

PROPS: Learning Stack Patterns for ROP Detection on Legacy ARM-based Devices

József Sándor[✉], Roland Nagy[✉], and Levente Buttyán[✉]

CrySyS Lab

Department of Networked Systems and Services
Budapest University of Technology and Economics
{jsandor,rnagy,buttyan}@crysys.hu

Abstract. Return-oriented programming (ROP) remains a potent threat to legacy systems running on 32-bit ARM architectures, where modern hardware defenses like Pointer Authentication Code and Branch Target Identification are unavailable. We present PROPS (Precision ROP Scanner), a novel detection mechanism for identifying ROP attacks on ARMv7 and older ARM platforms. Using Linux kernel uprobes, PROPS monitors exclusively the return points of known (or potentially) vulnerable functions without modifying the existing software stack. PROPS applies Principal Component Analysis (PCA)-based anomaly detection to the stack content of monitored programs. By learning normal stack patterns from benign execution, it detects deviations that likely indicate ROP attacks, achieving 98.24% detection accuracy on generated ROP chains. This approach represents the first purely stack-based ROP detection method for ARM architectures, offering a practical solution for protecting legacy systems.

Keywords: ARM · Return-Oriented Programming · ROP Detection · Principal Component Analysis · Anomaly Detection

1 Introduction

Memory corruption vulnerabilities remain a critical security concern in modern computing systems, often leading to exploits such as arbitrary code execution, privilege escalation, and system compromise. These vulnerabilities typically arise due to programming errors (such as buffer overflows, use-after-free, or integer overflow) that allow attackers to manipulate memory and alter program execution. Among the various exploitation techniques, return-oriented programming (ROP) [28] has emerged as one of the most effective and sophisticated methods for bypassing traditional security mechanisms like Data Execution Prevention (DEP). ROP attacks leverage existing executable code within a program’s memory space, chaining together small instruction sequences (gadgets) to execute arbitrary operations without injecting new code. This technique makes ROP attacks particularly challenging to detect and mitigate, as they do not introduce

easily identifiable artifacts into the memory, neither they require any illegitimate changes to memory access permissions.

ROP attacks pose a significant threat to embedded devices as well, which are often deployed in critical applications and operate under tighter resource constraints than general-purpose systems. Although numerous ROP mitigation and detection mechanisms have been developed for desktop environments, particularly Intel x86-64 platforms, these approaches are rarely suitable for embedded systems due to architectural differences, limited computational resources, and operating system restrictions. For instance, even Address Space Layout Randomization (ASLR) is inherently weaker on 32-bit architectures (common in embedded systems) due to the significantly smaller address space, which reduces entropy and makes brute-force probing more feasible. Many existing defenses depend on hardware-assisted control-flow monitoring features unavailable on legacy embedded processors, require recompilation of software that is often proprietary or closed-source, or rely on OS kernel modifications that are impractical due to vendor limitations and stability concerns. These challenges collectively contribute to a significant gap in practical ROP mitigation and detection solutions for embedded systems, including legacy ARM devices, and motivate the development of specialized approaches tailored to their unique constraints and operational environments.

Recent ARM architectures introduce hardware features that significantly raise the bar against code-reuse attacks. Pointer Authentication Code (PAC) cryptographically signs return addresses and function pointers using secret keys, preventing attackers from substituting gadget addresses without triggering authentication failures [20]. Branch Target Identification (BTI) enforces control-flow integrity by requiring that indirect branches land only on designated landing pads, thereby reducing the availability of gadgets in executable code sections[2]. Memory Tagging Extension (MTE) enforces tag checks on pointer dereferences, detecting many spatial and temporal memory safety violations before they can be exploited [27]. However, these mechanisms are available only on recent ARMv8 and ARMv9 processors; the vast majority of ARMv7 and early ARMv8 embedded devices lack PAC, BTI, and MTE entirely. As a result, legacy systems remain fully vulnerable to code-reuse attacks, highlighting the need for complementary software-based detection methods.

To be viable in real-world environments, a security solution must be straightforward to deploy and compatible with existing systems. Legacy hardware and heterogeneous, vendor-customized operating systems make this particularly challenging, as many embedded devices run outdated or specialized OS versions that cannot be easily modified or upgraded. Our approach is designed to overcome these constraints: it operates without requiring changes to the underlying firmware or toolchain and avoids dependencies on recompilation, kernel modifications, or proprietary hardware features. As a result, it can be seamlessly integrated into a wide range of ARM platforms, including legacy systems where traditional defenses are impractical and new protection mechanisms are unavailable.

To address ROP detection under these constraints, we propose PROPS (Precision ROP Scanner), a novel detection mechanism that examines stack content to identify the presence of ROP chains. PROPS is motivated by the realities of legacy embedded deployments, where vulnerabilities cannot always be patched promptly, and software updates may be operationally infeasible. In such environments – common in critical infrastructure such as substations, power plants, and manufacturing systems [21] – vulnerable components must continue to operate despite known memory-safety issues. By leveraging knowledge of specific software vulnerabilities (e.g., from CVE databases), PROPS performs precision detection (hence the name) tailored to the exploitation context. Our solution targets 32-bit ARMv7 and older architectures, which represent a broad class of deployed ARM devices and remain widely used in industrial and embedded systems where hardware upgrades are impractical.

At its core, PROPS combines approaches that examine stack content [26] with anomaly-detection techniques [15]. Rather than attempting to enumerate or model the broad and evolving space of ROP techniques, we take a different approach: we learn only the characteristics of benign stack content and classify deviations as potential ROP activity. Our approach draws inspiration from EigenROP [10], which applies anomaly detection based on Principal Component Analysis (PCA) to identify ROP attacks on Intel architectures. However, PROPS differs in key aspects: unlike EigenROP, our system operates without requiring binary instrumentation, and we demonstrate improved detection performance on ARM platforms. To achieve this, we train a PCA model on clean stack snapshots captured from benign program execution, enabling the detection of anomalies that deviate from learned benign patterns. The trained model is then deployed directly on the target device, providing on-device ROP detection without modification of the existing software stack. To the best of our knowledge, PROPS is the first ROP defense to apply exclusively stack-based anomaly detection as a countermeasure on ARM architectures, demonstrating that models trained on benign stack patterns can effectively identify ROP chains without requiring attack-specific assumptions.

The remainder of this paper is organized as follows. Section 2 introduces ROP attacks. Section 3 reviews related work in ROP detection. Section 4 and Section 5 present the design and implementation details of PROPS. Section 6 evaluates detection performance and runtime overhead. Section 7 discusses trade-offs and limitations. Finally, Section 8 concludes this work.

2 Background

Memory-corruption vulnerabilities have long been among the most pervasive and exploited classes of software flaws, with early exploits often relying on simple stack-based buffer overflows to redirect control flow to a shellcode placed on the stack by the attacker. In response, modern systems introduced a combination of hardware- and software-based defenses to prevent the execution of injected code, including hardware mechanisms such as No-eXecute (NX) on AMD, Execute

Disable (XD) on Intel, Execute Never (XN) on ARM, and Execute Inhibit (XI) on MIPS. At the OS level, similar policies are enforced through DEP on Windows and the Write-XOR-Execute principle on Unix-like systems.

