LLVM-based overlapped executable code generator

V. Aranov
Institute of applied mathematics and mechanics
SPbSPU
Saint-Petersburg, Russia
vladik@d-inter.ru

A. Terentiev
Institute of applied mathematics and mechanics
SPbSPU
Saint-Petersburg, Russia
alterterrific@gmail.com

***Abstract*—Overlapped executable code is an attractive artifact of obfuscation technology not yet widely covered and researched. Overlapped code and opaque predicates technologies together allows creation of prominent software obfuscation technologies featuring both obscure executable code and code protected from patching due to hard-to-track relations with other code. The paper provides polynomial algorithm to generate overlapped executable code using LLVM framework and discuss results of the generation implementation.**

Keywords—obfuscation; LLVM; code transformation; code generation; reverse engeneering

I. INTRODUCTION

The two main approaches to overcome software piracy threats are used: administrative one, including legislation support and organization piracy countermeasures and technical, which include different kinds DRM, registration keys, software activation technologies and so on. We are to concentrate our efforts on the technical aspect of this problem. The need of obfuscating code transformation in the industry is clear: a lot of pirated software available in the Internet shows inefficiency of current technical protection methods and techniques. There is no need to go far away to find examples. The first KMS activator for Windows 7 appeared in less than 3 months after operating system gone alive. For Windows 8.1 the same took place less than one month: in Oct. 17, 2013, the OS was published and around Oct. 25 KMS activation solution was readily available for everybody to download in the Internet [1]. The client activation code for MS Windows starting with Windows XP uses an asymmetric cryptography, so it is impossible to generate the valid activation response. However, the valid KMS server can be bought by a client for local activation and the code from it was used to create KMS activator back in 2010 and 2013 years. No need to tell KMS client and server codes in both products were protected with anti-debugging techniques and properly obfuscated, but reality tells us “not enough did”. This is only one story, but with best “impact factor” which calls us for new code generation methods for code execution in insecure environment.

Another example is WinRar – a popular data compression product. Key-code generation algorithm or specifically private key for registration verification code was never publicly available, but counterfeit copies of WinRar are still available despite of all measures taken by Eugene Roshal and his team. The reason is simple: the code is either patched to ignore key code check altogether (loosing archive authentication feature), or public part of registration checking part of the executable code was replaced with one in keygen [2]. These examples demonstrate the need for patch-proof code that cannot be easily modified by either third party or legal customer of the product. Current obfuscation technologies include mostly virtual machines, different morphing technologies, garbage code insertion and code encryption with runtime decryption coupled with heavy anti-debugging technologies, but every encrypted code has to be decrypted before execution and therefore can be patched. In addition, most anti-debugging technologies are well known; morphing and garbage insertion do not prevent code modification at all. Obfuscation virtual machines still provide serious challenges for hackers, but still could be defeated with enough efforts. So, something completely new should be invited. Overlapped code is promised to be one of such solutions.

II. OVERLAPPED CODE

A. Attacker’s model

From now on we are going to use Bruce Schneier archetypes [3]. Let’s assume Eve as a person with malicious intension to modify a program developed by Alice. Alice has transferred to Eve full program consisting of executable modules, dynamic linking libraries and data files. Eve has full control over execution environment which means that she can:

- Modify any and every byte of executable program at any given time.
- Set breakpoint at the any point of Alice application.
- Perform full snapshot of all address space Alice application is running in.
- Record execution traces.
- Perform backtrack debugging.
- Alice cannot react to Eve actions.

Therefore, Eve is like omnipotent Supreme Being relative to Alice code. However, no Eve actions except for the first one break execution logic of Alice code. While modifying the code, Eve supposes she does not break the logic of other parts of the code except for that were just modified. However, two technologies break this assumption: making check sums and overlapped code.

Unfortunately, the code check sums are easy to defeat: many platforms have hardware “Page guard” breakpoints to

This work is licensed under the Creative Commons Attribution License.

assist Eve. “Page guard” breakpoint only triggered when CPU reads specific memory page, but not when executes. Therefore, overlapped code is the only valid option.

