# ROPdefender

A Detection Tool to Defend Against Return-Oriented Programming Attacks

ROPdefender: a tool that is able to detect/prevent ROP attacks.

make use of known techniques such as *instrumentation*, particularly instrumentation based on a just-in-time (jit) compiler, which adds extra code to a program’s binary at runtime with the purpose to observe the program’s behavior during execution.

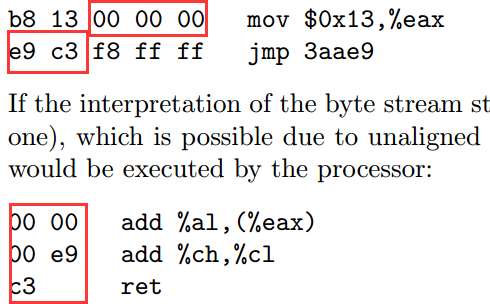
our tool can even detect execution of *unintended* instruction sequences when the adversary is able to tamper with the control flow.

**adopt already existing techniques：**

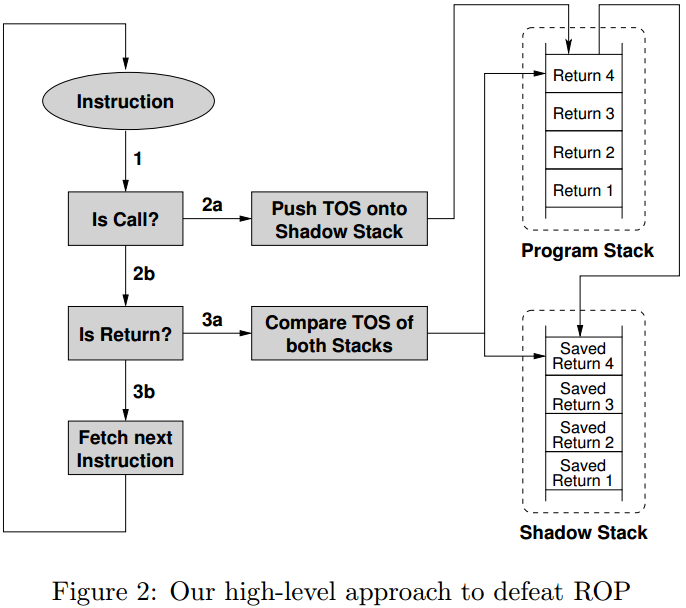
1. **shadow stack** [12, 43, 21] for return addresses,
2. the concept of **binary instrumentation** as used in **taint tracking** [34, 14] or return address protection [22, 13].
3. use the jit-based binary instrumentation framework **Pin** [31].

ROPdefender is able to detect execution of unintended instruction sequences, specifically unintended return instructions. This is **achieved by enforcing a return address check on any return instruction issued during program execution**.

**Unintended Instruction Sequences:**



**Our Approach to Detect/Prevent ROP:**



TOS: Top Of the Stack

In order to evaluate each return instruction issued during program execution, we store a copy of the return address onto a separate shadow stack (similar to [12, 43, 21]) once the program issues a call instruction. We instrument all return instructions that are issued during program execution and **perform a return address check** as described in the following:

1. Before an instruction will be executed by the processor, our solution intercept the instruction and evaluates the instruction’s type and target. In practice, this can be accomplished with a binary instrumentation framework as we will explain in Section 3.3 and 3.4.
2. First, we check if the instruction is a call instruction. If this is the case, we store a copy of the pushed return address onto our shadow stack (transition 2a in Figure 2). Otherwise, we check whether the intercepted instruction is a return instruction (transition 2b in Figure 2).
3. If the instruction is a return instruction, our solution checks if the top return address on the shadow stack equals the return address on top of the program stack. If the two return addresses are different, we conclude that a return address corruption has occurred and program execution is redirected to instructions not intended by the programmer.

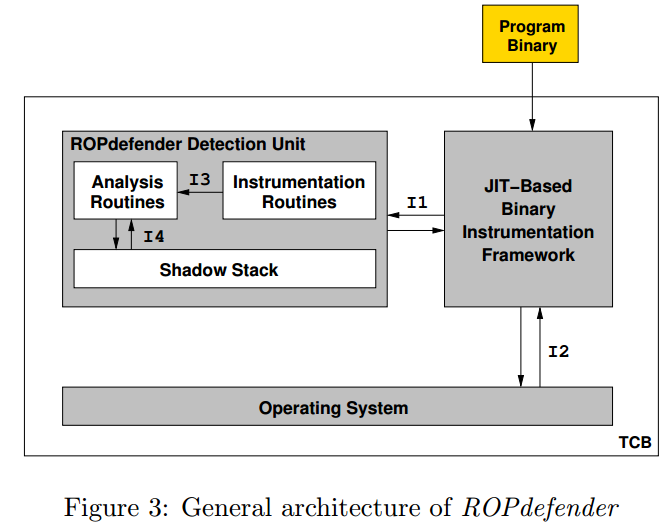
Our solution prevents traditional buffer overflow attacks that are launched by modifying a return address.

The program stack will hold the start address of the second instruction sequence. However, since our shadow stack holds only return addresses that are pushed by call instructions themselves, **it cannot contain the start address of this instruction sequence starting somewhere in the middle of a function**.

**Obviously, in our approach we assume that instruction sequences issued during a ROP attack end in return instructions**.

One idea to bypass our solution is to use instruction sequences ending in an indirect jump/branch instruction.

The architecture of *ROPdefender* consists mainly of two components, *ROPdefender* detection unit and a binary instrumentation engine.



