Detecting Return-Oriented Programming on Firmware-Only Embedded Devices Using Hardware Performance Counters

Adebayo Omotosho
University of Passau
Passau, Germany
adebayo.omotosho@uni-passau.de

Gebrehiwet B. Welearegai
University of Passau
Passau, Germany
gebrehiwet.welearegai@uni-passau.de

Christian Hammer
University of Passau
Potsdam, Germany
christian.hammer@uni-passau.de

ABSTRACT

Return-oriented programming (ROP) relies on in-memory code sequences ending in return instructions to chain together arbitrary malware. ROP is one of the most dangerous security exploits because, if wittingly crafted, it can be used to wreak havoc on the system, network, and nodes connected to it. It is not surprising that ROP has been studied on architectures such as x86 and ARM, mostly with an operating system (OS). Xtensa is one of the most popular industry standards for digital signal processors and it is present in many resource-constrained firmware-based embedded WiFi home automation devices, which operate by reading instructions directly from flash memory. Despite leveraging no real OS, Xtensa is not immune to ROP, and there have been reports of buffer overflow vulnerability exploitations leading to ROP in Xtensa.

Therefore, we present the first detection of ROP, and its variant *Jump-oriented programming* (JOP), in a firmware-only environment using *hardware performance counters* (HPCs). Our approach discerns the variations in the HPC micro-architectural events triggered by ROP attacks and benign program execution. We implemented attack scenarios using instrumented programs and exploits that perform tasks similar to those in a known microprocessor benchmark programs. We recorded micro-architectural events to train a machine learning binary classifier. The learned model identifies relevant HPCs, which could serve as predictors of ROP/JOP execution even in embedded firmware-only configurations, where features atypical to conventional processors, like instruction memory and data memory, are available. Our evaluation results indicate a high precision, recall, and accuracy of the classifier predictions.

CCS CONCEPTS

• **Security and privacy** → **Malware and its mitigation**; • **Computing methodologies** → *Supervised learning*; • **Computer systems organization** → *Embedded and cyber-physical systems.*

KEYWORDS

Microprocessor, Xtensa, ROP, JOP

ACM Reference Format:
Adebayo Omotosho, Gebrehiwet B. Welearegai, and Christian Hammer. 2022. Detecting Return-Oriented Programming on Firmware-Only Embedded Devices Using Hardware Performance Counters. In *The 37th ACM/SIGAPP Symposium on Applied Computing (SAC '22), April 25–29, 2022, Virtual Event.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3477314.3507108

SAC '22, April 25–29, 2022, Virtual Event

1 INTRODUCTION

The Internet of Things (IoT) and embedded devices make use of dedicated commercial off-the-shelf microprocessors and many of these devices largely depend on firmware with a C-language codebase and software development kit [2]. As expected, the battle of wits between the engineers creating offensive and those inventing defensive strategies is a never-ending one, and attackers are increasingly and desperately bug hunting these devices using memory corruption techniques such as buffer overflows and heap corruption, which are common in C applications, to find a way to evade device security. Frequently, attackers technically use an approach called *Return-Oriented Programming* (ROP) [10], one of the most dangerous security exploit techniques to take advantage of software weaknesses such as buffer overruns (e.g., in the C language), overwriting the call stack, and gaining control over the program's control flow [32]. Omitting the need to inject malicious binary code, the attackers meticulously select and execute multiple tiny sequences of machine instructions (called *gadgets*) that are already in memory [16, 28]. The attacker will construct a payload based on the addresses of the selected gadgets and corrupt the stack such that the return address of the topmost stack frame points to the first gadget. Since each ROP gadget ends in a return instruction, gadgets can be chained together to build complex exploits by ensuring that the next return address on the stack points to the succeeding gadget. The major challenge to preventing ROP is that the gadget's instructions are located in the executable memory area of the original program and therefore ROP can circumvent mitigation mechanisms such as *data execution prevention* and coarse-grained *address space layout randomization.* A popular variant of ROP is *Jump Oriented Programming* (JOP) that involves chaining gadgets ending in a jump instruction and controlling the control flow via a special gadget called the *dispatcher gadget* [5].

Most of the existing ROP countermeasure techniques focus on ARM and x86 architectures e.g in [13, 14, 20, 24], processors for embedded devices like the Xtensa core have only been investigated rudimentary. Xtensa is a Tensilica processor platform manufactured by Cadence®, with highly customizable and configurable processors that found wide application in HiFi audio and voice digital signal processors. The Tensilica core family includes the Xtensa LX and NX processors, and different versions of the core have been adopted by vendors like Microsoft, AMD, and Espressif [30, 34]. Millions of

IoT devices (including industrial IoT devices, e.g., eModGATE[1] and Moduino X Series[2]) and embedded systems using ESP 32 (LX6) and its predecessor ESP 8266 (LX106)—which are economical and low-power systems on a chip—are based on Xtensa core. Firmware-only devices are characterized by deterministic interrupt-driven tasks due to a lack of a scheduler (no OS). The firmware is typically stored in rewritable, nonvolatile memory (flash), without fine-grain privilege separation and execution isolation available in a conventional OS [11]. Usually, they are supported by manufacturer header files, and the absence of third parties drivers/firmware is believed to add trust and control [19]. However, this restricted the use of custom security, and being resource-constrained(e.g low memory/storage), it is challenging to deploy a sophisticated solution against memory corruption attacks on them.

Motivation: First, ROP attacks on Xtensa are *not well documented*, even though the chips are present in almost every WiFi-based home automation device. However, pieces of evidence of buffer overflow vulnerability exploitation leading to ROP exist. To name a few, on Expressif's ESP8266 (with Xtensa LX106 core), an attack allowed bypassing the network credentials, making the device perform unauthorized operations [29]. A similar memory exploitation attack with the *common vulnerabilities and exposure* number *CVE-2019-12588* was used to crash Xtensa (LX106 and LX6) WiFi devices, causing a denial of service to legitimate users [18].

Second, micro-controller devices are often resource-constrained but there are very few studies on ROP on IoT platforms leveraging no operating system but firmware executing directly from flash memory. Likewise, almost all of the previous work on ROP detection using HPC *considered devices with a full-blown configuration*, where it is possible to inspect the usual suspects in terms of hardware events, like cache-level events or return misses [12, 26] that are generally not available or inapplicable on embedded platforms. Also, a most recent low-level detection study on a micro-controller proposed the use of control flow integrity but did not provide implementation or results to support the feasibility of the approach [24]. In contrast, we investigate an alternative cost-effective approach using HPC events on a low configuration Xtensa processor.

These shortcomings, together with the danger that IoT devices may not receive security updates as frequently and lastingly as personal computers or mobile devices, serve as the basis for this research. This paper combines ML predictive capability using HPC, which generally provide architectural characteristics of the underlying hardware. Although HPCs were originally designed for debugging during development, they have proven useful for performance analysis and validation [33].

The objectives of this study are to 1) briefly explore the Xtensa architecture and define valid gadgets, 2) demonstrate how to implement ROP and JOP attacks on Xtensa, 3) implement these attacks on the Xtensa LX7 architecture running only firmware and to uncover the HPC events indicative of an attack, 4) train a ML model to automatically detect such behavior using HPC events, and 5) evaluate the model's performance on benchmark programs running on Xtensa.