Consequently, attackers adapted by reusing existing executable code already present within a program’s address space rather than injecting new payloads. This evolution led to the rise of ROP, in which an attacker chains together short instruction sequences ending in indirect branches to achieve arbitrary computation without injecting new code. Over time, ROP has become a central technique for bypassing modern memory-safety mechanisms [4], motivating extensive research into its detection and mitigation.

Common ROP attack strategies typically center on manipulating processor state to invoke sensitive functionality using only existing code. A frequent technique is constructing gadget chains that set up registers for a targeted system call, such as preparing an `execve("/bin/sh", 0, 0)` invocation by loading the appropriate argument registers (e.g., `r0-r2`) and the syscall number (e.g., `r7=11`) before issuing `svc`, which spawns a shell on the ARM platform. Attackers also routinely leverage libc-based ROP, invoking functions such as `mprotect()` or `mmap()` to create executable memory regions. These approaches highlight how ROP enables the execution of complex, high-impact operations without requiring the injection of new code, thereby circumventing modern memory-safety defenses.

Figure 1 illustrates a simple ROP attack on an ARM binary, in which a stack-based buffer overflow is leveraged to invoke `system("/bin/sh")`. After overflowing the buffer (step 1), the attacker overwrites the saved return address with the address of a suitable gadget ending in an indirect branch, such as `pop {r0, r4, pc}`. When the function returns, execution jumps to this gadget (step 2), causing the processor to pop attacker-controlled values from the stack into registers. By placing the address of the string `/bin/sh` at the correct position, the gadget loads it into `r0`, ARM’s first argument register, and subsequently sets the program counter (`pc`) to the address of `system()` (step 3). As a result, the attacker gains arbitrary command execution without injecting code, demonstrating how ROP can bypass non-executable memory protections on ARM architectures.

Actively used defenses offer some protection against memory-corruption attacks but remain insufficient against determined adversaries. Stack canaries [6] insert guard values between local variables and control-flow metadata, allowing the program to detect simple stack-based buffer overflows before the return address is overwritten. ASLR randomizes the locations of executable regions, making it harder for an attacker to predict gadget addresses for ROP construction. However, both defenses can be circumvented in practice. Canary values and randomized library addresses can often be leaked through secondary vulnerabilities, such as format-string bugs, allowing attackers to reconstruct the expected stack layout and bypass these protections. As a result, canaries and ASLR provide only partial mitigation and cannot prevent ROP attacks once information disclosure occurs.

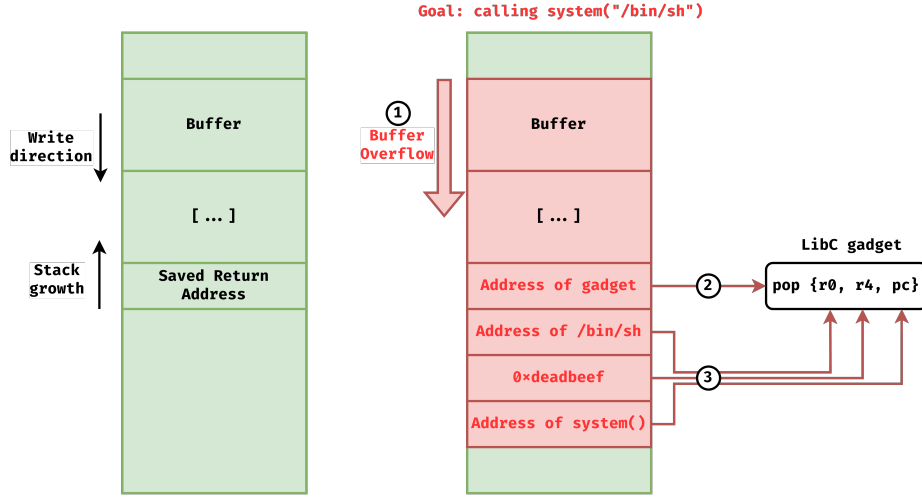


Fig. 1. Example of a simple ROP attack on ARM. A stack buffer overflow overwrites the saved return address with the address of a `pop {r0, r4, pc}` gadget, allowing the attacker to load controlled values into registers and redirect execution to `system()` with `/bin/sh` as its argument, resulting in arbitrary command execution without injecting code.

Modern ARM architectures introduce several hardware-assisted defenses designed to mitigate code-reuse attacks. PAC, introduced in ARMv8.3-A [20], protects both forward- and backward-edge control flow by cryptographically signing return addresses and function pointers with a per-process secret key. When a pointer is used, the CPU verifies the embedded signature; any modification – such as overwriting a return address with a gadget address – results in authentication failure and an exception, effectively preventing classic ROP chains. BTI, standardized in ARMv8.5-A [2], restricts indirect branches so they may only target instructions annotated with valid landing pads. This mechanism blocks control-flow redirection into arbitrary instruction sequences, significantly reducing available gadget surfaces. MTE, also part of ARMv8.5-A [27], associates architectural tags with memory allocations and enforces tag matching on pointer dereferences, enabling the detection of spatial and temporal memory safety violations that often underpin the injection of code-reuse payloads.

While PAC, BTI, and MTE substantially raise the difficulty of exploiting modern ARM systems, a vast ecosystem of legacy devices (before ARMv8) – particularly in embedded, industrial, and IoT deployments – lacks these hardware features entirely. These systems, therefore, remain fully vulnerable to ROP techniques, underscoring the continued relevance of complementary detection mechanisms such as the one presented in this work.

3 Related Work

ROP countermeasure techniques can be broadly classified into two categories: mitigation and detection. Mitigation techniques focus on preventing or limiting the impact of ROP attacks by modifying binary structures, randomizing memory layouts, or enforcing control flow integrity (CFI). Detection techniques, on the other hand, aim to identify ROP attacks by analyzing program behavior, memory contents, or control flow patterns. Since our work focuses on detection rather than mitigation, we present a taxonomy of detection solutions proposed against ROP attacks.

Detection techniques can be deployed in diverse environments, including legacy systems and resource-constrained platforms, where implementing mitigation strategies may be impractical. For instance, ROP mitigation mechanisms that stop the execution of the program when a control flow integrity violation is detected cannot be used in industrial environments where the availability of the functionality and services provided by the program may be a critical requirement. A pure detection-based defense can also tolerate false positives – inherent in anomaly detection mechanisms – better, as they do not necessarily lead to invoking defense mechanisms, which may interfere with critical operations, erroneously. Additionally, detection techniques can complement existing mitigation strategies, adding an extra layer of security against ROP attacks, and can be categorized based on three key aspects: implementation level, hardware reliance, and detection focus.

3.1 Implementation Level

ROP defenses can be categorized by their implementation level: hardware-based approaches, execution environment modifications, and application-level instrumentation.

Hardware-based approaches provide efficient and low-overhead solutions for detecting ROP attacks. CFI enforcement techniques, such as those proposed in [8] and [11], enhance security by ensuring that program execution adheres to a predefined legitimate control flow. The integration of dedicated monitoring hardware, as demonstrated in [18], enables real-time detection of anomalous control flow patterns with minimal performance overhead. Similarly, lightweight hardware-assisted solutions, such as those in [1], leverage architectural extensions to detect and prevent code reuse attacks effectively. For ARM-based systems, approaches like [17] and [23] introduce architectural enhancements to strengthen defenses against ROP attacks without imposing high computational costs.

