software protection mechanism.

In case of more complicated checks, one can simply simulate a counterfeit server and create fake packets to trick the application into behaving normally as it would in case it received a positive response from the server regarding the validation of the software.

VII. Digital Watermarking

Digital watermarking refers to hiding of validation specific data within a program. The server or a local system such as the virtual machine checks for the watermark in the file and responds accordingly to the validation of the program.

These watermarks are strategically placed within the program so the cracker cannot easily locate the validation mechanism. Sometimes, even multiple watermarks are placed in the application to prevent crackers from easily cracking it. All of this depends on the size of the organization crafting the program or the amount of security an individual or organization wants to incorporate into the product.

Digital watermarking is one of the most efficient methods of validation of a program. This not only doesn't affect the execution time as the server checks for a hash value or a set of bits within the program which barely consume any system resources, but also provides effective protection against easy cracking of the piece of software.

VIII. Cracking Epidemic & Viable Approaches To The Problem

A. Why Cracking Is Difficult To Defeat?

The answer to this question lies in the simplicity of the system that lies underneath all the abstraction presented to the user. Every application can be translated to a sequence of assembly programs where the assembly generated is vendor specific depending on the CPU that's running underneath. These CPUs' functionalities are extremely well documented so that the programmers can program them to accomplish multiple tasks at hand. This also means that if a user has enough knowledge, s/he can modify any program to accomplish any task that s/he wants by bypassing the protections in place. The opcodes are one to one conversion to assembly programs which the CPU understands and hence all the programs run in hexadecimal form which in turn are crammed representations of 1's and 0's. Generally, the programmers try to incorporate into their product protection mechanisms with the help of abstraction layers present on top of the base hardware. These techniques fail as all the layers of abstraction can be eventually broken down into assembly and the protection mechanism eliminated from the final program.

B. Simple Approach To Combat Cracking

An easy approach to combat cracking is to manufacture CPUs with the presence of a special purpose reprogrammable instruction set chip with a unique hardware identifier number which is essential to the system to function normally. This chip should provide the unique hardware identifier number when a special instruction is encountered which takes care of the validation of the system. Such anti-cracking methods have already been used in the past, such as in the Sega Saturn where the wobble of the curly carved CD-Disk was used

to verify the legitimacy of the inserted CD. This was a revolutionary approach at the time of its creation. In general, such hardware checks are very difficult to bypass, if not impossible. This when added to the implementation of the above-mentioned methods, the benefits of presence of a reprogrammable instruction set and using the unique hardware identifier number for verification, creates completely impenetrable software that runs on it. Our goal is to create the same but, in an environment, leaning more towards changes in software than changing a lot of hardware components since such hardware dependent checks and flexibility at the CPU level are absolutely impossible to incorporate in every system that's functional currently. If there is such a need by the software, unless it is extremely useful or irreplaceable or serves a very special purpose, it will be commercially unsuccessful in today's market.

IX. Safe Software Deployment Methods

A. VM & Software Coupling

The software should be released with the underlying virtual machine instead of releasing them as separate packages. This ensures that the virtual machine doesn't become general purpose and hence cannot be easily cracked or analyzed in depth.

B. Unique VM For Each User

Each virtual machine's opcode list should be encrypted using a key specific to that user. Asymmetric encryption techniques should also be used to decrypt the opcodes within the program. This protects the application from leakages as an individual can be tracked back using the details of the virtual machine as every key generated will be unique and bound to the user.

C. Couple Checking Apart From Server Checking

The virtual machine should incorporate methods to check the program running for changes in its hash value or watermark locations, as should the program to the virtual machine. This in turn should be reported to the server to update the values. This value should be updated every time the program ends its execution.

D. Multiple Digital Watermarks Stored In Random

Digital Watermarks should be stored in the application and extra ones should also be stored in directories dictated by the server on startup. These watermarks should be modified, directories deleted, and new ones stored upon the next startup.