[1]https://iot-industrial-devices.com/category/esp32/
[2]https://moduino.techbase.eu/

Table 1: Registers in Call0 and Windowed Register ABI

Registers	Call0 ABI	Windowed Register ABI
a0	Return address	Return address
a1	0 or (sp) Stack Pointer (callee-saved)	Stack pointer
a2 - a7	Function Arguments	Incoming arguments
a7		Callee's stack-frame pointer (optional)
a12 - a15	Callee-saved	
a15	Stack-Frame Pointer (optional)	

```
1 main:
2 60000bcc:  addi a1, a1, -16
3 60000bcf:  s32i.n a0, a1, 0
4 60000bd1:  l32r a2, 60000904 (600007b8 <
      _clib_rodata_end>)
5 60000bd4:  call0 60000c0c <printf>
6 60000bd7:  movi.n a2, 0
7 60000bd9:  l32i.n a0, a1, 0
8 60000bdb:  addi a1, a1, 16
9 60000bde:  ret.n
```

Listing 1: Call0 ABI assembly

The main contributions of this paper are: First, we show how ROP and JOP attacks could be orchestrated on Xtensa processors. Second, we present HPC events that distinguish ROP and non-ROP program executions on Xtensa. Third, in contrast to existing studies and to the best of our knowledge, this paper presents the first practical work on detecting ROP and JOP in a firmware-only embedded system using HPC.

Threat model: Our focus is on detecting memory corruption attacks through buffer overflows in the firmware or application that result in a ROP/JOP execution on bare-metal Xtensa. We assume that the device has limited hardware functionality but can be protected using a machine-learning classifier based on the HPC events. The attacker is assumed to be able to access the device either physically (through I/O) or through a network connection. Since we are interested in the attack that takes place when executing firmware from flash memory, the attacker uses a payload to execute gadget instructions that are at the static address range. In practice, gadgets in the Xtensa Boot ROM are mapped to the static range, irrespective of the platform being used. Our approach feeds the captured HPC events from a potentially manipulated execution to the machine-learned classifier to predict ROP/JOP behavior that may arise when an attacker exploits a vulnerable firmware function.

2 XTENSA ARCHITECTURE AND REGISTERS

The Xtensa processor architecture is a Harvard architecture with instruction and data memory separate that provide fast simultaneous access to both memories. The Xtensa processor architecture targets embedded system-on-a-chip applications, and the *Instruction Set Architecture* (ISA) specifies a 32-bit RISC-like architecture expressly designed for embedded applications. The Xtensa core ISA is implemented as 24-bit instructions, providing about a 25% reduction in code size compared with 32-bit ISAs [8]. Instructions can be represented as 16 or 24 bits, which results in high code density and also means that any byte is a valid jump target. The instructions provide access to the entire processor hardware and support special functions, such as a single-instruction *compare and branch*, which reduces the number of instructions required to implement

various applications. Xtensa has three distinguishing features and the first is *extensibility*. This addition of architectural enhancements allows easy and efficient extension of the processor architecture with application-specific instructions. The second is *configurability*, which supports creating custom processor configurations that make it easy to specify whether (or how much) pre-designed functionality is required for a particular product. The third is *retargetability*, which allows mapping of the architecture onto hardware to meet the different speed, area, and power targets in different processes. These features make Xtensa unique and in demand for embedded systems design. Xtensa supports 16 address registers a0 to a15, where the functionality of these registers differs slightly depending on the *application binary interface* (ABI) in use. An ABI is a set of rules describing what happens when a function is being invoked, how its parameters are processed, and defining the stack layout for the function call. Xtensa supports two ABIs: *Call0* ABI and the *Windowed Register* ABI, for which Table 1 presents the registers and their functions. The Call0 ABI works with all Xtensa processors, and it has a better context switch time than the Windowed Register ABI. ROP and JOP attack orchestration on both ABIs are similar [21] despite the differences in register usage.

In this paper, our principal target is the Call0 ABI processor configuration, which has been hit by some memory corruption attacks in recent years. We, therefore, demonstrates ROP and JOP on this ABI configuration to provide a foundational understanding of Xtensa architectural behavior. The Xtensa architecture also has a 32-bit program counter, which – similar to x86 and in contrast to ARM – cannot be directly accessed. Generally, Xtensa's instruction format follows the pattern:

`mnemonic` <*dest_reg* >, <*operand_1*>,<*operand_2*>

The destination register *dest_reg* stores the result of the operation specified by the opcode *mnemonic* on the first *operand_1* and second *operand_2* operand. Xtensa's instruction set is flexible in that not all of the instructions set require all of the fields in this template. More specific details about the Xtensa LX hardware instruction set architecture can be found in [9].

2.1 Xtensa ABI Assembly

In this section, we briefly present the Call0 ABI assembly. To this end, we leverage the configurability feature of the Xtensa processor. We created, built, and installed a configuration for CALL0 ABI using Xtensa Xplorer version 8.0.10.3000 running on Windows 10. Xtensa Xplorer's Integrated Development Environment is based on Eclipse and comes with a pre-installed Software Development Kit for processor configurations and programming. We then compiled a simple *helloworld.c* program to illustrate the call0ABI assembly. The corresponding assembly is shown in Listing 1.

It is worth noting that some instructions in the Call0ABI use the *.n* suffix, which is the Xtensa processors' optional code density feature that provides 16-bit versions of some commonly used instructions. Technically, the compiler and the assembler use narrow instructions where possible to achieve better code density [9]. In `line 1`, the stack is decremented by 16 bytes (space allocation), and in `line 2`, the return address is saved to the top of the stack `*(a1+0)`. At the end of the assembly, the reverse is done before `ret.n`, i.e. the return address is restored into `a0` in `line 7`, and the stack is incremented by 16 bytes (space deallocation) in `line 8`. These two steps are similar in almost all Xtensa assemblies and will serve as the basis for chaining gadgets.

```
1 6000a383: 1148   l32i.n a12, a1, 4
2 6000a385: 4149   l32i.n a13, a1, 8
3 6000a387: 4128   l32i.n a14, a1, 12
4 6000a389: 3108   l32i.n a0, a1, 0
5 6000a38b: 30c112 addi   a1, a1, 16
6 6000a38e: f00d   ret.n
```

Listing 2: Return gadget

```
1 60000210: 0338      l32i.n a3, a3, 0
2 60000212: 7149      s32i.n a4, a1, 28
3 60000214: 0003a0    jx  a3
```

Listing 3: Jump gadget

The Xtensa assembly language opcodes used throughout this paper will be limited to a small subset of the entire Xtensa instruction set. Our interest covers mainly the *return* (`ret`), *jump* (`jx`), *call* (`callx`), *load* (`l32i` and `l32r`), *store* (`s32i`), *move* (`mov`), *add* (`add`), and *subtract* instructions (`sub`).