B. Overlapped code idea

How one can make a patch-proof code in this case? At first, such task seems to be impossible as soon as Eve has full control over execution environment with specified capabilities. However, there is a way showed on Fig. 1.

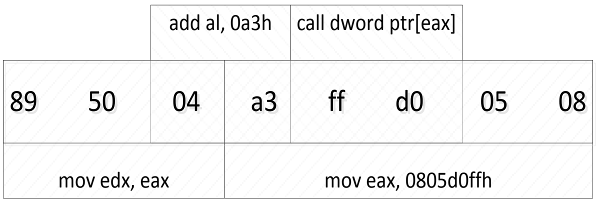


Fig. 1. Overlapped code with 4 bytes overlapped and 2 bytes shift

Bytes on the Fig. 1 encode two sets of instructions at once:

```
mov edx,eax
mov ax, 0805d0ffh
```

and

```
add al, 0a3h
call dword ptr[eax]
```

Patching any overlapped byte will implicitly change meaning of another instruction in other code execution path. If this code path is not discovered by Eve, yet such code change may even go unnoticed because the task of discovering all executing control paths is not solvable for arbitrary case. In most cases using common tools like IDA, Hex-Rays and OllyDbg second layer code will not be even discovered using static code disassembly analysis, which means this approach not only having unclear way to defeat but also being hard to detect.

III. OVERLAPPING CODE QUALITY

Before starting overlapping code generation it is important to define exact goals of such generation, i.e. define a criterion answering the question: which of two pieces of overlapped code of the same functionality is better.

Let’s define requirements for such criterion with the following assumptions: P – is a program of n size generated by reference LLVM compiler, Q – is a program of m size generated by overlapped code generator with same functionality as P, $x_1, .., x_m$ – each byte usage count in program code, $W_P(Q)$ – target quality measure:

- $\forall P, i = 1..n: x_i \equiv 1$. We assume reference compiler neither generate overlapped code nor use alignment skips.
- $\forall Q, m = n, x_i = 1, i = 1..n: W_P(Q) \equiv 1$.
- $\forall Q, m < n, x_i = 1, i = 1..m: W_P(Q) > 1$. We do not want a huge program size. The shorter program code, the larger $W_P(Q)$.
- $\forall Q, m > n, x_i = 1, i = 1..m: W_P(Q) < 1$. The larger program code, the lower $W_P(Q)$.
- $\forall Q : m = n, x_i = N, i = 1... m : W_P(Q) \equiv N$. The imaginary program of the exactly same size but with every byte used exactly in N instructions will have N as value of criterion.
- The more overlapping bytes in the code, the larger $W_P(Q)$.

The task of creation desired criterion is not too complex as it can be derived from the series of logical assumptions:

- Calculate the average overlap as $(A) = \frac{1}{k}\sum_{i=1}^{k} x_i$, where k is the size of the program A and x_i is a number of instructions i-th byte participate into.
- Calculate the specific average overlap over the size of the code as $A_{spec}(P) = {\frac{\sum_{i=1}^{k} x_i}{k}}\Big/_{k}$.
- Define the quality function f for programs A, B: $\mathrm{f}(\mathrm{A},\ \mathrm{B}) = \frac{\mathrm{A_{spec}(A)}}{\mathrm{A_{spec}(B)}}$. This function allows to compare to programs A and B. So when $\mathrm{f}(\mathrm{A},\ \mathrm{B}) > 1$ the program A is considered better than the program B, when $\mathrm{f}(\mathrm{A},\ \mathrm{B}) < 1$ the program B is considered better than the program A and $\mathrm{f}(\mathrm{A},\ \mathrm{B}) \equiv 1$ means quality of programs A and B are identical.
- For the regular program P created by the reference code generator provide by LLVM with 1 byte alignment the specific average overlap will be $A_{spec}(P) = \frac{1}{n}$, considering the fact that $x_i \equiv 1$, where n is the size of the program P.
- The final formula will take form $W_P(Q) = f(P, Q) = \frac{(\sum_{i=1}^{m} x_i)*n}{m^2}$, where n is the size of the program P and m is the size the program Q.

