Trading Off a Vulnerability: Does Software Obfuscation Increase the Risk of ROP Attacks

Harshvardhan P. Joshi, Aravindhan Dhanasekaran and Rudra Dutta

Department of Computer Science, North Carolina State University
Raleigh, NC 27695-8206, USA
Email: {hpjoshi; adhanas; rdutta}@ncsu.edu

Received 13 February 2016; Accepted 25 February 2016;
Publication October 2015

Abstract

Software obfuscation is a commonly used technique to protect software, especially against reverse-engineering attacks. It is a form of security through obscurity and is commonly used for intellectual property and Digital Rights Management protection. However, this increase of security may come at the expense of increased vulnerabilities in another direction, hitherto unsuspected. In this paper, we propose and investigate the hypothesis that some of the most popular obfuscation techniques, including changing the control flow graph and substituting simpler instruction sequences with complex instructions, may make the obfuscated binary more vulnerable to Return-Oriented Programming (ROP) based attacks. ROP is a comparatively recent technique used to exploit buffer-overflow vulnerabilities. We analyze the ROP gadgets present in both obfuscated and un-obfuscated versions of well known binaries. We show that the number of ROP gadgets in a binary significantly increase after certain obfuscations, and it can potentially make ROP-based exploits easier.

1 Introduction

As computers, and the software to utilize them, perform increasingly pervasive and critical functions in society, the security of software assumes ever-increasing importance. The sophistication of malicious actors in developing

Journal of Cyber Security, Vol. 4, 305–324.
doi: 10.13052/jcsm2245-1439.444

techniques to exploit vulnerabilities also continues to increase, requiring ongoing re-examination of existing software and security practices.

Software developers have two broad security concerns: (1) vulnerabilities that lead to exploitation of the software (e.g. buffer overflow); (2) reverse engineering (e.g. software or music piracy). The first concern (software exploitation) is based on the threat model where the software runs on a benign host, and the goal is to protect the software in order to protect the host system and the information the software has access to. An example of this threat model is an unprivileged attacker exploiting vulnerability in software to gain access to restricted information. The second concern (reverse engineering) however, is based on a very different threat model where the host or the privileged user is malicious and tries to subvert the restrictions set by the software developers. An example would be reverse engineering Digital Rights Management (DRM) protection software for music or movies piracy.

Some developers use static analysis tools to prevent vulnerabilities such as buffer overflow; however, many others rely on OS level protections such as Address Space Layout Randomization (ASLR) and Write or Execute Only (W$\oplus$X) pages [1, 2]. Protection against reverse engineering is generally achieved via software obfuscation tools and cannot leverage OS support.

Given the pervasiveness of W$\oplus$X in operating systems, attackers have begun using return-oriented programming. Return-Oriented Programming (ROP) is a technique through which an attacker can introduce changes to a program's control flow using many short code snippets, called gadgets, present in a program's address space [3]. ROP allows attackers to exploit buffer overflow vulnerabilities even when W$\oplus$X protection is enabled. The goal of the attacker using ROP is to exploit a vulnerability (buffer overflow) in client software in order to compromise the host system. Thus, it is a technique used by attackers in the software exploitation threat model.

Even the software that requires protection against reverse engineering should be secure against the software exploitation threat model in order to protect the benign hosts it may run on. However, some of the software obfuscation techniques used in practice may impact the vulnerability of the software to the software exploitation threat model. In other words, in their overriding concern to protect their software against the user (who may be, but is not typically, malicious), software developers may be unwittingly exposing the integrity of the users platform to sundry attackers, who are most certainly malicious. In this paper, we assert a potential impact of software obfuscation on the program's vulnerability to ROP based attacks, and do a preliminary study comparing the impact of two different obfuscating tools.

The rest of the paper is organized as follows. Section 2 gives some background on ROP and code obfuscation techniques. Section 3 provides a brief overview of our work, which outlines the design and implementation details, followed by evaluation methods and results in Section 4. Section 5 includes related work and finally the concluding remarks are in Section 6.

2 Context

2.1 Software Obfuscation Techniques

Software obfuscation is achieved by a sequence of transformations on the code in order to obscure the purpose of the code while maintaining its original behavior. There have been several obfuscating transformations identified in literature [4, 5]. These include inserting opaque predicates which are difficult to evaluate at compile time; inserting dead or irrelevant code; cloning, splitting or merging functions; control flow flattening; etc. More details about some of these obfuscation techniques are discussed in Section 3.2.1.

2.2 Return-Oriented Programming

Return-Oriented Programming is made possible by making use of small instruction sequences called "gadgets" [3]. Each gadget ends with a 'return' or 'jump' instruction, which is used to chain together several such gadgets to alter the program's behavior. Each gadget might perform a different operation, e.g. a load and add operation followed by a jump. A return-oriented program consists of several gadgets arranged carefully to meet the attacker's goal. These gadgets must be in the memory, in the address space of the executing program or in the address space of a library used by the program. On the whole, return-oriented programming can be viewed as the generalized notion of return-to-libc [6] types of attacks. In W$\oplus$X model, a memory page is either writable or executable, but not both, which prevents all types of code injection attacks. In return-oriented programming attacks however, the attacker does not inject any code and just alters the program execution by executing already existing code in an arbitrary fashion. While ASLR can be used to mitigate ROP attacks, it can be circumvented if information about the address layout is leaked [7].

