TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Thwarting ROP Attacks and Unveiling User Level Stack Tampering through Kernel Shadow Stack

Marco Calavaro • P.Caporaso • G.Bianchi • F.Quaglia

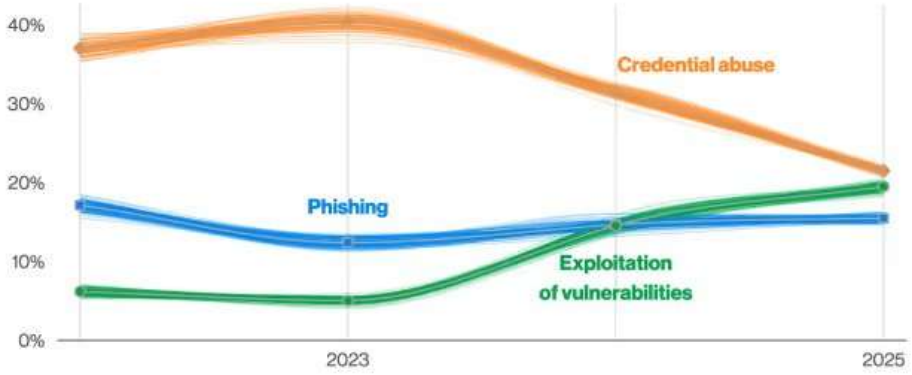
23rd IEEE International Symposium on Network Computing and Applications

NCA 25

Introduction

Numerous cyber threat intelligence reports show:

- Most attacks involve **network-based exploitation**
- There is a growing trend of attacks targeting **endpoint vulnerabilities**

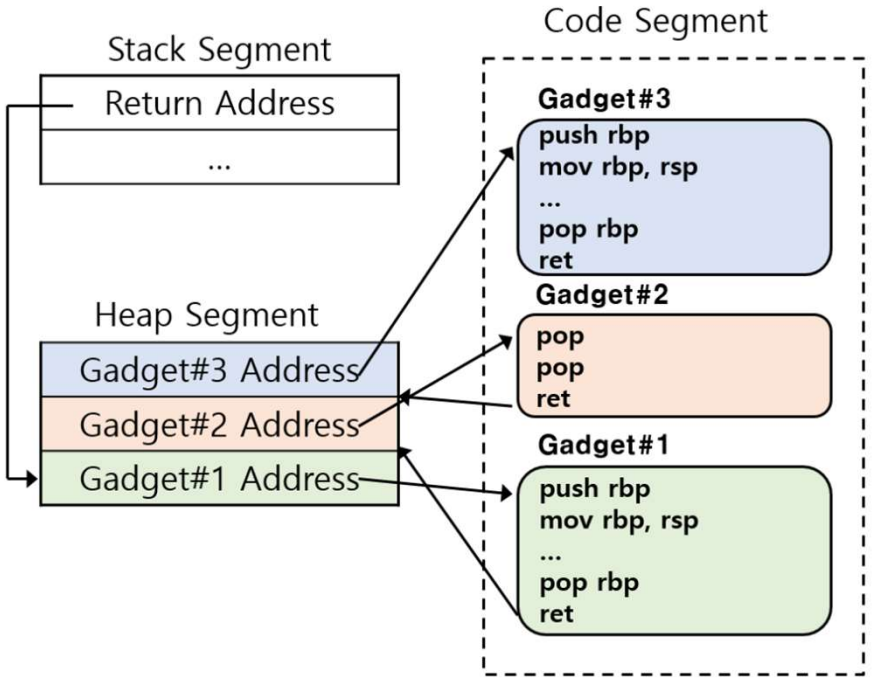


Oligo

Kaspersky

Introduction - ROP

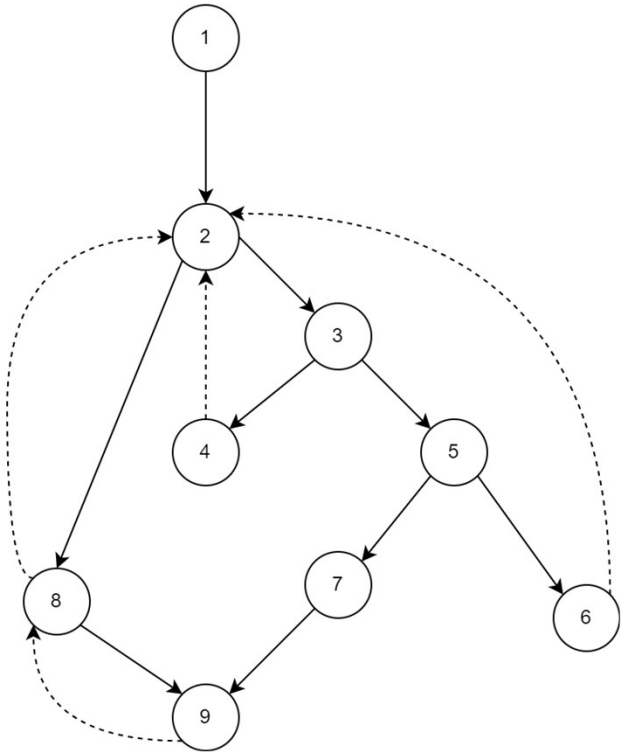
- Return Oriented Programming (**ROP**) attacks
- Utilizes small sequences of executable instructions to operate (**Gadgets**)
- The attack is executed by chaining gadgets together, forming what is known as a ROP chain