Detection mechanisms at the **execution environment level**, including OS modules and hypervisors, employ various approaches to identify ROP attacks. Static analysis techniques, such as those in [26,30,32,33], analyze code structure and linkability to detect ROP chains before execution. Machine learning-based methods, including [10,19], and [36], improve detection accuracy by learning program characteristics and identifying anomalous execution patterns. Runtime monitoring techniques, such as [5,22,24], and [25] leverage performance counters

and branch tracing to detect deviations in control flow. Additionally, hypervisor-based solutions like [13] create an isolated execution environment, enforcing security policies and detecting attacks without risking the protected system. PROPS, presented in this work, operates at the execution environment level through a Linux kernel module without modifying the target application.

Beyond the execution environment, detection mechanisms can also be implemented by **instrumenting the target application**. Binary patching techniques, as demonstrated in [9], [12], and [18], modify the application’s binary code to embed detection logic that monitors CFI at runtime. While compile-time and source code-based approaches also fall under application instrumentation, they are primarily designed for mitigation rather than detection.

3.2 Hardware Reliance

ROP detection techniques vary in their reliance on hardware features, ranging from exotic and specialized hardware components to common architectural capabilities and purely software-based methods.

Some approaches leverage **dedicated hardware components** to enhance security and minimize performance overhead. For instance, [18] proposes an ARM-based SoC integration of the ROP monitoring system, enabling real-time attack detection with minimal system disruption. Similarly, [1] introduces a lightweight hardware-assisted scheme for securing embedded systems against code reuse attacks, reducing reliance on software-based defenses. In [11], a specialized control flow monitoring mechanism is employed to protect embedded systems, providing robust defenses against control flow hijacking without significant performance degradation. These methods utilize custom hardware modifications to enforce stricter security policies against ROP attacks.

Several techniques exploit **widely available hardware features** such as performance monitoring counters, branch tracing, and microarchitectural properties for ROP detection. Statistical learning-based detection methods, such as [10] and [36], utilize program execution characteristics to identify anomalies associated with ROP payloads. Approaches like [5] and [24] employ branch tracing mechanisms, including Last Branch Record (LBR), to track control flow deviations in real-time. Hardware Performance Counters (HPCs) are leveraged in [35] and [40] to detect signature-based and performance anomaly-driven indicators of ROP attacks. Additionally, [17] and [23] introduce metadata-driven and architectural refinements in ARM-based multiprocessors to strengthen defenses without excessive computational overhead. Other works utilize lightweight hardware-based runtime integrity checking to prevent exploitation in embedded and constrained environments [13,22]. These methods offer effective and practical ROP detection by leveraging hardware features commonly found in modern processors.

Software-based ROP detection techniques focus on static and dynamic analysis without relying on hardware support. Static analysis-based methods, including [30], [32], and [33], detect ROP gadgets by analyzing binary code structures and linkability patterns. Machine learning-based approaches learn attack

patterns from training data without depending on any hardware-specific features [19]. Runtime software instrumentation techniques, like those in [9] and [12], modify application binaries to enforce CFI. Similarly, speculative execution techniques in [26] attempt to detect ROP payloads without relying on architectural counters or tracing mechanisms. PROPS also operates purely in software, using kernel-level stack monitoring and PCA-based anomaly detection to identify ROP payloads without requiring specialized hardware features, performance counters, or architectural extensions.

3.3 Detection Focus

ROP detection mechanisms vary in their focus, targeting different artifacts and behaviors associated with ROP attacks. These methods can be broadly categorized into payload detection, control flow monitoring, side-channel analysis, and detection based on other artifacts.

Payload-focused techniques aim to identify ROP payloads by analyzing memory regions for suspicious patterns, such as chained return addresses or code pointers in data segments. Static analysis-based methods, including [30,32] and [33], utilize structural analysis and linkability evaluation to identify ROP gadgets before execution. [26] proposes a speculative execution-based approach to detect ROP payloads dynamically, while [13] introduces a hypervisor-based solution. Network-level payload detection, as demonstrated in [31], extends the scope of ROP detection beyond individual systems by analyzing network traffic for payload signatures [14]. Additionally, machine learning-based techniques train models to recognize previously unseen payloads through deep neural networks or data stream analysis, as demonstrated by DeepReturn [19] and Horus [37]. PROPS uses this payload-focused approach by analyzing stack content at vulnerable function return points, using PCA to detect anomalies in stack structure that indicate the presence of ROP chains.

Control flow-based approaches aim to detect deviations from expected execution patterns, focusing on maintaining execution integrity. Shadow stack-based methods, such as [9], track return addresses to prevent hijacked control flow redirections. Fine-grained CFI solutions, including [8,11], and [39], enforce strict execution constraints to detect and mitigate runtime memory attacks. ARM-based solutions like [17] and [18] incorporate monitoring IPs to enforce CFI at the hardware level, while [1] and [23] introduce lightweight metadata-driven mechanisms to secure embedded systems. Additionally, runtime integrity checking techniques, such as [12] and [22], continuously monitor execution flow to detect unauthorized control flow modifications.

Some detection methods utilize **hardware artifacts**, such as branch history, memory access patterns, and performance counters, to identify ROP attacks. Statistical learning-based methods, like [10], leverage program execution characteristics such as memory locality, register traffic, and memory reuse distance to detect anomalies associated with ROP execution. Techniques in [7] and [36] use HPCs to monitor execution for patterns indicative of control flow hijacking. Branch tracing mechanisms, such as LBR, are employed in [5] and [24] to track

indirect branches and detect unusual control flow transitions. Signature-based approaches analyze HPC-based signatures to recognize known ROP attack patterns, including [35] and [40]. Additionally, [16] and [25] propose methods that utilize instruction trace data and branch records to enhance ROP detection precision.

4 Design

In this section, we present the design of PROPS, our ROP detection approach, which relies on the observation that ROP payloads manifest as distinct anomalies in stack memory. In typical exploitation scenarios, the attacker’s payload is injected into the stack, and the attack is triggered at precisely defined control-transfer points, most notably at `pop pc` instructions on ARM platforms, where the program counter is loaded directly from stack data. By monitoring the stack at these critical moments, we can treat ROP behavior as deviations from the statistical and structural properties of clean stack content.

Our overall system architecture, shown in Figure 2, illustrates the flow from clean stack collection and feature extraction to model training and runtime detection. The following subsections detail our feature extraction strategy, the training of our model on benign stack samples, and the runtime detection mechanism used to identify ROP payloads. Details on clean stack collection are provided in Subsection 6.1.

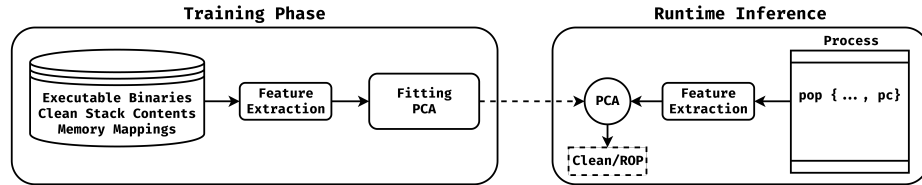


Fig. 2. Overview of PROPS. In the training phase, we collect executables together with clean stack contents and their corresponding memory mappings, apply feature extraction, and fit a PCA model. The model is then deployed to the embedded device. At runtime, the device invokes feature extraction whenever a `pop pc` instruction is executed by the monitored process, feeds the resulting feature vector into the model, and classifies the event as either clean or ROP.

