DeTRAP: RISC-V Return Address Protection With Debug Triggers

Isaac Richter
Department of Electrical and Computer Engineering
University of Rochester
Rochester, New York, USA
isaac.richter@rochester.edu

Jie Zhou[†]
Department of Computer Science
George Washington University
Washington, DC, USA
jie.zhou@gwu.edu

John Criswell
Department of Computer Science
University of Rochester
Rochester, New York, USA
criswell@cs.rochester.edu

***Abstract*—Modern microcontroller software is often written in C/C++ and suffers from control-flow hijacking vulnerabilities. Previous mitigations suffer from high performance and memory overheads and require either the presence of memory protection hardware or sophisticated program analysis in the compiler.**

This paper presents *DeTRAP* (Debug Trigger Return Address Protection). DeTRAP utilizes a full implementation of the RISC-V debug hardware specification to provide a write-protected shadow stack for return addresses. Unlike previous work, DeTRAP requires no memory protection hardware and only minor changes to the compiler toolchain.

We tested DeTRAP on an FPGA running a 32-bit RISC-V microcontroller core and found average execution time overheads to be between 0.5% and 1.9% on evaluated benchmark suites with code size overheads averaging 7.9% or less.

I. Introduction

Modern microcontroller software is mainly written in C and C++. Unfortunately, these languages are type-unsafe, and programs written therein may have memory safety vulnerabilities that can be exploited by control-flow hijacking attacks [57]. While restricting control flow by enforcing Control-flow Integrity (CFI) [1] can mitigate these attacks, previous work has shown that advanced control-flow hijacking attacks are still possible if the integrity of function return addresses is not protected [14], [26], [33]. Worse, programs on all mainstream architectures, such as x86, ARM, and RISC-V, all suffer from this problem [21].

Previous work has taken one of two approaches to mitigate these sophisticated attacks. The first approach protects the integrity of function return addresses [13], [69], [75]. However, all of these systems induce execution time and memory overheads. Some [69] utilize hardware memory protection features that are intended for privilege domain separation, such as switching to supervisor mode when manipulating the shadow stack, but otherwise executing in user mode, requiring a context switch on every call to a non-leaf function. The most efficient of these systems is Silhouette [75] which imposes 1.3% performance overhead on CoreMark-Pro [32] and 3.4% performance overhead on BEEBS [52]. Worse, the code size overheads for these same benchmarks are 16.5% and 5.3%, respectively, when the size of untransformed code (namely `libc` and `libm`) are removed from the baseline.

[†]Contributions made while at the University of Rochester

A second approach is to detect corruption of the return address; such detection must account for the path that control flow has taken through the call graph in order to mitigate sophisticated control-flow hijacking attacks. μRAI [4] is such a system and has comparable performance overhead to Silhouette, averaging 2.6%[1], but has code size overheads of 54.1%. μRAI also requires computing a complete call graph at compile time, which requires sophisticated whole-program analysis [44]. These overheads hinder adoption in microcontrollers. Additional overheads can force manufacturers to choose between higher security at the cost of utilizing more expensive hardware with faster processors and more memory or greater cost-efficiency at the cost of security.

Additionally, we seek a solution that requires no new hardware features. New hardware features must be thoroughly tested by manufacturers and ratified by standards bodies before they are implemented and deployed. Since new hardware support requires "buy-in" from multiple entities, a solution that uses features already approved by manufacturers and standards bodies is more likely to gain adoption.

Modern processors, such as ARM [6], [8] and RISC-V [56], provide sophisticated processor watchpoint features that can generate a debug watchpoint trap when certain conditions, configured by software, occur. Unlike earlier processors, these new debugging facilities can generate a watchpoint trap when the program counter or a load or store address is within an arbitrary range, or when the processor is executing a particular instruction. Furthermore, conditions can be chained together so that a trap occurs only when multiple conditions are met. While previous work [60] has employed these hardware features to implement execute-only memory, we observe that we can use these features to implement more dynamic security policies, such as write-protected shadow stacks, which must distinguish stores that save return addresses from other stores within a program.