3 XTENSA GADGETS

ROP and JOP gadgets in Xtensa usually end with a `ret` and `jx` instructions respectively. Although it is also possible to use codes ending with an indirect `callx` and `branch` instructions to an address stored in a register. For gadgets discovery, we extended the *xrop* tool[3] to extract and return valid Xtensa gadgets ending with the preferred instruction types. From our programs we extract and design Turing complete gadgets that perform data movement, arithmetic operations, branching, and function calls. The programs are compiled with Xtensa Xplorer, which outputs executable and linkable format binaries supported by the Xtensa LX7 board. Assuming the stack has been overwritten and preloaded with attacker-controlled addresses and values, it is, e.g., possible to use the *return gadget* in Listing 2 to load arbitrary values from the stack at `a1+0`, `a1+8` and `a1+12` into the registers `a12`, `a13`, and `a14` respectively before returning to the designated address.

Xtensa gadgets support very limited direct operations on memory addresses. Therefore, gadgets' addresses must either be loaded into a register from another register or the stack. One of the most common instructions in Xtensa binaries allowing direct memory operands is `l32r`, which can directly load the address of a string literal from memory. Thus, gadgets used for exploits are usually longer and with more side effects on registers, when compared to ARM. For instance, an equivalent of the gadgets in Listing 2 in ARM could be as short as:

`0x00010578 : pop(r3, r4, r5, pc)`

An example of a *jump gadget*, which loads the content of the address stored in `a3+0` into `a3` and jumps to it is given in Listing 3.

Other types of instructions that can be used as gadgets are the *branch gadgets* and the *call gadgets*, an example of the latter, which performs an indirect call operation to a subroutine address in a

[3] https://github.com/jsandin/xrop

```
1 60000e01: 4108      l32i.n a0, a1, 16
2 60000e03: 0e4d      mov.n a4, a14
3 60000e05: 0000c0    callx0 a0
```

Listing 4: Call gadget

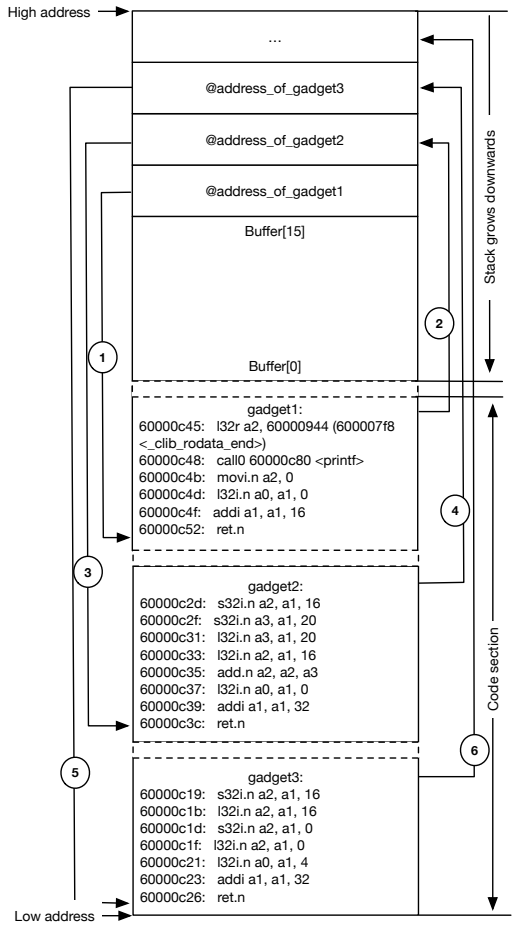


Figure 1: Xtensa ROP attack process.

register is shown in Listing 4. The `l32i` instruction loads an address from the stack at `a1+16` into `a0` and then jumps via a procedure call to that address.

3.1 Xtensa ROP Attack Process

Usually, more than one gadget is required to perform a complex exploit, the general process of chaining ROP gadgets is shown in Fig. 1, which represents a stack that grows downwards (the buffer grows in the opposite direction) from a higher memory address to a lower memory address. The figure has two sections – the code section (memory region that is non-writable but executable), and the stack section (memory region that is writable but not executable). The code section contains the executable gadgets while the stack section stores the addresses of these gadgets (plus potentially values to be read into registers by gadgets). Since we are exploiting a buffer overflow vulnerability to hijack the program's control flow via ROP, the gadgets' addresses must be placed behind the buffer starting at the current stack frame's return address. Let us assume that we have a function that is not performing a bounds check and that the function is using an unsafe C function like *gets* or *strcpy* to initialize a 16 bytes buffer variable. We can exploit this function by feeding it a payload that overflows the buffer and writes to the stack the addresses of *gadget1* (@address_of_gadget1), *gadget2* (@address_of_gadget2), and *gadget3* (@address_of_gadget3) respectively. In this example, these addresses will be 20 bytes, 24 bytes, and 28 bytes, respectively, from the beginning of the buffer. More importantly, the address of the first gadget (@address_of_gadget1) overwrites the return address in `a0` and the stack pointer `a1` becomes the gadgets counter i.e the number of times `a1` is incremented is equivalent to the number of gadgets executed. This whole process is depicted in Fig. 1. Each gadget executes, increments the stack pointer, and returns to the address on the top of the stack until all the gadgets in the ROP payload have been executed. The order of execution of the instructions after the control flow is hijacked is labeled from *1* to *6*. In the end, the execution of these gadgets prints a string, adds the contents of two registers, and initializes a register with a value from another register. If these gadgets were functions, it would be important not to use the starting address at the beginning of a function because this would result in infinitely returning to the same function – we do not want the `ret` to chain a gadget to itself repeatedly but to other meaningful gadgets. Essentially in Xtensa, the instructions occupying the first 5 bytes of every function reserve space for the function on the stack. This characteristic is the foundation of gadget chaining and simulating ROP behavior in Xtensa. Therefore, we always skip these instructions and addresses when crafting a payload for a ROP attack.

Besides, as shown in Fig. 1, the actual gadgets normally reside in non-consecutive locations (as indicated by the dotted lines between them) in the code section of the memory, while the gadget's addresses could be in a consecutive location on the stack. Some garbage addresses may be included as part of the actual gadget addresses on the stack which solely serve the purpose of address padding so that each gadget address is positioned at the top of the stack. Likewise, within the executable gadgets, some instructions exist as side effects, meaning they are not part of the intended exploit but they can also alter register states. Examples of such instructions were in the previous gadgets presented e.g the line 2 of both Listing 3 and Listing 4 were unintended.

3.2 Xtensa JOP Attack Process

JOP uses gadgets ending with an indirect jump to an address in a register as demonstrated in Listing 3. However, the steps involved differ, as jump gadgets cannot be redirected to the stack with a return instruction, so to logically connect gadgets, we adopted the JOP model from [5]. That approach recommends that a *dispatcher gadget* is needed to link all jump gadgets, i.e., a *trampoline* gadget that relays from one jump gadget (also called *functional gadget*) to the next. In that scheme, the dispatcher gadget may maintain a *dispatch table*, which stores the JOP gadget addresses that should be executed sequentially, and each gadget must always point back to the dispatcher after execution. Fig. 2 shows how these iterative steps can be implemented for JOP in Xtensa, and the label *1* to *8* represents the order of execution of instructions once JOP is