4.1 Feature Extraction

A ROP attack begins when a vulnerable function returns, using attacker-controlled stack data as the return address to redirect control flow. At the moment of this return, the ROP payload is already positioned at the top of the stack, with

subsequent gadget addresses arranged to form the chain. When the function executes a `pop pc` instruction, the program counter is overwritten with the first gadget address, initiating the ROP chain. To analyze this critical boundary, we examine the state of the stack immediately before the `pop pc` executes, focusing on the semantic meaning of the values that will guide subsequent control flow.

Raw 32-bit stack values provide little discriminative information on their own. We therefore perform feature extraction to capture the semantic characteristics of words on the stack. On ARMv7, addresses are 4-byte aligned, allowing each 32-bit word to be interpreted as a potential pointer. Because ARM and Thumb modes differ in instruction width, we analyze either a 40-byte or 20-byte window from the target address. The least significant bit of the address determines the execution mode, and each window corresponds to roughly ten instructions. This provides a mode-robust view of the region referenced by the stack.

For classification, we examine the first 32 words on the stack and assign each a semantic label according to the nature of the region it references. The label set consists of ten categories designed to identify attacker intent, memory semantics, and control-flow-relevant behavior:

1. **F**: points to a `libc` function;
2. **S**: points to a string literal;
3. **N**: disassembly encounters an invalid instruction;
4. **!**: disassembly encounters an `svc` instruction (supervisor call);
5. **G**: disassembly encounters a `pop {..., pc}` instruction;
6. **J**: disassembly encounters a branch-to-register instruction;
7. **X**: valid executable region not matching any special case above;
8. **W**: points to a writable (but not executable) memory region;
9. **R**: points to a readable (but not writable or executable) region;
10. **?**: address does not correspond to any mapped memory region.

This transformation abstracts away concrete addresses and produces a symbolic representation of the stack from the ROP perspective. The resulting label sequence characterizes the structural and semantic properties of the potential payload, enabling reliable discrimination between benign and malicious stack states. The approach remains effective under variations in ASLR, compiler output, and concrete memory layout.

4.2 Training Phase

Designing heuristic rules to characterize clean stack behavior proves impractical because labels that might indicate ROP activity (**G**, **J**, **!**, **F**) also occur naturally in benign execution. Instead, we adopt a systematic machine learning approach that learns the complete distributional patterns of legitimate stack content. In the training phase, we construct a model that learns exclusively from features extracted from *clean* stack content. This follows a semi-supervised learning paradigm in which the model is exposed only to benign samples and must internalize the structural and statistical properties of legitimate execution

contexts. By training solely on clean stack semantics, the model develops a representation of normal behavior, enabling it to detect deviations introduced by ROP payloads during inference.

To capture these structural patterns, we employ PCA, a dimensionality reduction technique that identifies the principal components (directions of maximum variance) in the training data. PCA transforms the high-dimensional stack feature vectors into a lower-dimensional subspace that preserves the most significant variation observed in benign execution. During detection, we project new stack samples into this learned subspace and reconstruct them back to the original feature space. The reconstruction error (the difference between the original sample and its reconstruction) serves as an anomaly score: benign samples that conform to learned patterns exhibit low reconstruction error, while ROP payloads, which deviate from normal stack structure, produce significantly higher errors.

After training, we determine a detection threshold by evaluating the model on the clean training set and computing the empirical distribution of reconstruction errors. The threshold is selected to achieve an acceptable balance between false positives and false negatives by choosing an appropriate quantile of this error distribution. Once the threshold is established, we deploy the trained PCA model together with its detection threshold to the embedded device.

4.3 Runtime Inference

At runtime, the embedded device hosts the fitted PCA model together with the corresponding detection threshold determined during the training phase. Leveraging prior knowledge of software vulnerabilities (e.g., from CVE databases), PROPS monitors only the return points of vulnerable functions rather than all function returns in the system. Specifically, we configure the detection mechanism to activate exclusively at the `pop pc` instructions corresponding to known (or potentially) vulnerable functions. This targeted approach significantly reduces runtime overhead while focusing detection on the specific locations where ROP exploitation is most likely to occur.

When execution reaches a monitored address, the system collects the necessary stack data required for feature extraction. The system applies the same feature extraction procedure used during training to derive the semantic feature sequence from the top of the stack. This feature vector captures the structural patterns present at the point of potential exploitation. This feature vector is then passed to the deployed PCA model for anomaly detection.

The model projects the feature vector into the learned subspace defined by the principal components and reconstructs it back to the original feature space. The reconstruction error quantifies how closely the current stack state resembles the benign patterns learned during training. This reconstruction error serves as an anomaly score: if the error falls below the predefined threshold, the stack state is classified as clean and execution continues normally; if the error exceeds the threshold, the system flags the event as a potential ROP attack and triggers the appropriate response (e.g., logging, alerting, or terminating execution).

5 Implementation

In this section, we describe the implementation details of PROPS. We begin by explaining our data collection mechanism for capturing stack snapshots at monitored points. We then present the practical heuristics employed in our feature extraction method to derive semantic representations from raw stack content. Finally, we detail the specific PCA configuration and parameters used for ROP detection.

5.1 Triggering Mechanism and Data Collection

Our solution must be able to stop the execution of any userspace process at any given point, read the content of its stack (and other memory regions), and access information about the memory mappings of the process. This functionality can be easily implemented using a debugger; however, making sure that all processes that must be monitored are run by a debugger, or debugged by another process, presents integration, performance, and security issues. To avoid these obstacles, we chose another mechanism to implement our solution, called **uprobes**.

Uprobes are a tracing functionality of the Linux kernel¹, which allows placing probes at arbitrary addresses of any userspace application. Uprobes are typically used together with eBPF, which allows loading code into the kernel memory, which is triggered when execution reaches the probes. eBPF code, however, has limited access to the kernel memory; for example, accessing information about the memory mapping of the process is prohibited. To overcome these limitations, we implemented our solution in the form of a Loadable Kernel Module that can be loaded into the kernel anytime and has complete access to kernel memory.

When loaded, our kernel module receives the executable file name and target addresses as parameters. Whenever the specified executable runs, probes are placed at the designated addresses. Whenever a probe is reached, execution is transferred to our uprobe handler function, which collects data about the current state of the process. We collect the top 32 words of the stack content, try to interpret each of them as pointers, save 40 bytes of data from the pointed location, and record the current memory mappings of the process. When data collection is finished, the execution of the application can continue.

In order to integrate this kernel module with the inference system, we use `ioctl`. This is a mechanism of the Linux kernel that allows the transfer of complicated data structures between the user and kernelspace. Our kernel module creates a character device file that a userspace process can read. Whenever a read operation happens, it blocks until there is data to be passed to the userspace. When the handler function finishes data collection, it notifies the process that

¹ Our implementation is Linux-specific, though the detection methodology could be ported to other operating systems with similar tracing capabilities. Importantly, Linux is widely deployed in the legacy embedded systems targeted by PROPS, making this implementation choice highly relevant.

tries to read the device file, data is passed to the userspace, and feature extraction and inference can begin. In the meantime, the monitored application continues its execution, so it is blocked only while data is collected.

