Run-Time ROP Attack Detection on Embedded Devices Using Side Channel Power Analysis

Jinyao Xu, Danny Abraham, Ian G. Harris

***Abstract*—Return-oriented programming (ROP) have emerged as great threats to modern embedded systems. ROP attacks can be used to either bypass credential verification or modify RAM contents. In this paper, we introduce a simple side-channel technique for run-time ROP detection. We use processors' power consumption pattern as an indicator for potential ROP attacks, which can be deployed across different platforms. We avoid the computational complexities of training machine learning models by using a simple linear comparison algorithm to compare known and unknown power patterns to discern anomalies. For evaluation, we implement both ROP attacks in multiple scenarios on benchmarks with various complexity levels. We demonstrate the robustness of our approach and also outline some potential overheads that the approach incurs for run-time ROP detection.**

***Index Terms*—Return-Oriented Programming, Embedded System Security, Threat Detection.**

I. Introduction

Embedded systems—low in power consumption and versatile in functionality—play vital roles in building modern "smart" societies. Given the extensive existence of embedded systems, ranging from avionic bootloaders to cars' remote fobs, securing these bare-metal devices has always been an important topic investigated by security researchers [1]. Lacking OS protections for stacks, these bare metal embedded systems are commonly exploited by attackers through Return-Oriented Programming (ROP), a code execution hijacking technique where an attacker overflows the call stack to re-write return address and gaining control over the program's control flow [9]. By hijacking the program counter to execute arbitrary code in program memory, attackers exploit instruction "gadgets" that end with another return instruction. Together, these gadgets can lead to attacker-defined malicious behaviors that are hard to detect at run time.

In this paper, we address the issues above by 1) providing an overview of the power side channel power analysis and its ability to be used on security defenses, 2) showing how to deploy the technique to an embedded platform using power analysis hardware, 3) evaluating the technique's performance for run-time ROP attack detection, 4) outlining the technique's potential trade offs in its deployment.

II. Threat Model

We assume that there may exist a software vulnerability which allows a malicious party to execute a ROP attack against the embedded system. Such vulnerability usually comes in form of unchecked buffer.

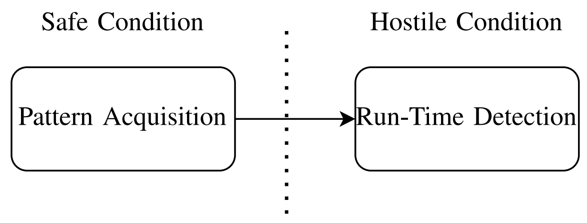

Fig. 1. Overview of Attack Detection Scheme

1) Existing research assumes a common rate of 11 attacks out of 1000 testing iterations (around 1%) [3]. However, other research also uses a 1:1 ratio of attacking vs. normal scenario to test its technique [4]. In a real world scenario, attacks occur sporadically. For this paper, we assume that attacks are sporadic (*non-clustered*), which are harder to detect given their low frequency. We assume that they happen 1% of the time during the validation process.
2) Attackers typically chain multiple code gadgets. For embedded devices, a chain of gadgets can be as short as 10-15 instructions to move a register's value to another [10]. In our experiment, we consider the worst possible attacking case where the attacker only uses 1 ROP gadget and such gadget contains no more than 6 assembly instructions.

III. Model Framework

Side Channel Power Analysis aims to gain program's execution insights via its CPU's power consumption over time [5], [6]. It is an non-intrusive technique that is easy to set up for both defense and attacking purposes [5]. In order to measure power of the CPU, a parallel circuit can be added between MCU's Vdd pin and the power supply. To avoid shorting the circuit, the new circuit requires an additional resistor between two measuring points. The change of voltage can be obtained through Ohm's law, where the voltage changes correspondingly to the current and resistance, we apply as constants to the circuit.

A. Framework High-Level Overview

Shown in figure 1, we first operate the device under the safe condition to acquire valid power patterns of a specific program. Once the program does not identify new branching patterns), the device is placed in the hostile environment where it's power is monitored and compared to the known patterns.

