Exploring Jump Oriented Programming Exploits on XTensa Architecture's Windowed ABI

Aidan Erickson

October 11, 2026

Abstract

This paper details the application of Jump Oriented Programming as an alternative to Return Oriented Programming on XTensa architecture's Windowed ABI. XTensa is used in a variety of IoT applications, and the space affected space of the suite, and especially the ESP32, is quite large. This paper outlines the effectiveness of these attacks on program control flow, and outlines vulnerability areas still present in Xtensa windowed ABI chips.

1 Security Matters Across Scales

With all kinds of research, a justification is in order for the purpose and operational space of the paper. If these devices are small and low bandwidth, who cares if they are insecure. With the number of IoT devices projected to surpass 75 billion active devices by 2025[1], there is no denying that these devices are everywhere and will continue to be so for the perceivable future.

From here there a definition switch is in order. Low bandwidth $\neq$ unimportant. In early 2020, a couple in Illinois awoke to find that an unusual male voice was speaking to their child via the baby monitor. Their baby monitor, which was of course connected to the internet, had been hacked along with their thermostat and smart cameras.[2] Later that year, a family in Wisconsin had a similar incident where a hacker attacked their smart home device cranking up the heat and playing vulgar music through a camera. [2]

While these cases can be summed up as low damage black hat harassment, they are indicative of a larger problem. Crimes like stalking have been committed on many occasions. Hackers within China have stolen footage from tens of thousands of security cameras and are selling them online as 'home video packages'.[3] A problem here is the sheer space of IoT. Security is expensive, and when your goal is to create low cost computer systems, secure procedures are often left to the wayside to save on computation.

A facet of the large IoT space is of course the diversity of chips. Once such example of a small scale chip is the XTensa set of microcontrollers. XTensa is an architecture with the goal of addressing the high costs and low customization of traditional architectures. Developed by Tensilica Inc., XTensa is used in a variety of IoT (Internet of Things) applications often due to it's configurability giving way to lower power, and cheaper, chips. [4]

One such problem for IoT microcontrollers is Return Oriented Programming (ROP)– an exploitative technique meant to leverage the running code already within the system by diverting control flow.

2 What is Return Oriented Programming and How Does it Work

Return Oriented Programming is a code reuse attack where the fundamental question as an attacker is not "how do I introduce and execute my code" but more "how do I divert control flow in a predictable way".[5]

The fundamental aspect of return oriented programming is abusing the return instruction. This instruction pops the return address off of the stack and jumps to its location. Using a classic buffer overflow vulnerability, an attacker could compromise the return address and jump to a location of the attackers choice.

A buffer overflow is when an attacker inserts data into the system in a way that is outside of the allocated space for that data. If there are no checks that the inserted data are inside that region, other areas could be written to. This includes the return address. The address telling the computer where to jump to in the source code upon the completion of the current function. For this reason it is often the center of system level cyber-attacks.

In a classic buffer overflow exploit, the attacker often uploads their own routines into the data section of memory then jumps into there. With the advent of more sophisticated memory protections, specifically the doctrine of any given part of memory being writable or executable but not both by a program, this approach is not viable.[6, 5, 7]

A few security experts thought that this measure was enough to prevent malicious routines from being executed. However, "Preventing the introduction of malicious code is not enough to prevent the execution of malicious computations"[7] and this was not true. This is where it gets interesting from a Return Oriented Programming perspective.

Return Oriented Programming circumvents the problem of not being able to jump into an attacker's routines by instead using the exploited program's already loaded in routines. What is necessitated are short snippets of code followed by a return instruction, often call gadgets, that define a small bit of functionality that the attacker might want to utilize. By chaining together these gadgets, an attacker would be able to simulate larger functionality with these small gadgets. This class of attack is called a code-reuse attack.

The power of course comes in the sequence in which these gadgets are executed. This is defined entirely by the stack, so a degree of control over the stack pointer is necessary. By pushing stack frames returning to attacker dictated gadgets, arbitrary control over the system is possible.[6, 8, 5, 9]

With large code bases, the capability of ROP increases. ROP has been shown to be turning complete in a variety of environments including libc.[6, 10]

To begin a Return Oriented Programming attack, a great deal of stack control is necessary. The idea is to push malicious return addresses and manipulate the stack pointer. With enough return addresses, a level of arbitrary behavior is achievable.

