Watchdog Monitoring for Detecting and Handling of Control Flow Hijack on RISC-V-based Binaries

Toyosi Oyinloye[1], Lee Speakman[2], Thaddeus Eze[3] and Lucas O'Mahony[4]
Department of Computer Science
University of Chester, Chester, UK[1,3,4]
School of Science, Engineering and Environment
University of Salford, Manchester, UK[2]

***Abstract*—Control flow hijacking has been a major challenge in software security. Several means of protections have been developed but insecurities persist. This is because existing protections have sometimes been circumvented while some resilient protections do not cover all applications. Studies have revealed that a holistic way of tackling software insecurity could involve watchdog monitoring and detection via Control Flow Integrity (CFI). The CFI concept has shown a good measure of reliability to mitigate control flow hijacking. However, sophisticated attack techniques in the form of Return Oriented Programming (ROP) have persisted. A flexible protection is desirable, which not only covers as many architecture structures as possible but also mitigates known resilient attacks like ROP. The solution proffered here is a hybrid of CFI and watchdog timing via inter-process signaling (IP-CFI). It is a software-based protection that involves recompilation of the target program. The implementation here is on vulnerable RISC-V-based process but is flexible and could be adapted on other architectures. We present a proof of concept in IP-CFI which when applied to a vulnerable program, ROP is mitigated. The target program incurs a run-time overhead of 1.5%. The code is available.**

***Keywords*—Watchdog; return oriented programming; RISC-V; control flow integrity; software security**

I. Introduction

Securing software from hijacking and exploitation is a major step in software development lifecycle and has been faced with persistent challenge especially in the area of control flow hijacking. Attacks via Return Oriented Programming (ROP) [1] remain a source of concern in spite of existing basic and sophisticated protections. Basic protections like DEP/NX [2], which are generic do not mitigate ROP. This is because they are centered around blocking execution of injected code. Although these provide some measure of protections, attacks that stem from code reuse [3] like ROP which are not detectable by memory protection mechanisms, cannot be stopped by data execution prevention. On the other hand, the original CFI [4] relies on the precision of the Control Flow Graph (CFG). The CFG facilitates the detection of abnormal behaviour in the protected process but [4] inaccuracies in the CFG is one of the limitations cited by [4]. Besides, the classic CFI is a non-generic solution. Another non-generic CFI-based solution is Modular Control Flow Integrity (MCFI) [5] which offers a reasonably high level of precision. Recent studies also proffered solution in the form of PUFCanary FIXER [6], a hardware-based CFI which is also limited due to possible information leak where the PUFCanary FIXER inherits known canary limitations. There are other variations of CFI implementation which have contributed positively to the efforts to combat Control Flow Hijacks (CFH) but limitations exist. This could be due to specificity in the architecture that the solution was built on, as we have it in [4], or variation in source code language of target program as we have in MCFI [5], or general cost of implementation for hardware reliant fixes as exist in [7, 8], or the overhead incurred. Gaps and limitations in the realistic adoption of existing solutions inspire the continuous search for adaptable protection for vulnerable applications against attacks like ROP.

Aside from these limitations in existing protection techniques, studies on software protection are mainly implemented on specific system architectures focusing on elements that are involved in the execution of applications. Previous studies on protection measures have mostly focused on earlier architectures like x86 [4] and ARM [9]. This is justifiable because in past years, attention has been given to securing computers and servers which are mostly built on x86. However, in recent years, new technologies have emerged which require more options of protections. The RISC-V [10] technology is one of such which in recent years has gained popularity among producers of CPUs for automotive, smart devices, health tracking devices, etc., [11] because it is open source and more affordable. Also recently, the first laptop running on RISC-V processor has been introduced [12]. With these advancements in technology, a proactive measure of protection is desirable for vulnerable applications and the infrastructures on which they reside. RISC-V is an open-sourced instruction set architecture (ISA) and it was built on the already-established RISC technology [13]. Unlike most other ISAs, RISC-V was designed by academics and was made to be flexible and affordable, not only for use in academic research but also for deployability in hardware and software designs without incurring any royalties. For this reason, producers of embedded device, smart devices, etc., have opted for it. Not much attention has been given to securing RISC-V compared to other architectures like x86. Existing protections might not adequately provide the needed protection for RISC-V- based programs and systems.

Recent studies [14, 15] have highlighted gaps in existing protections especially for RISC-V programs and specifically against ROP as a result of hidden execution paths in the Control Flow Graph applied for implementing CFI. This study was embarked on with the goal to fill the gaps by increasing adaptability of protection mechanism for surmounting ROP. The concept implemented in this paper builds on a previous study [16] which proposed the possibility of securing vul-

nerable processes via inter-process communication. A novel approach was derived as Inter-process CFI (IP-CFI) which adopts the CFI concept alongside inter-process signaling and watchdog monitoring to detect abnormal behaviour in vulnerable applications. The vulnerable process is monitored by another process during execution. If a deviation is suspected in the control flow of the process, a watchdog function and inter-process signaling is triggered to further handle control flow monitoring.

According to [16], in-line CFI could be implemented by inserting labels to mark the start and end of each function with some additional instructions to perform checks on the flow of execution. In building on this technique, the watchdog adopts the time-out concept to extend monitoring whenever the process exceeds a stipulated time frame. The idea of watchdog monitoring is not new for securing systems. For example, study by [17] presents the *grenade* for monitoring mobile apps especially against denial of service (DoS) attacks. The research [17]'s *grenade* uses a countdown timer which is not reloadable once it begins to countdown. The author in [17] opted for this same technique to avoid a hijack of processors where a malicious program is able to extend its own life. In a similar line of thought, we avoid a possible extension of any malicious code execution by running a waiting time based on the intended purpose of the protected process. This is because unlike *grenade* which relies on the operating system timer, IP-CFI uses a monitoring program that is dedicated for monitoring a target process to increase flexibility. Some waiting time is also triggered in the target process and the monitor to achieve inter-process signaling. If a CFH is detected, further exploits can be prevented by an outright halting of the process.