5.2 Heuristics of Feature Extraction

Our feature extraction process incorporates several practical heuristics that reflect the characteristics of ARM-based ROP payloads. First, when identifying strings referenced from the stack, we classify an address as pointing to a string if it resolves to a sequence of at least four consecutive printable ASCII characters terminated by a `null` byte. This constraint reduces false positives arising from incidental byte patterns that resemble text.

Second, when examining executable regions, we heuristically disassemble a fixed-length window of up to ten instructions. This length is more than sufficient for characterizing ROP gadgets, as such gadgets typically contain only a few instructions and rarely exceed a length of two to six instructions [29]. The 10-instruction window, therefore, provides sufficient context to determine whether the target address corresponds to a valid gadget while still keeping the analysis computationally lightweight.

Finally, we use a feature vector consisting of the first 32 stack words interpreted as potential addresses. This choice is informed by the fact that addresses on 32-bit ARM systems are 4-byte aligned, and real-world ROP chains are typically much shorter than 32 entries. As a result, a 32-element feature vector captures the full structure of most payloads while maintaining a compact and fixed-size representation suitable for machine learning models.

5.3 PCA-Based Anomaly Detection

To enable PCA-based anomaly detection, we first transform the extracted semantic features into a numerical representation suitable for statistical analysis. Each feature vector consists of 32 categorical labels (one per stack word), where each label can take one of 10 possible values corresponding to the semantic categories identified during feature extraction. We apply one-hot encoding to convert these categorical labels into binary vectors: each of the 32 positions is expanded into a 10-dimensional binary vector with exactly one element set to 1 (indicating the active category) and the remaining are set to 0. This yields a 320-dimensional feature vector that captures the complete semantic structure of the stack snapshot in a suitable format for PCA.

We fit the PCA model exclusively on feature vectors extracted from clean, benign execution traces. The fitting process computes the covariance matrix of the 320-dimensional training vectors and performs eigen decomposition to identify the principal components. These are the orthogonal directions that capture the greatest variance in the benign data. Since benign stack patterns exhibit inherent structure and regularity (e.g., consistent arrangements of return addresses, frame pointers, and local variables), the top principal components effectively encode these normal execution patterns. Lower-variance components, by contrast,

primarily capture noise and minor variations that are less informative for distinguishing benign from malicious behavior. Through hyperparameter search, we determined that retaining the top 207 principal components out of 320 achieves optimal detection performance. These 207 components collectively explain 99.6% of the variance observed in benign stacks, while discarding the remaining 113 components, which contribute minimal information and may introduce sensitivity to noise.

At runtime, we project each incoming 320-dimensional feature vector into the 207-dimensional PCA subspace and reconstruct it back to the original 320-dimensional space. The reconstruction error, computed as the mean squared error between the original feature vector and its reconstruction, serves as the anomaly score. Benign samples that conform to learned patterns exhibit low reconstruction errors, as their structure is well-represented by the principal components. ROP payloads, which introduce anomalous gadget chains and deviate from normal stack semantics, produce significantly higher reconstruction errors that fall outside the benign distribution. To establish a detection threshold, we compute the reconstruction errors for all samples in the clean training set and set the threshold at the 98th percentile of this error distribution. This configuration ensures that 98% of benign samples are correctly classified as clean while maintaining sensitivity to anomalous patterns. The choice of the 98th percentile was determined through hyperparameter search to balance false positive and false negative rates on our test set. Samples with reconstruction errors exceeding this threshold are flagged as potential ROP attacks.

6 Evaluation

To evaluate our approach, we first constructed a dataset that captures both clean execution behavior and realistic ROP attack traces by combining feature vectors as described in Subsection 6.3. Using this dataset, we assessed the detection performance of our PCA model, which is trained offline. The trained model is then installed on our testbed device, a Raspberry Pi3 Model B (RPI3) running the 32-bit version of Raspberry Pi OS Lite. Finally, we measured the runtime performance overhead of our detection method on the RPI3.

6.1 Clean Stack Collection

To fit the PCA model, we require a large and diverse corpus of clean stack samples. Since the model is exposed exclusively to benign stack content during training, the quality and variety of these samples directly influence its ability to learn the structural properties of non-malicious execution.

Our data collection procedure targets all executable binaries present on the RPI3 system, totaling approximately 900 programs. For each binary, we identified all `pop pc` instructions (where a ROP attack could potentially begin) and placed breakpoints at these locations. Each program is then executed under GNU Debugger (GDB). To capture the complete runtime memory context, we

generated a full memory dump once the program reaches its `main()` function, at which point all relevant libraries are typically loaded. Additionally, every time a breakpoint was hit, we extracted 128 bytes from the top of the stack, representing both the region from which a legitimate return address would normally be fetched and the location where a ROP payload would be injected during an attack. This procedure yielded roughly 400,000 stack snapshots, of which 290,000 are unique at the binary level. After applying the feature extraction method described in Section 4.1, we obtained approximately 22,000 unique clean feature vectors that form the basis for fitting the PCA model.

6.2 ROP Chain Generation

Constructing a representative ROP test dataset is a known challenge, as noted in prior works [34]. To generate realistic and diverse ROP payloads, we rely on the Angrop tool [38]. The output of which we validated using a small custom vulnerable program to ensure that the generated chains execute correctly and have the expected effect on our ARM32 platform. For our evaluation, we selected 60 commonly used programs and used Angrop to generate ROP chains that correspond to three typical attacker objectives: spawning a shell, creating a new executable memory region, and making the stack executable. For each objective, we attempted to generate chains using four different methods, which involve calling:

1. functions (`system`, `mmap`, `mprotect`) using gadgets from `libc`,
2. functions (`system`, `mmap`, `mprotect`) using gadgets from the target program,
3. system calls using gadgets from `libc`,
4. system calls using gadgets from the target program.

Because ARM binaries may contain both ARM and Thumb-mode code, we generated each chain in both instruction-set modes whenever possible. Across all combinations of programs, goals, methods, and modes, Angrop produced a total of 1,234 ROP payloads, of which 828 were unique. After applying our feature extraction procedure, we obtained 284 unique ROP feature vectors. Not all cases yielded chains, as Angrop is unable to generate valid ROP payloads for certain binaries or objectives.

6.3 Detection Performance

To evaluate the effectiveness of our approach, we combined clean and ROP feature vectors in a manner that reflects realistic runtime conditions. For each clean feature vector, we insert a ROP feature vector at the stack position corresponding to the instruction at which the clean snapshot was captured. For example, if the triggering instruction is `pop {r4, r5, r6, pc}`, the correct offset is 3, ensuring that the first gadget address appears at the slot that would be popped into the program counter.

We split the clean dataset into 80% for training (17,600 samples) and 20% for testing (4,400 samples). To ensure robustness, we performed repeated K-fold

cross-validation across all experiments (5 splits, 2 repeats). For ROP evaluation samples, we ensured that the clean feature vectors into which ROP payloads were inserted were never part of the training set, preventing data leakage. This resulted in more ROP feature vectors than clean samples, so we randomly selected 4,400 ROP features to balance the test set.