Introduction - CFI

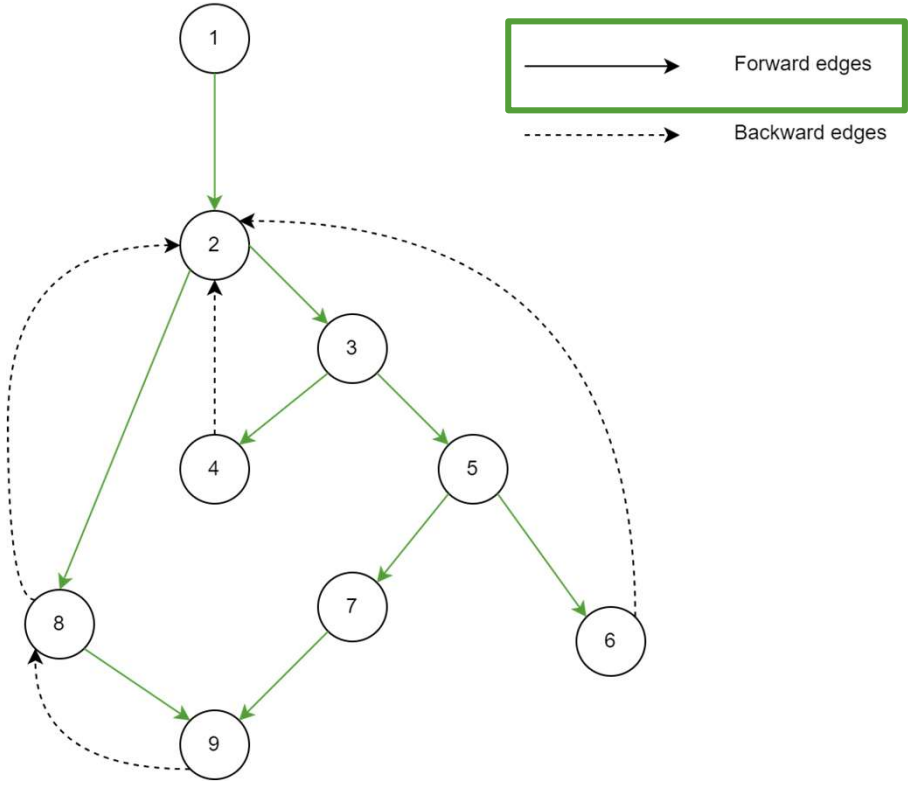
- **Control flow** determines the order in which individual instructions, statements, or function calls are executed in a program
- **Control Flow Graph** (CFG) is a graphical representation of all paths that might be traversed through a program during its execution
- **Control flow integrity** (CFI) techniques protect against attacks that alter program execution, such as code injection or jump hijacking

CFG:



Introduction – CFI – Forward edges

What it secure	• Indirect calls/jumps (function pointers, virtual calls)
Common attack	• Function pointer / vtable hijacking, COOP
Technique	• Target validation (type-based or label-based CFI)
Performance	• Low overhead
Example defenses	• LLVM CFI, VTV



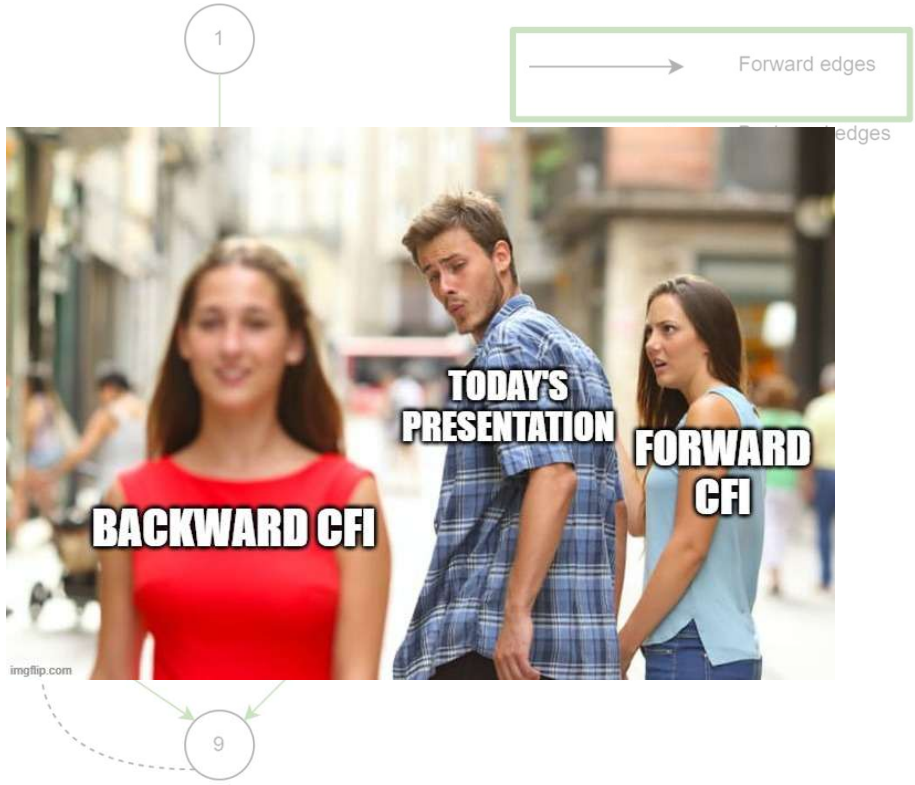
Forward edges - CFI: Not Our Main Character