The detection is made possible by initially analysing the vulnerable program to chart its intended execution path representing the CFG of the program, through static and dynamic analysis. In IP-CFI, values are passed from in-line CFI into shared memory where the monitor performs status check of the vulnerable process. Inter-process communication is achieved using atomic operation via semaphore and mutex on shared memory. Values that are used in the monitoring processes are stored in immutable registers and set in assembly code before completing compilation. Since IP-CFI is a software-based implementation which involves addition of enhancement code, the target program would require rebuilding after appending the enhancement code to implement the new protection.

This paper discusses the use of static analysis, dynamic analysis, RISC-V assembly coding, insertion of in-line checks for IP-CFI which provides a behaviour-based detection, and handling of CFH via Inter-process signaling on programs built and run on the RISC-V architecture. Most similar existing solutions are centered on the x86 system architecture and lack capacity to protect applications running on other CPUs like RISC-V. For this reason, the solution presented here is built around the RISC-V architecture but could be adapted for programs running on other CPUs. The rest of this paper is structured beginning with related works discussed in Section II. The methodology and implementation details are held in Section III while Section IV highlights the outcome of implementation, evaluation and application. Section V is a conclusion on this study and possible future works.

II. Related Works

Studies on software exploitation and protection have revealed control flow hijacks as a major source of concern in software security. Researchers have identified strengths and weaknesses of existing mitigation techniques by demonstrating various instances and concepts. The issue of CFH is particularly complex because there are different factors that need to be considered in proffering a lasting solution. This could be the consideration of the programming language of build, the low precision of CFG, CPU architecture on which the applications are running, and the cost of applying hardware solutions. The protection offered through CFI has potentials if applied alongside external enhancements. According to [18], CFI being a concept is flexible and could be enhanced by additional operations.

The classic CFI which was implemented on x86 architecture by [4] presented a promising solution to the challenge of CFH. A concept that relies on expected behavior, detection of deviations from expected behavior and trustworthiness of detector/enforcer. The classic CFI concept makes use of CFGs to apply inline reference monitoring (IRM) with which the protected application is rewritten. This was however found to be inefficient in its fine-grained form and not realistically implementable. Since the outcome of study by [4], other implementations have been studied in [5, 19, 6, 8], etc. These held some reliable outcomes with variations in structure, model, and platform but the mechanism still involves cross-checking flow of execution in comparison to intended flow. The integrity of the process is then enforced by introducing a halt or other forms of handlers to the situation.

Aside from existing limitations is realistic implementation of the classic CFI, recent studies [14, 15] have revealed the possibility of Hidden Execution Paths (HEP) which are not detectable and therefore omitted in the mapping when a CFG is built. While addressing ROP as a threat model on RISC-V, [14, 15] identified how ROP could persist on RISC-V platforms as a result of gadgets that could be obtained from overlapping code. It is therefore desirable to have a protection that is capable of an overview of the protected program. IP-CFI does not seek to know what gadgets are involved in the attack but to ensure the continuity of genuine execution and the termination of illegitimate flow in execution. Previous forms of CFI [4, 20] have their protection mechanism lying within the protected binaries which to an extent provides an impactful protection but the CFI itself might be unreliable due to low precision in CFG. In the case of [20], HEPs that were recently identified [14] could enable attack bypassing the checks. In this study, an additional monitoring process is adopted so that the in-line CFI could be monitored from outside of the target process while a watchdog is triggered if a deviation is suspected.

We present here a software-based protection. The classic CFI [4] was also software-based and was accomplished without recompilation of the program and no access to source code. This was made possible on the x86 implementation because CFG that was used in performing CFI checks were built using Vulcan [21]. Vulcan is not yet compatible with the RISC-V environment and as the program used here is a simple one, the CFG was done using Ghidra and Gdb for analysis. On the other hand, there exists compiler-based implementations like Gfree [20] which requires a part of the binary to be rewritten

and recompiled because the solution aims to eliminate gadgets that are based in libraries. Similarly, the protection here applies some additional lines of instruction to the protected target at the assembly level and also requires a full compilation into executable after the enhancement code has been inserted. The insertion of code has been automated but in-line checks are still inserted manually. A RISC-V target C-source code can be passed as argument into a startup script to be compiled with the enhancement code. The monitor however runs separately and need to be run concurrently as with the target program.

A state-of-the-art study presented in [18] observed that all eleven software-based CFI that were examined could be bypassed, although they each provided some protections in one way or another. They identified fine-grained CFI as a strong defense but incurs high overhead because of the use of shadow stack. Coarse-grained CFI on the other hand is a looser form that checks if control-flow transfers has originated from a return instruction and if its destination can be targeted. Three hardware-based CFI were examined and they were identified as difficult to be realistically implementable as such approach requires changes to the IT ecosystem that would incur additional cost on the system. [18] made conclusions that a hybrid form of CFI that combines existing protections might improve security in a CFI protected program. This study aims to adopt this suggestion by combining the use of In-line CFI, Inter-Process Monitoring (IPM), and watchdog time-out.

Another recent study [6] presents FIXER for protecting RISC-V applications using hardware. Subsequent improvement on FIXER involves the use of a PUFCanary [22]. This however had its own limitations in that a diversion may occur before the canary check, also FIXER does not protect against memory disclosure and it may cause the custom instruction to be bypassed.

[17] came up with a study back in 2000 with foresight on the advancement in technology in the future which we are now in. They foresaw a future where the use of computerised devices would become the norm and based on this, they presented the idea of *grenade* based on the concept of watchdog timer to protect against malicious mobile apps and ensure stability in services running on vulnerable systems. The use of a watchdog timer is a reasonable option for combatting a variety of attacks including Denial of Service (DoS) attacks. This is a relatable scenario as we find that ROP attacks on RISC-V, when chained in some particular order could lead to denial of service. The protection presented here adopts a similar approach by applying the watchdog concept alongside in-line CFI to ensure that such DoS attacks are detected and handled.

During this study, we identify RISC-V ROP gadgets that cause the denial of service. Among numerous possibilities of the outcome of ROP, DoS could occur when chained ROP gadgets don't include an instruction to redirect execution to a location where other chained gadgets could be executed. This would normally involve the use of a *ld* instruction to change the value of the previous return address to the next destination. For example, using *ld ra sp(40)* to load the malicious address from a stack under attackers control into the *ra* register to be fetched as next destination. If the gadget does not include this type of instruction, then the execution iterates over the last bunch of instructions via the previous value that is in the *ra* register causing a loop. In this case, execution results in a loop over the last bunch of instructions in the chained gadgets. The author in [17] also relate with a similar circumstance by giving another practical scenario where a bunch of code running on an electricity meter device should trigger a reload of credits to sustain service. However, an interference in transaction between user's bank and the meter, due to malicious code that leads to an endless looping of a bunch of code would not ensure that the meter is turned off if the user's account is not credited. This could be detrimental to the service provider as well as users.