In this paper, we leverage these debugging features to build *Debug Trigger Return Address Protection (DeTRAP)*: a system that combines a novel compilation strategy with modern processor debug facilities to provide an efficient write-

[1]μRAI's overhead is 2.6% when a compiler transformation that serendipitously improves performance is also applied to the baseline against which μRAI is compared.

protected shadow stack. Unlike prior work [27], [75], DeTRAP has minimal hardware requirements: it requires no memory protection, address translation, or privilege mode hardware. Furthermore, DeTRAP only uses functionality *already specified* in the RISC-V ISA [56], [72], [73]; no new hardware needs to pass through standards committees. Finally, DeTRAP does not need sophisticated whole-program call graph analysis.

We prototyped DeTRAP by enhancing the RISC-V Rocket Core [10] to fully implement the complete RISC-V debugging facilities [56] (adding just 0.87% to the core pipeline) and by enhancing the LLVM compiler [43] to implement a write-protected shadow stack using these debugging features. Our experimental results show that DeTRAP outperforms previous work such as Silhouette [75] and μRAI [4]: DeTRAP incurs execution time overhead of just 0.5% averaged across the benchmarks in CoreMark-Pro and 0.8% for the BEEBS benchmarks evaluated by Silhouette [75], an improvement of 0.5% and 8.5% respectively. Our results also show that DeTRAP incurs a code size overhead of 7.9% and 6.7%, respectively. On CoreMark, DeTRAP's execution time overhead is 1.9%, an improvement of 5.7% against μRAI [4]; and it has a flash size *decrease* of 2.7%, a ∼40% improvement. We further evaluated against Embench [53] and found performance overhead of 1.4% with a code size *decrease* of 3.5%.

We evaluated a modification to the rocket core pipeline to implement the parts of the RISC-V ISA [56] needed for DeTRAP, and found that it can be done using just 0.14% additional cell area (pre-routing) in core pipeline.

To summarize, the main contributions of this paper are:

- The design of the first system that uses modern processor debugging facilities to implement efficient write-protected shadow stacks
- A return address integrity system that can support unmodified untrusted leaf functions in precompiled code and handwritten assembly
- A DeTRAP prototype that implements our design on the RISC-V Rocket Core [10]
- An evaluation of the hardware changes to Rocket Core that would be needed to support DeTRAP.
- An evaluation of DeTRAP's performance and code size overheads, showing that DeTRAP provides the same protection as previous work with less performance and memory overhead. Unlike previous work [4], [75], our evaluation methodology removes serendipitous code layout changes as the source for improved performance in our evaluation results.

II. Background on RISC-V Debug Triggers

The RISC-V architecture [56] provides a rich set of primitives for specifying the conditions under which the processor should trigger a breakpoint exception. Breakpoints can be configured to fire prior to entering a trap handler, after a configurable number of instructions has been executed, or based on a comparison against a program counter, load/store address, instruction opcode, and/or data value loaded from or stored to memory.

Breakpoint comparisons are not limited to equality checking; comparisons can also be configured to trigger if a value is less than, greater than or equal to, or unequal to another value [56]. It is also possible to define a **bitmask match**, where selected mask bits of an input value are checked against the same bits of a stored pattern. This functionality can be used to raise an exception if a specific instruction is to be executed regardless of the registers encoded in the opcode.

What makes RISC-V breakpoints particularly powerful is that multiple debug triggers can be chained together so that the processor only traps if all conditions in the chain are met [56]. This feature permits trapping on conditions that are more complex than can be described by a single comparison. For example, trapping when executing code within an arbitrary region can be done by using two triggers, one each for the region's lower and upper bounds. The Debug ISA also allows chained triggers to mix and match what is being compared. For example, a data value trigger can be chained with a store address trigger, trapping when the code attempts to write a specific value to a specific memory location.

Together, these features provide efficient conditional breakpoints, alleviating the need to check for conditions in the exception handler. Furthermore, because debug triggers operate in parallel with execution, they perform checks without any per-check performance penalty.

To balance functionality with performance and cost, current implementations generally only include a few triggers: SiFive's FU540 [64] and FU740 [65] chips only include two debug triggers per hardware thread (hart). Since triggers are configured per-hart, this substantially limits their usability, as applying a policy across all harts requires duplicating the configuration across them as well, so all policies targeting these devices must collectively fit into just two triggers.