X. Multiple Foolproof Methods

A. Using Internet's Resources To Run An Application

With the rise of internet speeds, it's very much viable to be able to download an application more than once if needed or equally much viable to have an online virtual machine serving multiple clients connected to it. The clients so connected should have an encrypted copy of the program which can only be decrypted by the server which holds the virtual machine. Since the decryption process of the application using the user specific key occurs first, the server can quickly run the opcode check for the program before launching it. After this process, the server can then launch the application. The problem here is that it will be referred to as a web-based application instead of a real one using the computation power of the system. After its initial usage, the server should then provide a new copy of the application to the client with newly encrypted opcodes and should update its own database for its key entry to be used next when the application is supposed to launch again. This is extremely efficient for general purpose computing but cannot be used to perform operating system specific actions. Apart from this, since servers are powerful computational systems and can serve multiple clients at once and with the current rate of decreasing server buying and economical leasing prices, it serves as an effective solution to the problem.

B. Using MOV To Decrease Overall Cost & Making Technically Uncrackable Applications

The seminal paper released by Stephan Dolan outlines the Turing Completeness of the MOV instruction in any of the systems currently in use[1]. This is implemented by Chris Domas in his movfuscator[2] which aims at compiling any C program into a bunch of MOV instructions instead of the general assembly. In case this is implemented in the CPU systems that are upcoming in the future, it will not only reduce the cost of building the circuitry by manifolds but will also make it technically impossible to make any sense of the compiled program whatsoever. The absolute frustration of a reverse engineer assigned to a duty of deciphering the meaning of a couple million MOV instructions is unimaginable. This when coupled with 1- or 2-layer virtual machine and encryption-based program protection literally makes it uncrackable under any circumstance. Apart from this, the ingenious IDA-Pro hack released by Chris Domas, called REpsych[3], makes static analysis of any program an absolute nightmare if using IDA-Pro, which is one of the most commercially used tools specifically designed for reverse engineering.

A general GCC output of a program is understandable by someone who is aware of assembly programming in 8086. The output also logically makes sense as each instruction is different and refers to certain operations taking place within the system, however the movfuscator's output and control flow graph make close to zero sense under ideal situations. This when further obfuscated produces illogical and unreadable output. The control flow graph generated by the movfuscator is nothing but a straight line without any branching unlike the GCC control flow graph as analyzed by the static analyzer IDA-Pro.

C. Implementing Rootkit & Virus Technology For The Good

A rootkit is a piece of software that is intended to gain access to a system without the knowledge of the user to perform monitoring of data or similar actions. In this case, the rootkit designed should be of special purpose and should only oversee the program and virtual machine that the program runs on (Such an incident took place when Sony installed their own version of a rootkit in systems to

monitor the misuse of data and were sued for the same). It should report back to the server regarding any tampering of the application, in which the server automatically cancels the license of the user while the virus technology implemented can delete the files from the system. Performing this illegally is a huge task but it can easily be implemented in the form of a driver in case of Windows, responsible for monitoring the applications in question and performing ring 0 level operations if needed to sanitize the program and the virtual machine. The rootkit should then delete itself from the system to avoid getting analyzed. This rootkit should be updated regularly to avoid being detected by anti-malwares or by other users interested to analyze the program who are aware of the presence of the rootkit as well.

To implement this in an illegal manner would mean exploiting operating system specific routines which in itself is a reverse engineering challenge but is possible as discussed in the previous sections as the operating system is just a piece of software running on the CPU which can be analyzed in parts and the underlying meaning extracted and evaluated together.

Some of the common steps of creating a rootkit includes, but is not limited to infecting the bootloader, altering the system call table, modifying the kernel. These acts are however illegal and hence one should stick to legitimate driver signing techniques whenever required to gain ring 0 access to a windows machine.