An ROP based attack depends on stringing together gadgets in the code in order to perform arbitrary action. In [8], a catalog of x86 gadgets are identified for a Turing-complete ROP. A software with a larger number of gadgets from

this catalog is likely to be easier to exploit using ROP by an attacker, assuming they find an entry through vulnerabilities like buffer overflow.

In this paper, we hypothesize that *software obfuscation makes it more susceptible to ROP attacks*. This is based on our observation that obfuscation tools add redundant code to the software, change control flows, add conditionals, etc. which can increase the number of gadgets in the binary.

We analyze a set of open source software binaries in terms of the catalog of gadgets they contain, and compare this with the gadgets found in obfuscated versions of the same software. Based on our experiments, we show that obfuscation significantly impacts the number of gadgets in a binary, which in turn makes an ROP attack easier.

3 Overview and Experimental Design

In order to study the susceptibility of a software to ROP attacks, we identify the catalog of gadgets in the code. It may seem that a software with a larger number of gadgets would be more susceptible to ROP attacks. However, there is limited advantage of repetitive gadgets, and variety is more important in order to chain them together and accomplish a useful task. Hence, we only consider unique gadgets in the code.

To evaluate our hypothesis, we find and compare gadgets in obfuscated and un-obfuscated versions of a large number of binaries. We use a set of open source software of different size and type including executables and libraries. We build the obfuscated version of the software for Linux using LLVM with the same flags and optimizations as used to build the un-obfuscated binaries, in order to make our analysis of both set of binaries comparable. LLVM [9] is a compiler infrastructure and toolchain that enables code analysis and optimizations for arbitrary languages. It is increasingly becoming popular both for academic research and in commercial products along with Clang, its native C/C++ compiler.

3.1 Binaries Selection

Since we believe that the increase in the size of binary due to obfuscation may impact its ROP susceptibility, we want to evaluate on binaries of different sizes. The source code of the software likely to be obfuscated in practice is not made openly available, as that defeats the purpose of obfuscation. However, we decided to use software that is security sensitive but openly available. This allows us to use well-known software for evaluation and also to build binaries with various levels of obfuscation for consistent comparison.

We use two sets of software, GNU Coreutils [10] and OpenSSL [11] as target binaries for our evaluation. We use OpenSSL, and specifically its libraries libssl and libcrypto as an example of commonly used security sensitive libraries.

GNU Coreutils include utilities like `cp, mv, ls, date,` etc. which are included in nearly all Linux distributions. Some of these utilities can also be considered to be part of these Linux distribution's trusted computing base (TCB) since they are frequently run as root. The Coreutils package includes 106 utilities of varying size and thus provides us with a good base for our evaluation. For more detailed analysis we select a subset of these utilities again based on their security sensitiveness, and their potential to do harm. These include `link, chroot, shred, touch, date, cp.` We select touch and date due to recent vulnerability CVE-2014-9471 that may allow arbitrary code execution or denial of service attack. We select shred due to expectation of secure removal of files, while the others (`link, cp, chroot`) for their potential for misuse.

3.2 Obfuscation Tools

Since C/C++ is most commonly used in practice among languages vulnerable to buffer overflow and ROP attacks, we focus our attention on C/C++ obfuscators. While there are several software obfuscation tools available for C/C++, the majority are neither open-source nor free. The free obfuscation tools for C/C++ include the Tigress [12] and Obfuscator-LLVM [13].

3.2.1 Obfuscator-LLVM

We chose to use Obfuscator-LLVM as it is an open source project to build an obfuscator for LLVM tool-chain and is currently maintained. It works on LLVM's intermediate representation (IR) level, and so it can take advantage of LLVM's front-end and back-end which support many languages including C and C++, and architectures such as x86, arm, mips, etc. Since, this is a fairly new project, it has limited obfuscating transforms available. These are:

- Instruction Substitution (SUB): This obfuscation technique relies on substituting one set of instructions with another set of instructions while maintaining the same functionality.
- Control Flow Flattening (FLA): This obfuscation flattens the control flow graph of the program so that the structure of the program cannot be easily understood by static analysis like disassembly. This is achieved by identifying and moving blocks in a function which are at nested levels,

next to each other. The selection of control flow to a particular block is done using a switch statement and a control variable that keeps track of the state of the program.
- Bogus Control Flow (BCF): This obfuscation modifies a function call graph by adding a basic block before the current basic block. This new basic block contains an opaque predicate and then makes a conditional jump to the original basic block.

3.2.2 The Tigress Obfuscator

The Tigress is a C language obfuscator and virtualizer that offers several defenses against both static and dynamic reverse engineering [12]. Designed for educational purposes, it is a source-to-source obfuscator, applying obfuscating transforms on the C source code to generate obfuscated C source which can be examined for effects of transformations. Some of the important transforms offered by the Tigress include:

- Virtualize: Turns a function into an interpreter with a virtual instruction set, a bytecode array, and virtual program counter and stack pointer.
- Jit: Turn a function into a sequence of instructions that dynamically builds up the function at runtime.
- Flatten: Flatten a function.
- Merge: Merges two or more functions.
- RndArgs: Randomize and add bogus arguments to a function.
- AddOpaque: Add opaque predicates to split up control-flow.
- EncodeLiterals: Replace literal integers and strings with less obvious expressions.
- InitBranchFuns: Create branch functions.
- AntiBranchAnalysis: Replace branches with other constructs.
- AntiTaintAnalysis: Insert implicit flow such that dynamic taint analysis becomes less precise.
- AntiAliasAnalysis: Replace direct function calls with indirect ones making alias analysis less precise.