What it secure	• Indirect calls/jumps (function pointers, virtual calls)
Example defenses	• LLVM CFI, VTV

Forward-edge targets might not even exist yet...

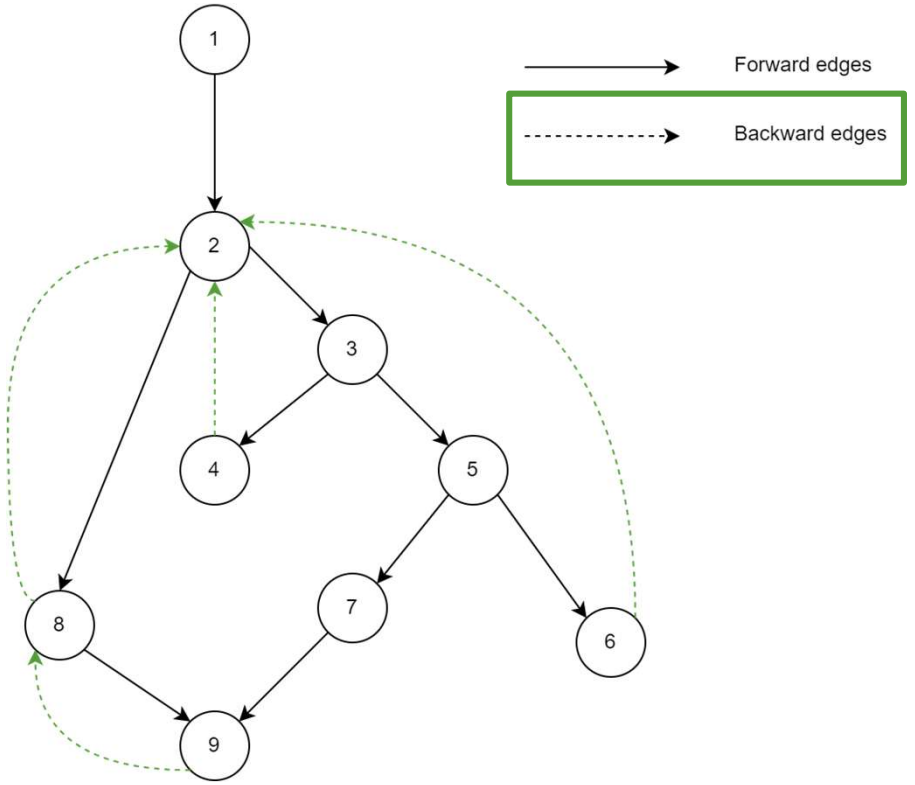
But backward-edge targets are *always there* — right on the stack.

Let's focus on what's always present: backward edges!



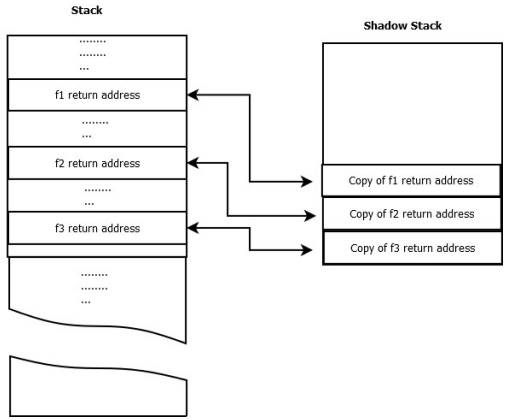
Introduction – CFI – Backward edges

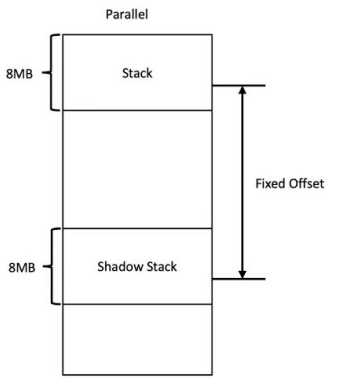
What it secure	• Return addresses on the stack
Common attack	• Stack smashing, Return-Oriented Programming (ROP)
Technique	• Shadow stacks or signed return addresses
Performance	• Moderate unless hardware-assisted
Example defenses	• Intel CET, ARM PAC

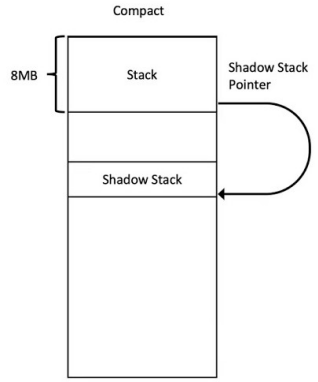


Shadow Stack: Introduction

- Shadow stack is a separate stack for return addresses
- Designed to protect the control flow
- Types [SoK : SP19]:
  - Parallel
  - Compact
- Hardware support has emerged only recently