XI. An Unique Approach To Solve The Problem

Let PR_s denote the server's private key and PU_s denote the server's public key for client *C*. Let *K* denote the key of the hardware dongle which is essential for this approach. Let *O(X)* denote the application of obfuscation operation on *X* and $o_{x1}....o_{xn}$ denote the unique obfuscated outputs resulting from such an operation where o_x is obfuscated version of the software *X*. Let *S* denote the software and let *V* denote the virtual machine that the software runs on. Let $l_1...l_n$ be the lines of code of the software *S* in opcode format. Let *O(V)* produce *p* outputs in the form of $o_{v1}...o_{vp.}$ Let Y_V denote the number of obfuscations per operation executed in the software. If $p>n$ then we set $Y_V = \lfloor p/n \rfloor$, else we set $Y_V = 1$. Let *V* be initially in the form of o_{vd} where $d \geq 1$ and $d \leq p$ and o_{vd} is the longest possible code length of *V* amongst its *p* obfuscated forms that are possible/present.

Constraints: The user U will have to own a copy of the dongle in order to run the software S.

Initially, the server's database contains PR_s, PU_s, *K*, *V* and *S*. The user initially doesn't have *S*, *V* and PU_s, however he has the dongle in which key *K* is stored.

The following steps are performed in order to execute the program securely and to prevent it from being cracked:

1. *C* sends $K(HWID_{dongle})$ to the server to match it with the server's entry of *HWID* where *HWID* refers to the unique hardware identifier number and $K(HWID_{dongle})$ denotes encryption of the unique hardware identifier number of the dongle with dongle key *K*.
2. The program continues execution if the *HWID* matches with the server's entry of *HWID*. Otherwise the program execution stops and the license for *C* is cancelled.
3. *C* receives *V* in the form of $PR_s(K(V))$ where $K(V)$ represents the encryption of *V*'s opcodes using dongle key *K* and $PR_s(K(V))$ represents the encryption of $K(V)$ by private key of the server PR_s.
4. *C* receives the PU_s in the form of $K(PU_s)$ where $K(PU_s)$ represents encryption of PU_s using dongle key *K*.
5. *C* forwards the $K(PU_s)$ to the dongle where the operation $K^{-1}(K(PU_s))$ occurs and results in PU_s which is then stored by *C* for the next few operations.
6. *C* receives l_1 from the server in the form of $PR_s(K(l_1))$.
7. *C* decrypts it using PU_s to acquire $K(l_1)$.
8. *C* decrypts $PR_s(K(V))$ using PU_s to acquire $K(V)$ where $K(V)$ is *V* in the obfuscated form o_{vd} and check for opcodes are encrypted in the form of *K(Opcode)*.
9. The opcode for which $K(V)$ matches with $K(l_1)$ is executed and if there is any reply *R* for the server for the same execution, it is encrypted with $PU_s(R)$ and sent to the server where it is decrypted and the next line of code l_2 is evaluated based on *R*.
10. *C* sends $K(HWID_{dongle})$ to the server to match it with the server's entry of *HWID* where if it matches, the program execution continues further, else the *C*'s license is cancelled and program execution ends.
11. The server sends the new dongle key K_1 in the form of $PR_s(K(K_1))$.
12. This is decrypted by *C* to $K(K_1)$ and is forwarded to the dongle where $K^{-1}(K(K_1))$ occurs and *K*'s definition is updated using the rule $K = K_1$.
13. Dongle sends a key successfully set message to *C* which is then forwarded to the server.
14. The server then updates PR_s, PU_s and *K* to PR_{s1}, PU_{s1} and K_1 respectively.
15. The server then performs $O(V)$ X_V times and sends new *V* definitions in the form of $PR_{s1}(K_1(V))$.
16. *C* updates the definitions of V by using a technique called *Self-modification of Code* as explained in the next section.
17. *C* receives new PU_{s1} from the server in the form of $K_1(PU_{s1})$.
18. *C* forwards the $K_1(PU_{s1})$ to the dongle where $K_1^{-1}(K_1(PU_{s1}))$ occurs and results in PU_{s1} which is then stored by *C* for the next few operations.
19. The execution then proceeds as normal from S*tep 6* until the end of code is reached.
20. If the end of code is reached, then the server generates a new set of keys in the form of PU_i, PR_i, and K_i where K_i is sent to *C* in the form of $PR_{i-1}(K_{i-1}(K_i))$. *C* then decrypts the same using PU_{i-1} and forwards $K_{i-1}(Ki)$ to the dongle where the operation $K_{i-1}^{-1}(K_{i-1}(K_i))$ occurs and *K*'s