3.3 ROP Gadget Analysis

A few algorithms for discovering gadgets in code have been described in literature, including in [3] and [8]. We use an ROP gadget finding tool, ROPgadget [14], to identify potential gadgets in binary. A list of all unique gadgets found in the un-obfuscated binary is created, and then compared against the list from obfuscated binary.

Once we identify gadgets in a binary, we also categorize these gadgets based on the instructions they contain and the function they can serve. The categories are

- Memory: These are instructions and gadgets that facilitate load and store operations from/to memory and registers.
- Arithmetic: These are gadgets that contain arithmetic instructions and which can be used to perform operations such as add, sub, neg, etc.
- Logic: These are gadgets that contain instructions or can be used to perform operations such as and, or, xor, shift, rotate, etc.
- Control: These are gadgets that can control the flow, such as conditional or unconditional jumps.
- Other: These are gadgets that could not be categorized in one of the above categories.

4 Results and Discussion

As discussed in Section 3, we perform our evaluations on binaries found in the GNU Coreutils package as well as the libraries in OpenSSL. The GNU Coreutils and OpenSSL packages are built using LLVM 3.4, and the same compiler is used to build the obfuscated versions of those binaries. Both Obfuscator-LLVM and Tigress obfuscators are used for evaluation, but the use of Tigress obfuscator is limited to a few binaries in Coreutils, used for more detailed analysis. This is because Tigress is a source-to-source obfuscator and is not able to handle certain compile-time macros commonly used in Coreutils source code, requiring manual modification of the source code.

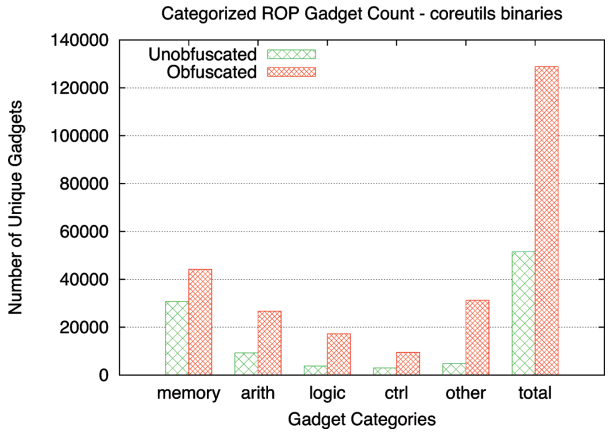
We built different versions of the obfuscated binaries, for different types of obfuscations supported by Obfuscator-LLVM and discussed in Section 3.2, and a binary with all the obfuscations. In our discussion, unless otherwise specified, an obfuscated binary refers to a binary built with all the supported obfuscations by Obfuscator-LLVM. For Tigress obfuscator, as default we use transforms that are similar to the ones offered by Obfuscator-LLVM, namely Virtualize, Init/Add/UpdateOpaque, and Flatten transforms. These transforms are applied on all functions in the source file. Since Virtualize and JIT are the two main transforms of Tigress, we also analyze with JIT applied to the main routine for each binary.

While the OpenSSL libraries libssl.a and libcrypto.a are a collection of object files, we created a small test program and then statically linked the objects from both libraries with this program to create a binary which we can

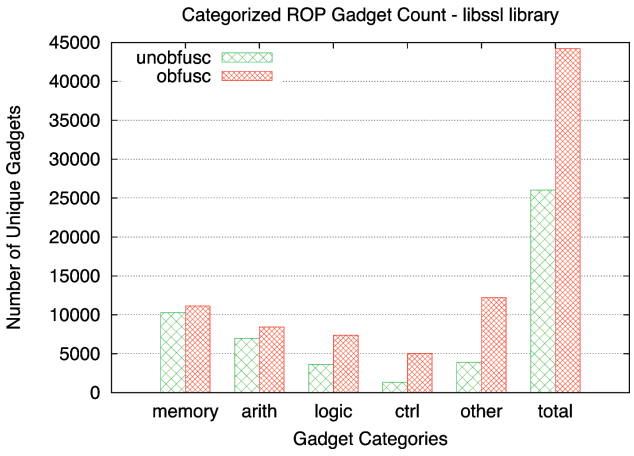
use for evaluation. We refer to this combined binary of objects from libssl and libcrypto as libssl.

4.1 Impact of Obfuscation on Gadget Count

Figure 1a shows the categorized as well as the total gadget counts aggregated for all the binaries in Coreutils with and without obfuscation. We can see that obfuscation more than doubles the total gadgets in the binaries, however the impact varies across categories. A similar graph for libssl is shown in Figure 1b, where the total increase is more moderate. We believe



(a) Gadget count for all binaries in `Coreutils`



(b) Gadget count for `libssl` library

Figure 1 Gadget count for various binaries with Obfuscator-LLVM.

this was because the obfuscator we are using was not able to handle some of the more complicated and large source files in libssl, and hence abandoned obfuscations on them. However, both these graphs show that obfuscation significantly increases the number of gadgets in a binary.