There are various ways of evaluating new protections. The choice of percentage run-time performance evaluation is selected here because the executable binary changes after additional instructions have been added to it. A watchdog waiting time is included which inevitably increases run-time. In addition to these, the target now has to communicate with the monitor by passing out data via shared memory. All of these would impact the run-time as the program now does more than it was originally built to do. It is important to present the impact of the new technique with such useful detail so that the technique would adequately represent itself among other possible options. As producers of technological devices continue to build devices with variation in purposes, continuous study of possible protections for vulnerable software is needful. This continues to provide options in protections for users and vendors to choose from. This study has selected a distinct feature of watchdog waiting time combined with in-line CFI checks via inter-process monitoring as another means of enforcing CFI.

III. Threat Model

Previous studies have revealed ROP as a persisting threat to vulnerable programs. There could be other threats occurring in form of UAF [23] and double free [24], etc. This study focuses on ROP as a threat particularly when the chained ROP gadget ends up in a denial of service. The sample C program for this study was written with the bugs that are relevant for simulating the threat model. The program accepts input from user at some point in execution and also has a buffer overflow vulnerability which was leveraged upon to mount ROP attacks. Two new gadget finders were written to extract gadgets from the sample program and selected gadgets were chained to be passed into the target program as input to mount ROP attacks. The outcomes of the ROP attacks differed because of the difference in the ret gadgets and the order in which the gadgets were chained. A more detailed discussion on this would be featured in our future works. One of the outcomes from the various ROP attacks was selected for use here as threat model to demonstrate how the IP-CFI works. The gadgets were chained in a planned order such that when the byte stream is passed as input into the target process, a trap is hit where the program runs into a loop causing a denial of service.

IV. Summary of the IP-CFI Approach

In this section we discuss an overview of the IP-CFI approach. More details to elements in the protection system are given in Section V. The IPC-CFI is built as a protection system where the vulnerable process is monitored by another

process and values relevant for protection exist within the target process which is monitored by another process. The goal is to monitor the execution flow upon entry and exit from each function. The first step taken was to use GCC to compile the vulnerable C program with libraries inclusive so as to increase the possibility of having useful gadgets in the binary. Next, simple static and dynamic analysis on the vulnerable program to identify what elements are critical in the execution path of the process. Analysis tools were Ghidra for reverse engineering, Objdump for static analysis and Gdb for dynamic analysis. The analysis gave us a clear mapping of the intended control flow and the choice was made to insert lines of assembly instructions (enhancement code) to label all function prologues and epilogues.

The choice of *777*, *888* and *0* as values to be set as labels was made for experimental purpose and future works will introduce how the values could be hidden through encryption or applied at run-time. *777* is used to identify intended function prologues while 888 marks the unintended functions. The value *0* is passed at the epilogues to trigger a switch off in the in-line CFI value. The labels serve as values for in-line CFI checks as well as values for inter-process CFI checks. For inter-process, these values get written into shared memory through atomic operation of semaphore and mutex. The values are interpreted by the monitor as flags to indicate the status of the functions within the target during execution. This makes up the first part of the IP-CFI protection.

The second part of the protection involves the monitoring where the status value that was written to the shared memory is harnessed for further CFI checks. To achieve this, a C program was written which applies atomic operations to read into the shared memory. The program also implements a watchdog timer based on the status value read from the shared memory and halts the target process if a CFH is detected. In evaluating the effectiveness of the method, we analysed the run-time performance overhead. This was done by running two timed versions of the program with normal input 110 times. One version was the original program and the second version was one that had the enhancement code for protection applied before compilation. Data cleaning was done to eliminate 10 outliers form each data set and statistical analysis were done to validate the result of the two data sets of the run times in seconds. An average was calculated for each data set of 100 run-times. The average values were then applied to the formula:

Overhead = (Run-time with IP-CFI – execution time without IP-CFI)/execution time without IP-CFI

The overhead was calculated without the waiting time of 5 seconds which is required by the target process to enable inter-process communication. The overhead obtained is reasonable considering that some lines of code were inserted into the target for the new protection. The information obtained in the implementation were then used to make deductions and propose possible future works.

A. Exploitation and Protection Implementation Environment

The exploitation and protection implementation environment was set on a Linux Fedora computer system within which an embedded Linux Fedora RISC-V64 QEMU emulator (Fedora EM) was running as shown in Fig. 1. The use of the QEMU was necessary as the RISC-V architecture has not yet been adapted for direct run on PCs. The Fedora EM once started, was used to create, edit, compile and recompile the target program with all enhancements.

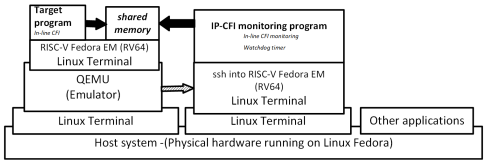

Fig. 1. Setting up a RISC-V System on QEMU Emulator

B. The Target Program

The target program used in this project is vulnerable to buffer/stack overflow and ROP. To simulate exploitation, we pass a of chain of ROP gadgets into the target. Details of this is given in Section V. Protection of the target in the first instance, adopts a CFI concept which makes use of checks inserted at the function prologue and epilogue of all direct functions. A comparison between the value that marks the intended function and the expected value from where it is called makes it possible for the CFI to detect a hijack. If the condition is not met, then the next instruction is executed, which halts the program, thereby avoiding further exploitation. As this In-line CFI on its own does not adequately protect the process from attacks stemming from ROP, we introduce other relevant protection concept in form of watchdog timing out where Inter-Process Communication (IPC) is used to establish IPM as a supervisory mechanism over the in-line CFI. In this case, the target process writes the in-line CFI value out into a shared memory and that value serves as the status value for an independent process to read and determine what action to take based on the status.