If we need to prioritize either overlap or generate code size the suitable generalized criterion will be:

$$W_P(Q) = \sqrt[d]{\frac{n*\sum_{i=1}^{m}(x_i^d)}{m^2}}\,, \qquad (1)$$

where d – is an arbitrary float parameter from $d \in (0, +\infty)$, where $d = \varepsilon$ and ε is a small positive number, means we do not care about overlapping at all and $d \gg 0$ means we prefer overlapping over the code size. Further we are going to use formula (1) with $d \equiv 1$.

In general, the more $W_P(Q)$ value, the better result.

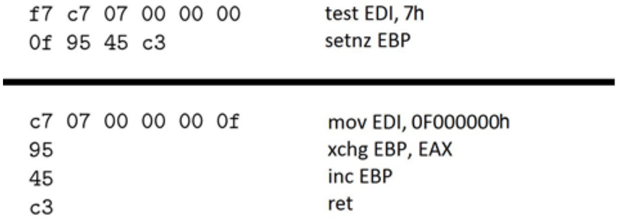
This work is licensed under the Creative Commons Attribution License.

IV. GENERATION OF OVERLAPPED CODE

The ROP (Return-oriented programming) [4] technique had been employed for overlapping code generation task. This technique uses control over an exploited program to execute an arbitrary code in vulnerable application. However, we are to employ this technique for good. ROP defines sequences of instructions ending with flow control instruction and not containing flow control instructions as *gadgets*. It is worth to mention, any instruction capable of modifying instruction pointer register can be used as gadget finish instruction. According to ROP, the gadgets are usually searched in an application executable code or in dynamically linked libraries.

During ROP attack, Mallory[3] usually overwrites executing program stack and creates gadgets library. The first is not important for us and covered by R. Hund [5], but the latter is the way to go for our purpose. Let's consider two major ways to create a gadget library:

- Explicit instruction sequences. Explicit sequences are widely discovered in standard library functions. According to Roemer [4], libc library contains more than 4000 different potential gadgets capable to implement almost arbitrary algorithm, while the library size is only 1.3 Mbytes. However, explicit sequences are not important for us because of not increasing criterion (1).
- Implicit instruction sequences. These are instruction sequences we are looking for, since each byte these instructions consist of will increase (1). There sequences are obtained through looking for specific byte (or bytes) in code (for example: 0C3h – ret instruction) and backward disassembly starting with this specific byte. One such byte(s) can usually produce more than one gadget. This approach would provide even more gadgets than explicit instruction case. However, one should be accurate with relocation items addresses. Fig. 2 provides good example of implicit gadget.



```
f7 c7 07 00 00 00        test EDI, 7h
0f 95 45 c3              setnz EBP

c7 07 00 00 00 0f        mov EDI, 0F000000h
95                       xchg EBP, EAX
45                       inc EBP
c3                       ret
```

Fig. 2 Implicit gadget example

The main difference from standard ROP is that initially we do not have any code to create gadgets from, because our compilation unit is empty. The "Overlapped code generator" algorithm pseudo code is proposed to get around this problem:

```
In: $funcs = ar[n] of ByteFunction$
Out: $newFuncs = ar[n] of ByteFunction$
Algorithm:
$gadgetList$=nil
$newFuncs$[0]=$funcs$[0]
for $i$=1 to $n$ do
$FindNewGadgets$($gadgetList$, $newFuncs$[$i$ - 1])
$newFuncs$[$i$]=$InsertGadgets$($funcs$[$i$],$gadgetList$)
end for
```

,where $FindNewGadgets$ has following pseudo code:

```
In/Out:
$gadgetList = array of Gadgets$
In:
$f = ByteFunction$
Algorithm:
for $i$ = 0 to $sizeof$($f$) do
  if $f$[$i$] == $ret$ then
   //Add gadgets ending with i$^{th}$ byte
   $FindGadgets$($i$, $maxGadgetLength$, $gadgetList$)
  end if
end for
```

Function $FindNewGadgets$ looks for all bytes with specific instruction codes (*ret* in this example) in machine bytes forming function f. If specific byte sequence has been found all byte sequences ending by this instruction are disassembled (backward disassembly). Disassembly is considered being successful if the last byte of disassembled instruction sequence is byte [i]. If disassembly successful, the disassembled instruction sequence is added as a gadget into $gadgetList$.