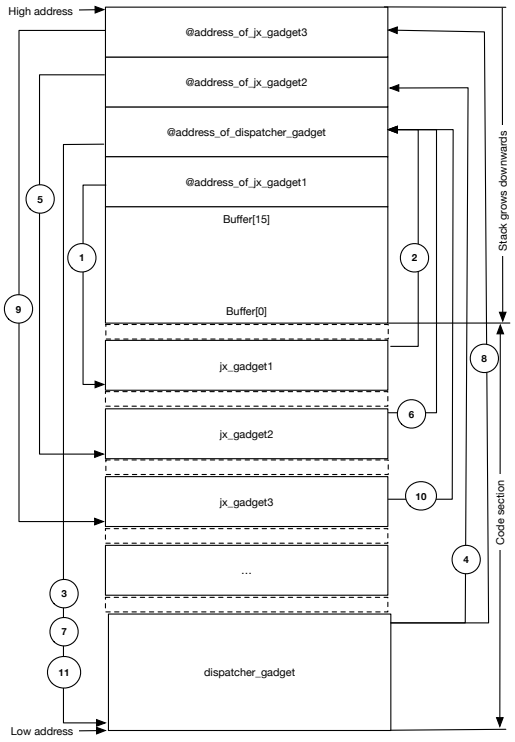


Figure 2: Xtensa JOP attack process.

```
1 60001525:  addi.n a15, a15, 4
2 60001527:  add.n a1, a1, a15
3 60001529:  l32i.n a3, a1, 0
4 6000152b:  sub a1, a1, a15
5 6000152e:  jx a3
```

Listing 5: A dispatcher gadget

```
1 60001555:  l32r a2, 600010ac
2 60001558:  call0 60001688 <printf>
3 6000155b:  jx a14
```

Listing 6: A functional gadget

initiated. Similar to ROP, a vulnerable function not performing bounds checking can be used to launch the sequence, in that the return address of the vulnerable function is overwritten to point to the first jump gadget or the dispatcher, depending on the intent of the attack. An example of a dispatcher gadget in Xtensa is shown in Listing 5, this dispatcher gadget increases the value of `a15` – a regular register leveraged as an instruction pointer – by a constant (4), then points the stack pointer (`a1`) to the next address, loads the new address into `a3`, and jumps to this next functional gadget every time it is executed. This means that the dispatcher gadget can compute the addresses and jump to each of the functional gadgets `jx_gadget1`, `jx_gadget2`, ..., respectively.

An example of a gadget that could serve as a functional gadget is shown in Listing 6. The functional gadget loads the address of a string literal into `a2`, prints the string, and jumps back to the dispatcher gadget at `a14`. Unlike ROP, the addresses of JOP gadgets (other than the first) do not have to be placed on the stack, as they can be computed by the dispatcher gadget, in which case there is no need to retrieve the addresses of the executed instructions from the stack.

4 HARDWARE PERFORMANCE COUNTERS

Many modern microprocessors are equipped with special-purpose registers known as hardware performance counters (HPCs). These registers serve as additional logic added to the CPU to track low-level events within the processor accurately and with minimal overhead when compared to software profilers. Their original purpose was for debugging purposes, but they can be leveraged to serve other purposes, such as detecting program modification at a low cost [22, 33]. They have also been used extensively in non-embedded processors for malware detection with high detection accuracy [1, 23, 35]

HPCs are not standardized and they are therefore manufacturer dependent; on a different microprocessor, even different models of the same processor family, HPCs may have different names, numbers, and functionality. Even though modern processors support a large number of events, only a fraction of these events can be monitored at any time. The number of events that can be monitored simultaneously is determined by the number of available HPCs which is low compared to the overall number of possible events. For example, ARM Cortex-A5 has just 2 HPCs and ARM Cortex-A8 only 4, meaning they can simultaneously monitor only 2 and 4 events, respectively. Additionally, some kind of kernel, operating system or API level support will be required to setup and access these counters. There are several libraries for manipulating HPCs on Linux based embedded/IoT devices e.g `perf` [4], `PerfSuite` [5], and `PAPI` [6]. On Xtensa the performance monitoring library allows a total of 8 events to be monitored simultaneously and they can be accessed using the `xt_perfmon` API. Xtensa LX7 supports 30 main events with a total of 125 masks or sub-events, which represent the microarchitectural state triggered by the running program. A counter will, however, increment only once (by one) if more than one condition corresponding to a set mask bit occurs. The available HPC events on Xtensa are shown in Table 2, and it should be noted that the labels and arrangement of the HPCs are arbitrary. Also, the two prominent sources of noise generally associated with HPC readings are related to the *program design* – noise caused by other internal or external instructions and programs – and the *HPC access* – noise caused by the reading the HPCs. We aim at reducing these noises as described in Section 5.0.2.

5 EVALUATION

In this section, we discuss research questions of interest, the evaluation methodology, the experimental set-up, and the results of the evaluation of the ML classifier.

5.0.1 ***Research Questions:*** We present the research question as follows:

RQ1: What are the top HPCs in terms of indicating ROP and JOP behavior on Xtensa? We do not intend to use all HPC events

[4] https://perf.wiki.kernel.org/index.php/Tutorial
[5] http://perfsuite.sourceforge.net/
[6] https://icl.utk.edu/papi/

Table 2: List of available HPCs on Xtensa

Label	HPC: XTPERF_CNT_...	Interpretation
F1	COMMITTED_INSN	Instructions committed
F2	BRANCH_PENALTY	Branch penalty cycles
F3	MULTIPLE_LS	Multiple Load or Store
F4	INSN_LENGTH	Instruction length counters
F5	CYCLES	Count cycles
F6	PREFETCH	Prefetch events
F7	INSN	Successfully completed instructions
F8	PIPELINE_INTERLOCKS	Pipeline interlocks cycles
F9	D_ACCESS_U1	Data memory accesses (load, store, S32C1I, etc; load-store unit 1)
F10	D_ACCESS_U2	Data memory accesses (load, store, S32C1I, etc; load-store unit 2)
F11	D_ACCESS_U3	Data memory accesses (load, store, S32C1I, etc; load-store unit 3)
F12	D_STORE_U1	Data memory store instruction (load-store unit 1)
F13	D_STORE_U2	Data memory store instruction (load-store unit 2)
F14	D_STORE_U3	Data memory store instruction (load-store unit 3)
F15	D_LOAD_U1	Data memory load instruction (load-store unit 1)
F16	D_LOAD_U2	Data memory load instruction (load-store unit 2)
F17	ICACHE_MISSES	ICache misses penalty in cycles
F18	DCACHE_MISSES	DCache misses penalty in cycles
F19	OUTBOUND_PIF	Outbound PIF transactions
F20	OVERFLOW	Overflow of counter n-1 (assuming this is counter n)
F21	D_STALL	Data-related GlobalStall cycles
F22	I_STALL	Instruction-related and other GlobalStall cycles
F23	BUBBLES	Hold and other bubble cycles
F24	I_TLB	Instruction TLB Accesses (per instruction retiring)
F25	EXR	Exceptions and pipeline replays
F26	IDMA	iDMA counters
F27	D_TLB	Data TLB accesses
F28	I_MEM	Instruction memory accesses (per instruction retiring)
F29	INBOUND_PIF	Inbound PIF transactions
F30	D_LOAD_U3	Data memory load instruction (load-store unit 3)

even if we could, but only a minimal number that results in a good prediction. Using fewer HPC events means requiring fewer resources for detection.