Moreover, implementations are not required to include all functionality from the specification. For example, SiFive's chips only support matches against the program counter or load/store address and do not support bitmask matches [62], [64], [65]; we know of no implementation that matches against the instruction opcode or loaded/stored data value. Furthermore, due to hardware tradeoffs, the debug specification [56] anticipates that implementors may want to restrict the complexity of supported triggers. Indeed, many implementations [10], [62], [64], [65] limit chaining to just two registers. Supporting longer chains requires additional chip area and could reduce the maximum pipeline clock frequency.

As Section IV-C discusses, DeTRAP's design is intended for single-core microcontrollers, similar to SiFive's FE310, which supports 8 triggers on its single hart [62] and allows up to four two-trigger-chain rules to be defined.

III. Threat Model

Our system protects a single embedded bare-metal application running without an operating system kernel or supervisor. For simplicity, we assume that the application is single-threaded and does not utilize traps to modify control flow or to context switch to other tasks, threads, or processes. As

with many embedded applications and processors, we assume a single address space application running in privileged mode without any hardware memory protection mechanisms. The single application is benign and has no runtime-loadable code, but may have exploitable spatial [57], [68] and temporal [2] memory safety errors that can corrupt control data such as return addresses and function pointers. Non-control data attacks [15] are out of scope.

Physical attacks, such as connecting an external debugger or modifying the data on volatile or non-volatile memories, are also out of scope. The damage from attacks on non-volatile storage can be mitigated through signature checking implemented by trusted boot running from non-reprogrammable mask ROM [47], [49]. We do not mitigate attacks via separately-programmed devices that can autonomously modify system memory, except to the extent that such requests may be initiated after the processor writes to a memory-mapped register of the peripheral (e.g., DMA engines [54]).

IV. Design

DeTRAP provides a write-protected compressed shadow stack which securely stores return addresses. Such a system has **trusted code**, which is permitted to save return addresses to the shadow stack, perform I/O operations, and write to security-critical data needed for DeTRAP's operation; the rest of the code is **untrusted code** that should be unable to write to the shadow stack or security-sensitive registers.

In this section, we first present the requirements that DeTRAP must meet to enforce return address integrity (Section IV-A). We then explain how DeTRAP lays out the address space to minimize the number of debug triggers it needs (Section IV-B) and how it configures those debug registers to prevent untrusted code from modifying the shadow stack or other security-critical data (Section IV-C). Next, we explain how DeTRAP transforms code to implement a write-protected shadow stack (Sections IV-E and IV-F). Finally, we discuss additional precautions that DeTRAP takes to ensure that its protections will operate as intended (Section IV-G–Section IV-H).

A. Security Requirements

Our design employs a compressed shadow stack [18]. Silhouette [75] summarizes three high-level invariants that any shadow stack-based approach for return address integrity must maintain: (1) A return address is stored either in the shadow stack or in a register that is never spilled to memory. (2) The shadow stack and the register for return addresses cannot be corrupted. (3) A function's epilogue always retrieves the return address that is stored by the function's prologue. For DeTRAP, we identify five security requirements that must be met to maintain the three invariants.

First, return addresses are always stored to a trusted location for use by function epilogues. Specifically:

Requirement 1 *Return addresses used for control flow can only be stored in a dedicated CPU register and the shadow stack, and the writes of return addresses can only occur in a function's prologue.* (Sections IV-D and IV-H)

Second, a write-protected shadow stack provides no protection unless a function's epilogue reads the return address from the correct location within the shadow stack. DeTRAP uses a shadow stack pointer (SSP) which reads/writes return addresses from/to the shadow stack. The SSP is stored in a dedicated hardware register. However, this design could inadvertently access an incorrect return address or an arbitrary value from outside the write-protected shadow stack if the SSP points to the wrong location. This leads to

Requirement 2 *Return addresses must always be retrieved from an uncorrupted CPU register or via an uncorrupted shadow stack pointer.* (Sections IV-D, IV-E and IV-G)

Additionally, as DeTRAP protects the shadow stack using the processor's debugging facilities, the following requirement must be met:

Requirement 3 *Software cannot reconfigure the processor's debug registers to modify its trap conditions.* (Section IV-H)

Furthermore, we need to ensure that forward control-flow transfers (e.g. calls via function pointers) can only target predetermined destinations. Not only does this mitigate various control-flow hijacking attacks, e.g., return-to-libc attacks [68], but it also permits DeTRAP to use code scanning techniques [25], [36], [59] to ensure that untrusted code does not use instructions that our design deems unsafe. Therefore:

Requirement 4 *Indirect function calls/jumps always branch to the beginning of a function, and intra-function indirect jumps always use a destination address loaded from their precomputed jumptable.* (Section IV-F)

Finally, as previous work has noted [24], [71], restricting control flow is useless if an attacker can modify executable code or CFI metadata. DeTRAP therefore enforces:

Requirement 5 *All executable code and any data used to enforce CFI cannot be corrupted.* (Sections IV-B, IV-C and IV-E)

B. Memory Layout

Binary executables typically have separate sections for code, read-only initialized data (rodata), initialized writable data (data), and uninitialized writable data (bss). DeTRAP must prevent corruption of code and security-critical data. DeTRAP therefore further divides the data sections to distinguish between those whose integrity it must protect, i.e., sections for which corruption would violate a security requirement, and those that can be modified safely by untrusted code.

Thus, we have both **untrusted stack** and **shadow stack** sections, as well as the **untrusted data/bss** and **trusted data/bss** sections. Since the rodata section may include data used for control flow, such as lookup tables for `switch` statements, DeTRAP must also protect its integrity. Most systems include a **memory mapped I/O (MMIO)** area for access to peripherals and configuring the processor. As some peripherals, such as direct memory access (DMA) engines [54], could be used to violate invariants, DeTRAP must also protect the MMIO area from untrusted writes.

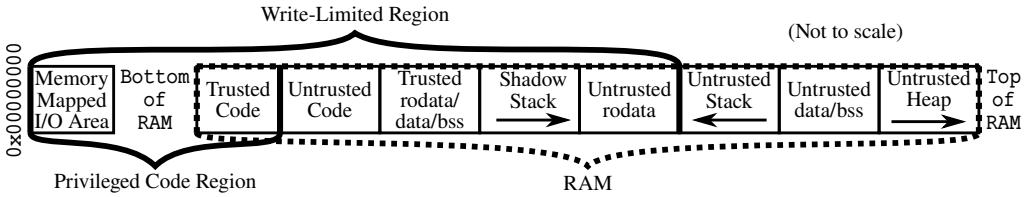

Fig. 1: DeTRAP memory layout for systems without separate code and data memories

Like with data, DeTRAP separates code into two sections: the **trusted code** section and the **untrusted code** section, respectively. DeTRAP prevents untrusted code from corrupting the shadow stack, trusted data/bss, rodata, and both code sections. Thus, each section has two DeTRAP-specific attributes: (1) whether the section must be *write-limited* because corrupting it could violate a requirement, and (2) if the section contains *privileged code*, comprising instructions that are trusted not to violate requirements when writing to memory.

To reduce the number of debug triggers needed, DeTRAP groups sections with identical protection requirements into contiguous regions. We thus define contiguous write-limited and privileged code regions and their implied complements (non-write-limited and unprivileged code), and lay out the sections within their respective regions.

Fig. 1 shows DeTRAP's memory layout. DeTRAP needs to define a trigger chain that detects whether an instruction in the *unprivileged code region* attempts to write into the the *write-limited region*. Since we target a system that only allows up to two triggers to be chained together, we can only use a single trigger to match against each region. We anchor each region at the bottom or top of the address space and use $<$ and $\geq$ trigger matches, respectively.

Anchoring the write-limited region at the beginning of the address space has the added benefit of automatically including the memory mapped I/O area, protecting against DMA attacks [54]. Additionally, by placing the non-write-limited region at the top of the address space and placing the untrusted stack at the bottom of the non-write-limited region, DeTRAP can provide free detection of stack overflow in untrusted code; stack overflows attempt to write to the write-limited region, generating a trap.

DeTRAP also anchors the privileged code region to the bottom of the address space, and the privileged code region includes the memory mapped I/O area. This has the benefit that ROM routines, which are vetted during hardware design, can directly write to MMIO addresses.

While there are no hardware-enforced memory protections against execution outside of code regions, DeTRAP's forward CFI protection (Section IV-F) makes it impossible to transfer control flow outside of the code sections. Hardware memory protection to control which memory sections are executable is therefore unneeded.