C. The IP-CFI Monitoring Program

The monitor is written in C program and values to be checked will be fetched from the relevant registers. It is an external independent process that is run concurrently with the IP-CFI-enabled target program to keep track of its execution. The external process consists of supervisory routine and a watchdog timing-out function which are implemented to ensure that the process maintains its intended flow. This Monitoring process communicates with the target program by reading its status from shared memory.

D. Shared Memory

In the Linux environment, there are two APIs that could be used to facilitate IPC- System V and POSIX. Both APIs provide IPC objects for reading and writing, but POSIX is safer to use as it does not permit execution for any category of user. According to [25], POSIX APIs are multithreaded-safe and we find it relevant for this project. POSIX APIs are also implemented with a backing file and we use that approach here to ensure compatibility, portability and persistence while the monitor accesses shared memory. In setting up shared memory, we mapped a shared file into the memory region *shm_open* using *mmap()*. The file could persist unless we delete it using

shm_unlink. We used *shm_open* to open the shared memory object and we used semaphore to avoid race conditions. The semaphore is also used as a mutex to lock/unlock access for the monitoring process and the target process.

E. Control Flow Integrity

CFI hinges on the ability of protection tool to observe the behavior pattern of the protected program during execution, detect anomalies and enforce the control flow integrity. The steps to it as implemented in IP-CFI is as follows:

1) Observing Program's Behaviour: The factors that are of importance in implementing this concept is a prior knowledge of expected flow of execution. This is a core step to the success of this method and it is achieved by carrying out thorough analysis on the vulnerable process. Each process varies in purpose, ability and vulnerability. The approach here is to establish the purpose of the process and identify vulnerabilities that are tied to the procedure by which that purpose is established. For example, a program that interacts often with users would have a higher attack surface. Also, processes that run for longer times will tend to be more vulnerable than short lived processes. The program is analysed by admin to identify the critical spots that lie within. The behavioural pattern is obtained from the CFG of the program prior to execution.

2) Detect Deviations: The success of a CFI-based protection depends on its ability to promptly detect a deviation from the expected pattern. Factors that are of importance here are the ability of the process to log in its status report into shared memory and to monitor delays in getting the status report value updated. The in-line value that is stored in the immutable registers are fetched and used to identify the status of the process. With this status, the watchdog is able to take the necessary action depending on the value read from shared memory. This also involves admin intervention prior to installation of the program. The time lapse that is permitted between the checks done by the watchdog is set prior to compilation.

3) Enforcing CFI: To enforce the integrity of the target process, CFI demonstrated here focuses on the external protection against CFH in a situation where attack bypasses in-line monitoring. This involves the insertion of instructions that enforce the CFI of the process. The enhancement code is inserted in assembly code into the protected binary. The needed elements for achieving these are relevant instructions and storage for the label values that are used in checking the legitimacy of each called function. The effectiveness of IP-CFI also lies in the trustworthiness of the detector/enforcer. To build an enforcer we combine two different sets of code in assembly language that are inserted into the target program - in-line CFI checks and Inter-process signaling code, and then a newly built external independent program that monitors the in-line CFI and inter-process status values.

F. The First Set of Enhancement Code

This fulfils the checks and enforcement of CFI and functions fully as in-line CFI. This relies primarily on the strategic positioning of checks in the target code. The positioning is determined by obtaining CFG of the program to see the possible pathway of execution. This involves a way of mapping out all subroutines in the program and identifying the direction of flow as intended by the programmer. A CFG could be built by making use of relevant reverse engineering tools. The authors in [4, 26] made use of Vulcan in developing a CFG but Vulcan is limited in use and is not implementable on RISC-V. Other useful reverse engineering tools are Ida and Ghidra which are effective in the x86 as well but none of these are yet to be effectively tailored to reverse engineer RISC-V-based applications. We have access to the sources code for our sample program here and that makes it a more straightforward process. However, in order to be able to use IP-CFI for programs that has been compiled without access to source code, a useful workaround that we applied was to use Ghidra on Fedora Linux running in x86 to reverse the x86 version of the same program. We found that this was useful foresight for easy analysing of the program in the RISC-V environment.

Further to this, static analysis informed us on the requirement for inserting the checks into the target. With these information, essential checks were inserted into the target in assembly code. The inserted in-line checks consist of fixed labels that mark intended functions along the execution pathway with matching values. Unintended functions along the execution pathways are marked with different values. At the beginning of execution, the value is set to *0* and would remain as 0 until a function call is made. If a function call is made, there would be two possibilities to the value which would either be a trigger to match the intended pathway or unintended pathway. The way labels are inserted would vary with the architecture of the underlying system. The RISC-V which is adopted here allows for straightforward storage of the values needed to achieve this process of the labels. The important objects for setting the values here are registers while the function prologue and function epilogues are used to position the in-line CFI checks and relevant action. Other pieces of code as shown in Fig. 2 that work in this phase are geared towards halting the target process based on the in-line CFI checks.

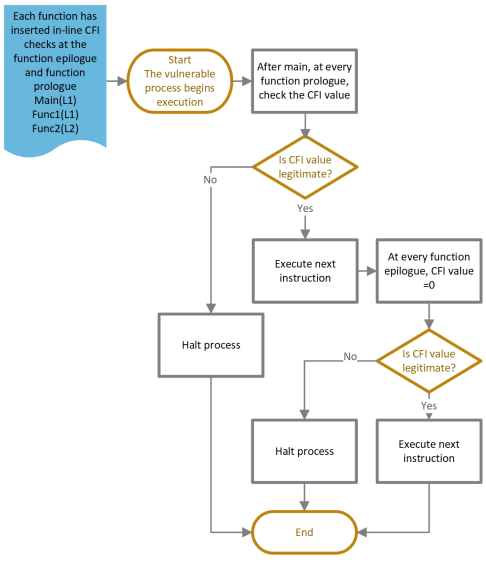

Fig. 2. Flow Chart of Target Process with In-Line CFI.

1) Wrongmove: This is a two-line instruction block that performs the halt to the vulnerable process by making a system call. For a sensitive program like the target, a proper handler needs to be triggered once a CFH is detected. Hence for this protection, the in-line CFI hands over control to the operating system by making an *ecall* on RISC-V. The *ecall* is an established RISC-V system call that ensures the transfer of control to the kernel so that the attacked process can be halted. This is achieved by setting the relevant value of *93* into a register *a7* in assembly code.