The receiver operating characteristic (ROC) curve for one representative fold is shown in Figure 3a. Across all folds, the PCA model achieved an average area under the curve (AUC) of 0.9868. We determined the operational detection threshold as the 98th percentile of reconstruction errors observed in the clean training set. Using this threshold, our model achieved an average detection accuracy of 98.24%, along with 97.16% precision, 99.38% recall, and a 98.26% F1-score, as shown in Figure 3b. Performance remained consistently high across all test folds.

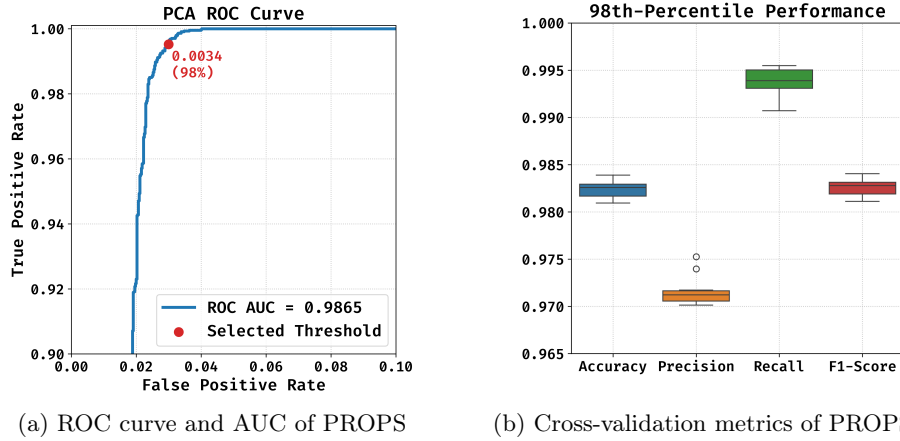


Fig. 3. The left subfigure shows that lowering the detection threshold increases the true positive rate but also raises the false positive rate. Our evaluation used a threshold set at the 98th percentile of reconstruction errors observed on the training set, which achieved the balance reflected in the performance metrics. The boxplots in the right subfigure show the distribution of accuracy, precision, recall, and F1-score across all cross-validation folds.

6.4 Runtime Performance

The runtime overhead of PROPS is determined by the time the monitored process spends blocked during kernel module data collection triggered by the placed uprobes. Once data collection completes, the process resumes execution while feature extraction and PCA inference occur asynchronously in userspace via the `ioctl` interface. Thus, the overhead perceived by the application consists only

of the data collection phase within the kernel module. The processing time for data collection is largely constant per invocation, except for the variable cost of traversing the process’s memory mappings, which depends on the number of memory regions allocated by the program. The frequency of kernel module interrupts depends on both the program being monitored and its execution arguments, as these factors determine how often the monitored return points are reached during execution. In practice, the blocking time per invocation is minimal: the kernel module completes data collection in a mean of 53 μ s, with 95% of invocations under 106 μ s (based on the handler invocations across the test programs evaluated in the following paragraph).

To evaluate overall runtime performance, we measured the overhead of PROPS on 60 common utility programs from `/usr/bin/`. For each program, we established a baseline execution time by running it 100 times without uprobes and computing the average. We then placed 1, 3, 6, and 12 uprobes at different addresses and measured the resulting overhead. All measurements were performed using the `perf` tool, with each configuration executed 100 times to obtain reliable averages. Table 1 presents the detailed results, showing baseline execution times and percentage overhead for each uprobe configuration.

Table 1. Runtime overhead of PROPS with varying numbers (1, 3, 6, 12) of monitored uprobes. Baseline shows average execution time in milliseconds; overhead columns show percentage increase relative to baseline. The dash (–) indicates the program does not have the specified number of reached uprobes.

Program name	Base. (ms)	1up. (%)	3up. (%)	6up. (%)	12up. (%)	Program name	Base. (ms)	1up. (%)	3up. (%)	6up. (%)	12up. (%)
base64	4.67	-19.95	-16.62	1.06	–	ln	3.85	8.71	9.76	53.96	–
cat	3.54	4.82	10.26	15.68	–	make	6.06	-24.33	-23.17	-20.57	-7.41
chmod	4.48	-15.35	-12.56	-8.95	2.16	md5sum	3.60	5.72	9.12	11.68	274.36
chown	3.91	6.40	7.35	22.65	30.68	mkdir	5.33	16.64	7.65	4.09	–
clear	3.99	5.18	8.46	–	–	mv	5.96	152.74	8.69	172.80	29.19
cp	8.78	-9.96	-8.53	-8.28	-2.53	pgrep	51.86	0.04	1.87	1.92	–
cpio	4.27	40.27	35.33	15.46	25.82	ping	7.67	0.49	2.76	7.92	1339.64
curl	52.50	5.00	5.95	4.86	5.53	printf	3.38	8.42	13.90	28.90	–
cut	4.50	-15.68	-6.27	-9.79	7.48	ps	42.41	-1.38	1.44	-0.48	1.48
date	4.79	-18.76	-13.82	11.55	-2.69	rm	4.33	-1.93	10.71	12.61	73.96
dd	243.44	1.40	26.76	27.86	92.12	scp	6256.87	-0.76	-0.68	-0.98	-0.97
df	4.95	6.68	16.52	31.75	35.74	sed	4.87	6.67	10.24	1491.05	24.50
diff	5.19	7.14	17.46	13.30	22.02	sftp	6185.63	0.23	0.28	0.59	0.42
du	67.94	-0.69	0.59	0.47	1.20	sha256sum	4.85	-23.11	-20.58	-13.21	-5.82
echo	4.66	-20.94	8.02	9.64	–	sleep	4.97	-21.16	-18.73	–	–
env	3.52	8.62	13.05	23.65	–	sort	4.94	-18.05	11.26	-8.45	-2.64
expr	5.03	-16.80	-13.93	-10.77	-4.82	ssh	6170.44	0.13	0.21	0.35	0.12
false	3.07	4.91	–	–	–	tail	3.70	7.89	11.33	24.88	26.48
find	26.20	-1.86	-3.08	-2.71	-0.85	tar	9.98	76.24	-15.33	852.91	-15.40
free	9.78	-18.73	-4.51	-1.17	–	test	3.47	10.02	11.59	16.88	–
gawk	10.10	-12.98	-13.03	-14.17	-9.53	top	216.93	-1.83	-0.67	0.62	-2.22
git	16.25	-1.51	2.53	1.04	6.95	touch	4.80	-21.56	-19.20	4.08	–
grep	4.71	26.51	7.11	11.96	18.09	true	4.06	-21.12	–	–	–
head	4.88	-17.62	-16.84	-8.59	-6.57	uniq	3.64	7.99	11.64	42.03	62.00
hostname	3.23	2.27	10.06	–	–	unzip	7.43	4.52	7.67	20.65	10.18
id	8.13	-18.54	-0.78	-3.86	–	uptime	7.41	2.37	8.83	–	–
less	7.30	18.34	2.93	0.90	11.29	wc	3.60	4.52	8.05	14.40	25.14
link	4.25	21.57	16.25	5.94	–	wget	57.42	-36.16	-35.03	-19.99	598.94
ls	6.29	5.70	10.82	12.70	22.13	whoami	3.81	7.68	15.47	15.65	–
lsblk	15.72	1.55	1.51	5.41	8.64	zip	9.05	-5.42	-1.82	3.14	13.11

Monitoring a single return point introduces an average overhead of just 2.02%, which remains nearly unchanged at 2.04% with 3 uprobes. Overhead grows more substantially with 6 uprobes (53.06%) and 12 uprobes (67.07%). However, these averages mask significant variation in individual program performance. Short-lived utilities with fast baseline execution times (e.g., `cat`, `clear`, or `cpio`) show higher percentage overheads because the fixed cost of data collection dominates their total runtime. Conversely, long-running programs with substantial computation between return points (e.g., `ssh` with 6170.44 ms baseline) show minimal overhead even with 12 uprobes. Some programs exhibit negative overhead values, indicating that instrumented runs occasionally completed faster than baseline measurements. These values fall within measurement noise and reflect the inherent variance in timing short executions, as well as fluctuations in system load between measurement runs.

