FIXER: Flow Integrity Extensions for Embedded RISC-V

Asmit De
School of EECS
The Pennsylvania State University
University Park, USA
asmit@psu.edu

Aditya Basu
School of EECS
The Pennsylvania State University
University Park, USA
aditya.basu@psu.edu

Swaroop Ghosh
School of EECS
The Pennsylvania State University
University Park, USA
szg212@psu.edu

Trent Jaeger
School of EECS
The Pennsylvania State University
University Park, USA
trj1@psu.edu

***Abstract*—With the recent proliferation of Internet of Things (IoT) and embedded devices, there is a growing need to develop a security framework to protect such devices. RISC-V is a promising open source architecture that targets low-power embedded devices and SoCs. However, there is a dearth of practical and low-overhead security solutions in the RISC-V architecture. Programs compiled using RISC-V toolchains are still vulnerable to code injection and code reuse attacks such as buffer overflow and return-oriented programming (ROP). In this paper, we propose FIXER, a hardware implemented security extension to RISC-V that provides a defense mechanism against such attacks. FIXER enforces fine-grained control-flow integrity (CFI) of running programs on backward edges (returns) and forward edges (calls) without requiring any architectural modifications to the RISC-V processor core. We implement FIXER on RocketChip, a RISC-V SoC platform, by leveraging the integrated Rocket Custom Coprocessor (RoCC) to detect and prevent attacks. Compared to existing software based solutions, FIXER reduces energy overhead by 60% at minimal execution time (1.5%) and area (2.9%) overheads.**

Keywords—Buffer overflow, ROP, Shadow Stack, RISC-V

I. Introduction

A. Security Vulnerabilities

Traditional computing systems are inherently vulnerable to a wide attack surface from the topmost application level to the systems architecture level, leading to serious security and integrity concerns such as leaking private SSH keys or launching Denial-of-Service (DOS) attacks. Programming languages like C which are closer to the hardware, provide a lot of flexibility in terms of memory and IO access to facilitate system and device level programming. However, this also means that such languages often tend to have inherent security deficiencies and can lead to vulnerabilities if not used with proper and secure practices. Buffer overflow is the most commonly exploited vulnerability that can cater to a wide attack surface. In a program without bounds checking, an adversary can overload a user input with excess data that can overrun the buffer capacity and overwrite nearby memory locations with potentially malicious data (Fig. 1), leading to several attack scenarios, such as return-oriented programming (ROP), VTable hijacking, function pointer manipulation and even violation of data flow in program.

B. Defense Mechanisms

Stack canaries [1] are *sacrificial* words placed on the stack at stack frame boundaries to detect potential return address overwriting. If an adversary overflows a buffer in order to overwrite the return address, the canary word will also be overwritten. Before returning in call stack, canary word is checked, and if modified, the return address is assumed to be compromised, and the program is halted.

Data Execution Prevention (DEP) [2] is employed to prevent an adversary from injecting malicious code onto the stack. Memory pages are marked W⊕X, meaning, a page can either be executable (code) or be writable (stack, heap), but not both. This prevents an adversary from executing malicious code from the stack. However, an adversary can return to existing code in the program or functions in the linked library using gadget chains (return-to-libc attack).

Address Space Layout Randomization (ASLR) [3] randomizes the code, stack, heap, and shared library locations on the address space, to make it difficult for the adversary to determine the specific addresses and launch attacks. However, buffer over-read and side-channel vulnerabilities can be used by an adversary to reverse engineer the randomized address.

Control Flow Integrity (CFI) [4] involves statically computing a valid control flow graph (CFG) of the program and ensuring that during runtime, the program abides by that CFG. A coarse-grained approach to ensuring control flow integrity while returning from functions is the use of a shadow stack (a separate stack residing in a secure memory location) [5]. On each function call, the return address is saved on the shadow stack alongside being put on the stack normally. While returning from a function, the return address on the stack is validated against the one on the shadow stack. On mismatch, it is assumed that the return address has been compromised and the program execution is halted. However, a shadow stack can be expensive and can hurt performance since the pages housing the shadow stack may not be present in cache and will require hundreds to thousands of cycles to bring the page onto the cache and perform the validation. Several software and compiler level systems have been proposed in literature for supporting shadow stack [6-7].

