ROPfuscator: Robust Obfuscation with ROP

Giulio De Pasquale[1] Fukutomo Nakanishi[2] Daniele Ferla[3] Lorenzo Cavallaro[4]

[1]King's College London
[2]Toshiba Corporation
[3]Università di Bologna
[4]University College London

TOSHIBA

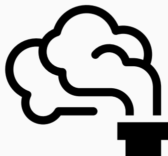
Obfuscation 101

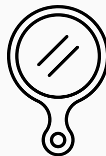
Definition

to **obfuscate**

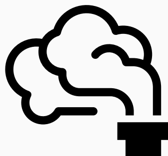
verb

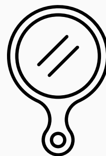
- make obscure, unclear, or unintelligible
- to make something less clear and harder to understand, **especially intentionally**





Obfuscation can be applied at different stages of the software development process, targeting source code or compiled binaries





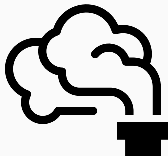
Obfuscation can be applied at different stages of the software development process, targeting source code or compiled binaries

Obfuscators

- Tigress
- VMProtect
- Loki
- MOVfuscator

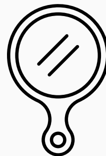
Applications and Tools

Obfuscation can be applied at different stages of the software development process, targeting source code or compiled binaries



Obfuscators

- Tigress
- VMProtect
- Loki
- MOVfuscator



Use Cases

- Digital Rights Management (DRM)
- Software Licensing
- Intellectual Property Protection
- Anti-Tampering

Obfuscation Techniques

Some of the most common obfuscation techniques include:

- Dead code insertion
- Control flow flattening
- Code transposition
- Virtualization
- Mixed-Boolean Arithmetics (MBA)

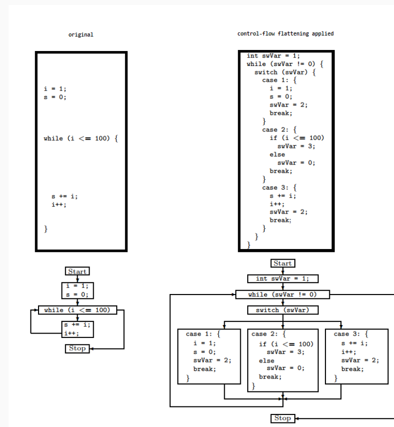


Figure 1: Control flow flattening [1]

[1]From Obfuscating C++ Programs via Control Flow Flattening, Laszlo and Kiss, SPLST 2007

Introducing ROPfuscator

Return oriented programming (ROP) is a binary exploitation technique which reuses existing code in programs to execute arbitrary algorithms

Code written using ROP is called **ROP chain** and it is composed of *gadgets*, a sequence of instructions followed by a final return instruction

When ROP is used to exploit a program, the ROP chains are usually injected through a buffer overflow vulnerability to control the instruction pointer

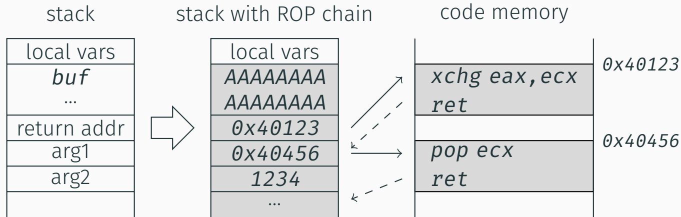


Figure 2: ROP chain execution on the stack

ROP is inherently harder to analyze than sequential code as it is threaded code [1] making it a great candidate for CFG disruption.

Hence the question:

Can ROP be repurposed as a defensive technique?

[1] Bell, Communications of the ACM, 1973

ROPfuscator[2], a fine-grained obfuscation framework for C/C++ programs using ROP