This is an extension of return-into-libc (often abbreviated as ret2libc), an attack that manipulates the stack to input return addresses and parameters to return into a libc function, primarily system() or exec() to launch a reverse shell. An advantage of ROP in this case is that it circumvents a lot of detection systems, often looking for rouge calls to subroutines like exec() and system(), by executing these malicious instructions within the context of the program. [10]

With Return Oriented Programming taking a place in the forefront of exploit development, protections against ROP have been becoming more common. Among these are intrusion detection techniques that look for consecutive execution of short code snippets followed by a return instruction and taking appropriate action, like terminating the program.[10] It should be noted, however, that with all runtime protection measures there is an expense with computing power. An expense that low power chips like the ESP32, a chip running XTensa windowed ABI, cannot necessarily afford– though hardware implementations, like performance counters, make this considerably cheaper.[9] There are also compile time protections, such as removing return statements completely– thereby removing the potential of ROP entirely.

Jump Oriented Programming deals with these shortcomings of ROP while still adhering to the code reuse technique.

3 Microcontrollers Come In Many Shapes and Sizes

Microcontrollers are incredibly variable. With our interactions with technology growing more and more frequent and involved, developments in microcontrollers have risen to the challenge. And due to the variant nature of our interactions with them, different settings have necessitated the need for different chips. This has proven to be quite costly, and different products and techniques have come to fill in the market gap.[4]

3.1 The XTensa Suite of Microcontrollers

To fill this gap, we have the XTensa set of microcontrollers. XTensa is an architecture with the goal of addressing the high costs and low customization of traditional architectures. Developed by Tensilica

Inc., XTensa is used in a variety of IoT (Internet of Things) applications often due to it's configurability giving way to lower power, and cheaper, chips. [4]

The goal of XTensa is to provide a great deal of customizability and extendability while maintaining the same instruction set architecture across devices. This principle has led to XTensa becoming one of the premier chipsets in both the IoT space but also technology at large.[4] This can be seen in the automotive industry, where Cadence Tensilica XTensa processors with 'flexlock capability' earned the ASIL-D designation– the highest automotive security standard within ISO 26262.[11]

One prototypical example used quite extensively is the ESP32, the successor to the ESP8266 microcontroller. The ESP32 has been and is being used in a variety of contexts. It is a chip used by hobbyists, academics, and developers. The ESP32 microcontroller is used in a variety of circumstances where cost is a large issue, a common trend in IoT devices. The ESP32 chip comes with builtin Wi-Fi and dual mode Bluetooth, which combined with its aforementioned low cost makes it an attractive choice in many circumstances.

For this paper, we will be utilizing the ESP32 as a case study due to its prototypical nature and accessibility.

4 ROP is a problem on XTensa

With return oriented programming being on the frontier for new cyberattacks, it is no great surprise that it is a prevailing issue for XTensa. Return oriented programming has been demonstrated on XTensa chips with both windowed and Call0 Application Binary Interfaces (ABIs).[8, 6] XTensa is dominated by two ABIs– Call0 and Windowed. An Application Binary Interface is the interface that on which the program runs. It defines a very complete set of behaviors like calling conventions and system services.[12] These ABIs change how gadgets are chained together as the ABIs dictate how code is interfaced onto the hardware.[12, 6] The ESP32 microcontroller utilizes the windowed ABI in an effort to cut computation costs.[6]

One paradigm of many XTensa chipsets is the 'system on a chip', the fact that many chips run their programs directly on the firmware as opposed to on an operating system.[13] What this means in many cases is the lack of certain operating system services, such as an interactive shell. This lack of a shell means that more rudimentary ret2libc attacks become obsolete as there is no inherent capability to create a reverse shell simply by executing the system("/bin/some_shell"). For this reason all of the attacks will have to be done within the context of the uploaded program, as opposed to executing any kind of system binaries found in more conventional environments. The ESP32 is one such chip.[13]

This limitation makes ROP significantly more attractive as an attack vector as opposed to something like ret2libc.

4.1 ROP Chaining the Call0 ABI

First, before we get into the exploitation lets talk about the Call0 ABI. The Call0 ABI is somewhat standard with a few unique quirks. The fundamental difference that changes it as far as calling and returning procedures go, something important to keep in mind with ROP, is that a0 must be saved on the stack and restored from the stack before the return.[6]

Since there are no push or pop stack instructions within the instruction set, there is full reliance on the load and store instructions. The return address of the function is referenced via the stack pointer and an offset, which is not universally defined and can differ between compilers and even functions.[6]

For a return oriented attack to work, there needs to be a popping mechanism on the stack. Given that it is not an implicit instruction, this behavior must be replicated.