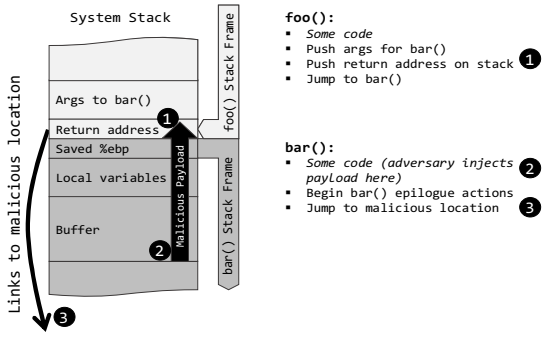


Fig. 1. Buffer overflow exploit.

Even with the presence of a shadow stack, an adversary can bend the control flow of a program. To prevent such incorrect control flows for indirect calls, the program is first analyzed to compute a coarse-grained or fine-grained CFG [4]. A control-flow policy matrix can then be created from the CFG that specifies the allowed call targets for each call site. During execution of the program, for each indirect call, the policy matrix is looked up to determine the validity of the call target. However, this approach still suffers from similar performance degradation if the policy resides in memory. Compile-time and runtime enforcement of CFI have been shown in [8-9]. Lazy CFI [10-11] can somewhat alleviate the performance loss, but that leaves room for generating false negatives.

Secure hardware platforms e.g., ARM TrustZone [12] and Intel Software Guard Extensions (SGX) [13] isolate the hardware so that the access to systems assets are restricted. Hardware acceleration of security validation has been proposed to address the performance impact partially while covering a subset of security threats e.g., Intel CFI Enforcement Technology (CET) [14] to protect against control-flow hijacking. Intel Memory Protection Extensions (MPX) [15] with extended instruction set architecture is developed to prevent memory safety violations such as buffer overflow, heap overflow and pointer corruption. Intel Transactional Synchronization Extensions (TSX) [16] exposes and exploits hidden concurrency in multi-threaded applications. Intel PT [17] logs TSX events when a transaction begins, commits or aborts. It has been shown in [18] that tagging of code and data using software-defined metadata and processing the tag using custom designed processor can detect ROP, code reuse, buffer overflow, code injection, memory safety violation and pointer corruption. Although effective, this new architecture cannot be readily deployed due to lack of re-configurability, and, area, energy and performance overhead. Other hardware-assisted techniques to protect forward and backward edges in control flow are proposed in [19-22]. Data flow protection in stack and heap using hardware assistance is also proposed [23-24]. Specialized hardware stack redundancy systems have also been developed for embedded systems [25-28], however these are architecture dependent and cannot be updated post-deployment.

The common challenges associated with the existing secure hardware platforms include design overhead, lack of provisions to patch the design and keep pace with rapidly evolving threats, need of code changes or instrumentation of the program binaries, compiler modifications, and, lack of adaptability to adjust the security level in runtime as needed. Furthermore, these platforms are associated with performance impact. To alleviate these issues, a decoupled architecture using hardware performance monitors implemented on a RISC-V coprocessor has been proposed in [29].

In this work, we propose Flow Integrity eXtensions for Embedded RISC-V (FIXER), a low energy, low overhead security solution that ensures integrity of backward and forward edge control flow of programs running on a RISC-V core. FIXER decouples the security architecture from the RISC-V core architecture, enabling a highly flexible security system design. In the target deployment platform, the unmodified RISC-V core will be a hard IP, while the dynamically reconfigurable FIXER coprocessor will be implemented on an on-chip FPGA. Such an approach has the potential to be scaled to hybrid processor designs e.g., a Xeon + FPGA core [30]. In such designs, the primary core can be completely unmodified, while the re-configurable FPGA core can be utilized to implement the security architecture. The FPGA also provides the flexibility to change and update the security architecture in demand to new threats, without a complete redesign of the primary computing core. With the number of vulnerabilities rapidly increasing, it demands an efficient low-power flexible and scalable security solution that is sustainable for long periods of time. FIXER potentially unlocks the design capability to protect our systems from such cybersecurity threats. Software based CFI techniques are also limited by the size of the address space, which can be overcome by FIXER's flexible FPGA implementation. Compared to NILE [29], FIXER achieves better performance. Although NILE uses an unmodified RISC-V core similar to FIXER, the core-coprocessor interface is modified for the coprocessor to tap into more resources of the core. Table I shows a qualitative comparison of FIXER with the state-of-the-art memory protection solutions. The major contributions of this work are, (a) a decoupled and flexible coprocessor based design for security assurance; (b) enforcement of backward edge and forward edge CFI protection; (c) low energy overhead than [29]; (d) ease of re-configurability to address new security threats and attacks.