4.2 Impact of Obfuscation Type

Table 1 shows the binary sizes for some of the utilities in GNU Coreutils along with their size with different obfuscations. From Obfuscator-LLVM, the instruction substitution obfuscation is referred to as sub-obfusc (or SUB), bogus control flow as bcf-obfusc (or BCF) and control flow flattening as fla-obfusc (or FLA), and details of these obfuscations are discussed in Section 3.2.1. As expected, substitution does not have as big an impact on binary size as the other two. The size increases in binaries with full obfuscation ranged from about $3\times$ to $5\times$. The effects of Virtualize transform of the Tigress obfuscator is similar, with $2\times$ to $4\times$ increase in binary size. The JIT transform on the other hand, applied only to the main routine, adds about 2 to 2.5 MB to the binary size due to the 'jitter' code needed to dynamically generate the function at execution-time. For smaller binaries like link, this can mean a size increase of more than $45\times$.

The impact of different obfuscation transforms of Obfuscator-LLVM on gadgets is shown in Figure 2. It is clear that while overall the impact of BCF and FLA are higher than SUB, their impact within specific categories of gadgets varies. Since aggregating across all of Coreutils binaries is difficult to analyze, we do more detailed analysis on a few selected binaries in Coreutils as identified in Section 3.1. For these binaries, we also use the Tigress obfuscator with Virtualize transform, such that it is comparable to the obfuscations of Obfuscator-LLVM. The graphs for `cp` and `link` are shown in Figures 3a and 3b. On very small binaries like `link` the Tigress Virtualize obfuscations only marginally increase the gadget count, while on larger binaries like cp the

Table 1 Binaries size (in KBs) with different obfuscation transforms

			Obfuscator-LLVM			Tigress	
Binaries	Original	SUB	BCF	FLA	All	Virtualize	JIT
link	44	48	81	92	149	87	2040
chroot	59	59	111	131	223	182	2567
shred	85	97	169	185	313	405	2374
date	101	113	217	277	473	235	2890
cp	169	185	381	425	757	384	2614
coreutils	11264	9318	17408	20480	41984	–	–

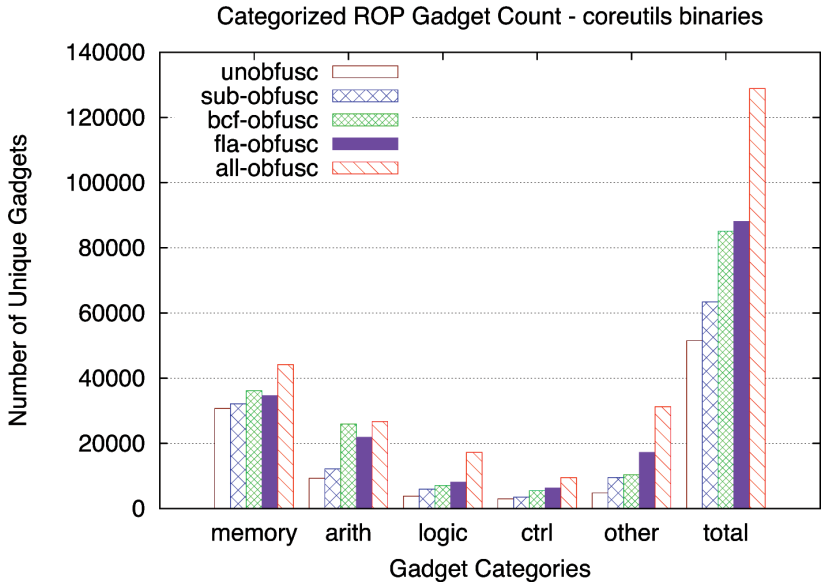
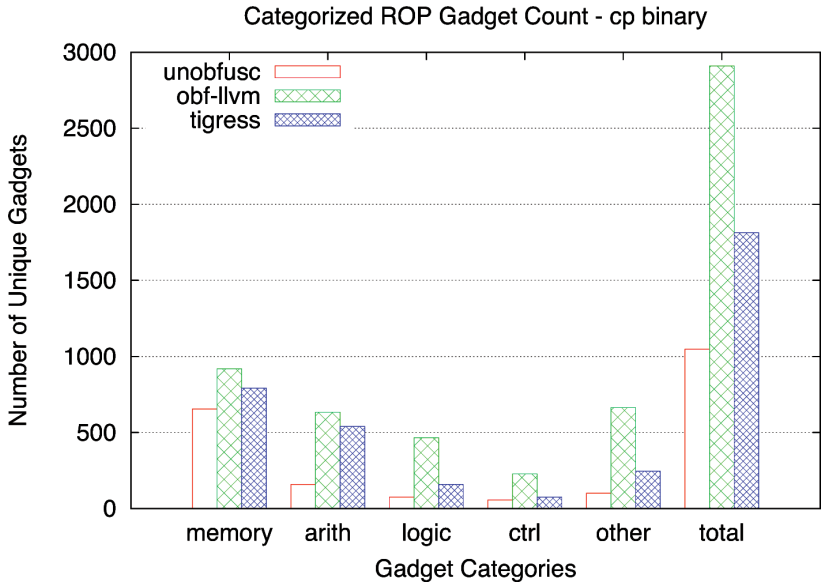