B. Side Channel Power Analysis For Return Anomalies

Based on the power pattern differences over a program's execution, we can monitor the behavior after a program executes the "return" instruction from either a function call or if/else statement. This means that a hardware trigger is needed from the embedded device to tell the sample collector when to start sampling. Most embedded devices have General Purpose Input/Output (GPIO) ports that communicate with external devices. Driving the GPIO signal to 3.3 V before return and jump instructions, the detection device will be notified to sample via its ADC and transmit the collected sample to the host machine via UART, where data will be processed. Because ROP gadgets have different instructions from the normal program's execution, the power pattern will be different. In the actual industrial setting, it is the programmer's job to write codes that drives the GPIO pins at security-critical parts of the program.

Figure 3 shows an example of a ROP attacked pattern vs a normal pattern for a sample searching. In this proof of concept, we used the Binary Search program where the attacking gadget contains 6 ALU operations. The x-axis shows the sample points overtime whereas the y-axis measures the voltage, calculated by equation (1) From sample No. 60 to No. 80, a clear power difference between ROP and Non-ROP attacked program can be spotted: the normal program executes returns from its call stack whereas the attacked program yields additional ALU operations before returning from the last recursive call (sample No.80 to No.100). Gadgets could be chained to form longer ROP chains, yet in this proof of concept, a gadget as short as 6 ALU instructions can be captured by the power analyzer.

$$voltage = -current * resistance * gain. \qquad (1)$$

IV. Power Profile Matching

Instead of using machine learning model or Hardware Performance Counter, we use a simple linear comparison algorithm to check the similarities of power patterns. When comparing power profiles such as those shown in Figure 3, each corresponding data point is compared with a "tolerance of difference" of 0.1 mV. To ensure pattern integrity, we also specify the minimum valid length as a parameter. Only when a consecutive matching pattern $>= min\_sequence$ can such pattern matching be considered valid. Algorithm 1 shows the pseudocode for our matching score calculation function. The linear comparison Algorithm 1 has a $O(n)$ complexity and runs in a predictable time for run-time data analysis.

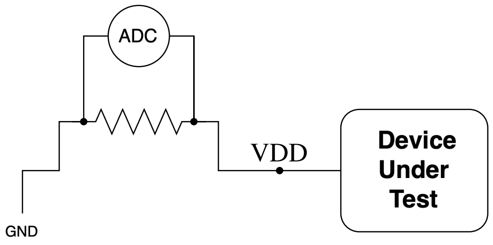

Fig. 2. Power Measuring Schematics

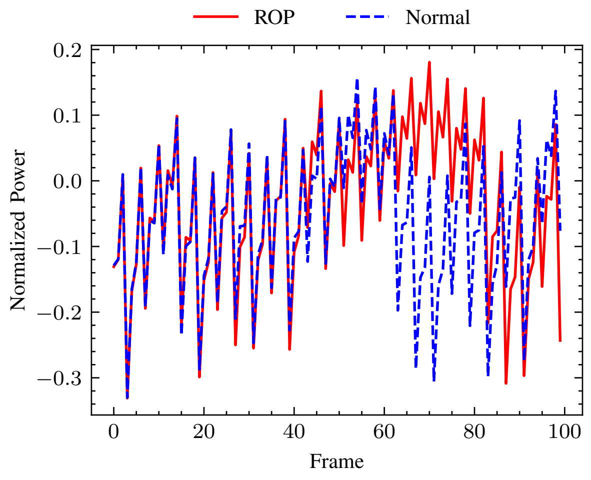

Fig. 3. Normal Execution v Rop Execution

Algorithm 1: linear comparison

```
Input: Pattern1, Pattern2, threshold, min sequence
Output: Matching Score
/* initialize variables */
score=0
counter=0
/* iterate through the pattern */
for each sample1, sample2 in Pattern1, Pattern2 do
    if abs(sample1-sample2) < threshold then
        /* aA single sample match */
        counter+=1
    else
        if counter >= min sequence then
            /* Matching Sequence is Long
               enough, add to score */
            score+=counter
            counter=0
        else
            counter=0
return score
```