TABLE I. QUALITATIVE COMPARISON OF FIXER WITH RELATED WORKS

	Canary [1]	ASLR [3]	CFI [4]	PUMP [18]	HAFIX [20]	GRIFFIN [22]	HDFI [24]	NILE [29]	**FIXER**
Control flow hijacking protection	✓	✓	✓	✓	✓	✓	✓	✓	✓
Data flow hijacking protection	`×	✓	×	✓	×	×	✓	×	×
Maintains high-performance	×	×	×	×	✓	×	✓	✓	✓
Low energy overhead	×	×	×	×	×	×	✓	✓	✓
No architecture modifications	✓	✓	✓	×	×	✓	×	✓	✓
No source code pre-processing	✓	✓	✓	✓	✓	✓	×	×	×
No compiler modifications	×	✓	×	×	×	✓	×	×	✓
Software flexibility	✓	✓	✓	✓	✓	✓	✓	✓	✓
Hardware flexibility	×	×	×	×	×	×	×	✓	✓
Dynamic patching	✓	✓	✓	×	×	×	×	×	✓

The paper is organized as follows: Section II provides an overview of the RocketChip and the Rocket Custom Coprocessor architecture. Section III describes the FIXER design flow and implementation. Experimental results are presented in Section IV. Security implications are discussed in Section V and conclusions are drawn in Section VI.

II. OVERVIEW OF THE ROCKETCHIP ARCHITECTURE

FIXER architecture is based on Rocket Chip [31] (written in CHISEL [32]), an open source parameterized system-on-chip (SoC) design generator. We use the RocketChip generator to generate synthesizable RTL for the standard Rocket Core SoC, a six-stage single-issue in-order pipeline processor that executes the 64-bit scalar RISC-V ISA (Fig. 2(a)). The Rocket Tile consists of the scalar core, the L1 instruction and data caches, and the Rocket Custom Coprocessor (RoCC). The RoCC acts as a user customizable accelerator for the core and can be triggered by a set of custom instructions capable of communicating between the core and the RoCC over the RoCCIO interface.

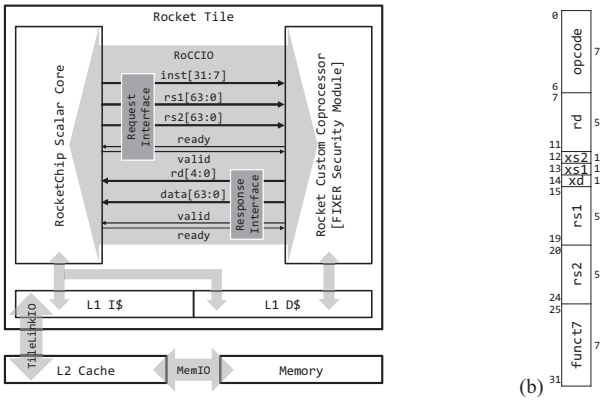


Fig. 2. (a) RocketChip architecture. FIXER coprocessor is also shown, (b) RoCC instruction encoding.

TABLE II. RoCC Instruction Opcodes

RoCC Instruction	Opcode
custom0	0001011
custom1	0101011
custom2	1011011
custom3	1111011

RoCC Instructions: In general, 32-bit RoCC instructions extend the RISC-V ISA and are encoded as shown in Fig. 2(b). The four custom instructions supported by Rocket Chip is shown in Table II. The *xs1*, *xs2*, and *xd* bits control read and write of the core registers by the RoCC instruction. If *xs1* is 1, then the 64-bit value in the integer register specified by rs1 is passed to the RoCC. If the *xs1* bit is clear, no value is passed over the RoCCIO interface. Similarly, *xs2* bit controls the read of register specified by *rs2*. If the *xd* bit is 1 and *rd* is not 0, the core will wait for a value to be returned by the coprocessor over the RoCCIO after issuing the instruction to the coprocessor. The value is then written to the register specified by *rd*. If the *xd* is 0 or *rd* is 0, the core will not wait for a value from RoCC. The *opcode* field specifies the custom instruction for the RoCC, and the *funct7* field further specifies a user-defined function implemented in the RoCC. The RoCC is responsible for signaling illegal instructions to the core.