"Overlapped code generator" works on function-based level following next steps:

a) For very first function in compilation module the code generated as usual using a normal LLVM codegenerator, however no new .CODE section is created for each function to disable function-level linkage and to enable cross-function gadgets. For the same purpose alignment bytes are not inserted between functions.

b) Inside every generated function new gadgets are discovered and added to $gadgetList$.

c) For every gadget added this way it's LLVM representation pattern is being created and added to instruction list to enable this gadget used as a normal instruction in the every case suitable.

d) Finally instruction selector priorities are being manupulated to force instruction selector choose gadget type instructions over ordinary ones.

The greedy approach is used while inserting gadgets into newly generated code: if we can insert longer gadgets we continue adding first suitable instruction into gadgets as much as possible. Such approach could potentially lead to miss of longer gadgets, however, experiments does not show big loss of the criterion (1) value, while avoiding of exhaustive search is very important. As soon as we can add instructions to match our gadget no more, we completely remove generated gadget code replacing it with call or jump to gadget found. The overview of the algorithm is provided on the Fig. 3.

This work is licensed under the Creative Commons Attribution License.

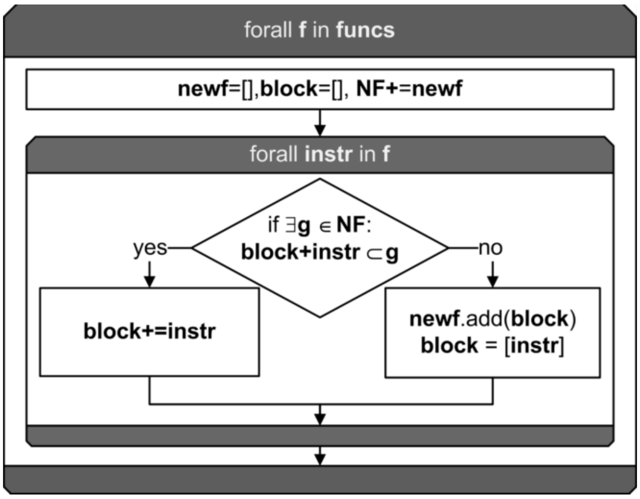


Fig. 3 The "Overlapped code generator" algorithm overview

Unfortunately, the existing LLVM structure was not suitable to implement "Overlapped code generator" algorithm. It order to increase number of gadgets the modification of LLVM pipeline showed on Fig. 4 have been implemented. Unfortunately current implementation of LLVM pipeline modification is not optimal and quite slow.

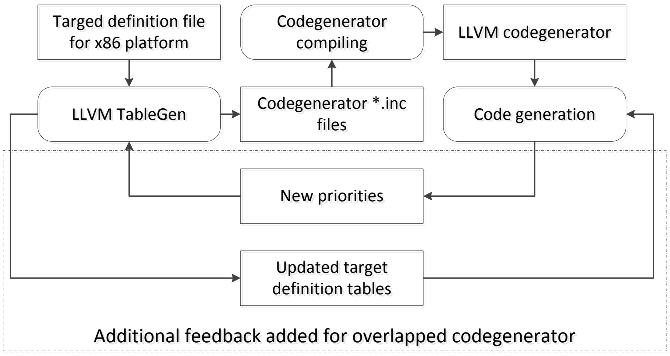


Fig. 4 Modification of LLVM pipeline for purpose of overlapped code generation

Having the aforementioned approach in mind it is possible to calculate time complexity of the approach. Disassembly of the size limited sequence takes O(1), the gadget list creation – $O(n)$. Insertion gadgets into

Having the aforementioned approach in mind it is possible to calculate time complexity of the approach. Disassembly of the size limited sequence takes O(1), the gadget list creation – $O(n)$. Insertion gadgets into the code – $O(n^2)$. Therefore in the worst case the total time complexity of all actions performed is $O(n^2)$.