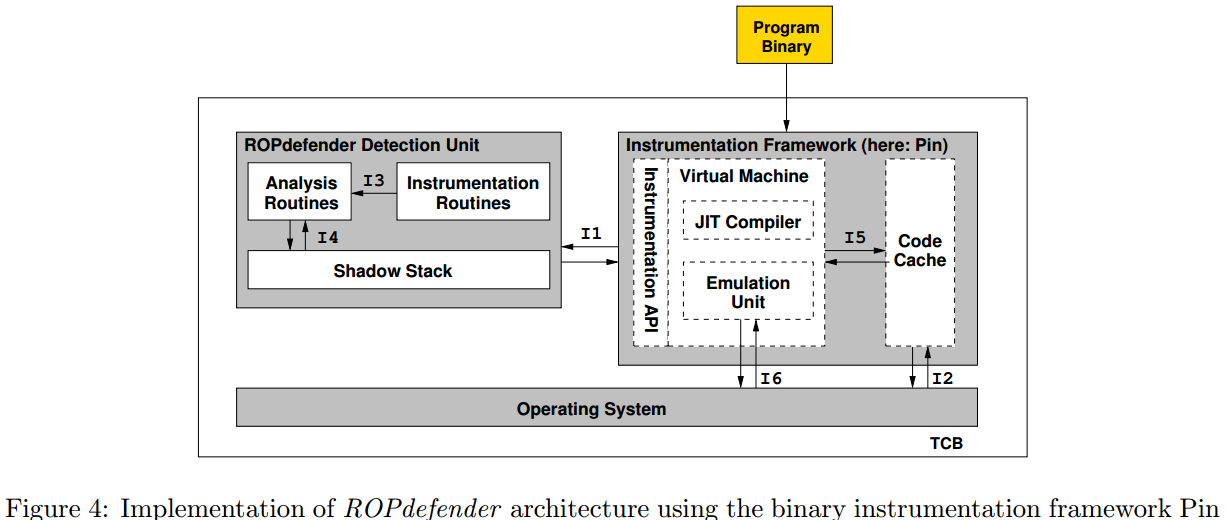
TCB: Trusted Computing Base, include the both components, as well as operating system

***ROPdefender* Detection Unit.** Our *ROPdefender* detection unit consists of instrumentation and analysis routines, and a shadow stack. Instrumentation routines usually inspect the current instruction in general (e.g., check if the current instruction is a call, a return, a branch, etc.). The instrumentation routines are able to call different analysis routines through interface I3 based on the type of the current instruction. For instance, if the instruction is a call instruction, an analysis routine can be invoked that performs further analysis of the call instruction. Is the current instruction instead a return instruction, then a second analysis routine can be called performing analysis different from the call analysis. The detection unit also includes the shadow stack. Analysis routines can push and pop return addresses onto and off the shadow stack through interface I4.

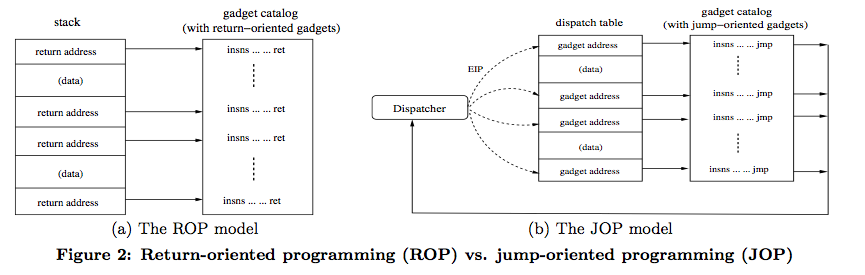
**Implementation:**

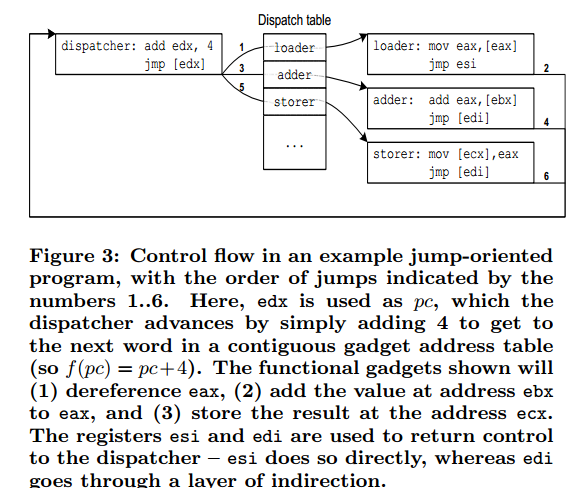
jit-based binary instrumentation framework Pin in version 2.6-27887

our implementation of the *ROPdefender* detection unit is one C++ file consisting of 80 lines of code.



# JOP





**Dispatcer Gadget:**

The dispatcher gadget plays a critical role in the JOP technique. It essentially maintains a virtual program counter, or pc, and executes the JOP program by advancing it through one gadget after another. Specifically, each pc value specifies an entry in the dispatch table, which points to a particular jump-oriented functional gadget.

We consider any jump-oriented gadget that carries out the following algorithm as a dispatcher candidate.



**Functional Gadgets:**

to launch other gadgets, which we call functional gadgets. To maintain control of the execution, all functional gadgets executed by the dispatcher must conclude by jumping back to it, so that the next gadget can be launched.

More formally, a functional gadget is defined as a number of useful instructions ending in a sequence that will load the instruction pointer with the result of a known expression.

# JOP-Alarm: Detecting Jump-oriented Programming-based Anomalies in Applications.

In this work, we propose a novel, easily implementable algorithm, called JOP-alarm, that computes a score value to assess the potential for JOP attack, and detects possibly harmful program behavior.

We study the most common characteristics of JOP such as length of functional gadgets, indirect jump (or call) distances, and tune our algorithm to adjust its scoring based on such details. can be implemented relatively easily in software with small amounts of code instrumentation or via simple hardware logic.