RoCCIO Interface: The RoCC interacts with the Rocket core and the shared memory system via the standard RoCCIO interface (Fig. 2(a)). The core initiates a coprocessor command by passing the RoCC instruction directly to the coprocessor via *inst*, as well as the relevant register values via *rs1* and *rs2*. If the instruction supplied to the RoCC set the *xd* bit, then the RoCC must eventually supply a response value over the RoCC response interface via *data*.

III. FIXER Security Architecture

A. FIXER Design for Backward-Edge CFI

The first security primitive implemented in FIXER to prevent a memory corruption vulnerability is a Shadow Stack. C programs compiled with the GNU GCC Toolchain for RISC-V target architecture do not provide any protection against memory corruption vulnerabilities such as, buffer overflow. An adversary can provide malicious inputs to a program and is capable of overwriting the return address of a function and redirecting the control flow of the program. The Shadow Stack security primitive can enforce CFI at the backward edge (return to functions). The RoCC is used to implement the Shadow Stack, thus preventing the need to modify the core system architecture. The Shadow Stack is designed as a hardware memory on the RoCC. Fig. 3 shows the steps for detecting CFI violation using a Shadow Stack. The return address is pushed on the system stack by default when a function call is made in the program. During this time, same return address is sent using a RoCC custom instruction to the RoCC to push it on the Shadow Stack as a backup. The return address is popped from the system stack to the instruction pointer register for execution when returning from a function. During this return the RoCC Shadow Stack is queried to retrieve the backup return address and compare against the one from the system stack. If they match, the program proceeds with normal execution, else a potential memory corruption is detected and program execution is stopped. Note that compared to HAFIX [20] where Shadow Stack is part of core, FIXER implements it in the coprocessor leaving the core architecture untouched. It is to be noted that FIXER is complementary to existing DEP protection, since the FIXER instructions must be tamperproof to ensure protection.

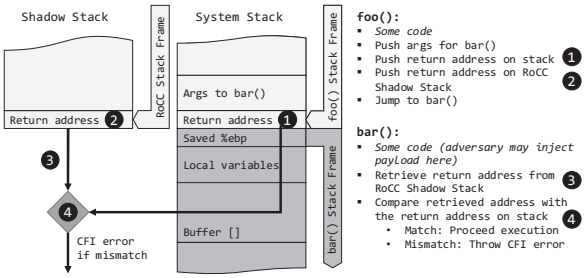


Fig. 3. CFI violation detection using a Shadow Stack.

Fig. 4(a) details the software design flow for FIXER. The source code is first marked with CFI tags (for saving to shadow stack and validation) and compiled to an intermediate assembly code using the RISC-V GNU toolchain. The assembly code is parsed by expanding the tags and injecting the required RoCC instructions in the assembly. The lifted assembly code is generated using a custom parsing script or a compiler pass and then assembled and linked to produce the fully compiled RISC-V binary. These steps are further elaborated in Section II.B.

Fig. 4(b) shows the hardware design flow for FIXER (coded in CHISEL [32] as a RoCC). The hardware implementation of FIXER in RoCC is described in Section II.C. The relevant configuration files for RoCC targeting the FPGA platform are also written. The RocketChip with the RoCC is then compiled with the RocketChip Generator to output the synthesizable Verilog code, from which the FPGA bitstream is compiled. The required RISC-V Linux system image, the FPGA devicetree and the generated bitstream is then deployed to the FPGA to run the RocketChip system. This FIXER assisted RocketChip system can successfully protect against CFI violations on the RISC-V programs compiled with FIXER assisted compilation process.

B. RISC-V Software Design with FIXER

Any program that needs to be backward-edge CFI enforced, is compiled and processed by the following steps:

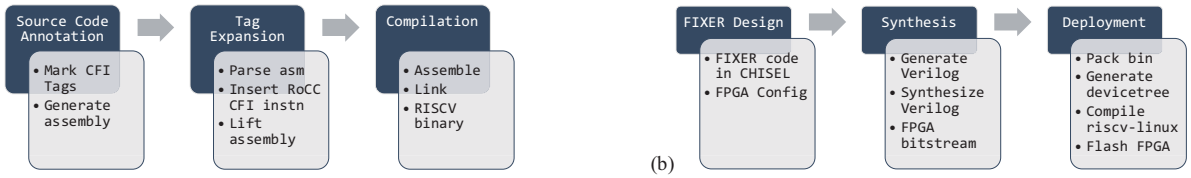


Fig. 4. FIXER design flow in (a) software and (b) hardware.