2) Storing the Label: The values used in the in-line CFI are set as numbers *777* and *888* to demonstrate the protection as these could be other values, provided that the value held by each of the intended nodes match the legitimate caller. An important decision that was made here is where to store the value to ensure that it is preserved throughout each function.

In RISC-V, values are stored in registers but the important register for storing our values were selected based on specific attributes. As the value in the label will be used for monitoring control flow status, it is important for the value to be unchangeable within its function. For this reason, the *s2-s11* registers are found suitable. These registers are referred to as saved registers. They have the ability to preserve the values stored in them within the function. Subroutines do not normally change the values and If they do, they will have to save the value and restore at the end of its execution.

For this purpose, only registers in this category can be used as other registers do not share the same attribute. They will either have specific roles or are temporary registers which means that the value that they hold will not be preserved. Apart from preservation, it is important and ideal to have labels that cannot be manipulated by attackers. This limitation is being studied for future improvement on the protection method. This would make those registers a strong tool for the implementation of this protection technique on RISC-V. For the demonstration here, the *s3* and *s4* registers were selected. Another option to using a known value as the label is to generate a scrambled value at run-time. This is however outside the scope of this study but is an area that could be considered for future studies. Outline of in-line checks as as follows:

G. The Second Set of Enhancement Code

This fulfils inter-process communication by logging status report from the target program to a shared memory. The status report is fetched from the outcome of the in-line CFI mechanism and values held in labels that have been set to mark the main execution path of the process, taking cognizance of its entry into and exit from critical nodes. This code is a new function that consistently writes the value contained in the *s3* and *s4* registers into a log to share the status of the target with a monitoring process.

1) Status Logging: The instructions that handle this step is inserted into the target assembly and it carries out *open*, *write* and *close* system calls to achieve this. It also applies an atomic operation involving a semaphore to these calls to avoid race conditions. Portions of this code were retrieved from [27] examples of shared memory.

H. IP-CFI

The CFI monitoring is enhanced by attaching an additional monitoring method involving another process attributed as not vulnerable as it runs with zero user interaction. The monitoring process performs the function of observing the target process by implementing an atomic inter-process signaling. The main tool that the monitor uses is information read from memory shared with the target process. In the case of ROP attack, it was observed that the status could appear legitimate whereas, a hijack has occurred undetected. For this reason, monitoring is extended to watchdog timing out.

1) The Watchdog Routine: The watchdog routine sets a counter to keep track of the target process. The demonstration in this report gives allowance of three cycles of checks by the watchdog. The first and second cycles could be enabled to restart the process without user intervention while the third cycle puts a halt to the target process. The number of allowable cycles can be adjusted to suit the performance or function of the protected program. For the demonstration here, the restart is not included for any of the cycles.

V. Implementation

As this protection is aimed at combating memory compromise through buffer/stack overflow that lead CFH via ROP input, promptness and accuracy in detecting deviations is very important it. The earlier a protection system is able to detect deviation and enforce integrity, the higher the chances of establishing a secure process.

A. Exploiting the Target

The first step to demonstrating the protection is to demonstrate an exploit. ROP is dependent on availability of gadgets and the ability of attacker to craft a byte stream to accomplish their malicious goal. Implementing ROP on RISC-V is more complex than x86 but is achievable. The aim is to control the execution from the stack by passing carefully crafted input through buffer overflow.

B. Gadget Finders

In order to make a variety of gadgets available to us, we wrote two gadget finders, *RETGadget* and *JALRGadgets* in Linux scripts and applied with the target as an argument to extract gadgets from it. The gadget finders are available and can work on RISC-V-based programs.

C. Passing Chained Gadgets

Once the gadgets were extracted, we mapped out ROP chains in various order based on a theoretical approach by [28]. According to [28], we can pass gadgets that will help us to store values and addresses in the registers that we intend to use to mount the ROP. The author in [28] classified the gadgets as functional gadgets and charger gadgets. The functional gadgets will hold the instructions for the actual attack while the chargers(linkers) gadgets will load the registers with addresses of the functional gadgets and other useful values. The linkers can be used to create a fake frame as shown in Fig. 3 and 4, which we can exploit further to pass malicious values into registers and other elements on the stack. Each of the ROP

chains was passed as input into the target to see how it could be exploited. In Fig. 3, the charger simply creates the fake frame with values loaded or copied from one location or register to the other. An exit is then called.

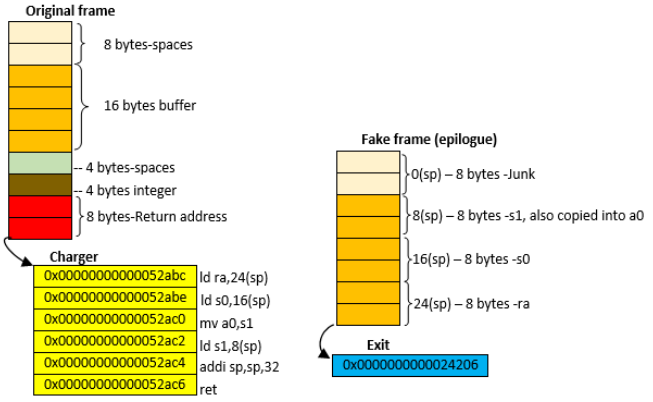

Fig. 3. Exploiting Target to Manipulate Some Registers and Exit Abruptly.

Further exploit is done as shown in Fig. 4 where two functional gadgets are chained to the linker. However, the outcome of this chained gadgets is different from that in Fig. 3 as this results in a loop. This is because we used a functional gadget that does not overwrite its previous value in *ra* register.