RQ2: At what precision and recall accuracy can a classifier predict unusual behavior caused by ROP/JOP within a running firmware code using the HPC events? In reality, the HPC data for positive attacks scenarios is just a small fraction of the actual application data. In addition, we prefer detecting the malicious behavior as soon as possible (before the end of the program's execution), therefore, a classifier that can predict malicious behavior with a high true positive rate from the HPC events is desired.

5.0.2 ***Evaluation Methodology:*** Answering the first research question requires recording the HPCs at the right place and correctly, though this step is important only for model learning and would not matter during testing on the evaluation set. Usually, not all of the instructions in a program under attack are malicious but only a fraction, other instructions that are not affected by the attack are regarded as noise. To address this question, we performed experiments using instrumented code in which the approximate position of the start of the attack marks the beginning of the ROP/JOP, and the code preceding and succeeding the attack are marked as benign. The benign section exhibits the program's actual behavior and a vulnerable function might be normal or malicious depending on whether or not it is exploited. Targetting the HPC readings close to where the ROP execution is initiated allows us to have less *program design* noise, and fine-grained and accurate HPC measurement. Each interrupt-driven program would usually run multiple times as both normal and malicious, and the ML classification method we use assigns binary labels to the HPCs associated with the different executions. Also, we discovered that constant noise values were automatically added to our readings by accessing the HPC, these noise values were accounted for in both the benign and ROP execution. Our instrumented programs contain different numbers of *push-pop* for ROP exploits and *jump* for JOP exploits, precisely 6.

To answer the second research question, the Xtensa ROP-JOP classifier is evaluated using the following standard ML classification metrics – precision and recall. The metrics are defined as follows, where TP, FP, TN, FN refer to true positive, false positive, true negative, and false negative respectively.

Precision: This measures the accuracy of the positive predictions and for this, we want to know how many of the classified characteristics, recorded as HPC belong to the positive class. This metric is computed as:

$$Precision = TP \;/\; (TP + FP) \tag{1}$$

Recall: This is also known as true positive rate, it is the ratio of positive instances that are correctly predicted by the classifier. In our case, we want to detect as many as possible samples in the positive classes with a reasonable precision score. The recall is computed as:

$$Recall = TP \;/\; (TP + FN) \tag{2}$$

5.1 Experimental Setup

The experimental setup consists of an Xtensa LX7 processor configuration designed with Xtensa Xplorer version 8.0.10.3000. This configuration runs on a Xilinx Zynq XCZ7020-1CLG484C[7] System on Chip (SoC) Module attached to a TE0703-06[8] carrier board. A Tincan Flyswatter2[9] debugger, which provides an external Joint Test Action Group (JTAG) standard interface is connected to the carrier board for direct debugging of the programs. An SoC module is necessary because we experimented with the latest Xtensa processor generation, which was not available on the market at the time of carrying out this research. The embedded hardware configuration used is also minimal as not all of the features available in high-end IoT systems were available. The processor configuration summary is online at GitHub[10].

5.1.1 ***Programs and input:*** We train our model on breadth-first search (*BFS*) algorithm-based programs, in which we instrumented six versions of attack codes using varying ROP chains. This decision was based on observations from our several experiments that merging and training with data from different programs introduces unwanted statistical bias, which negatively affects model convergence. Therefore, in these six programs *P1, ..., P6*, with a ROP chain length of 1, ..., 6 are exploited, respectively. The programs were written in the C language, compiled using the Xtensa C compiler `xt-xcc`, and they run directly on the Xtensa processor. `xt-xcc` uses the GNU preprocessor, assembler, and linker but in addition, it provides a superior and smaller compiled code [7]. Malicious modification is made to the programs by the addition of an extra function call that simulates attacks such as buffer overflow and return-to-libc [28]. To flout the LIFO mechanism of the stack so that the program's control flow is hijacked, we corrupted the stack and modeled the payload to not just include gadget addresses to divert the control flow but also potential function arguments. Our design is mainly

[7] https://wiki.trenz-electronic.de/display/PD/TE0720+Resources
[8] https://wiki.trenz-electronic.de/display/PD/TE0703+Resources
[9] https://www.tincantools.com/product/flyswatter2
[10] https://github.com/dbayoxy/xtensa_config/blob/master/config.html

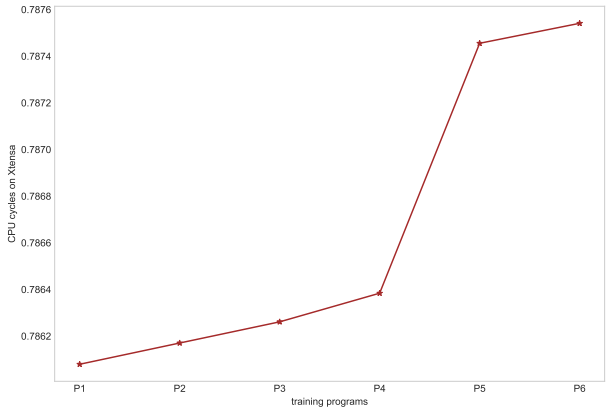


Figure 3: Training programs running time on Xtensa

targeted at buffer overflow attacks to exploit memory corruption vulnerabilities because it is the most commonly used method, even in IoT devices [3, 17, 27]. The aftermath of the attacks then launches a ROP/JOP sequence that results in the execution of codes and functions in the programs that were originally never invoked. In Fig. 3, the execution time overhead in the training programs appears to increase when there are more frequent indirect calls to functions consisting of a few instructions. This instrumentation impact is similar to what was obtained for the ARM embedded benchmark used in [24].

To benchmark our model, we use a blend of 10 programs listed in Table 3, each running with a different set of inputs and payloads. Similar programs can be found in the CTuning suite[11] and MiBenchmark[12].The total size of the 10 programs is ≈ 2.74 MB. In Table 3, *ET1*, *ET2*, *Ovd*, *#Rop*, and *O notation* represents the original execution time, execution time with HPC measurement, instrumentation overhead, the length of the rop/jop chain, and the time complexity of the programs, respectively. The maximum and minimum overhead recorded of 1.34% and 0.74%, respectively, look reasonable. We do not directly compare these programs' performance because they differ in input and runtime complexities, our main goal is to see how the ML model will perform against random programs like these.

5.1.2 ***Data and model selection:*** We recorded the events that were triggered during multiple executions of the exploited programs for ROP and JOP attacks, respectively. The HPC data was recorded at 5 frequencies (≈ 10 * 2^{12} to 50 * 2^{12}), which allows us to sample data from a wide corpus of cycles, and thus provides a sufficient dataset that best represents the behavior of the programs under attack. On Xtensa, the frequency parameter is expected to be a multiple of 2^{12} to prevent round-off errors. We started at ≈ 10 * 2^{12} because this frequency has been used to validate the integrity of program control flow via HPC with some promising results [22]. For any given program, the higher the frequency chosen, the lower the noise effect, as well as the number of HPC sample size recorded. Our data shape is (6061, 30), containing 6061 rows of event counts and 30 features. The support vector ML (SVM) algorithm is preferred because it