Step 1 - Source code annotation: We annotate the function calls and returns with a special tag to indicate the sites where the enforcement needs to take place. We use `CFI_CALL` tag before a function call and a corresponding `CFI_RET` tag just before a return from the called function, as shown in Fig. 5.

Step 2 – Tag expansion: We expand the CFI tags to actual RISC-V assembly instructions. During compilation, we intercept the intermediate assembly code of the program and inject the RoCC custom instructions to communicate with the RoCC. Fig. 6 shows the assembly instructions corresponding to `CFI_CALL` and `CFI_RET`, that are placed just before the `call` and `jr ra` (return) instructions respectively.

For `CFI_CALL`, we first retrieve the current value of the program counter from the instruction pointer register using the `auipc` instruction and add 14 bytes offset (instructions are variable length) to calculate the target return address. We save the computed return address in a temporary register *t0*. Then we craft the RoCC instruction `cfi_call` to push the return address from *t0* to the Shadow Stack. A generic 32-bit RoCC instruction extends the RISC-V ISA and is encoded in the format as shown in Fig. 3. There are four RoCC instructions available (*custom0-3*) that are identified by the 7-bit opcode field, as shown in Table I. The *funct7* field can be used to further specify a particular function of the RoCC instruction. We use *custom0* to implement the CFI instructions. We set the *funct7* field to b'0000000 (0) for `cfi_call` and to b'0000001 (1) for `cfi_ret`. We use the *rs1* field to set it to use the t0 register (b'00101), where we temporarily stored the computed return address and set the corresponding *xs1* bit to 1. The final crafted instruction word for `cfi_call` is represented by 0x0002a00b.

For `CFI_RET`, we set the *funct7* field to b'0000001 (1) and set the *rd* field to use the *t0* temporary register (b'00101) along with *xd* bit as 1. The final crafted instruction word for `cfi_ret` is represented by 0x0200428b. During a return from a function, the saved return address is popped from the system stack on to the link register *ra*. We then use the `cfi_ret` custom instruction to retrieve the backup return address from the RoCC Shadow Shack on to the temporary register *t0*. The value in *t0* is then compared against the value in the register *ra* using the `bne` instruction. If they match, the execution proceeds by completing the return (`jr ra`: jump register), else we throw a CFI error.

```
void main () {
    ...
    CFI_CALL
    myFunc();
    ...
}
```

```
void myFunc() {
    ...
    CFI_RET
    return;
}
```

Fig. 5. Source code annotation

```
# CFI_CALL
auipc   t0,0
add     t0,t0,14
.word   0x0002a00b
call    myFunc
```

```
# CFI_RET
.word   0x0200428b
bne     t0,ra,_cfi_error
jr      ra
```

Fig. 6. Tag expansion

Step 3 – Compilation: The final CFI enforced assembly code is passed to the compiler to assemble, link and generate the final executable binary of the program. No compiler modifications are necessary to embed the instructions in the final binary since we provided the custom instruction as a binary instruction word, and the RoCC instruction format is already supported by the GNU toolchain.

C. FIXER Hardware Implementation in RoCC

Fig. 7 shows the FIXER implementation in the RoCC. The program binary runs on the Rocket Core and sends RoCC instructions over the RoCCIO whenever a security validation is required. The RoCC instruction is first passed through the Cmd decoder, which extracts the individual fields of the RoCC instruction, and the contents of the two registers *rs1* and *rs2* if specified. The opcode field is decoded to the *custom0* instruction in our implementation. The *funct7* field is decoded to interpret a `cfi_call` or a `cfi_ret`.

For `cfi_call`, the contents of core register *t0* (the return address) is sent through the *rs1[63:0]* field of the RoCCIO interface. The shadow stack is implemented as a SRAM memory with 64-bit wide words. A top-of-stack register (ToS) holds the address of the top of the shadow stack. If a `cfi_call` is interpreted, the content of the ToS register is incremented by 1. The updated value in the ToS register is used to decode the write address for the shadow stack. The value in the *rs1* field is written to this address on the shadow stack. This operation is non-blocking, so the core can continue execution after issuing the `cfi_call` instruction. There is a command queue at the RoCCIO interface to prevent race conditions. If the instruction function is interpreted as `cfi_ret`, then the ToS register is read to obtain the address for the shadow stack. This address is used to read the saved return address from the shadow stack memory. The value is then sent back to the core