[2] *https://github.com/ropfuscator/ropfuscator*

Figure 3: An architectural view of ROPfuscator

Architectural Overview

Figure 3: An architectural view of ROPfuscator

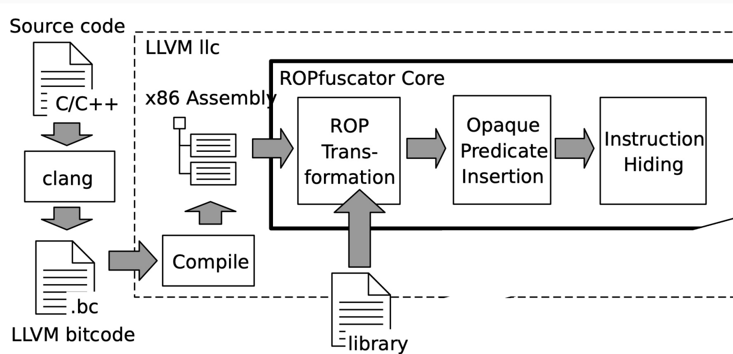


Figure 3: An architectural view of ROPfuscator

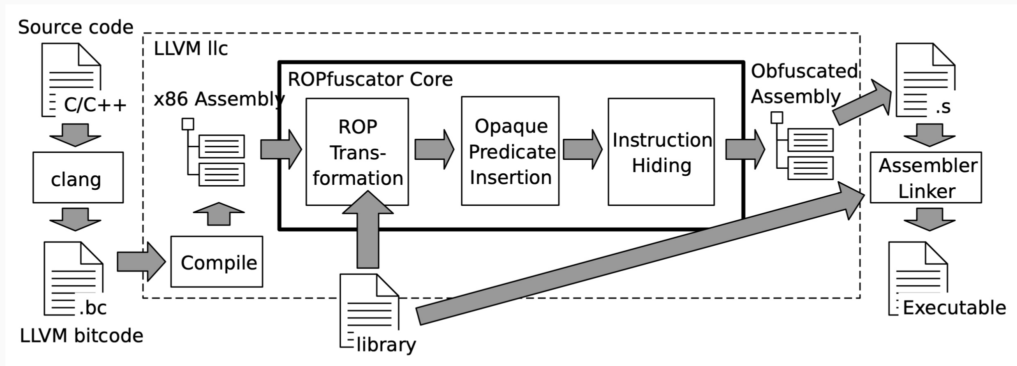
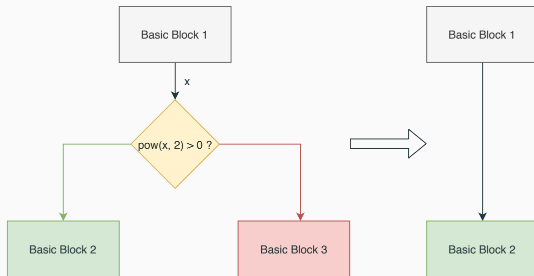


Figure 3: An architectural view of ROPfuscator

Opaque Predicates

An **opaque predicate** is a conditional statement that always evaluates to a specific value and deliberately designed to be difficult to analyze



These predicates are frequently constructed using NP-Hard problems to ensure the intractability of their analysis

Instruction Hiding

Instruction Hiding decomposes one or more instructions into smaller parts, rearranging them non-sequentially across neighboring locations

Original code	Hidden code inserted	Dummy code inserted
mov edx, 0xda598211	*mov edx, 0xda598211*	*mov edx, 0xda598211*
mul edx	*mul edx*	*mul edx*
(Insertion Point)	mov ecx, 123	add [esp], 456
cmp eax, 0x40527619	*cmp eax, 0x40527619*	*cmp eax, 0x40527619*
setne al	*setne al*	*setne al*
cmp edx, 0xde447238	*cmp edx, 0xde447238*	*cmp edx, 0xde447238*
setne dl	*setne dl*	*setne dl*

By applying this technique only to instructions that do not impact the calculation's outcome, instruction hiding makes it more difficult for attackers to understand the program's functionality and control flow

Experimental Results

We considered four different adversarial scenarios of increasing difficulties and realism:

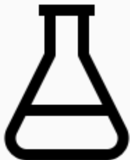
- **Threat A:** ROP-agnostic Static Analysis
- **Threat B:** Static ROP Chain Analysis
- **Threat C:** Dynamic Symbolic Execution (DSE)
- **Threat D:** Dynamic ROP Chain Analysis

- **RQ1: Completeness**. What is the highest code coverage that ROPfuscator can attain?
- **RQ2: Performance**. To what extent ROPfuscator affects performance?
- **RQ3: Correctness**. Are the semantics of the program preserved?
- **RQ4: Robustness**. How does ROPfuscator's robustness stand in regards to the threat model attacks?
- **RQ5: Practicality**. Is our approach applicable to real-world use cases?

The configurations are composed as follows:

- **Baseline**, non obfuscated binaries
- **ROPonly**, ROP transformation only
- **ROP+OP$_{\mathrm{Basic}}$**, ROP transformation with basic opaque predicates
- **ROP+OP$_{\mathrm{DSE}}$**, ROP transformation with DSE-resistant opaque predicates
- **ROP+OP$_{\mathrm{DSE}}$+Hiding**, ROP transformation with opaque predicates and instruction hiding

Evaluation Test Cases



We applied ROPfuscator to different applications and test sets:

- GNU **binutils**
- **SPEC CPU 2016** benchmark suite
- **VLC** media player

RQ1: Completeness

libc version	Status	readelf	c++filt
2.27-3 ubuntu1	Obfuscated	77.24%	74.99%
	Unobfuscated (No gadget / reg)	11.80%	11.70%
	Unobfuscated (Other)	10.96%	13.31%
2.27-3 ubuntu1.2	Obfuscated	36.02%	26.07%
	Unobfuscated (No gadget / reg)	53.02%	60.62%
	Unobfuscated (Other)	10.96%	13.31%
2.31-0 ubuntu9	Obfuscated	82.69%	80.93%
	Unobfuscated (No gadget / reg)	6.35%	5.77%
	Unobfuscated (Other)	10.96%	13.31%