[11] http://ctuning.org/
[12] https://vhosts.eecs.umich.edu/mibench/

Table 3: Benchmark programs

Program	ET1 (μs)	ET22(μs)	Ovd(%)	#Rop	O notation
DFS	4421664	4481007	1.34	2	$O(V+E)$
Kruskal	2273937	2297769	1.05	3	$O(E \log E)$
RabinKarp	642475	647202	0.74	4	$O(mn)$
Huffman	2343248	2368711	1.09	5	$O(n \log n)$
Mergesort	933301	941449	0.87	6	$O(n \log n)$
LCS	677565	688501	1.61	1	$O(mn)$
Prim	1577938	1596889	1.20	2	$O(E \log V)$
BinaryS	507317	511553	0.83	4	$O(\log n)$
FloydWarshall	1748544	1765759	0.93	5	$O(n^3)$
BellmanFord	1448642	1460045	0.79	6	$O(VE)$

excels for data in high dimension spaces and it is relatively memory efficient. SVM is an excellent binary classifier if data is balanced but because the positive cases are less than the negative cases, with about a factor of 7, we use a weighted SVM. The weighted SVM modifes the SVM penalty parameter C_i to fit the model for each instance i, so that the weight w_i is proportional to the class distribution. We use 10-fold cross-validation, i.e., 10% of the data is used for testing, which is a standard procedure to ensure the validity of the learned classifier. The model performance measured by the mean of the ROCAUC score is 0.94, which is well above 0.5, this means the classifier has a predictive ability.

We conducted the ML experiments on a MacBook Pro with a 2.9 GHz Intel Core i7 processor and 16GB RAM.

5.2 Discussion

In this section, we discuss our findings and their contribution to the research questions.

Important HPC events: Of the 30 main HPC events in Table 2, feature engineering found that the readings for events F1, F3, F4, and F7 are the same values irrespective of the number of times a program executes either as benign or attacked code. An explanation for the similarities in the HPC values could be that the low-level events (masks) in these main events being accounted for occurred at almost the same count rate. Therefore, F1, F3, F4, and F7 serve as the pivot for the permutation with the remaining events, together with all the sub-events, to determine which events are dependent on them. Notwithstanding, the HPC values recorded are reproducible for any given program, the reasons for the deterministic nature of some of the counted and recorded events could be the result of (a) running the programs on a bare board with no operating system or kernel and (b) there are no running background services, which reduces the effect of interference in the readings.

The distribution of the 8 candidate HPC events (used in our final SVM model) in the benign and attack code is represented in Fig. 4 by violin plots, which are combinations of box-and-whisker plots and probability density functions (PDF). The violin plot shows the density and distribution of the readings for each of the 8 selected HPC events. -1 and 1 represent HPC events in the benign and attack executions respectively. Fig. 4a, Fig. 4b, Fig. 4c, Fig. 4d, Fig. 4e, Fig. 4f, Fig. 4g and Fig. 4h are the distributions for F1, F2, F5, F8, F12, F15, F25 and F27 respectively. While at a glance some features such as F2, F12, F15, and F27 might look close for both the benign and ROP runs, dropping them degrades the model's performance (the overall recall falls from 70% to 58%.) However, it may of course be possible, in the future, to use fewer than 8 HPC events, but our interest at the

moment is to identify the best events that Xtensa's 8 performance counter registers could monitor simultaneously, and which could give a strong indication of ROP/JOP execution on an embedded system running a firmware. In the sequel we give an overview of the selected HPC events:

F1 in Xtensa is equivalent to the number of retired instructions and it is the number of instructions reaching the *W* stage without being killed at a given sampling interval. At the *W* stage, the effect of an instruction on the architectural state is irreversible. The wider PDF region of the ROP infected run, which is at the same time above the third quartile of the benign run, is abnormal. The median of the ROP run also lies above the boxplot of the benign run, meaning that the two HPC data belong to a different group.The ROP run HPC values occur frequently in one PDF region and this is as a result of the execution of small gadgets performing little tasks and leading to slightly more/faster-committed instructions per interval.

F2 relates to the number of branch penalty instruction events in a given sampling interval. The pipeline will be stalled if more branch instructions are executed than they are taken. All of the branch instructions in the benign programs were correctly executed while the malicious program executes only a few selected branch instructions to accomplish its malicious intention. The boxplots look like they are slightly in the same range and symmetrical but the PDF region of the ROP-affected run shows the HPC values occur more frequently and this can be attributed to the more mispredicted branches.

F5 distribution appears to cover two and three PDF regions in the benign and ROP-affected run respectively. For the ROP-affected run, the median is almost identical to the third quartile which is why they overlap, this is likely because of a large proportion of low values of *F5* events. The ROP-run skipped some instructions and this could be responsible for the very low variance in this HPC.

F8 relates to the number of stalls in the pipeline in a sampling interval. In Xtensa, the number of interlocks refers to the number of R-stage holds arising from register dependencies and interlock-specific instructions. Register dependencies are low because gadgets skip several normal instructions. The median of the ROP-affected run is also unsurprisingly outside the box of the benign run and the boxes' variance shows that this pipeline delay varies more in the benign run than in the ROP-affected run.

F12 records the number of stored instructions events (such as store misses and cached store) from the data memory in a sampling interval. The ROP run has significantly more variance and inverted PDF and this is likely caused by a reference to data not in the data memory.

F15 records the number of the load instruction events (such as load misses and cache load) from the data memory in a sampling interval. The data memory, unlike the instruction memory, is both readable and writeable. The normal run distribution have slightly more variance than the ROP-affected run, however, the PDF is an indication that the ROP-run performs data manipulation operations and load data operation more frequently.

F25 can record, for example, the number of the exceptions, interrupts, and replays resulting from TBL misses, load and store errors, illegal instructions, etc. It is not surprising that despite the median being the same for the two boxes, the ROP-affected run is severely skewed and the unusual PDF width shows that the majority of the