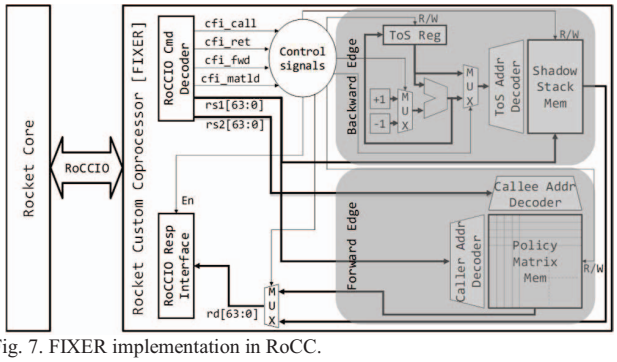


Fig. 7. FIXER implementation in RoCC.

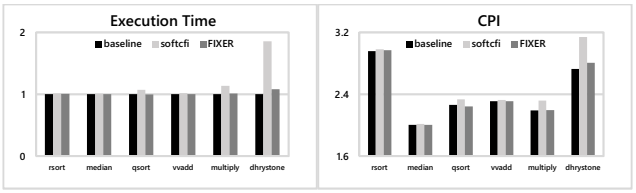
by writing to the *rd[63:0]* field of the response interface of the RoCCIO, which writes the value to the *t0* register on the core as indicated by the RoCC instruction. Our proof-of-concept implementation of the shadow stack can accommodate 1000 addresses. However, this can be updated on demand by simply reconfiguring the FIXER module on the FPGA, a benefit exclusive to our implementation. The size of the shadow stack will be limited by the memory available on the target FPGA.

D. Forward-edge Protection with FIXER

A shadow stack only protects control flow on return boundaries. However, programs often use function pointers to jump to multiple function addresses. To ensure the validity of such function calls using function pointers, a pre-computed call policy is enforced. A static or runtime analysis is performed on the program to construct a control flow graph (CFG). The CFG is represented as a policy matrix that indicates the valid call targets for each function call made using a function pointer. The policy matrix is loaded in memory and at runtime, it is queried to validate the call target for every indirect function call. This forward-edge protection is implemented as another FIXER security module (Fig. 7). The policy matrix memory is created in the RoCC along with peripheral caller and callee address decoders. Our proof-of-concept implementation has 64 rows (each represents an originating call site address) in the matrix and each row holds a 64-bit policy vector (each bit represents a call target address). A set (unset) bit indicates that the call is valid (invalid) for that (caller, callee) pair. A RoCC instruction `cfi_matld` is used to load the policy bitmap into the FIXER module prior to the program execution. A RoCC instruction `cfi_fwd` is inserted before every indirect function call in the source code. The `cfi_fwd` instruction sends the caller and the dereferenced function pointer (callee) addresses to the RoCC for validation. The forward-edge FIXER module then validates the action using the policy matrix and sends back a 1 or 0 indicating allow or disallow respectively. Similar to the shadow stack implementation, the policy matrix size can also be updated post-deployment by reconfiguring the FPGA.

IV. Experimental Results

We implemented FIXER on a Xilinx Zynq FPGA. The hardware architecture of the security module is coded in CHISEL in the RocketChip Generator. The high level CHISEL code is translated to synthesizable Verilog code using the available tools in the RocketChip Generator. We prepared a FPGA system image using the generated Verilog and ran the system on a Zybo board. A sample program is written with 1 billion iterations of function calls and returns. One version of the code implemented a simple software version of the shadow stack (*softcfi*). The software shadow stack is created as a regular stack in the address space. During function calls, the return address is simultaneously placed on the system stack as well as the shadow stack. Another version instrumented the code with the proposed RoCC CFI instructions (*FIXER*). We compiled the *baseline* (base code with no CFI checks), the *softcfi* and *FIXER* versions using the RISC-V GNU GCC compiler. The three versions of the program were run on the RocketChip system running on the FPGA. The base code takes 19 seconds to execute, whereas the software enforced CFI code takes 74 seconds. FIXER takes 29 seconds resulting in ~1.5X overhead over the base code and ~2.55X lower overhead compared to the pure software enforcement. The FPGA on idle draws 370mA current, while on load (with the program running) draws 420mA current, resulting in 1.13X increase. The corresponding energy overhead is 3.89X for the pure software enforced CFI and only 1.53X for the FIXER (60.52% improvement). The FIXER RoCC module incurs only 2.9% area overhead over the vanilla RocketChip without RoCC.