Shadow Stack: State of the art

Different implementations have been proposed:

- Compiler and run-time system (Silhouette [USENIX20])
- On flash memory (FLAShadow [TIOT24])
- Thread-local storage (Buddy stacks [TOSEM22])
- Dynamic user library (TRUSS [08])

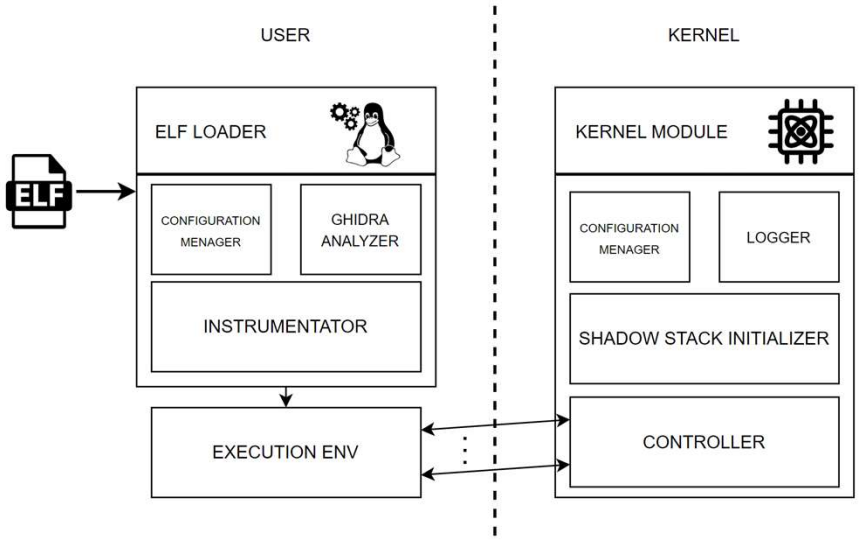
Still vulnerable to [1]:

- Information disclosure
- Side-channel

[1] N. Burow, X. Zhang and M. Payer, "SoK: Shining Light on Shadow Stacks," 2019 IEEE Symposium on Security and Privacy (SP)

Kernel Shadow Stack: Idea

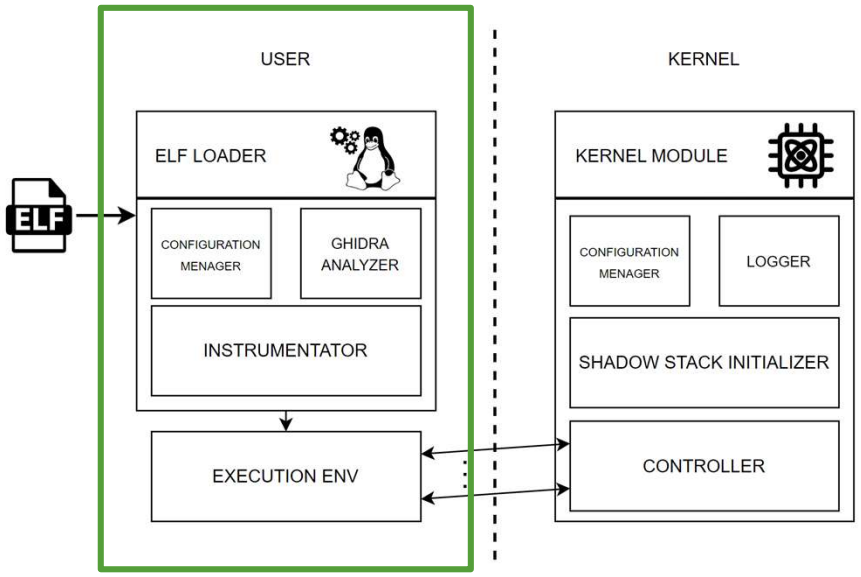
- Propose a **Kernel** implementation for Shadow Stack
- Two main components:
  - User space ELF loader
  - Kernel Module



Kernel Shadow Stack: Under the hood (ELF-L)

Core Objectives

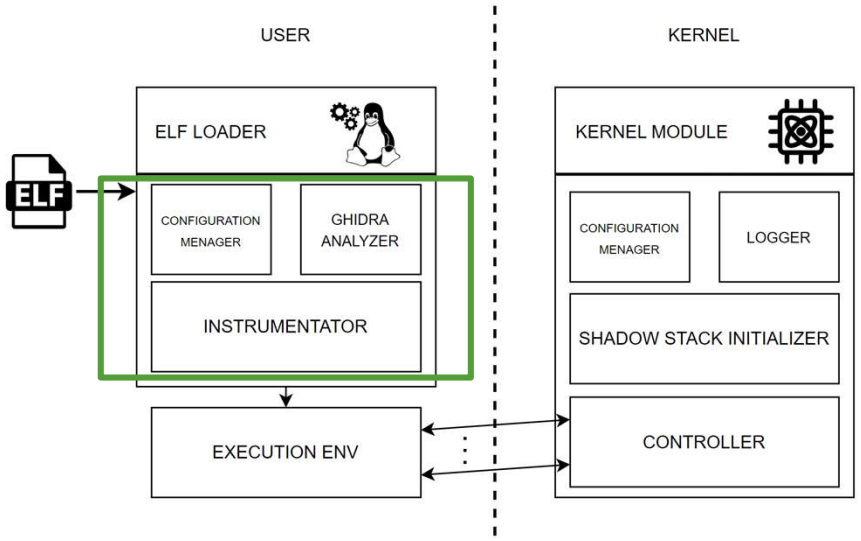
- Load ELF target program into memory
- Instrument CALL/RET instructions for kernel interaction
- Launch the instrumented program