V. PRACTICAL IMPLEMENTATION EVALUATION

The proposed approach has been evaluated using LLVM stress test kit. More than 1000 different programs has been generated and compiled using the standard LLVM code generator and the our code generator enhanced with approach proposed in this paper. The results are shown on Fig. 5. Value of criterion (1) here is the average value for all sample programs compiled.

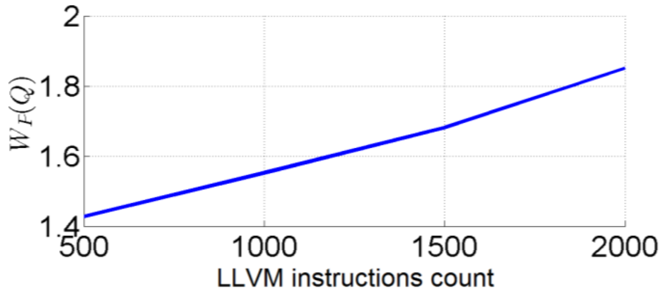


Fig. 5 Dependence of $W_P(Q)$ from compilation module size (d = 1)

For d = 1 (Fig. 5) we virtually prefer neither size of the program nor amount of instruction bytes being overlapped. Fig. 5 demonstrates with such choice of value *d*, that the quality of the code produced by the proposed code generator gradually increase with the increase of the amount of the code being compiled. This is the expected result because the more LLVM instruction the proposed code generator has the more probable is to discover gadget in the code already compiled and more versatile gadgets discovered are. However the aggressiveness of gadgets usage is limited by the size of output data considerations. The exact data is shown in table 1.

Table 1. Dependence of $W_P(Q)$ from the compilation module size.

Size	Avg. $W_P(Q)$	σ	Max. $W_P(Q)$	Min. $W_P(Q)$
500	1.428	0.0858	2.862	1.214
1000	1.608	0.0317	2.106	1.387
1500	1.735	0.0292	2.247	1.514
2000	1.850	0.0485	2.406	1.610

According to Callberg [6] it is important to mention to have performance of the obfuscated code measured compared to clear machine text versions.

To perform such tests each function has been called 100 000 times on Intel Core i7 2600K with thread and process affinity set and with power management disabled to minimize measurements fluctuations. Three different algorithms were tested: sine calculations using Tylor series, iterative factorial calculation and Fibonacci series (Fig.5).

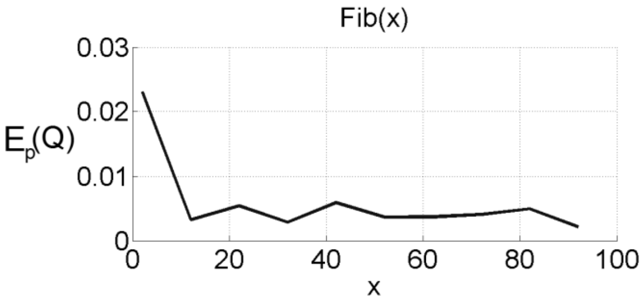


Fig. 6 Compiled program size reduction

The following marker values were calculated to estimate performance impact of overlapped code:

- $E_P(Q) = \frac{E(Q)-E(P)}{E(P)}$, where $E(P) -$ CPU cycles required to execute program P and

This work is licensed under the Creative Commons Attribution License.

- $T_P(Q) = \frac{T(Q)-T(P)}{T(P)}$, where $T(P)$ – time is seconds required to execute program P.

The full results are provided in Table 2:

Table 2. Performance of the overlapped code.

Test	*T(P)*,sec	*T(Q)*,sec	$T_P(Q) * 100\%$	$E_P(Q) * 100\%$
Sine	36.149	36.044	-0.29	+0.07
Factorial	3.334	4.477	+4.2	+4.5
Fibonacci	3.211	3.301	+2.8	+2.7

Table 2 demonstrates the obfuscated code sometimes executes *faster*, rather than original code, however such effect is unreliable. Anyway proposed approach does not impose large performance drawback. The “optimization” can be explained through overall program size reduction and therefore better CPU cache performance.