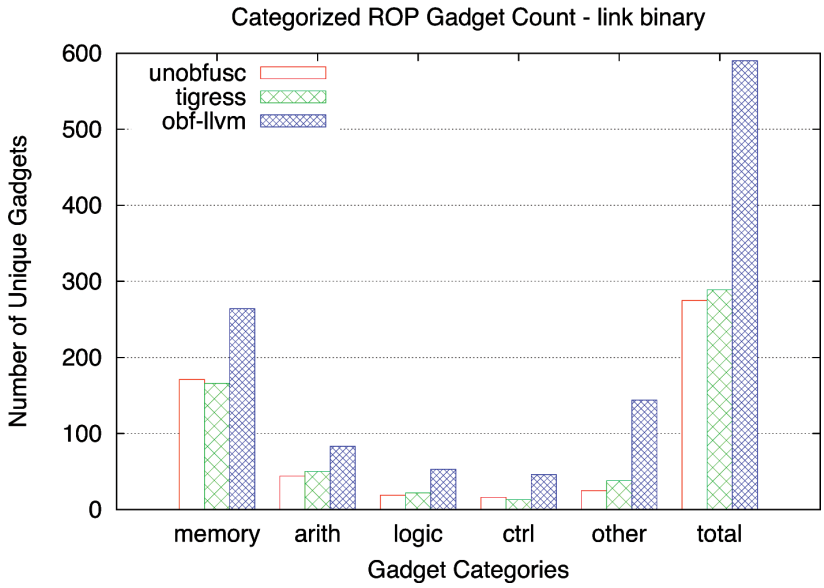
Figure 2 Different obfuscation transforms of Obfuscator-LLVM for all binaries in `Coreutils`.

increase is significant. On the other hand, Obfuscator-LLVM has big impact on the gadget counts for all binaries. We have not shown the gadget counts for Tigress JIT obfuscation, but for smaller binaries like *link*, the increase in gadget count can be more than 20×. The impact of JIT obfuscation are explored more in Section 4.3.

The two obfuscators generate considerably different binary images, and it is quite difficult to quantitatively evaluate and compare the effectiveness of each obfuscation through static analysis. Since obfuscation is expected to make recognizing original code difficult, one quantitative measure could be the correlation between binaries before and after applying obfuscating transforms. However, a high score on this measure can be achieved by inserting large amount of unused random instructions that is never called while keeping the original binary content. Such obfuscation is unlikely to be very effective in practice when faced with determined adversary. A more useful, but qualitative, measure could be the change in control flow graph of a function. In Figure 4 we show the flow graph of a single function, `is_root()` from `chroot`, in unobfuscated and obfuscated binaries. The simple flow graph of the unobfuscated function in Figure 4a is significantly obfuscated and complicated in Figures 4b and 4c. The effects of bogus control flow can

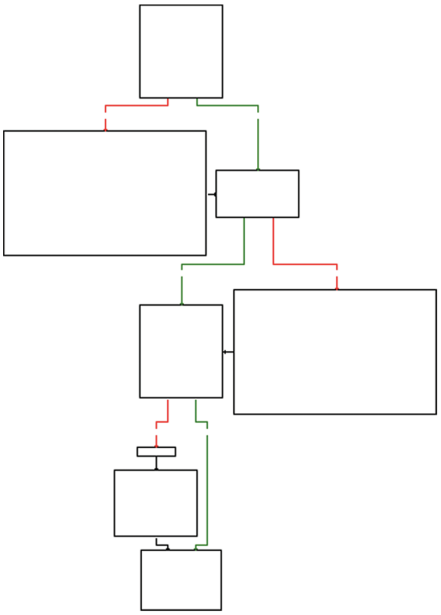


(a) Categorized gadget count for a large binary in Coreutils: `cp`

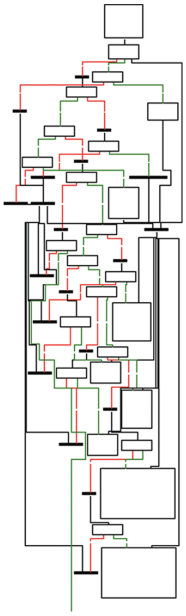


(b) Categorized gadget count for a small binary in Coreutils: `link`

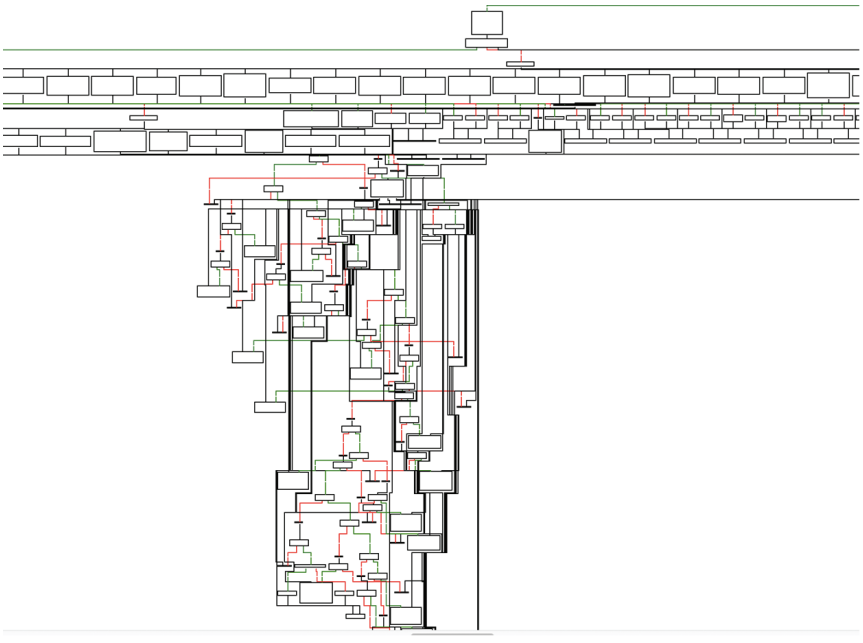
Figure 3 Impact of different obfuscations on gadget count for small and large binaries.