Kernel Shadow Stack: Under the hood (ELF-L)

Instrumentation Strategy

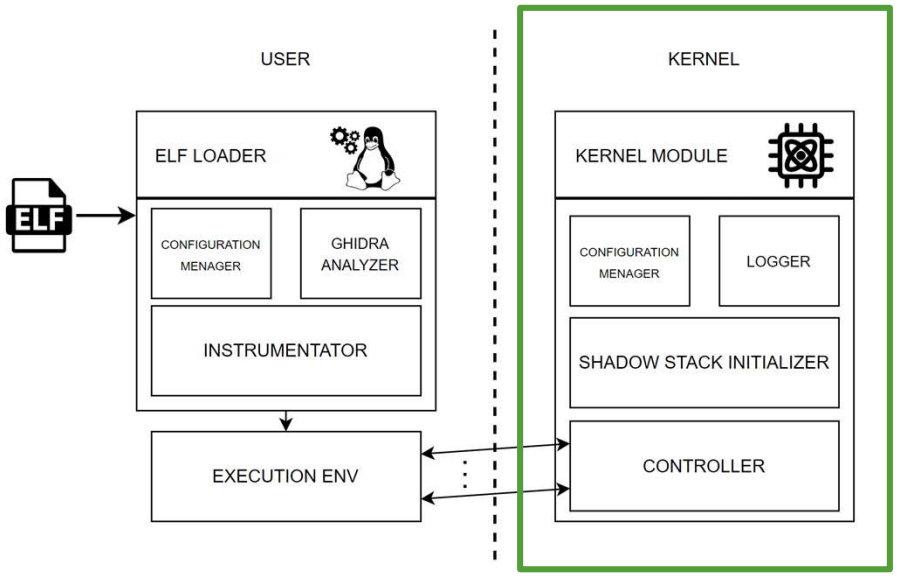
- Uses **Ghidra analysis** to locate CALL/RET
- **Minimal rewriting**: only overwrite the bytes originally occupied by CALL/RET (avoid full binary rewriting)



Kernel Shadow Stack: Under the hood (LKM)

Core Objectives

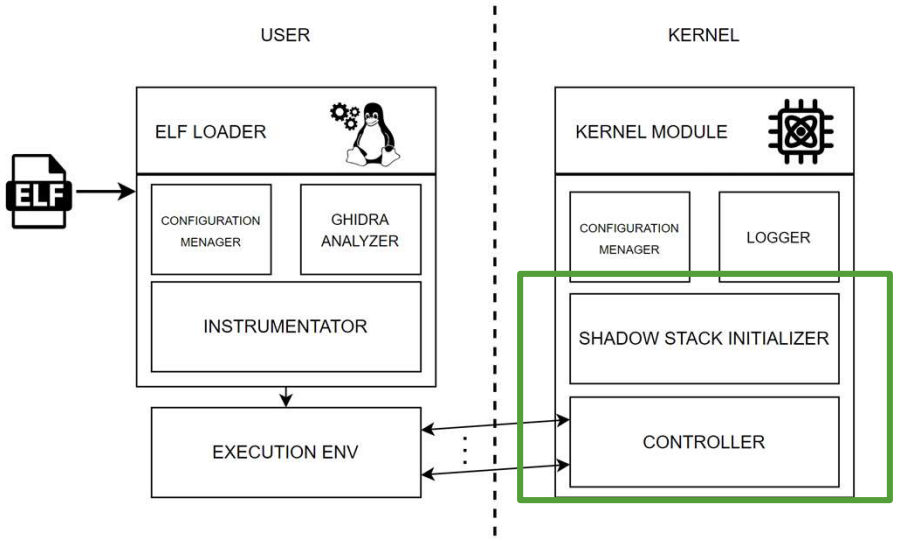
- Initializing the kernel structures needed for application monitoring and logging
- Managing the per-thread shadow stacks, based on instrumented CALL/RET invocation



Kernel Shadow Stack: Under the hood (LKM)