Figure 4: Ratio of instructions obfuscated in GNU binutils with different library versions

On average, ROPFuscator transforms about 60–80% of the instructions into ROP chains

Compiler optimizations influence the coverage, and, when not applied, result in better higher coverage

The same applies to the **library version** from which the ROP gadgets are extracted

RQ2-3: Performance and Correctness

metric	obfuscation	absolute value		ratio (Baseline=1)		ratio (Roponly=1)	
		readelf	c++filt	readelf	c++filt	readelf	c++filt
time	Baseline	0.39s	0.30s	1.0	1.0	0.09	0.01
	ROPonly	4.23s	30.6s	11.0	102	1.0	1.0
	ROP+OP$_{\mathrm{Basic}}$	41.1s	337s	107	1118	9.7	11.0
	ROP+OP$_{\mathrm{DSE}}$	66.4s	761s	172	2527	15.7	24.8
	ROP+OP$_{\mathrm{DSE}}$+Hiding	57.1s	611s	148	2030	13.5	19.9
size	Baseline	1.1MB	1.1MB	1.0	1.0	0.10	0.07
	ROPonly	10.5MB	15.7MB	9.6	14.1	1.0	1.0
	ROP+OP$_{\mathrm{Basic}}$	895MB	1407MB	828	1269	86.6	89.7
	ROP+OP$_{\mathrm{DSE}}$	1530MB	2411MB	1417	2175	148	154
	ROP+OP$_{\mathrm{DSE}}$+Hiding	1283MB	2063MB	1188	1861	124	132

Figure 5: Runtime slowdown and code size of obfuscated programs for binutils for each obfuscation algorithm

RQ4: Robustness

Obfuscation Algorithm	Robustness against Attack Algorithm				Performance	
	A) Static Analysis	B) Static ROP Analysis	C) DSE	D) Dynamic ROP Analysis	Slowdown ratio	Size ratio
Baseline	○	○	○	○	1	1
ROPonly	●	○	○	○	10–200	10-16
ROP+OP$_{\mathrm{Basic}}$	●	●	○	○	100–2000	900–1500
ROP+OP$_{\mathrm{DSE}}$	●	●	●	○	200–4000	1500–2500
ROP+OP$_{\mathrm{DSE}}$+Hiding	●	●	●	◕	150–3000	1200–2000

Figure 6: Robustness and performance of each algorithm in ROPfuscator against attacks

○: Breakable, ●: Robust, ◕: Mostly Robust

ROPonly is robust against Threat A but not against Threat B, C, and D

Introducing **opaque predicates** (ROP+OP$_{\mathrm{DSE}}$) fortifies the programs against Threats B and C

Instruction hiding (ROP+OP$_{\mathrm{DSE}}$+Hiding) makes the obfuscated binaries resistant against Threat D

To balance robustness and performance, we considered obfuscating functions selectively: the **Balanced** configuration

Config	Time [s]	CPU Usage	Played Smoothly?	Size [MB]
Baseline	30.2	12.4%	Yes	0.034
ROPonly	30.2	23.3%	Yes	0.38
ROP+OP$_{\mathrm{DSE}}$	110.2	97.0%	No	48.5
ROP+OP$_{\mathrm{DSE}}$+Hiding	120.7	95.3%	No	41.3
Balanced	30.2	23.2%	Yes	18.4

Figure 7: Performance statistics of VLC Media Player using libdvdcss

Using *Balanced*, title key derivation functions are obfuscated with ROP+OP$_{\mathrm{DSE}}$+Hiding and the rest of the library with ROPonly

Takeaway

The spatial and time overhead introduced by ROPfuscator are significant

However, it is possible to achieve robustness for an acceptable performance loss when selecting **sensitive functions** to be strongly protected

Future Directions and Developments

Implemented Changes

- Build process and experiments **reproducibility** with **Nix**[3]
- Independence from the system's *`libc`* with *`librop`*[4]

Future directions

- Move from **microgadgets** to a gadget extraction engine
- Extended architectures support (*x86_64*, *ARM*, ...)

---

[3] *https://nixos.org*
[4] *https://github.com/ropfuscator/librop*

Conclusion

We present **ROPfuscator**[5], a C/C++ obfuscation framework with a unified threat model. We address code coverage, overhead, correctness, robustness, and practicality challenges through principled reasoning

Come meet us at the Demo session!

[5] *https://github.com/ropfuscator/ropfuscator*