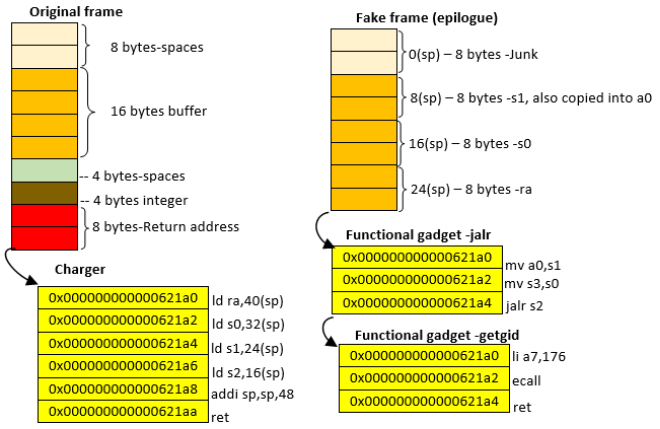

Fig. 4. Exploiting the Target to Cause a Loop.

The order in which gadgets are crafted in RISC-V would determine the outcome of the exploit. This has a lot to do with the value of the *ra* register that gets overwritten from time to time as execution steps into and out of library functions or other functions that get called within a function. This can only be detected during dynamic analysis and remains undetectable to user as no feedback is written to standard output but the process appears to be hanging as it doesn't crash. This a typical attack that could lead to denial of service and the IP-CFI protection is able to detect and handle it.

D. Protecting the Target

With the attack in place, we then applied the IP-CFI protection. The monitoring process is run concurrently with the target. Each time a new function is called in the target, the status is updated by the target process via the shared memory as shown in Fig. 5. The status value indicates what sort of function is being executed and at what stage of the function the execution is. Once the monitor reads into the shared memory, it

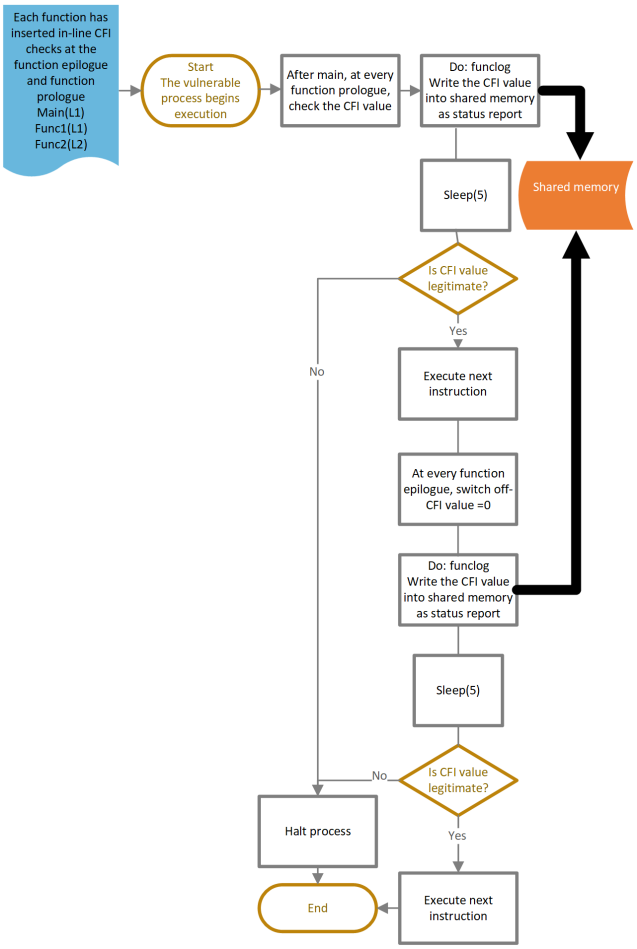

Fig. 5. Target Writes into Shared Memory.

takes the necessary action depending on the value. The denial of service is promptly identified and stopped. The possible flow of execution of the monitor based on the possible values that could be held in the status report is shown in Fig. 6.

E. Applying IP-CFI to a Source Code

There are three steps in setting up a program to use IP-CFI. We begin with the C source code and end up with an executable. The steps include two stages of compilation with the insertion of enhancement code between stages. The relevant scripts:*IP-CFI-make.sh* and *IP-CFI-full-compile.sh*, the monitor program *IP-CFI-monitor-watchdogv1*, and the enhancement code *IP-CFI-enhancement-code.s* are required.

Step 1:
Run *IP-CFI-make.sh* passing the C source code as argument

Step 2:
Find the resultant assembly (.s) and manually insert CFI-checks. Instructions are mapped out as follows:

Within main function:
Insert the following lines before the first call to a function

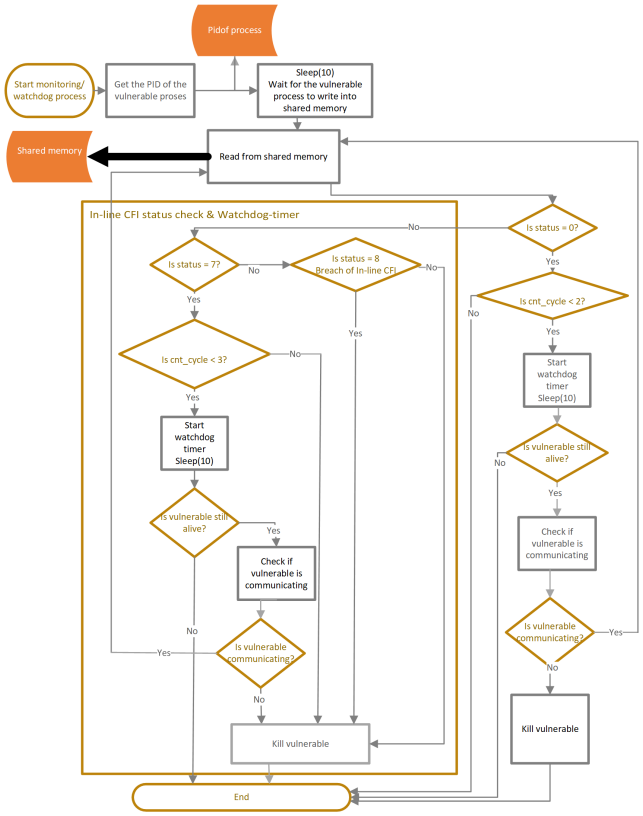

Fig. 6. Flow Chart Showing Inter-Process Monitoring as Status is Read from Shared Memory.

li s3,777 #set monitoring label for legit function

Within all intended functions that are not main:
Insert the following lines at the prologue:

Line 1:

li s4,777 #inserted for legit function

Line 4:

call IPCFIfunclog #log status

bne s4,s3,.Wrongmove. #enforce in-line CFI

Insert the following lines at the epilogue (just before *ra* register is loaded or before call to *exit*):

li s4,0 #Switch off CFI label

call IPCFIfunclog #update status

Within all unintended functions:
Insert the following lines at the prologue:

Line 1:

li s4,888 #inserted legit label

Line 4:

call IPCFIfunclog #log status

bne s4,s3,.Wrongmove. #enforce in-line CFI

Once all checks have been inserted save the file and exit.