Pattern Acquisition: At pre-deployment phase, We repeatedly collect samples for 100 rounds to run the program with random inputs so we can cover different branches, return/jump instructions with conditions, etc. At this phase, a sample pattern will be checked against existing samples using Algorithm 1. Should the pattern be "new"—a linear comparison that yields a score lower than 70% to any known ones—the new pattern will be added to the pattern collection.

A. Runtime Attack Detection

To determine the matching score threshold for the actual attacks, we run the programs and collect their power profiles, matching them against the known legal patterns to get a matching score distribution. Equation (2) is used to calculate the score for distinguishing normal and attacked programs.

$$mean(scores) - std.deviation(scores) \quad (2)$$

V. Evaluation

Based on the following research questions, we present our evaluation methodologies.

1) **RQ1:** Using side-channel power analysis techniques, what accuracy and false positive rate at run-time can we achieve for detecting ROP attacks?
2) **RQ2:** What are the computation trade offs for using side-channel power analysis? How does such trade off relate to the programs' complexity.

To address the first question, we benchmark our technique on 10 different benchmarks with various code structures and complexities. The results are evaluated based on its accuracy and false positive rate. To see the trade offs for the power side channel approach, we vary delay times and measured degradation of accuracy.

A. Experiment Setup

For the experiment, we use the ChipWhispererLITE (CWLite) with its default XMEGA target [7]. The CWLite has an Xilinx SPARTAN 6 chip as its main processing unit and runs on its 5.7.0 firmware version and can be interfaced with Jupyter notebook. ChipWhisperer is an open-source, low-cost solution to expose weaknesses that exist in embedded systems [2].

The target board has an Atmel AVR instruction set architecture and connects to the CWLite via its measure port JP10, over a shunt resistor. The trigger is connected via CWLite's GPIO D pin, on both the target and the CWLite device. The testing programs are cross compiled using the Brew AVR GCC compiler on a Mac OS Ventura 13.4 host machine. Programs are subsequently loaded onto the XMEGA target via CWLite's serial port. Configuration of the CWLite device such as sampling size, sampling speed, uart protocols is done via CWLite's python API functions.

Programs and Inputs: We use the same benchmarks used in Omotosho et al's paper [8], including various known algorithms like Depth First Search, Binary Search, Kruskal, etc. All benchmark programs are written in C with different run time complexities and vulnerability points. The average run time for a full iteration and the programs' complexity are listed in the second and third columns of Table I.

Inputs to these benchmarks are randomly generated at each iteration via the $rand()$ and other key functions from the AVR standard library. For example, in the binary search program, the input array is first generated by the $rand()$ function and are then sorted in-place using $qsort()$.

Vulnerability and Gadgets: Vulnerability points of a program are identified after return or jump instructions that can be impacted directly by the input. For this paper, this is done manually via code inspections. We first inspect the pseudo-codes of the benchmark programs and identify major components of them. However, this process can be automated via static analysis tools.

Sampling Triggers: Return-oriented attacks alter system behavior at return instructions, so we sample power profiles starting at the execution of return instructions. The sampling starting point will be notified via a hardware trigger. Before each vulnerable return or jump, the hardware trigger on the target board (GPIO D) goes high to notify the CWLite (GPIO D) port to sample a fixed amount of power samples, measured in Volts. Trigger's (dis)activation is implemented via CWLite's software API $trigger\_high()$ and $trigger\_low()$. The $trigger\_high()$ drives the target board's GPIO D port to 3.3V where $trigger\_low()$ resets the port to 0V.