(a) Original unobfuscated flow graph



(b) Obfuscator-LLVM generated flow graph



(c) Tigress generated flow graph

Figure 4 Flow graphs for *is root*() function from *chroot* with different obfuscations.

be seen in both Figures 4b and 4c, while the top part of Figure 4c clearly shows the effects of flattening transformation and the resulting large block of `switch` statement.

While in this paper we focus on the static analysis for measuring the effectiveness of obfuscation, dynamic analysis may be able to provide more insight. In [15], a comparison of the effectiveness of obfuscation is done by analyzing the dynamic behavior of code to find similarities between obfuscated and unobfuscated code. This work also uses the same obfuscation tools, Obfuscator-LLVM and Tigress.

4.3 Impact on Gadget Categories

The impact of obfuscations on different types of gadgets is not evenly distributed. We have observed that in general binaries have plenty of memory gadgets but relatively few logic and control gadgets. The impact of obfuscation is more pronounced on these fewer logic and control gadgets than on memory gadgets. This is not only because of the smaller baseline, but also because of the characteristics of the obfuscations. Both, bogus control flow and control flow flattening obfuscations significantly increase control flow and logic related instructions like conditional and unconditional jumps. For memory gadgets, the increase with obfuscations is comparatively small as can be seen in Figure 5. However, for logic and control gadgets, the increase in numbers is quite high and can be more than 6x the number of gadgets in unobfuscated binaries. As we can see in Figures 6 and 7, small binaries like `link` have very few (nearly single digit) logic and control gadgets and obfuscation can push the number up 4 to 5 times. If we consider very complex dynamic obfuscation transform like Tigress JIT, then the increase can be up to 20 times. This can potentially impact the feasibility of ROP attack on that binary.

In addition to analyzing the number of gadgets, we looked at the actual gadgets for these binaries in order to do a qualitative analysis. We compared the gadgets found in each utility against their obfuscated binaries. There were some common gadgets between these binaries, usually at the lower memory address where the common preambles in ELF can be expected. However, the majority of the gadgets were very different in both sets of binaries and it was difficult to make any generalizations about them.

5 Related Work

Both obfuscation and ROP have been studied in literature previously, though not together.

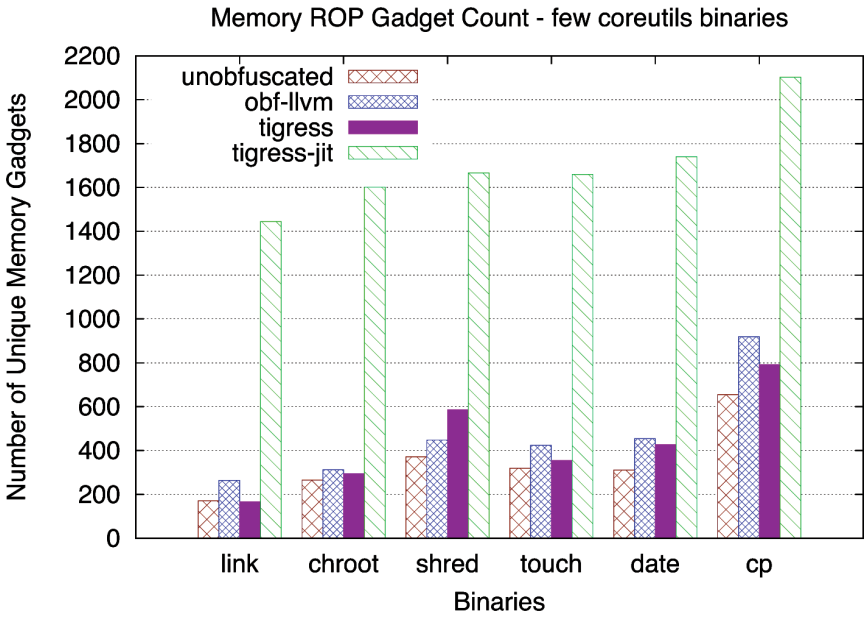


Figure 5 Memory gadget count for few 'coreutils' binaries.