definition is updated to K_i in the form of $K = K_i$.

21. The server then does $O(V)$ X_V times and keeps a version of V ready in the form of o_{vm} for the next time the application is launched by C.

XII. Self-Modification Of Code

Self-modification of code in this context refers to the modification of the code segment of the program using an assembly program of the form:

```
programSegment segment read write execute

virtualMachineProgram proc

mov rax, rcx
;Used to move the opcode to be processed from rcx to rax
vmLabel:
;Steps to decipher the opcode here
nop
nextRoundLabel:
jmp getNextLabel
;Other code here
getNextLabel:
call getOpcodeList
mov dword ptr vmLabel, [opcode_list[0]]
…
mov dword ptr vmLabel+n, [opcode_list[n]]
ret

getOpcodeList:
;Get data from the internet and store it in [opcode_list]
ret

virtualMachineProgram endp

programSegment ends
```

This code firstly reads an opcode which it then moves to the EAX register for use. Then it performs various checks to see which opcode matches with the opcode specified. On finding a match, it executes the operations underneath and jumps to getNextLabel, where it calls getOpcodeList function, which gets the opcodes from the internet and stores them in a list called the opcode list. Then the execution continues with the program rewriting its vmLabel section with the new virtual machine definitions which are retrieved from the opcode list.

XIII. Result & Advantages of The Method

After discussing almost all the currently implemented methods that are currently in use and discussing some of the potentially foolproof ones, we decided to introduce our own algorithm. This algorithm not only takes care of secure software delivery as each line of opcode to be processed is provided individually after a certain number of checks are performed to detect tampering or absence of hardware, but also ensures that the application is be technically uncrackable because of the following conditions:

1. Assuming the number of obfuscations possible for a virtual machine containing n lines of code is x. This means that each time the virtual machine's definition is updated, each opcode will perform one of x possible underlying operations assigned to that opcode at any given time.
2. The opcode list of the virtual machine is encrypted each time by a dynamically changing key which only makes reverse engineering of such an application harder as not only does the opcode list of the underlying operation change in the form of $PR_{sp}(K_p(l_p))$ which translates to line p encrypted by dongle key K_p and further encrypted by private key of server PR_{sp}, but also this when decrypted by the client C results in encrypted opcode which matches with the then definition of virtual machine and no other definition of any other virtual machine. This when added to the everchanging definition of the virtual machine makes it technically impossible to figure out for what opcode a certain operation is performed.
3. A server-based approach to software delivery is always beneficial as the product can be updated on the fly without relying on users to manually update applications which takes care of patching further loopholes and bugs in the software itself rather than its implementation method.

This approach tackles the problem of cracking using a lot of mechanisms including hardware-based authentication, encryption of opcodes, usage of virtual machines to run software and most importantly server-based software delivery which is essential for maintaining the security of the software. As the opcodes of the software are fed line by line by the server to the client, the client cannot analyze the complete program flow at once which is possible in case the software is delivered directly to the client machine. Extra mechanisms such as deletion of code after execution can be incorporated to further tighten the security factor of the software being deployed.