Nevertheless, these results validate PROPS’s precision monitoring approach: the system achieves low overhead when monitoring 1-3 function return points. Targeting only critical vulnerable functions, rather than comprehensive system-wide monitoring, is essential for practical deployment.

7 Discussion

While PROPS effectively detects ROP attacks on legacy ARM platforms, it has some limitations and practical considerations that merit discussion.

Detection vs. Prevention. PROPS focuses on detection rather than active prevention. If an attacker successfully escalates privileges to root before detection occurs, they could disable the detector entirely. This limitation is inherent to monitoring-based approaches and highlights the importance of complementary defensive measures.

Linear Disassembly Trade-offs. Our linear disassembly approach prioritizes speed over comprehensiveness, resulting in three related blind spots. First, gadgets containing backward jump instructions are misclassified as generic executable code (X) rather than explicit gadgets (G or J), since our disassembler does not follow the true control flow. Second, stack pivoting attacks are only partially visible: we capture the initial pivot gadget address, but subsequent chain execution in a different memory region remains undetected. Third, Jump-Oriented Programming (JOP) attacks, which replace traditional `pop pc` chaining with a dispatcher-based execution model using indirect branches like `bx reg` or `blx reg`, are only detectable during their initialization phase. Once the dispatcher begins invoking functional gadgets from an attacker-controlled table in a separate memory region (see [3] for detailed JOP mechanics), the execution chain becomes invisible to PROPS. More sophisticated disassembly techniques, such as symbolic execution, could address these limitations. However, such approaches would significantly increase analysis complexity and runtime overhead, undermining our core design goal of practical deployability on resource-constrained legacy systems. Our current method favors speed and simplicity while still effectively defending against the most common ROP attack vectors.

Deployment Considerations. Effective deployment of PROPS requires careful uprobe placement. Monitoring every `pop pc` instruction would provide complete coverage but introduces prohibitive overhead. Instead, PROPS achieves practical performance by monitoring only the return points of known (or potentially) vulnerable functions, identified through CVE analysis or vulnerability assessments. This precision monitoring approach enables operators to protect legacy systems without patching vulnerabilities or understanding their implementation details. This is particularly valuable in critical infrastructure environments where updates are operationally infeasible due to uptime requirements, certification constraints, or vendor support limitations.

PROPS has been evaluated primarily using artificially generated ROP chains to systematically assess detection capabilities across diverse gadget patterns. Future work will validate PROPS against real-world exploits and measure its actual runtime overhead on the corresponding vulnerable programs.

8 Conclusion

We presented PROPS, a novel ROP detection mechanism for legacy ARM systems that combines stack content analysis with PCA-based anomaly detection. By learning exclusively from benign stack patterns, PROPS effectively identifies ROP attacks without requiring attack-specific signatures. Our approach addresses a critical gap in the security of 32-bit ARMv7 and older platforms, where modern hardware defenses like PAC and BTI are unavailable. By leveraging Linux kernel uprobes and monitoring only the return points of known (or potentially) vulnerable functions, PROPS provides a deployable solution for embedded systems where patching is operationally infeasible. Our evaluation demonstrates that PROPS achieves 98.24% detection accuracy while maintaining practical runtime overhead.

Acknowledgments. The authors would like to thank Dorottya Futóné Papp and Gábor Pék for their invaluable feedback on the preliminary version of this paper, as well as the other members of CrySyS Lab for their insightful discussions related to this project. The authors also thank Zoltán Balázs, Dániel Kiss, Márton Bak, and István Lám for their valuable comments and suggestions on this paper. Finally, the first author was supported by the National Research, Development and Innovation Fund of Hungary (Doctoral Excellence Fellowship Program and NKKP-ADVANCED_25-153325).

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. An, Z., Wang, W., Li, W., Li, S., Zhang, D.: Securing Embedded System from Code Reuse Attacks: A Lightweight Scheme with Hardware Assistance. *Micromachines* (2023). <https://doi.org/10.3390/mi14081525>

2. ARM Ltd.: ARM Architecture Reference Manual for ARMv8.5-A: Branch Target Identification (2019), available at: <https://developer.arm.com/documentation/ddi0487/latest/>
3. Bletsch, T., Jiang, X., Freeh, V.W., Liang, Z.: Jump-oriented programming: a new class of code-reuse attack. In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (2011). <https://doi.org/10.1145/1966913.1966919>
4. Carlini, N., Wagner, D.: Rop is still dangerous: breaking modern defenses. In: Proceedings of the 23rd USENIX Conference on Security Symposium (2014)
5. Cheng, Y., Zhou, Z., Yu, M., Xuhua, D., Deng, R.: Ropecker: A generic and practical approach for defending against rop attacks. In: NDSS Symposium 2014 (2014). <https://doi.org/10.14722/ndss.2014.23156>
6. Cowan, C., Pu, C., Maier, D., Hintony, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q.: Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In: Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7 (1998)
7. Das, S., Chen, B., Chandramohan, M., Liu, Y., Zhang, W.: ROPSentry: Runtime defense against ROP attacks using hardware performance counters. *Computers & Security* (2018). <https://doi.org/10.1016/j.cose.2017.11.011>
8. Das, S., Zhang, W., Liu, Y.: A Fine-Grained Control Flow Integrity Approach Against Runtime Memory Attacks for Embedded Systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2016). <https://doi.org/10.1109/TVLSI.2016.2548561>
9. Davi, L., Sadeghi, A.R., Winandy, M.: ROPdefender: a detection tool to defend against return-oriented programming attacks. In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security. Hong Kong China (2011). <https://doi.org/10.1145/1966913.1966920>
10. Elsabagh, M., Barbara, D., Fleck, D., Stavrou, A.: Detecting ROP with Statistical Learning of Program Characteristics. In: Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy (2017). <https://doi.org/10.1145/3029806.3029812>
11. Francillon, A., Perito, D., Castelluccia, C.: Defending embedded systems against control flow attacks. In: Proceedings of the first ACM workshop on Secure execution of untrusted code (2009). <https://doi.org/10.1145/1655077.1655083>
12. Huang, Z.J., Zheng, T., Liu, J.: A dynamic detective method against ROP attack on ARM platform. In: 2012 Second International Workshop on Software Engineering for Embedded Systems (SEES) (2012). <https://doi.org/10.1109/SEES.2012.6225491>
13. Jiang, J., Jia, X., Feng, D., Zhang, S., Liu, P.: HyperCrop: A Hypervisor-Based Countermeasure for Return Oriented Programming. In: Information and Communications Security. Springer Berlin Heidelberg (2011). https://doi.org/10.1007/978-3-642-25243-3_29
14. Jämthagen, C., Karlsson, L., Stankovski, P., Hell, M.: eavesROP: Listening for ROP Payloads in Data Streams. In: Information Security. Springer International Publishing, Cham (2014). https://doi.org/10.1007/978-3-319-13257-0_25
15. Kim, S.K., Jin, H., Joo, K., Lee, J., Lee, D.H.: Dropsys: Detection of rop attacks using system information. *Comput. Secur.* (2024). <https://doi.org/10.1016/j.cose.2024.103813>
16. Koranek, D.F., Graham, S.R., Borghetti, B.J., Henry, W.C.: Identification of Return-Oriented Programming Attacks Using RISC-V Instruction Trace Data. *IEEE Access* (2022). <https://doi.org/10.1109/ACCESS.2022.3170479>