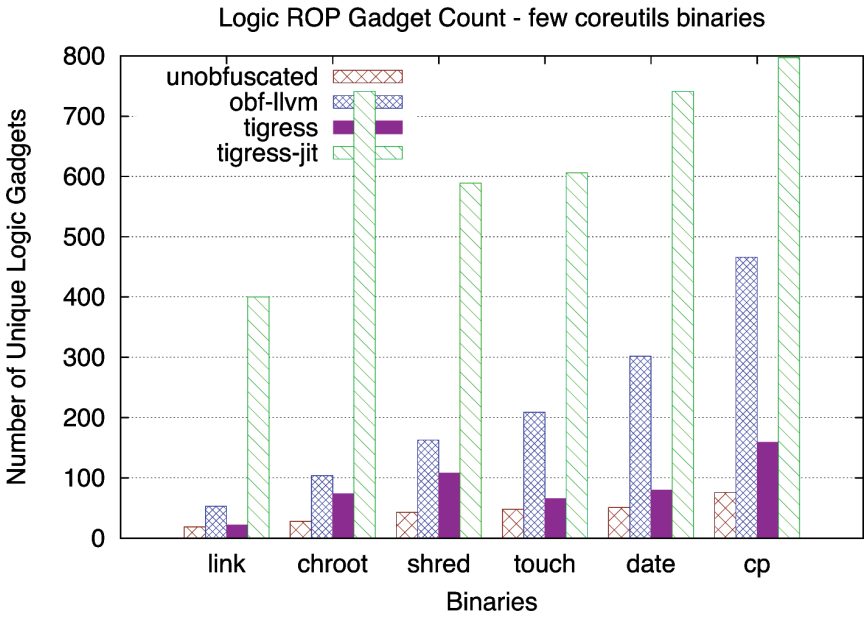


Figure 6 Logic gadget count for few 'coreutils' binaries.

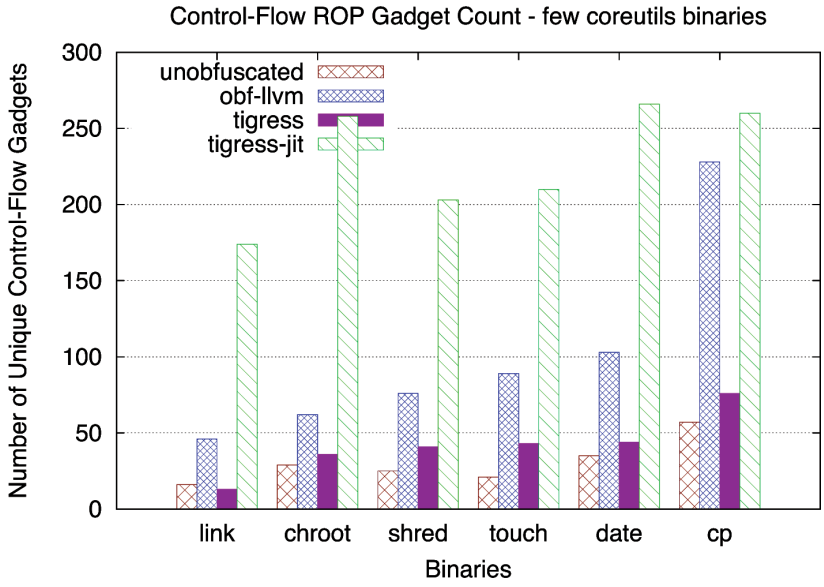


Figure 7 Control-flow gadget count for few 'coreutils' binaries.

In [16], the return-oriented programming is used for program steganography. This is a form of software obfuscation using return-oriented programming. In-place code randomization [17] is a technique that can be applied to third-party software to mitigate ROP attacks. It uses narrow-scope code transformations to randomize binaries and disrupt useful ROP gadgets. The mechanisms of this approach is similar to software obfuscation in the use of code transforms, however, the goal is not to obfuscate but to prevent occurrences of useful gadgets.

The attacks that fall under return-oriented programming paradigm are very broad, but still there are many defense mechanisms to mitigate or to prevent ROP based attacks. Abadi et al., [18] used control-flow integrity (CFI) to prevent stack based ROP attacks. kBouncer [19] is a light weight tool to prevent certain ROP attacks using hardware features and without requiring any modifications of the binary code. Another hardware based approach, ROPecker [20], prevents control flow based ROP attacks by examining the last branch taken registers, found in commodity processors. Carilini et al., [21] discuss three new ROP attacks that break existing CFI based defense mechanisms such as kBouncer and ROPecker. Schuster et al., analyze different defense mechanisms in [22] and show that with a little extra effort, it is possible to break ROPecker, kBouncer and ROPGuard.

However, to the best of our knowledge, the connection between obfuscation and ROP has not been studied or hypothesized in the literature previously.

6 Conclusion and Future Work

Software developers concerned about reverse-engineering attacks and piracy commonly deploy software obfuscation as a defense. The users of software however are more concerned about vulnerabilities in the software that may compromise their system. We show that there is a possibility of conflict between these two security goals if software obfuscation is used. We have shown that software obfuscation can significantly increase the number of gadgets (1.5× to 3×) in a binary. We have also shown that for logic and control flow related gadgets, the increase is much higher (up to 6×). For certain, especially smaller, binaries which have very small number of logic and control gadgets this increase can potentially make ROP attacks feasible, or at best, easier. Our evaluation with two openly available obfuscation tools show that certain obfuscation techniques may be preferable to reduce the impact on number of gadgets in a binary.

Our study is an early one, and is far from complete. In the future, this work can be extended by comparing the effectiveness of obfuscation tools and transforms quantitatively, along with comparing their impact on the number of gadgets. Such studies can be conducted on a much wider range of software for a broader range of platforms. It would also be interesting to implement an ROP-based attack on an obfuscated binary that would not be feasible without obfuscation. Finally, an allied field of research would consist of developing effective techniques for obfuscation that do not adversely impact the susceptibility to ROP attacks.

Acknowledgements

The work presented in this paper was made possible in part by grants CNS-1111088 and 1318594 from the US National Science Foundation.