As is commonplace, with the return of a function the return address must be loaded from the stack and the stack pointer adjusted– popping off the function parameters. The only difference here is that these actions aren't done intrinsically by the return instruction but instead must be done via other instructions. This mechanism is commonplace so that the function epilogues themselves can be used as ROP gadgets, since all that needs to be done is to load the return address into a register and adjust the stack pointer.[6]

The construction of any kind of ROP payload requires knowledge of where the return address is located. Because the return address is referenced by the stack pointer and an offset, they both must

be known by the attacker to properly return into a gadget. What must be done then is to keep track of the increments in each gadget to facilitate the actual return.[6]

4.2 ROP Chaining the Windowed ABI

Given that ABIs dictate how programs' instructions interact with the underlying firmware, special considerations for these differences must be made with respect to attack vectors. ROP, by virtue of its reliance on return instructions, has many shortcomings with respect to the windowed ABI that make it less viable than it would be otherwise.[6]

Windowed ABI is atypical as far as ABIs go. The premise is to have a sliding window of registers where each call moves the window down and each return moves the window up. The distance that the window moves is dependant on the call/return instruction used. While the call instruction dictates the number of registers to slide, this is not the instruction that actually does the sliding. The entry instruction itself does the sliding.[6, 14]

In the windowed ABI, you have access to the 16 general registers labeled a0 to a15. These registers, however, are just a subset of a larger register file varying in size between 32 and 64.[6, 14] As opposed to other instruction sets, the window sliding is variable with increments in 4, 8, and 12 on a per-call basis.

The reason for this system is that it reduces code size and increases performance of programs by doing away with register saves and restores at the entry and exit of subroutines.[14]

In order to keep track of how many registers the cpu must slide, the information is stored in the two most significant bits of the return address by the call instruction. During the return instruction's execution, the two most significant bits of the current program counter stay the same as those bits in the return address are used for window sliding. For this reason, you cannot return to a different 1GB segment in memory with a windowed return– something of note in extremely large applications.[6, 14] Importantly however, by manipulating the return address an attacker is able to manipulate how far the window slides.[6]

With the number of registers in the system being finite, sliding the window cannot go on forever. When this happens, and the number of physical registers is exhausted to the point of not being able to slide the window, register spilling occurs. This calls a handler which saves the affected registers to the program stack. This happens with both underflow and overflow, with loading and storing necessitated respectfully.[14] The register overflow exceptions are subdivided into the length of the window shift, so there exists an exception for increments 4, 8, and 12.[6]

The fundamental difference between register windowing and call0 as it relates to return oriented programming is that the function epilogue is now replaced with the sliding window mechanism. All hope is not lost however, as the windowed return itself can be exploited. When registers have to be reloaded from the stack due to a windowed underflow, the underflow exception handler can be used to load malicious register values into the current register window. Due to that, the stack pointer as well as the return address, registers a1 and a0 respectively, can be manipulated by the attacker.[6] The inherent challenge here then in initiating a ROP chain is triggering a windowed underflow.

With the attacker being able to control how far the window slides, an attacker is able to select how many registers to load in from the stack back into the registers upon an underflow.[6]

4.3 Jump Oriented Programming

Jump oriented programming is another type of code-reuse attack. The fundamental difference between Jump Oriented Programming (JOP) and ROP is that JOP does away with the reliance on the stack and return instructions– giving it more flexibility in situations where this control may not be feasible. By forgoing this reliance, JOP avoids a lot of the preventative measures problematic for ROP as they are primarily reliant on the stack and return instructions.

Instead, JOP relies on jump instructions– particularly plentiful due to its heavy use in branching. A shortcoming of this is that the structure guaranteed by the stack reliance is not present. With a return instruction, control is passed to wherever the return address points too and the frame is popped off. With multiple frames pushed onto the stack, this controlled structure is inherently guaranteed. JOP does not provide this luxury as all it does is jump to the instruction without context or an inherent means of structure.[10]

This shortcoming is addressed with the introduction of a 'dispatch' gadget. The dispatch gadgets job is to facilitate the chaining together of gadgets, each forming primitive operations, in such a way to provide the desired functionality.[10, 15] The dispatcher gadget is outright fundamental to the success of a jump oriented attack as the chaining's structure is dependant on it.