17. Lee, Y., Heo, I., Hwang, D., Kim, K., Paek, Y.: Towards a practical solution to detect code reuse attacks on ARM mobile devices. In: Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy (2015). <https://doi.org/10.1145/2768566.2768569>
18. Lee, Y., Lee, J., Heo, I., Hwang, D., Paek, Y.: Integration of rop/jop monitoring ips in an arm-based soc. In: 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE) (2016)
19. Li, X., Hu, Z., Wang, H., Fu, Y., Chen, P., Zhu, M., Liu, P.: DeepReturn: A deep neural network can learn how to detect previously-unseen ROP payloads without using any heuristics. Journal of Computer Security (2020). <https://doi.org/10.3233/JCS-191368>
20. Liljestrand, H., Nyman, T., Wang, K., Perez, C.C., Ekberg, J.E., Asokan, N.: PAC it up: Towards pointer integrity using ARM pointer authentication. In: 28th USENIX Security Symposium (USENIX Security 19) (2019)
21. National Vulnerability Database: CVE-2015-5374: En100 ethernet module denial-of-service vulnerability. NVD – National Vulnerability Database (2015), <https://nvd.nist.gov/vuln/detail/CVE-2015-5374>, accessed: 2025-11-26
22. Neugschwandtner, M., Mulliner, C., Robertson, W., Kirda, E.: Runtime Integrity Checking for Exploit Mitigation on Lightweight Embedded Devices. In: Trust and Trustworthy Computing. Springer International Publishing (2016). https://doi.org/10.1007/978-3-319-45572-3_4
23. Oh, H., Cho, Y., Paek, Y.: A metadata-driven approach to efficiently detect code-reuse attacks on ARM multiprocessors. The Journal of Supercomputing (2021). <https://doi.org/10.1007/s11227-020-03542-1>
24. Pappas, V., Polychronakis, M., Keromytis, A.D.: Transparent rop exploit mitigation using indirect branch tracing. In: Proceedings of the 22nd USENIX Conference on Security (2013)
25. Pfaff, D., Hack, S., Hammer, C.: Learning How to Prevent Return-Oriented Programming Efficiently. In: Engineering Secure Software and Systems. Springer International Publishing (2015). https://doi.org/10.1007/978-3-319-15618-7_6
26. Polychronakis, M., Keromytis, A.D.: ROP payload detection using speculative code execution. In: 2011 6th International Conference on Malicious and Unwanted Software (2011). <https://doi.org/10.1109/MALWARE.2011.6112327>
27. Serebryany, K., Stepanov, E., Shlyapnikov, A., Tsyurklevich, V., Vyukov, D.: Memory tagging and how it improves c/c++ memory safety. arXiv preprint arXiv:1802.09517 (2018)
28. Shacham, H.: The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In: Proceedings of the 14th ACM Conference on Computer and Communications Security (2007). <https://doi.org/10.1145/1315245.1315313>
29. Si, L., Yu, J., Luo, L., Ma, J., Wu, Q., Li, S.: Rop-hunt: Detecting return-oriented programming attacks in applications. In: Security, Privacy, and Anonymity in Computation, Communication, and Storage (2016). https://doi.org/10.1007/978-3-319-49148-6_12
30. Stancill, B., Snow, K.Z., Otterness, N., Monrose, F., Davi, L., Sadeghi, A.R.: Check my profile: Leveraging static analysis for fast and accurate detection of rop gadgets. In: International Symposium on Recent Advances in Intrusion Detection (2013). https://doi.org/10.1007/978-3-642-41284-4_4
31. Tanaka, Y., Goto, A.: n-ROPdetector: Proposal of a Method to Detect the ROP Attack Code on the Network. In: Proceedings of the 2014 Workshop on Cyber Se-

- curity Analytics, Intelligence and Automation (2014). <https://doi.org/10.1145/2665936.2665937>
32. Usui, T., Ikuse, T., Iwamura, M., Yada, T.: POSTER: Static ROP Chain Detection Based on Hidden Markov Model Considering ROP Chain Integrity. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (2016). <https://doi.org/10.1145/2976749.2989040>
 33. Usui, T., Ikuse, T., Otsuki, Y., Kawakoya, Y., Iwamura, M., Miyoshi, J., Matsuura, K.: ROPminer: Learning-Based Static Detection of ROP Chain Considering Linkability of ROP Gadgets. IEICE Transactions on Information and Systems (2020). <https://doi.org/10.1587/transinf.2019ICP0016>
 34. Wang, H., Singhal, A., Liu, P.: Tackling imbalanced data in cybersecurity with transfer learning: a case with ROP payload detection. Cybersecurity (2023). <https://doi.org/10.1186/s42400-022-00135-8>
 35. Wang, X., Backer, J.: SIGDROP: Signature-based ROP Detection using Hardware Performance Counters (2016). <https://doi.org/10.48550/arXiv.1609.02667>
 36. Welearegai, G.B., Hu, C., Hammer, C.: Detecting and Preventing ROP Attacks using Machine Learning on ARM. In: 2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC) (2023). <https://doi.org/10.1109/COMPSAC57700.2023.00092>
 37. Yang, G., Liu, X., Tang, C.: Horus: An Effective and Reliable Framework for Code-Reuse Exploits Detection in Data Stream. Electronics (2022). <https://doi.org/10.3390/electronics11203363>
 38. Zeng, K., Schloegel, M., Salls, C., Doupé, A., Wang, R., Shoshitaishvili, Y., Bao, T.: ropbot: Reimaging code reuse attack synthesis. In: Network and Distributed System Security (NDSS) Symposium (2026). <https://doi.org/10.14722/ndss.2026.240845>
 39. Zhang, M., Sekar, R.: Control flow and code integrity for cots binaries: An effective defense against real-world rop attacks. In: Proceedings of the 31st Annual Computer Security Applications Conference (2015). <https://doi.org/10.1145/2818000.2818016>
 40. Zhou, H., Wu, X., Shi, W., Yuan, J., Liang, B.: HDROP: Detecting ROP Attacks Using Performance Monitoring Counters. In: Information Security Practice and Experience. Springer International Publishing, Cham (2014). https://doi.org/10.1007/978-3-319-06320-1_14