Step 3:
Run *IP-CFI-full-compile.sh* passing the saved assembly (.s) file as argument. The resultant executable within current working directory can be run concurrently with the monitor. Work is in progress to fully automate these stages.

VI. Evaluation

In evaluating the protection, further observations were made with various forms of ROP chains passed into the target. An exploitable target was used. With the watchdog in place, the protection was applied to the target while different ROP chains were passed as input into it. The protection was found to be effective. The impact that the additional prologue and epilogue code might have on the execution time of the program was considered. In this section, we evaluate the new protection by obtaining the relative performance overhead with respect to our target program.

A. Results and Discussion

The outcome of analysing the run-time overhead of the new protection is discussed in this section. While IP-CFI effectively surmounts ROP with a run-time overhead of 1.5%, there could be variations in the outcome execution time based on the waiting time set in order to accomplish synchronisation between the monitor and the protected process. Details of how the run-time overhead is calculated are as follows:

Run-time type	***Average of run-time***
w/o IP-CFI	*3.26 ± 0.12 (milli seconds)*
w/ IP-CFI	*8.06 ± 0.16 (milli seconds)*

Run-time overhead w/ IP-CFI

((8.06 – 3.26)/3.26) = 1.47239264 (approx. 1.5%)

This presents a reasonable overhead when the additional pieces of code are included into the target. When the program is run concurrently with the monitor, some waiting time is required in order to establish communication between the target and monitor. This could vary from one program to another as it largely depends on the purpose of the program. For the sample program, a 5 seconds waiting time is applied to accommodate the time required for the monitor to read into the shared memory and take the necessary action.

While the full protection surmounts ROP, the waiting time applied appears to significantly increase the overall response time. However, the extra time incurred here is artificial. In reality, it is not additional run-time as the program would function fully without the waiting time. Waiting is needful to achieve interoperability between the two programs here and this outcome could differ if various scenarios are considered. In this instance, IP-CFI has been applied on a simple program and the outcome may vary more favourably with larger programs. On the path of the monitor, a 10 seconds wait is involved but this is independent of the target and does not impact the target run-time.

Furthermore, the run-time may vary slightly with the number of functions that accept the enhancement code. However, since the functions will only run one at a time, the overhead would not be greatly impacted. Also, optimization was set to *0*for the samples used here. The two options of level of protection could be applied to vulnerable process- one with full IP-CFI, and the other without the watchdog waiting time.

An alternative to the IPC via signalling, in order to monitor in-line CFI with less waiting time is the direct monitoring of the process from the kernel. A study by [31] presented this as a security measure but not with regards to CFI. According to [31], every process reports its logs somewhere in the file system and this could be harnessed as useful information for the kernel to monitor processes. If the kernel were to be used directly for monitoring the CFI, other related elements like the in-line CFI value and watchdog timing routine might need to be reconsidered.

A limitation of IP-CFI is that the values in the labels might leak and the registers that holds those values could be reused by an attacker. Although the impact of this with IP-CFI implementation against a CFH does not yet appear to be considerable. The possibility of locking the s3 and s4 registers is being considered for future works, as well as ways to encrypt the label values to generally improve on the IP-CFI protection.

Currently, RISC-V applications exist in highly sensitive eco-systems as they are commonly used and constantly running. IP-CFI is aimed as a broad spectrum of protection cutting across various eco systems. For efficiency, it is however aimed to protect applications that are built for long running services or those performing a single role. The RISC-V platform is well structured for such applications and is expected that the intended purpose of the application would inform the choice of protection. The RISC-V architecture is being implemented for several health monitoring devices. A recent study by [29, 30] presents a cutting-edge technology in form of an implantable medical device (IMD) for conditioning the human body electrical activity which runs on a RISC-V processor. [11] also produced a RISC-V-based microprocessor that could be used in devices for personalized health management aside from other devices like electronic voting machines, smart cards, etc. These devices match the category of devices that are dedicated for a single purpose and applications that are run on them might benefit from the IP-CFI protection.

VII. Conclusion and Future Works

Here, we have presented a proof of concept using IP-CFI, a new protection mechanism which is based on the concept of CFI combined with an external monitoring program. IP-CFI effectively resolves a denial of service from lingering when ROP is mounted. The main strength of the system is its ability to detect delays in change of the status value logged into shared memory where the monitoring process fetches information for taking actions towards maintaining the integrity of the protected program. With a prompt detection of delay, the target process can be halted to prevent furtherance of attack process.

The possibility of sustaining an execution while preventing furtherance of attack is an area that previous CFI solutions have not really addressed as CFI tends to halt processes once an attack is detected. One of the areas we explored in the process of this study is a way that the monitor could trigger a restart to the program rather than a halt. So far, we have no way of preserving existing data such that the restart of the process is done without side effects. This option would be explored in future works especially for environments that some of these vulnerable processes might require seamless continuity.

In this study, we have opted for a higher-level monitoring process to give us more control of the protection, as well as increase flexibility in the settings. The overall response time can be improved upon by optimising the IPC and setting the monitor to respond asynchronously. This is being considered for future works.

References

[1] H. Shacham, "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)," ACM conference on Computer and communications security, pp. 552-561, 2007.

[2] Microsoft Corporation, Microsoft Documentation, 31 May 2018. [Online] Available:https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention.[Accessed 22 April 2020].

[3] Tran, M., Etheridge, M., Bletsch, T., Jiang, X., Freeh, V., Ning, P, "On the Expressiveness of Return-into-libc Attacks," Sommer R., Balzarotti D., Maier G. (eds) Recent Advances in Intrusion Detection. RAID 2011. Lecture Notes in Computer Science,, vol. 6961, 2011.

[4] M. Abadi, M. Budiu, U, Erlingsson, and J. Ligatti, "Control Flow Integrity," in CCS '05: Proceedings of the 12th ACM conference on Computer and communications security, New York, United States, 2005.