Attack Simulation: Malicious modifications are made to the programs by the addition of an extra function call $ROP()$. These functions are placed at programs' vulnerability points with assembly instructions mimicking a buffer modification and register modifications. Test iterations are measured in terms of $ROP()$ function's execution: whenever a program executes through one of its vulnerability points, we consider this as a test instance where samples will be collected and matched against known ones. The ROP() function is triggered at random using hardware's random number generator.

TABLE I
Benchmark Program and Pattern Acquired

Program	Time(ms)	Complexity	# Vulnerabilities	Gadget Length
Binary Search	0.0015	$O(logn)$	3	6
Bellman-Ford	0.026	$O(VE)$	2	2
DFS	0.037	$O(V+E)$	2	6
Kruskal	0.014	$O(ElogE)$	4	6
Floyd Warsall	0.033	$O(n^3)$	2	5
Merge Sort	0.018	$O(nlogn)$	4	6
LCS	0.1036	$O(mn)$	3	5
Prim	0.004	$O(ElogV)$	2	3
Huffman	0.023	$O(nlogn)$	2	6
RabinKarp	0.008	$O(mn)$	4	4

Delays: Given the fact that benchmark evaluations are done on a Mac OSX platform using Python, we injected NOP delays before each trigger up event to give the operating system (OS) enough time processing the collected sample during the benchmark's runtime. Given the XMEGA device operates at 7.3 MHz clock [7], we stress tested each benchmark's minimum delay to detect the first correct ROP attack and uses it to evaluate each of our benchmark.

Attack Frequencies: In our benchmarking process, 4 out of 1024 samples are maliciously infected and each happens once randomly in every 256 test instances, yielding a 0.3% chance of encountering a ROP attack.

Validation: We use the matching threshold obtained during the training phase as the deciding point. For metrics, detection accuracy and false positive rate are calculated using the following equations.

$$accuracy = (TruePositive + TrueNegative)/Total$$

$$false\ positive = FalsePositive/Total$$

In these equations, the term "positive" means an attack, vice versa for the term "negative".

VI. Results and Discussion

In this section, we present our results and answer our research questions based on the result.

Performance: Evaluated among these benchmarks, our side-channel method has significant accuracy for embedded devices. Shown in Figure 4, accuracy tends to be high (average of 80%) and false positive rate tends to be low for all programs except the Longest Common Sub-sequence (35% accuracy) and (60% false positive). Although not a complicated program, its recursive branches are complicated in tree-structure representation. Hence the variations in recursive inputs affect accuracy. Similar for Kruskal's algorithm: the complexity in branching for Minimum spanning tree problem causes larger power variation, which makes the attacks harder to detect.

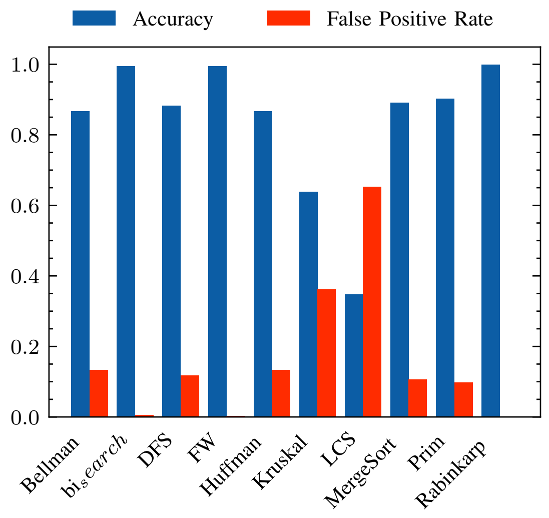

Fig. 4. Benchmarking Results

Trade offs: Depending on the program complexity, time delay required for host machine to process decision is measured by number of NOP delays, shown in Figure 5. In the x axis, we rank the program's computation complexity in an ascending order and the y axis shows the delay (in terms of NOP instructions) time. The graph roughly shows a positive correlation between computation complexity and time of delay.

Implications: With our findings, we list further research areas.