Efficient On-Demand creation

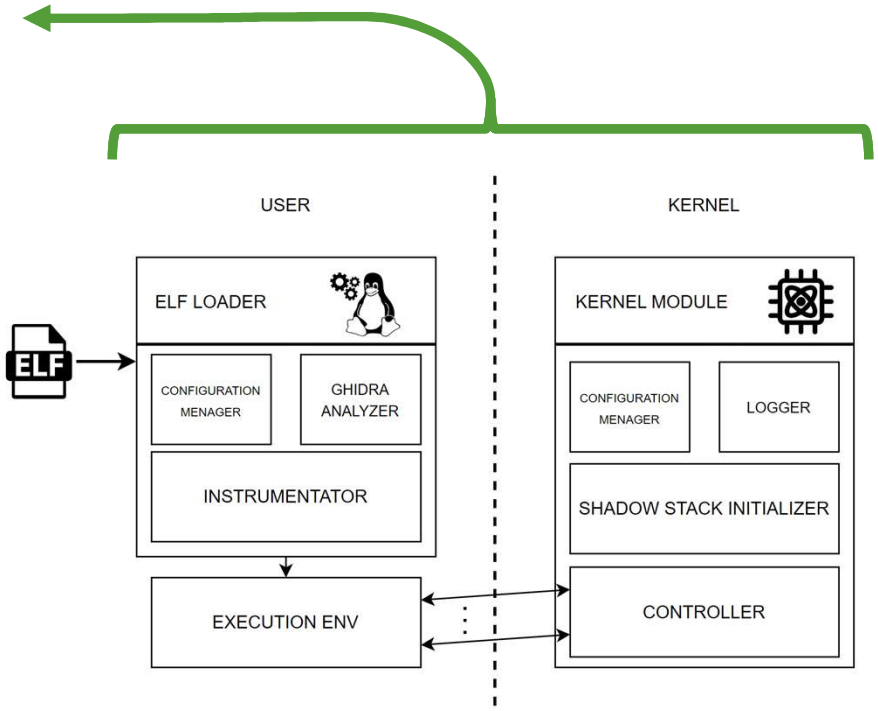
- Instrumentation information are passed true ioctl
- Hooks ensure per-thread structures are created only when required:
  - kernel_clone
  - do_exit
  - pick_next_task



Kernel Shadow Stack: Key Strengths

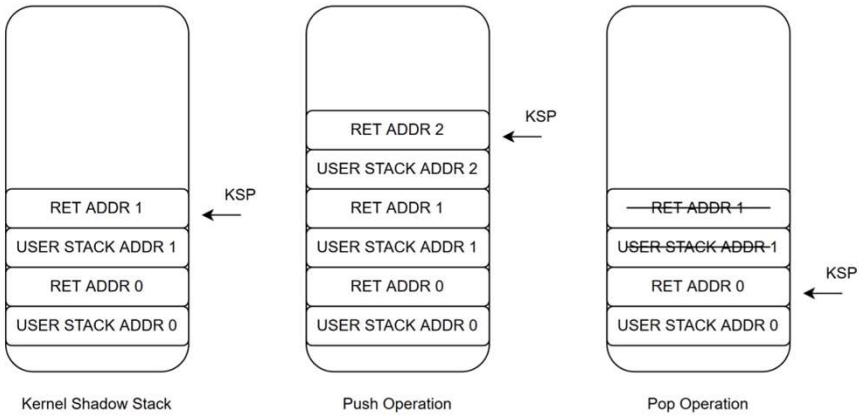
Transparency

- Binary-level analysis and instrumentation:
  - no source changes required
- Designed for broad applicability



Kernel Shadow Stack: Execution-flow verification

- **Goal**: verify instrumented RET did not observe user-stack tampering
  - on tamper → log and/or kill process
- **How**: LKM reads user return addresses (uses `copy_from_user(...)`, which temporarily disables SMAP) and compares them with kernel shadow-stack entries
- **Configurable depth**: how many return addresses to verify (1…N)
- **Partial-instrumentation tolerance**: if stack misalignment occurs due to partially instrumented CALL/RET pairs



Kernel Shadow Stack: Key Strengths

Beyond Anti-ROP Functionality

- Hardening of information disclosure
- Configuration flexibility
- Active analysis of anomalies:
  - Can analyze the entire user stack and call stack
- Monitoring and logging user stack

Kernel Shadow Stack: Experimental Data

EFFECTIVENESS AGAINST THE RIPE BENCHMARK

Attack Type	Target Pointer	Total	No KSS	With KSS	Block Perc.
direct	ret	12	12	0	100.0%
direct	baseptr	4	4	2	50.0%
direct	funcptrstackvar	10	10	0	100.0%
direct	funcptrstackparam	10	0	0	100.0%
direct	structfuncptrstack	10	9	0	100.0%
direct	longjmpstackvar	20	20	2	90.0%
direct	longjmpstackparam	20	20	2	90.0%
indirect	ret	10	10	0	100.0%
indirect	baseptr	4	4	2	50.0%
indirect	funcptrstackvar	10	10	0	100.0%
indirect	funcptrstackparam	10	10	0	100.0%
indirect	funcptrheap	10	10	0	100.0%
indirect	funcptrbss	10	9	0	100.0%
indirect	funcptrdata	10	10	0	100.0%
indirect	structfuncptrstack	10	9	0	100.0%
indirect	structfuncptrheap	10	10	0	100.0%
indirect	structfuncptrbss	10	10	0	100.0%
indirect	structfuncptrdata	10	10	0	100.0%
indirect	longjmpstackvar	10	10	0	100.0%
indirect	longjmpstackparam	10	10	0	100.0%
indirect	longjmpheap	10	10	0	100.0%
indirect	longjmpbss	10	10	0	100.0%
indirect	longjmpdata	11	10	0	100.0%
Total		**240**	**227**	**8**	96.7%