[5] B. &. T. G. Niu, "Modular Control-Flow Integrity," in PLDI '14: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2014.

[6] De, A, Basu, A, Ghosh, S.,& Jaeger, T., "FIXER: Flow Integrity Extensions for Embedded RISC-V," in 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE), 2019.

[7] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar & D. Song, "Code-Pointer Integrity," in The Continuing Arms Race: Code-Reuse Attacks and Defenses, Association for Computing Machinery and Morgan & Claypool, 2018, p. 81–116.

[8] C. Sadullah, D. Leila, Z. Boyou, J. Ajay & E. Manuel, "Efficient Context-Sensitive CFI Enforcement Through a Hardware Monitor," in Detection of Intrusions and M17th International Conference, DIMVA 2020, Lisbon, Portugal, June 24–26, 2020, Proceedings, Lisbon, Portugal, 2020.

[9] M. Neugschwandtner, C. Mulliner, W. Robertson, & E. Kirda, "Runtime Integrity Checking for Exploit Mitigation on Lightweight Embedded Devices," International Conference on Trust and Trustworthy Computing, vol. 9824, pp. 60-81, August 2016.

[10] A. Waterman, K. Asanovi & J. Hauser, "The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Docu- ment Version 2021120," 2021.

[11] IIT Madras, "IIT Madras, Indian Institute of Technology Madras," IIT Madras, 24 Sept. 2020. [Online]. Available: https://www.iitm.ac.in/happenings/press-releases-and-coverages/iit-madras-develops-and-boots-moushik-microprocessor-iot. [Accessed 08 July 2022].

[12] DeepComputing, "xcalibyte.com," xcalibyte, 28 06 2022. [Online]. Available: https://xcalibyte.com/roma-preorder/. [Accessed 04 07 2022].

[13] A. Samuel O, "An Overview of RISC Architecture," in Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing: Technological Challenges of the 1990's, Kansas City, Missouri, USA, 1992.

[14] G, Gu, and H. Shacham, "No RISC No Reward:Return-Oriented Programming on RISC-V," 29 July 2020.

[15] G-A. Jaloyan, K. Markantonakis, R. N. Akram, D. Robin, K. Mayes, and D. Naccache, "Return-Oriented Programming on RISC-V," ASIA CCS '20: Proceedings of the 15th ACM Asia Conference on Computer and Communications Security, p. 471–480, October 2020.

[16] T. Oyinloye, L. Speakman, T. Eze, "Inter-Process CFI for Peer/Reciprocal Monitoring in RISC-V-Based Binaries," in 20th European Conference on Cyber Warfare and Security, 2021.

[17] F. Stajano, R. Anderson,AT &T Laboratories Cambridge, "The Grenade Timer: Fortifying the Watchdog Timer Against Malicious Mobile Code, " in Proceedings of 7th International Workshop on Mobile Multimedia Communications (MoMuC 2000), Waseda, Tokyo, Japan, 2000.

[18] S. Sayeed, H. Marco-Gisbert, I. Ripoll, and M. Birch, "Control-Flow Integrity: Attacks and Protections," Applied Sciences, vol. 9, no. 20, p. 4229, October 2019.

[19] M. Zhang and R. Sekar, "Control flow integrity for cots binaries," SEC'13: Proceedings of the 22nd USENIX conference on Security, p. 337–352, August 2013.

[20] K. Onarligolu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, "G-Free: defeating return-oriented programming through gadget-less binaries.," in Proceedings of the 2010 Annual Computer Security Applications Conference, New York, NY, 2010.

[21] A. Srivastava, A. Edwards, and H. Vo., "Vulcan: Binary transformation in a distributed environment," Microsoft Research:Technical Report: MSR-TR-2001-50, 2001.

[22] De, A, Basu, A, Ghosh, S., & Jaeger, T., "Hardware Assisted Buffer Protection Mechanisms for Embedded RISC-V," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2020.

[23] Common Weakness Enumeration, "cwe.mitre," 19 July 2006. [Online]. Available: https://cwe.mitre.org/data/definitions/416.html. [Accessed 17 July 2022].

[24] Common Weakness Enumeration, "cwe.mitre," 19 July 2006. [Online]. Available: https://cwe.mitre.org/data/definitions/415.html. [Accessed 17 July 2022].

[25] Oracle, "Oracle Programming Interfaces Guide," 2012. [Online]. Available: https://docs.oracle.com/cd/E26502_01/html/E35299/svipc-posixipc.html#scrolltoc. [Accessed 04 06 2022].

[26] E. Göktaş, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out Of Control:Overcoming Control-Flow Integrity," in 2014 IEEE Symposium on Security & Privacy, San Jose, CA., USA., 2014.

[27] M. Kalin., "Inter-process communication in Linux: Shared storage," 2019.

[28] B. Deac, "InfoSec Write-ups," 14 March 2022. [Online]. Available: https://infosecwriteups.com/return-oriented-programming-on-risc-v-part-1-dd9817b52d2b. [Accessed 25 06 2022].

[29] A. Arnaud, M., Miguez, J. Gak, R. Puyol, R. Garcia-Ramirez, E., Solera-Bolanos, R. CastroGonzalez, R. Molina-Roblkes, A. Chacon-Rodriguez, R. Rimolo-Donadio, "A RISC-V Based Medical Implantable SoC for High Voltage and Current Tissue Stimulus," in 2020 IEEE 11th Latin American Symposium on Circuits & Systems (LASCAS), Costa Rica, 2020.

[30] A. Arnaud, M., Miguez, J. Gak, R. Puyol, R. Garcia-Ramirez, E., Solera-Bolanos, R. CastroGonzalez, R. Molina-Roblkes, A. Chacon-Rodriguez, R. Rimolo-Donadio, "Siwa: a RISC-V RV32I based Micro-Controller for Implantable Medical Applications," in 2020 IEEE 11th Latin American Symposium on Circuits & Systems (LASCAS),, Costa Rica, 2020.

[31] W. Kehi, G. Yueguang, C. Wei & Z. Tong, "The Research and Implementation of the Linux Process Real-Time Monitoring Technology," in 012 Fourth International Conference on Computational and Information Sciences, 2012.