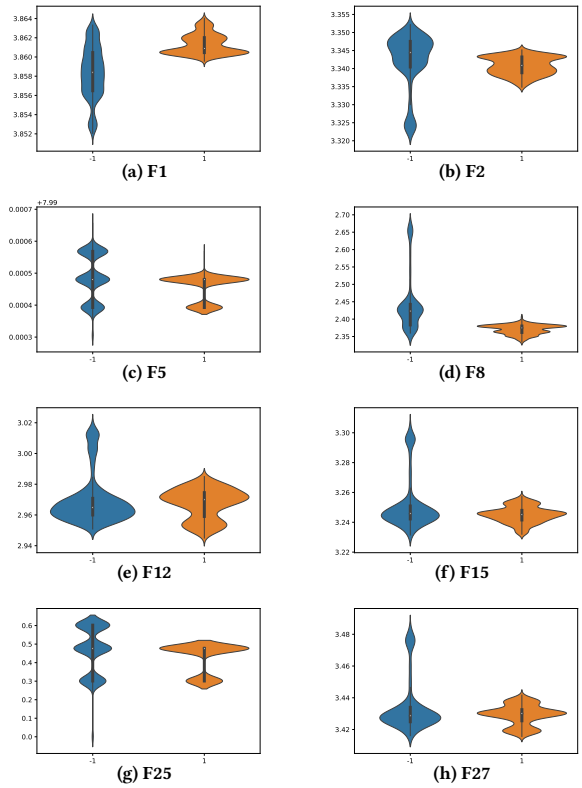


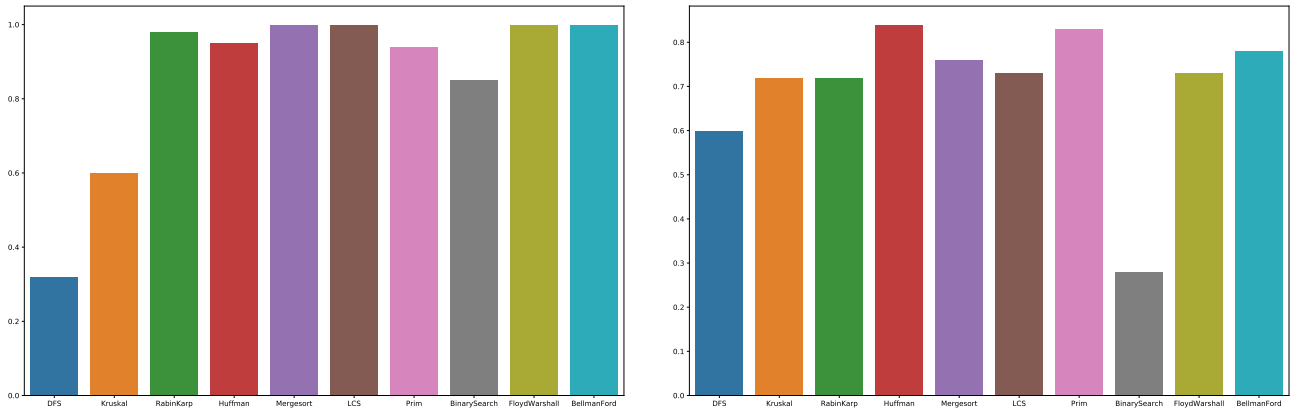
Figure 4: Box-and-whisker plots of the event counts for the benign (-1) and ROP executions (1)

F25 HPC readings are in one region. This implies that the error handling by this HPC occurs more frequently in the ROP-affected run.

F27 records the number of lookups to the data translation lookaside buffer (TLB). Unlike the von-Neumann architecture, the Harvard architecture can have separate memory access hardware - instruction TLB and data TLB. Data TLB hit helps to reduce the data access time from the data memory. This HPC for the two runs appears to be in the same group but the PDF is irregular with a high-frequency region for the ROP run, this is because each data TLB miss leads to a computationally expensive page table lookup for the physical addresses of data.

Model performance metrics: Our classifier is trained to predict if illegal instructions using *returns* and *jumps* have been exploited and this is put to test against some benchmark programs. The model is fed with HPC data of these programs running on the bare-metal Xtensa (on the FPGA).

Fig. 5a and Fig. 5b show the individual precision and recall of the classifier. In Fig. 5a the precision peaked for *floydwarshall, and bellmanford* where a higher number of illegal instructions was executed. While it is intuitive that the metrics improve with the increase in *returns/jumps*, we found this not to be always true if the exploited programs are different. That is why *BinaryS* with 4 rop/jop violations has lower precision (0.85) than *Prim* (1.00) and LCS (0.94) with 1 and 2 violations, respectively. We verified this and the metrics actually increase with more *returns/jumps* exploitation



(a) Precision **(b) Recall**

Figure 5: Evaluation on benchmark programs

in the same program. *DFS* has the lowest precision of 0.32, however, this is not a problem as long as the recall is high, this is based on the premise that in reality, the positive case is rare, and hence the percentage of the positive instances that are correctly detected should be optimized. The recall for DFS is almost twice as high, 0.60. The top three recall values are for *huffman* (0.84), *prim* (0.83) and *bellmanford* (0.78), these are very interesting because it means that almost all of the ROP instances are correctly detected. However, *BinaryS* has a low recall of about 0.30. We examined this and it appears the *BinaryS*, which is already fast with $O(n \log n)$ running time, also ran on a small input array of length 12. From Table 3 it appears that, even with the instrumentation overhead, it still runs fast. It appears that the complexity of the program did affect the capability of the classifier. However the overall detection average accuracy of 0.79 is significant.

Threats to Validity. We identify the following potential threats to validity:

- Examples of ROP/JOP exploit code for non-standard processors are not easy to find in practice, and therefore the instrumented programs used for our training may not be representative of all ROP programs' behaviors on Xtensa. Notwithstanding, the size of our data together with the results obtained could reasonably justify the reliability of our model in detecting attacks initiated through a buffer overflow, which is the most exploited vulnerability.
- Our ROP/JOP exploits do not consider specific attack cases such as accessing a shell, which might be the final intention of the attacker even though it may not be available in the context of embedded/IoT devices, which often only contain firmware.

6 RELATED WORK

This section presents some selected literature related to the problem addressed in our work.

ROP attack defense using HPC: Several works have justified the use of HPCs in detecting abnormal program behavior. HPCs can be used to detect ROP attacks by applying heuristics of ROP behavior or via a ML-based approach. HDROP [36], SIGDROP [31], and ROPSentry [12] are some of the papers that utilize HPCs to build a defense against ROP exploits using heuristics. In contrast, hadROP [26] and EigenROP [15] use ML-based approaches to learn relevant hardware events distinguishing ordinary execution from ROP gadget chain execution. Our approach also leverages HPCs and ML to detect the ROP attack. Unlike our proposal, most of the existing solutions were not targeted at low-level embedded devices running only firmware and have overheads that are not resource-constrained devices friendly, e.g., HDROP recorded an overhead as high as 38% and SIGDROP requires an OS.

Code reuse attacks: Several techniques use existing code on the system to craft exploit that attacks the system. These includes ROP [10], JOP [5], string ROP [25], blind ROP [4] and signal ROP [6]. However, most of these attacks have been tested on x86 or ARM processors rather than configurable processors dedicated to embedded systems/IoT. In contrast, our paper implements ROP and JOP attacks using compiled gadgets on the Xtensa LX7 architecture. Moreover, we are the first to design JOP attacks using a dispatcher gadget compatible with the Xtensa processor.

ROP on Xtensa: Little is public about how ROP works on Xtensa, but [21] presented a paper on the challenges of ROP on Xtensa architecture. The authors showed that Xtensa can be attacked by chained gadgets irrespective of the ABI in use. The authors additionally proposed a linked list approach to chain gadgets for Xtensa's Call0 ABI. Since the approaches to attacking either ABI are similar, we used the default ABI in our paper to demonstrate ROP/JOP on Xtensa. Their paper is also different in that it did not cover attack detection. Although based on a different method, a similar platform to ours is used in [24] who propose a solution called FPGA CFI for bare-metal ARM embedded devices. Their approach targets devices that read firmware instructions directly from the flash memory but unlike our work, ROP detection was not included in their work. Additionally, CFI solutions' memory requirements and overheads are generally considered impractical to secure resource-constrained embedded devices [31].

.

7 CONCLUSION

In this paper, we have been able to demonstrate the possibilities of how the Xtensa Call0 ABI processor configuration could be attacked using gadgets from an executable linkable format binary of user programs. We extracted valid gadgets, demonstrated gadget chaining scenarios for ROP/JOP, and carried out experiments with these attack scenarios on a minimal Xtensa hardware configuration running as a bare-metal embedded system. Our hardware configuration is minimal and targeted for low configuration embedded/IoT devices running instructions from the flash memory. Furthermore, we experimented with the available hardware performance counter events and trained a support vector machine classifier based on these HPCs to detect ROP and JOP readings in our test programs. By evaluating the model on unseen HPC data, we obtained a high precision and recall. We also identified some HPC that help in predicting the execution of these kinds of code reuse attacks. Our validation results prove the feasibility of the SVM and HPC detection methods for ROP/JOP on Xtensa from a functional perspective, thereby validating the capacity of this technique to detect code reuse behavior on a firmware-only Xtensa processor.