(a) (b)

Fig. 8. RISC-V benchmark evaluation for backward-edge protection w.r.t. (a) execution time (number of cycles), and (b) effective CPI.

TABLE III. Benchmark Instruction Overheads

	Backward-edge	Improvement over *softcfi*
rsort	1.000019X	0.0126%
median	1.000305X	0.2310%
qsort	1.00434X	3.1770%
vvadd	1.000622X	0.5080%
multiply	1.008037X	5.7140%
dhrystone	1.068607X	32.7930%

We evaluated FIXER by enforcing it on the set of benchmarks provided for testing RISC-V architecture. The benchmarks are modified to create three versions for performance comparison: (i) the *baseline* with no CFI enforcement, (ii) the *softcfi* with the software based CFI enforcement, and (iii) the *FIXER* with RoCC based CFI protection. We ensured that the benchmark code remains the same across all the three versions except the CFI enforcement code. We compiled the benchmarks with the RISC-V GNU toolchain without any compiler optimizations and ran the compiled binaries on the Zybo FPGA board. Fig. 9 show the evaluation results backward-edge FIXER. The corresponding instruction overheads are shown in Table IV. With the backward-edge protection, the execution time overhead with *softcfi* is ~18% on average across the six benchmarks compared to 1.5% with *FIXER*. The *softcfi* increases the CPI (cycles per instruction) by 4.6% over the *baseline*, while the *FIXER* increases the CPI by only 0.5%. With the forward-edge protection, the execution time overhead with *softcfi* is ~2% on average across the six benchmarks compared to 0.61% with *FIXER* and CPI reduces 0.4% on average, which is negligible.

V. Security Implications

Performance vs. Security: FIXER is targeted for hybrid architectures, e.g., CPU+FPGA, or ASIC+FPGA. Our current results are based on both the RocketChip and the RoCC accelerator being on the FPGA since we do not have access to such architecture. It is true that if the FPGA is off-chip, there could be performance degradation (due to speed gap between CPU and FPGA) if the checking is performed in a synchronous and fine-grained manner. One of the ways to reduce the performance issues is by making the checking asynchronous, by

using interrupts. In such cases the program can continue execution, until the FPGA raises an interrupt to halt the program. However, it cannot be guaranteed that the adversary has not been able to take control of the system before the FPGA detects the attack. When the FPGA is on-chip, e.g., Intel Xeon with embedded FPGA, the performance overheads can be alleviated due to QuickPath Interconnect (QPI) interface between the core and the FPGA for fast communication.

Security Vulnerabilities and Limitations: FIXER enforces protection for a single process only. For a simultaneous multi-process protection, the FIXER design can be expanded to accommodate multiple shadow stacks and policy memories for different processes. A round-robin scheduler on the FIXER module can assign the shadow stacks and policy memories to each process based on the process ID. The FIXER module on the FPGA also needs to be protected from tampering or data leaks. The current RocketChip implementation allows the entire code containing custom RoCC instructions to be run with supervisor privileges. However, this can be restricted via system calls so that the RoCC instructions are first verified and then run with supervisor privileges. It should be noted that FIXER is still vulnerable to buffer over-reads. Similar to HAFIX and NILE, FIXER will not enforce security if the adversary can modify binary to skip the custom instructions.

Security Guarantees and Benefits: FIXER implemented in the FPGA offers benefits compared to other core based or system level protection schemes. Designs e.g., NILE which use the virtual address space to house the shadow stack, are limited by the size of the address space, and cannot scale based on the branch sequence depth. HAFIX has a separate limited memory on the core to store the CFI tags. However, in case of FIXER, the design can be scaled up or down based on the actual workload of the system. Typically, embedded devices e.g., IoTs have a limited set of workloads, and FIXER module on the FPGA on the IoT's SoC can be scaled appropriately based on the workload. For example, if a new workload is being introduced to the system, which requires a larger shadow stack, the FPGA can be reconfigured to accommodate that (the maximum size being limited by available LUTs).

VI. Conclusions

We proposed FIXER, a CFI security architecture to implement a shadow stack and a policy memory in the RISC-V coprocessor for uninterrupted program flow without modifying or instrumenting the existing binary layout. FIXER implemented on FPGA can enable dynamic reconfiguration to allow flexible on-demand resizing of the shadow stack and policy memory, and also to adapt to new security threats. FIXER exhibits small energy footprint and significant performance gain over traditional software shadow stack.