References

[1] Bhatkar, S., DuVarney, D. C., and Sekar, R. (2003). "Address obfuscation: an efficient approach to combat a broad range of memory error exploits," in *Proceedings of USENIX Security*, Vol. 3, 105–120.

[2] Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., and Boneh, D. (2004). "On the effectiveness of address–space randomization," in *Proceedings of the 11th ACM Conference on Computer and Communications Security*. ACM, New York, NY, 298–307.

[3] Shacham, H. (2007). "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, *ser. CCS '07* (New York, NY: ACM), 552–561. Available at: http://doi.acm.org.prox.lib.ncsu.edu/10.1145/1315245.1315313

[4] Nagra, J., and Collberg, C. (2009). *Surreptitious software: obfuscation, watermarking, and tamperproofing for software protection*. Pearson Education, Upper Saddle River, NJ.

[5] Collberg, C. S., and Thomborson, C. (2002) Watermarking, tamper-proofing, and obfuscation tools for software protection. *IEEE Transact. Softw. Eng.*, 28, 735–746.

[6] Wojtczuk, R. (2001). The advanced return-into-lib(c) exploits: PaX case study. *Phrack Mag., 0x0b, Phile# 0x04 of 0x0e*.

[7] Snow, K. Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., and Sadeghi, A.-R. (2013). Just-in-time code reuse: on the effectiveness of fine-grained address space layout randomization," in *2013 IEEE Symposium on Security and Privacy (SP),* , 574–588.

[8] Roemer, R., Buchanan, E., Shacham, H., and Savage, S. (2012). "Return-oriented programming: Systems, languages, and applications," in *ACM Transactions on Information and System Security (TISSEC)*, Vol. 15, 2.

[9] Lattner, C., and Adve, V. (2004). "Llvm: a compilation framework for lifelong program analysis and transformation," in IEEE *International Symposium on Code Generation and Optimization, 2004 (CGO 2004)*, 75–86.

[10] GNU Coreutils. Available at: https://www.gnu.org/software/coreutils/

[11] OpenSSL. *OpenSSL: Cryptography and SSL/TLS Toolkit*. Available at: https://www.openssl.org/

[12] The Tigress C Diversifier/Obfuscator. Available at: http://tigress.cs.arizona.edu/index.html.

[13] Obfuscator LLVM. Available at: https://github.com/obfuscatorllvm/obfuscator/wiki

[14] ROPgadget. Available at: http://shell-storm.org/project/ROPgadget/

[15] Scrinzi, F. (2015). *Behavioral Analysis of Obfuscated Code*. [Online]. Available at: http://essay.utwente.nl/67522/

[16] Lu, K., Xiong, S., and Gao, D. (2014). "Ropsteg: program steganography with return oriented programming," in *Proceedings of the 4th ACM*

Conference on Data and Application Security and Privacy (New York, NY: ACM), 265–272.

[17] Pappas, V., Polychronakis, M., and Keromytis, A. D. (2012). "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *IEEE Symposium on Security and Privacy (SP), 2012*, 601–615.

[18] Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. (2005). "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security* (New York, NY: ACM), 340–353.

[19] Pappas, V., Polychronakis, M., and Keromytis, A. D. (2013). "Transparent rop exploit mitigation using indirect branch tracing." in *USENIX Security*, 447–462.

[20] Cheng, Y., Zhou, Z., Yu, M., Ding, X., and Deng, R. H. (2014). "Ropecker: a generic and practical approach for defending against rop attacks," in *Symposium on Network and Distributed System Security (NDSS)*.

[21] Carlini, N., and Wagner, D. (2014). "Rop is still dangerous: Breaking modern defenses," in *USENIX Security Symposium*.

[22] Schuster, F., Tendyck, T., Pewny, J., Maaß, A., Steegmanns, M., Contag, M., and Holz, T. (2014). "Evaluating the effectiveness of current anti-rop defenses," in *Research in Attacks, Intrusions and Defenses* (Berlin: Springer), 88–108.

Biographies

H. P. Joshi received his B.E. and M.B.A. degrees from Gujarat University, Ahmedabad, India in 2000 and 2002 respectively. He also received a Master of Science degree from North Carolina State University, USA in 2006. After working in industry for several years, he is currently pursuing a Ph.D. degree in Computer Science at North Carolina State University. His primary research interest is in networking and security.

A. Dhanasekaran received his B.Tech. in Information Technology from Anna University, India in 2009. He worked in industry before earning a Master of Science in Computer Science from North Carolina State University, USA in 2015. He currently works as a software engineer at Cisco Systems, Inc.

R. Dutta was born in Kolkata, India, in 1968. After completing elementary schooling in Kolkata, he received a B.E. in Electrical Engineering from Jadavpur University, Kolkata, India, in 1991, a M.E. in Systems Science and Automation from Indian Institute of Science, Bangalore, India in 1993, and a Ph.D. in Computer Science from North Carolina State University, Raleigh, USA, in 2001. From 1993 to 1997 he worked for IBM as a software developer and programmer in various networking related projects. He has been employed from 2001–2007 as Assistant Professor, from 2007–2013 as Associate Professor, and since 2013 as Professor, in the department of Computer Science at the North Carolina State University, Raleigh. During the summer of 2005, he was a visiting researcher at the IBM WebSphere Technology Institute in RTP, NC, USA. His current research interests focus on design and performance optimization of large networking systems, Internet architecture, wireless networks, and network analytics.