- Performance of this technique can be improved as gadgets get longer, or chained in a longer sequences. In our experiment, we test the worst case scenario– one ROP attack gadget chain only. We are uncertain about the matching score's correlation with larger size of malicious codes or gadget chain length.
- Currently, a testing pattern has to be tested with all acquired patterns before a concluding a potential attack. This scheme may be improved by using hash tables to avoid iterative matching each pattern. Doing this may introduce additional space overheads and increase overall resource usage, especially for resource constrained machines.

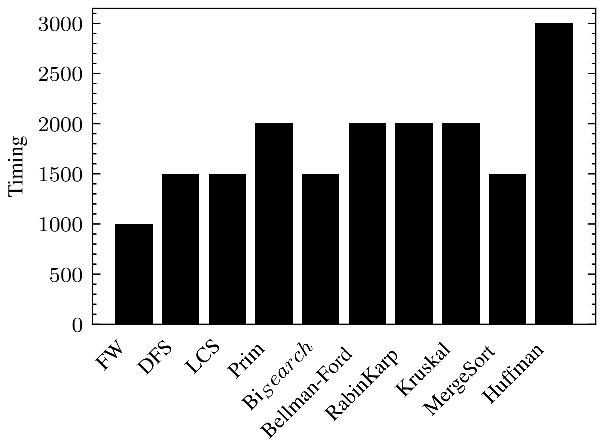

Fig. 5. Minimum Time Delay

- The actual power measurement can be done on a pure hardware than a full OS based machine. For example, a bare-metal FPGA implementation may be used, thus reducing the time delay needed for run-time detection.
- The NOP delays could potentially harm the run-time nature of this detection mechanism. Conducting further experiment, we identified that the bottle-neck of this approach is the UART communication between the power-measurement device and the host machine. To improve upon this and fully achieve run-time, the whole scheme can be implemented on a FPGA.

References

[1] Mitre's embedded capture the flag. https://ectf.mitre.org/past-competitions/. Accessed: 2023-08-22.

[2] Ayush Bansal and Debadatta Mishra. A practical analysis of rop attacks. *arXiv preprint arXiv:2111.03537*, 2021.

[3] Sanjeev Das, Bihuan Chen, Mahintham Chandramohan, Yang Liu, and Wei Zhang. Ropsentry: Runtime defense against rop attacks using hardware performance counters. *Computers Security*, 73:374–388, 2018.

[4] Mohamed Elsabagh, Daniel Barbara, Dan Fleck, and Angelos Stavrou. Detecting rop with statistical learning of program characteristics. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, CODASPY '17, page 219–226, New York, NY, USA, 2017. Association for Computing Machinery.

[5] Hasindu Gamaarachchi and Harsha Ganegoda. Power analysis based side channel attack, 2018.

[6] Navyata Gattu, Mohammad Nasim Imtiaz Khan, Asmit De, and Swaroop Ghosh. Power side channel attack analysis and detection. In *Proceedings of the 39th International Conference on Computer-Aided Design*, ICCAD '20, New York, NY, USA, 2020. Association for Computing Machinery.

[7] NewAE Technology Inc. Chipwhisperer-lite, 2024. Accessed: 2024-05-23.

[8] Adebayo Omotosho, Gebrehiwet B. Welearegai, and Christian Hammer. Detecting return-oriented programming on firmware-only embedded devices using hardware performance counters. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, SAC '22, page 510–519, New York, NY, USA, 2022. Association for Computing Machinery.

[9] Ye Wang, Qingbao Li, Zhifeng Chen, Ping Zhang, and Guimin Zhang. A survey of exploitation techniques and defenses for program data attacks. *J. Netw. Comput. Appl.*, 154(C), mar 2020.

[10] Nathanael R. Weidler, Dane Brown, Samuel A. Mitchell, Joel Anderson, Jonathan R. Williams, Austin Costley, Chase Kunz, Christopher Wilkinson, Remy Wehbe, and Ryan Gerdes. Return-oriented programming on a resource constrained device. *Sustainable Computing: Informatics and Systems*, 22:244–256, 2019.