Acknowledgment

This work is supported by Semiconductor Research Corporation (SRC) [2727.001], National Science Foundation (NSF) [CNS-1722557, CNS-1801534, CCF-1718474, DGE-1723687 and DGE-1821766] and DARPA Young Faculty Award [D15AP00089].

References

[1] Cowan et al. "StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks." In SSYM, 1998.

[2] Data Execution Prevention, https://msdn.microsoft.com/en-us/library/windows/desktop/aa366553(v=vs.85).aspx

[3] Team, PaX. "PaX address space layout randomization (ASLR), 2003." URL: https://pax. grsecurity. net/docs/aslr.txt

[4] Abadi et al. "Control-flow integrity." In Proc. ACM CCS, 2005.

[5] Park et al. "Microarchitectural Protection Against Stack-Based Buffer Overflow Attacks." *IEEE Micro*, 2006.

[6] Nishiyama et al. "SecureC: control-flow protection against general buffer overflow attack," C*OMPSAC*, 2005.

[7] Sinnadurai et al. "Transparent runtime shadow stack: Protection against malicious return address modifications," 2008.

[8] Zeitouni et al. "ATRIUM: runtime attestation resilient under memory attacks." ICCAD 2017.

[9] Iwainsky et al. "Compiler Supported Sampling through Minimalistic Instrumentation," ICPPW, 2014.

[10] Pappas et al. "Transparent ROP exploit mitigation using indirect branch tracing." In USENIX SEC, 2013.

[11] Cheng et al. "ROPecker: A Generic and Practical Approach For Defending Against ROP Attack." NDSS Symposium 2014.

[12] Alves et al. "TrustZone: Integrated hardware and software security." ARM white paper, 2004.

[13] McKeen et al. "Innovative instructions and software model for isolated execution." In HASP@ ISCA, 2013.

[14] Intel: Control-Flow Enforcement Technology Review, 2016.

[15] Ramakesavan et al. "Intel memory protection extensions (intel mpx) enabling guide," 2015.

[16] Yoo et al. "Performance evaluation of Intel® transactional synchronization extensions for high-performance computing." SC-Intl Conf for HPC, Networking, Storage and Analysis. 2013.

[17] Kasikci et al. "Failure sketching: a technique for automated root cause diagnosis of in-production failures." In SOSP, 2015.

[18] Dhawan et al. "Architectural support for software-defined metadata processing." SIGARCH Computer Arch News, 2015.

[19] Wang et al. "Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity." In IEEE S&P, 2010.

[20] Davi et al. "HAFIX: hardware-assisted flow integrity extension." in DAC, 2015.

[21] Jin et al. "Hardware control flow integrity." *The Continuing Arms Race*. ACM and Morgan & Claypool, 2018.

[22] Ge et al. "GRIFFIN: Guarding Control Flows Using Intel Processor Trace" In ASPLOS, 2017.

[23] Arias et al. "HA^{2}lloc: Hardware-Assisted Secure Allocator." in HASP, 2017.

[24] Song *et al*., "HDFI: Hardware-Assisted Data-Flow Isolation," *IEEE Symposium on Security and Privacy (SP), 2016.*

[25] Bresch et al. "A red team blue team approach towards a secure processor design with hardware shadow stack," *IVSW*, 2017.

[26] Bresch et al. "Stack Redundancy to Thwart Return Oriented Programming in Embedded Systems," in *IEEE ESL*, 2018.

[27] Panis et al. "Scaleable shadow stack for a configurable DSP concept," in *IWSOC, 2003.*

[28] Ming et al. "Shadow Stack Scratch-Pad-Memory for Low Power SoC," in *IEEE Intl Symposium on Embedded Computing*, 2008.

[29] Delshadtehrani et al. "Nile: A Programmable Monitoring Coprocessor," in *IEEE Computer Architecture Letters*, 2018.

[30] www.extremetech.com/extreme/184828-intel-unveils-new-xeon-chip-with-integrated-fpga-touts-20x-performance-boost

[31] Asanovic et. al, "The Rocket Chip Generator", technical report, www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[32] Bachrach *et al*., "Chisel: Constructing hardware in a Scala embedded language," In *DAC, 2012*.