EFFECTIVENESS AGAINST THE RECIPE BENCHMARK

Attack Type	target	Total	No KSS	With KSS	Block Perc.
OOBPtrHijack	retaddr	4	4	0	100.0%
BoundOFlow	oldebp	16	16	0	100.0%
BoundOFlow	retaddr	20	20	0	100.0%
PtrHijack	retaddr	4	4	0	100.0%
NBoundOFlow	retaddr	4	4	0	100.0%
Total		**48**	**48**	**0**	100%

Evaluation of **protection efficiency** using benchmark RecIPE and RIPE

- The 8 remaining successful RIPE attacks rely on artificial gadgets inserted by the benchmark (not real code).
- These gadgets are not instrumented by our system, hence undetected
- Not present in real-world application

Kernel Shadow Stack: Experimental Data

EXECUTION PROFILE OF SPEC CPU 2017 APPLICATIONS

Application name	User Time percentage	Kernel Time percentage
600.perlbench	99.28 %	0.72 %
602.gcc	95,90 %	4,10 %
605.mcf	99,20 %	0,80 %
620.omnetpp	99,38 %	0,62 %
623.xalancbmk	99,06 %	0,94 %
625.x264	96.50 %	3,50 %
631.deepsjeng	98,92 %	1,08 %
641.leela	99,45 %	0,55 %
648.exchange2	99,09 %	0,91 %
657.xz	99.24 %	0.76 %

EXECUTION TIME (SECONDS) OF SPEC CPU 2017 APPLICATIONS

Application name	Application Area	No inst.	Inst. 33%	Inst. 66%	Inst. 100%
600.perlbench	Perl interpreter	522	10999	18251	26844
602.gcc	GNU C compiler	674	5603	10104	14947
605.mcf	Route planning	767	855	1753	3283
620.omnetpp	Discrete Event simulation computer network	563	583	640	739
623.xalancbmk	XML to HTML conversion via XSLT	527	1804	5457	6671
625.x264	Video compression	252	1615	3151	3977
631.deepsjeng	Artificial Intelligence: Alpha-beta tree search	431	8446	13764	17701
641.leela	Artificial Intelligence: Monte Carlo tree search	518	1074	1971	4244
648.exchange2	Artificial Intelligence: recursive solution generator	289	668	715	1489
657.xz	General data compression	2635	2968	3450	4241

Performance overhead varies across benchmarks

CPU-bound nature show higher overhead, this not align well with KSS characteristics

Represents the **worst-case** scenario for KSS performance

Instrumentation reduction demonstrates that overhead can be tuned and optimized based on the specific use case

Kernel Shadow Stack: Experimental Data

EXECUTION PROFILE OF SPEC CPU 2017 APPLICATIONS

Application name	User Time percentage	Kernel Time percentage
600.perlbench	99.28 %	0.72 %
602.gcc	95,90 %	4,10 %
605.mcf	99,20 %	0,80 %
620.omnetpp	99,38 %	0,62 %
623.xalancbmk	99,06 %	0,94 %
625.x264	96.50 %	3,50 %
631.deepsjeng	98,92 %	1,08 %
641.leela	99,45 %	0,55 %
648.exchange2	99,09 %	0,91 %
657.xz	99.24 %	0.76 %

EXECUTION TIME (SECONDS) OF SPEC CPU 2017 APPLICATIONS

Application name	Application Area	No inst.	Inst. 33%	Inst. 66%	Inst. 100%
600.perlbench	Perl interpreter	522	10999	18251	26844
602.gcc	GNU C compiler	674	5603	10104	14947
605.mcf	Route planning	767	855	1753	3283
620.omnetpp	Discrete Event simulation computer network	563	583	640	739
623.xalancbmk	XML to HTML conversion via XSLT	527	1804	5457	6671
625.x264	Video compression	252	1615	3151	3977
631.deepsjeng	Artificial Intelligence: Alpha-beta tree search	431	8446	13764	17701
641.leela	Artificial Intelligence: Monte Carlo tree search	518	1074	1971	4244
648.exchange2	Artificial Intelligence: recursive solution generator	289	668	715	1489
657.xz	General data compression	2635	2968	3450	4241

Why SPEC CPU?

- Commonly used in literature for performance evaluation
- Not an effective benchmark for assessing real-world impact
  - As describe in CONFIRM [2]

[1] Xiaoyang Xu, Masoud Ghaffarinia, Wenhao Wang, Kevin W. Hamlen, and Zhiqiang Lin. **CONFIRM: evaluating compatibility and relevance of control-flow integrity protections for modern software**. 28th USENIX Conference on Security Symposium (SEC'19).