It is noteworthy to tell that in some cases proposed approach was able to produce code (Q) better than normal code produce by compiler (P) not only in terms of criterion (1) but in terms of the size in bytes too. This result was not intentionally pursued and appeared as a positive side effect demonstrated on Fig. 7. Fig. 7 shows the average reduction of the compiled program size for about 4% is unrelated to the size of the program being compiled. While the whole reduction is not large and depends on the actual code, it still worth to save about 700 bytes for 19Kbytes (roughly corresponds 1000 LLVM instruction program) of the compiled code.

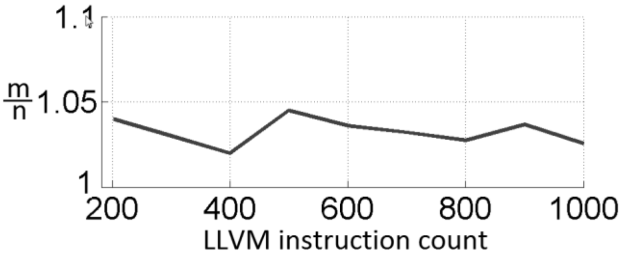


Fig. 7 Compiled program size reduction

Unfortunately current algorithm cannot guarantee a specific instruction to be overlapped, only some probability of such overlap. Tables 3 and 4 demonstrates the more code we have in compilation module the more gadgets we would find and better result we able to produce:

Table 3. Average overlap of the program *Q*.

Size	Avg. *A(Q)*	Max. *A(Q)*	Min. *A(Q)*
500	1.24	1.31	1.18
1000	1.28	1.41	1.21
1500	1.30	1.41	1.24
2000	1.33	1.49	1.25

VI. FUTURE WORK

The approach proposed by Joshua Mason [7] look like the most prominent way to improve criterion (1) and make the better overlapped code. Since we are interested in increase of criterion (1) we can use Viterbi algorithm [8] to traverse our collection of gadgets in conjunction with hidden Markov model to reconstruct most probable sequence of states used in HMM. Where each function being encoded in Markov model, whose states consist of unknown parameters (most suitable gadgets or ordinary glue instructions in our case) and known parameters (list of gadgets we are already have).

Such approach would allow us to avoid using greedy approach and has prominent potential to increase quality of overlapped code.

Usage of proposed approach for compilation of size critical code for SOCs and microcontrollers is a one of further research goals and can be further improved.

REFERENCES

[1] Vlad Dudau, Windows 8.1 activation has been bypassed, 2013, URL: http://www.neowin.net/news/windows-81-can-now-be-activated-with-kms-workaround-tool (accesses April 11 2014)

[2] Practical Reverse Engineering Tutorial - Cracking Winrar, 2011, URL: http://www.hackingalert.net/2011/09/practical-reverse-engineering-tutorial.html (accesses April 12 2014)

[3] Bruce Schneier, Applied cryptography (2nd ed.): protocols, algorithms, and source code in C, John Wiley & Sons, Inc., New York, NY, 1995

[4] Ryan Roemer , Erik Buchanan , Hovav Shacham , Stefan Savage, Return-Oriented Programming: Systems, Languages, and Applications, ACM Transactions on Information and System Security (TISSEC), v.15 n.1, p.1-34, March 2012 .

[5] Ralf Hund , Thorsten Holz , Felix C. Freiling, Return-oriented rootkits: bypassing kernel code integrity protection mechanisms, Proceedings of the 18th conference on USENIX security symposium, p.383-398, August 10-14, 2009, Montreal, Canada.

[6] Christian Collberg, Clark Thomborson, Douglas Low, A Taxonomy of Obfuscating Transformations, Technical report 148, Department of Cumputer Science, University of Auckland, July 1997.

[7] Joshua Mason , Sam Small , Fabian Monrose , Greg MacManus, English shellcode, Proceedings of the 16th ACM conference on Computer and communications security, November 09-13, 2009, Chicago, Illinois, USA .

[8] A. J. Viterbi. Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm. IEEE Transactions on Information Theory, 13(2):260--269, April 1967

This work is licensed under the Creative Commons Attribution License.