XIV. Limitations Of The Method

Any such computationally intensive algorithm comes with its fair share of limitations. In this case specifically, we do have a few as well which are as follows:

1. Internet speeds need to be higher than what they currently are in industries, households or the market that the software author is targeting in order to get realistic execution times.
2. The need for such a software should be high enough for the user to pay a premium to own a copy of the dongle to run the software which is unrealistic currently due to companies taking to digital-distribution of software and not relying on hardware for software distribution.
3. One of the major limitations that arises in the implementation of this method is when the number of lines of code of the virtual machine being deployed has very few lines of code. In this case the number of possible obfuscations will be very low and ultimately the working of such a virtual machine can be guessed by

comparing multiple definition updates of the virtual machine. This however can be combatted using the movfuscator designed by Chris Domas to compile the virtual machine. In such a case the number of the lines of code of the virtual machine will increase and as a result the number of possible obfuscations to code will increase as well.

4. One can argue that all the data packets can be captured which results in the definition files of the virtual machine and encrypted lines of the opcodes being stored by the user. This can later be analyzed, and the software can hence be cracked using all the data that's present to the user. This is possible but is highly unrealistic. General-purpose software generally span thousands of lines of code and the virtual machine responsible to run such a software should at least have ten thousand lines of code. In such a case thousand different encrypted versions of opcodes need to be stored and analyzed against $n \times 10000$ different definitions of the virtual machine running the software where n is the number of possible obfuscations of the opcode generated by the virtual machine. This when added to the complexity of following obfuscated code where labels aren't constant and operations dynamically changing in each execution, it is technically infeasible if the lines of code of the virtual machine is decently sized. This when added to the complexity of code that the movfuscator produces(if used) and the need to reverse engineer the dongle and create a counterfeit hardware simulator and counterfeit server for distribution of software packets, makes it technically impossible to crack such an application. Hence the only time when such an approach is possible is when the number of obfuscations of the virtual machine definitions are low in number and no other obfuscation toolkit has been used to increase the lines of code generated by the virtual machine.

XV. Conclusion

While this approach to the software delivery and development provides unparalleled protection standards, it can further be strengthened using other techniques such as code obfuscation using movfuscator to generate the underlying machine level opcodes for the virtual machine and the use of multiple layers of virtual machine, each of which incorporates this strategy for code execution to run the software on top of it, digital watermarking of the products at various locations for additional checks to be performed to continue execution of the program, all at the cost of system resources being used increasing exponentially with each layer of addition of virtual machines.

Although this implementation is more of an abstract idea rather than a useable alternative currently, but with the ever increasing speeds of the internet and the increase of computational power with faster and more efficient computational chips being developed every year, it technically won't take very long to actually convert the same into reality and implement such a strategy to enforce robust software protection.

Apart from this, with the emerging trends of quantum computers, quantum computing and better cryptographic algorithms being developed for both general purpose computers and quantum computers, better encryption algorithms can be applied to this algorithm to make the protection of the software even more robust and impenetrable at no cost of execution time of the software being affected whatsoever if such a strategy is implemented on a quantum computer.

Acknowledgement

Thanks to my father, Mr. Soumya Laha, mother, Mrs. Sima Laha and uncle, Mr. Soumya Mitra for introducing me to the fascinating world of computers and providing all the necessary support throughout my journey towards my inquisitiveness and thirst for knowledge and curiosity towards computers and computing.

I would especially like to thank my mentor Ms. Surbhi for her continuous support and guidance without which this project would never have been complete.

I would also like to thank each of my teachers and professors for their encouragement and guidance which really helped me in diving deep into the various aspects of computing at a much granular level than I initially knew of understood.

References

[1] Stephen Dolan (2013) mov is Turing Complete. https://www.cl.cam.ac.uk/~sd601/papers/mov.pdf. Accessed 20 February 2019.

[2] Chris Domas (2015) movfuscator. https://github.com/xoreaxeaxeax/movfuscator. Accessed 18 April 2019.

[3] Chris Domas (2015) REpsych. https://github.com/xoreaxeaxeax/REpsych. Accessed 18 April 2019.