Given the necessity of a dispatch gadget, an explanation is certainly warranted. The dispatch gadget's purpose is to navigate and jump to various points in the dispatch table. The dispatch table is a sequence of jump addresses that point to primitive gadgets, chained together to accomplish the desired functionality. The dispatch gadget functions by taking control of a register to act as a 'gadget counter' within the context of the disp: acting as a reference for the next point in which to jump to. Code snippets that are possible dispatch gadgets are ones that modify a memory address in a predictable way and jump to it. With the versatility of what constitutes a candidate, several viable snippets can be found in libc.[10]

The dispatch table, where the dispatch gadget will be pointed to, can exist in some attacker controlled memory such as the stack and the heap. This is easily accomplished as an overflow is not even required as nothing needs to be overwritten– it just has to exist in memory where it can be reached by a potential dispatch gadget.

An attacker can realistically launch an attack by overwritting the stack, a function pointer, or a setjmp() buffer– something that is more rigorously explored in other literature.[10]

5 What did we learn and why does it matter?

The fundamental principle here is that the devices that surround us should not be viewed as inherently secure if don't have a monitor. Devices such as the Esp32 should show us that security should be a priority in IoT devices instead of an afterthought. The truth is that while these smaller scale devices may not have access to the same kind of data that say, a production server may have, they still are gateways into our lives that we don't secure as much as we should. IoT devices can be exploited to do numerous things, especially in mass.

These harassment exploits could be just the tip of the iceberg. Given that these devices proliferate our lives to such an extent, there are so many of them in the average American's home at this point [1], these devices if coordinated in mass could have large scale effects on communities. These devices could be used as listeners, sniffing internet traffic for important information. These devices could be used in DDoS attacks against machines running in the internal network if enough of them are compromised. The limits of these attacks is only up to the imagination of the attackers– not a comforting thought in the grand scheme of things.

Acknowledgements

I would like to thank my mentor Professor Josh Siegel for providing me with the tools and support to create this. He was honest and frank when it mattered and that's the most I could ask of anyone.

References

[1] G. D. Maayan, "The iot rundown for 2020: Stats, risks, and solutions." [Online]. Available: https://securitytoday.com/Articles/2020/01/13/The-IoT-Rundown-for-2020.aspx?Page=1

[2] I. Rabinovich, "How to protect your iot-based smart home devices from hackers." [Online]. Available: https://www.iotforall.com/baby-monitor-hackers

[3] M. Zuo, "Hackers are stealing videos from private security cameras and selling them as home video tapes." [Online]. Available: https://www.scmp.com/news/people-culture/article/3127659/hackers-are-stealing-videos-private-security-cameras-and

[4] R. Gonzalez, "Xtensa: a configurable and extensible processor," *IEEE Micro*, vol. 20, no. 2, pp. 60–70, 2000.

[5] M. Prandini and M. Ramilli, "Return-oriented programming," *IEEE Security Privacy*, vol. 10, no. 6, pp. 84–87, 2012.

[6] K. Lehniger, M. J. Aftowicz, P. Langendorfer, and Z. Dyka, "Challenges of return-oriented-programming on the xtensa hardware architecture," pp. 154–158, 2020. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9217805

[7] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," p. 552–561, 2007. [Online]. Available: https://doi.org/10.1145/1315245.1315313

[8] Z. Xu, J. Zhang, S. Ai, C. Liang, L. Liu, and Y. Li, "Offensive and defensive countermeasure technology of return-oriented programming," pp. 224–228, 2021.

[9] C. H. Adebayo Omotosho, Gebrehiwet B. Welearegai, "Detecting return-oriented programming on firmware-only embedded devices using hardware performance counters." [Online]. Available: https://www.infosun.fim.uni-passau.de/ch/publications/sac22.pdf

[10] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," p. 30–40, 2011. [Online]. Available: https://doi.org/10.1145/1966913.1966919

[11] C. Newsroom, "Cadence tensilica xtensa processors address most stringent automotive functional safety requirements with full iso 26262 compliance to asil-d." [Online]. Available: https://finance.yahoo.com/news/cadence-tensilica-xtensa-processors-address-144500794.html

[12] "Understanding the application binary interface," pp. 373–378, 1991. [Online]. Available: https://doi.org/10.1007/978-1-4612-3192-9_25

[13] M. Babiuch, P. Foltýnek, and P. Smutný, "Using the esp32 microcontroller for data processing," in *2019 20th International Carpathian Control Conference (ICCC)*, 2019, pp. 1–6.

[14] "Xtensa instruction set architecture (isa)." [Online]. Available: https://usermanual.wiki/Document/Xtensa2020ASSEMBLER20GUIDE.1231659642

[15] F. Yao, J. Chen, and G. Venkataramani, "Jop-alarm: Detecting jump-oriented programming-based anomalies in applications," pp. 467–470, 2013.