Kernel Shadow Stack: Real-World Scenario

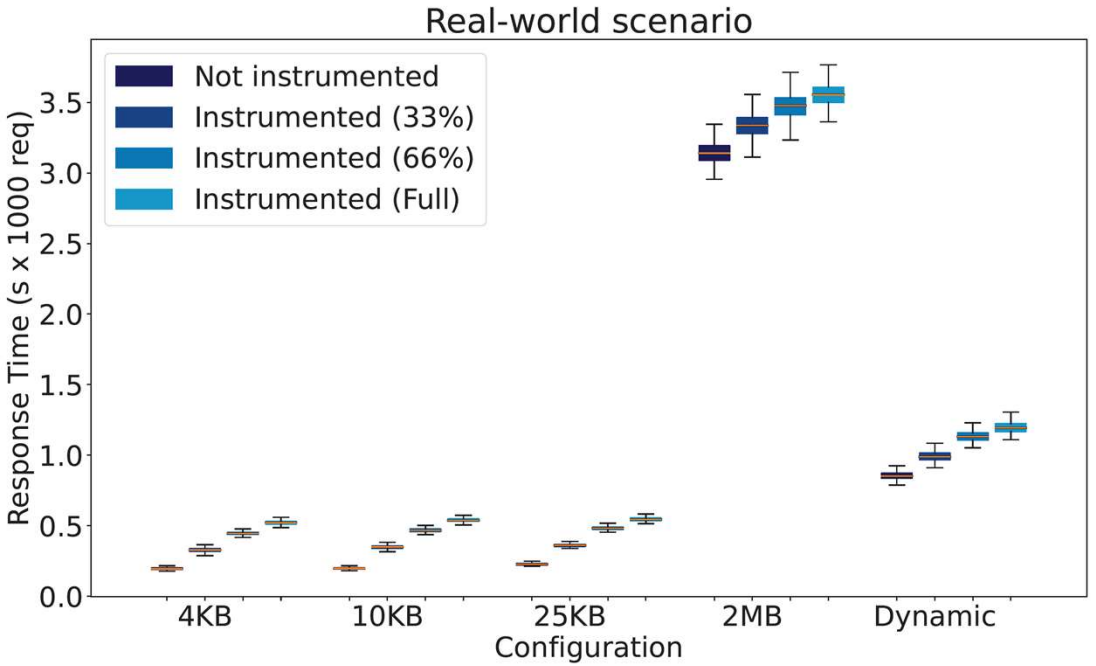
Experiment setup

- Apache2 Web-Server
- SqlLite3 database

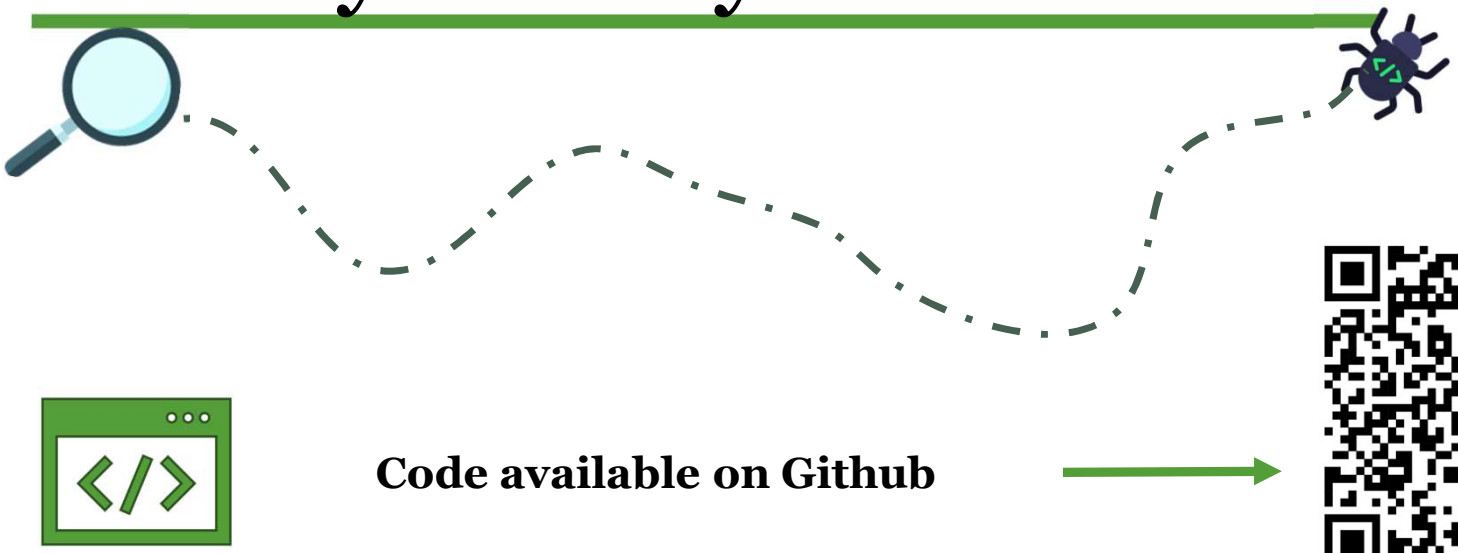
Worst possible overhead

- On full instrumentation
- Executed on the same machine to eliminate network delay, ensuring results are not artificially improved

Overhead approximately 20%



Thank you for your attention



Code available on Github