ACKNOWLEDGMENTS

This work was supported by the German Federal Ministry of Education and Research (BMBF) under research grant number 01IS18065D.

REFERENCES

[1] Manaar Alam, Sayan Sinha, Sarani Bhattacharya, Swastika Dutta, Debdeep Mukhopadhyay, and Anupam Chattopadhyay. 2020. RAPPER: Ransomware prevention via performance counters. *arXiv preprint arXiv:2004.01712* (2020).

[2] Mahmoud Ammar, Giovanni Russello, and Bruno Crispo. 2018. Internet of Things: A survey on the security of IoT frameworks. *Journal of Information Security and Applications* 38 (2018), 8–27.

[3] Atri Bhattacharyya, Andrés Sánchez, Esmaeil M Koruyeh, Nael Abu-Ghazaleh, Chengyu Song, and Mathias Payer. 2020. SpecROP: Speculative Exploitation of {ROP} Chains. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2020)*. 1–16.

[4] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. 2014. Hacking blind. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 227–242.

[5] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. 2011. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM, 30–40.

[6] Erik Bosman and Herbert Bos. 2014. Framing signals-a return to portable shellcode. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 243–258.

[7] Cadence. 2018. *Xtensa® C and C++ Compiler User's Guide*. https://ip.cadence.com/swdev

[8] Cadence. 2018. *Xtensa® Microprocessor Programmer's Guide*. https://ip.cadence.com/swdev

[9] Cadence. 2019. *Xtensa® Instruction Set Architecture*. https://ip.cadence.com/swdev

[10] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. 2010. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*. 559–572.

[11] Ang Cui, Michael Costello, and Salvatore J Stolfo. 2013. When Firmware Modifications Attack: A Case Study of Embedded Exploitation. In *20th Annual Network & Distributed System Security Symposium*. 1–13.

[12] Sanjeev Das, Bihuan Chen, Mahintham Chandramohan, Yang Liu, and Wei Zhang. 2018. ROPSentry: Runtime defense against ROP attacks using hardware performance counters. *Computers & Security* 73 (2018), 374–388.

[13] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. 2011. ROPdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. 40–51.

[14] Christian DeLozier, Kavya Lakshminarayanan, Gilles Pokam, and Joseph Devietti. 2020. Hurdle: Securing Jump Instructions Against Code Reuse Attacks. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 653–666.

[15] Mohamed Elsabagh, Daniel Barbara, Dan Fleck, and Angelos Stavrou. 2017. Detecting rop with statistical learning of program characteristics. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. 219–226.

[16] Mohamed Elsabagh, Dan Fleck, and Angelos Stavrou. 2017. Strict virtual call integrity checking for C++ binaries. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 140–154.

[17] K Virgil English, Islam Obaidat, and Meera Sridhar. 2019. Exploiting memory corruption vulnerabilities in connman for IoT devices. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 247–255.

[18] Matheus Eduardo Garbelini. 2019. *Easily crashing ESP8266 Wi-Fi devices*. https://matheus-garbelini.github.io/home/post/esp8266-beacon-frame-crash/

[19] Jin-bing Hou, Tong Li, and Cheng Chang. 2017. Research for vulnerability detection of embedded system firmware. *Procedia Computer Science* 107 (2017), 814–818.

[20] Xin Huang, Fei Yan, Liqiang Zhang, and Kai Wang. 2020. HoneyGadget: A Deception Based Approach for Detecting Code Reuse Attacks. *Information Systems Frontiers* (2020), 1–15.

[21] Kai Lehniger, Marcin Aftowicz, Peter Langendörfer, and Zoya Dyka. 2020. Challenges of Return-Oriented-Programming on the Xtensa Hardware Architecture. In *EUROMICRO Conference on Digital System Design*. 1–6.

[22] Corey Malone, Mohamed Zahran, and Ramesh Karri. 2011. Are hardware performance counters a cost effective way for integrity checking of programs. In *Proceedings of the sixth ACM workshop on Scalable trusted computing*. 71–76.

[23] Ganapathy Mani, Vikram Pasumarti, Bharat Bhargava, Faisal Tariq Vora, James MacDonald, Justin King, and Jason Kobes. 2020. DeCrypto Pro: Deep Learning Based Cryptomining Malware Detection Using Performance Counters. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, 109–118.

[24] Nicolò Maunero, Paolo Prinetto, Gianluca Roascio, and Antonio Varriale. 2020. A FPGA-based Control-Flow Integrity Solution for Securing Bare-Metal Embedded Systems. In *2020 15th Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*. IEEE, 1–10.

[25] Mathias Payer and Thomas R Gross. 2013. String oriented programming: when ASLR is not enough. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*. 1–9.

[26] David Pfaff, Sebastian Hack, and Christian Hammer. 2015. Learning how to prevent return-oriented programming efficiently. In *International Symposium on Engineering Secure Software and Systems*. Springer, 68–85.

[27] Marco Prandini and Marco Ramilli. 2012. Return-oriented programming. *IEEE Security & Privacy* 10, 6 (2012), 84–87.

[28] Hovav Shacham. 2007. 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*. 552–561.

[29] Thotcon. 2016. *The complete esp8266 psionics handbook*. https://speakerdeck.com/jsandin/the-complete-esp8266-psionics-handbook

[30] Gordon Mah Ung. 2013. *Everything You Wanted to Know About AMD's New TrueAudio Technology*. https://web.archive.org/web/20140711104556/http://www.maximumpc.com/everything_you_wanted_know_about_amd%E2%80%99s_new_trueaudio_technology_2013

[31] Xueyang Wang and Jerry Backer. 2016. SIGDROP: Signature-based ROP detection using hardware performance counters. *arXiv preprint arXiv:1609.02667* (2016).

[32] Ye Wang, Qingbao Li, Zhifeng Chen, Ping Zhang, and Guimin Zhang. 2020. A Survey of Exploitation Techniques and Defenses for Program Data Attacks. *Journal of Network and Computer Applications* 154 (2020), 102534.

[33] Vincent M Weaver and Sally A McKee. 2008. Can hardware performance counters be trusted?. In *2008 IEEE International Symposium on Workload Characterization*. IEEE, 141–150.

[34] Chris Williams. 2016. *Microsoft's HoloLens secret sauce: A 28nm customized 24-core DSP engine built by TSMC*. https://www.theregister.com/2016/08/22/microsoft_hololens_hpu/

[35] Boyou Zhou, Anmol Gupta, Rasoul Jahanshahi, Manuel Egele, and Ajay Joshi. 2018. Hardware performance counters can detect malware: Myth or fact?. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*. 457–468.

[36] HongWei Zhou, Xin Wu, WenChang Shi, JinHui Yuan, and Bin Liang. 2014. HDROP: Detecting ROP attacks using performance monitoring counters. In *International Conference on Information Security Practice and Experience*. Springer, 172–186.