C. Debug Triggers

We observe that we can leverage the debug triggers, as discussed in Section II, to enforce a security policy as a chain of trigger conditions. Since software running on the processor can configure the debug hardware, we can exploit the debugging facilities not for debugging but to enforce security policies. As Table I shows, we use three debug triggers in two chains—one chain with two triggers and the other trigger by itself—to implement policies that protect DeTRAP's sensitive data.

TABLE I: Configured Debug Triggers

Write Protection Chain (Two triggers)		
Program Counter	$\geq$	Bottom of Untrusted Code Section
Write Address	$<$	Bottom of Untrusted Stack Section
Shadow Stack Overflow Prevention Chain (One trigger)		
Write Address	$=$	Top of Shadow Stack

To properly protect the write-limited region, the processor must trap when code outside the privileged code region attempts to modify data in the write-limited region. DeTRAP chains two triggers together to enforce this policy. The first trigger matches the program counter greater than the top of the privileged code region, and the second trigger matches memory write addresses below the top of the write-limited region. The overall effect is that if untrusted code attempts to modify the write-limited region, the processor will raise a breakpoint exception. The runtime's trap handler (Section IV-E) will detect that exception and take corrective action.

Additionally, as discussed below in Section IV-G, we use a debug trigger to detect attempts by any code to write to the topmost location in the shadow stack, preventing the imminent overflow thereof.

D. Return Address Handling

DeTRAP's compiler generates the function prologue and epilogue code that saves and restores return addresses. For leaf functions, i.e., functions that do not call other functions, the prologue saves the return address in an ABI-defined register that is reserved for return addresses. Since the compiler generates no other code that writes to this reserved register, the return address remains safe. Non-leaf functions, though,

```
1 .section .text
2 foo:
3   j foo$trampoline
4 foo$postjump:            #post-jump label
5   addi sp, sp, -FRMSIZE #original prologue
```

Listing 1: DeTRAP Function Prologue

must use special prologue and epilogue code to safely save the return address to and restore it from the shadow stack.

Since function prologues write return addresses to the shadow stack, they must utilize code (called a *trampoline*) within the trusted code region. For each non-leaf function, the DeTRAP compiler generates runtime trampolines, located within the trusted code region, that writes the return address to the shadow stack. When a function (e.g., `foo`) needs to save its return address, its prologue (see Listing 1) first calls its associated trampoline (`foo$trampoline`) in the trusted code segment. This trampoline (see Listing 2) saves the return address, which was set by the `call` instruction, to the write-protected shadow stack, and then jumps back to the start of its associated function's original prologue (`foo$postjump`).

To optimize performance, when calling a function with a trampoline, the call is modified to directly branch to the trampoline (e.g., `foo$trampoline`). If a function has no external linkage and all calls to it are replaced by calls to its trampoline code, the compiler can further reduce code size by removing the jump to the trampoline (Listing 1 lines 2-3) from the function's prologue.

For non-leaf functions, DeTRAP inserts code into the function epilogue to restore the return address from the shadow stack (see Listing 3). Unlike function prologues, epilogues need no trampoline in the trusted code region because they do not write to the shadow stack. DeTRAP's code generator, CFI (Section IV-F), and code scanner (Section IV-H) ensure that the shadow stack pointer is never corrupted, guaranteeing that the epilogue always loads the correct return address from the write-protected shadow stack.

E. Trap Handling

If unprivileged code attempts to write to the write-limited region or the shadow stack overflows, the debug triggers described in Section IV-C will cause a trap. To ensure that this trap is handled properly by trusted code, DeTRAP therefore performs initial handling of all traps. If the trap is an exception within trusted code, or was caused by an attempt to violate DeTRAP's protections, the handler will terminate execution. However, not all traps are caused by violations of DeTRAP's security requirements, such as timer interrupts. In these cases, DeTRAP's handler will create a trap frame and invoke the application's (untrusted) handler.

To protect against corruption of potentially sensitive processor state, the trap frame is always saved to the shadow stack. When handling interrupts, this ensures that the untrusted handler cannot potentially subvert sensitive operations in-progress (e.g., return address handling code within prologues or epilogues). The untrusted handler can still modify data outside the write-limited region to, for example, set a flag or copy data into or out of an I/O buffer.

For exceptions in unprivileged code, the trap frame is copied onto the untrusted stack for the application's handler to modify, with some limitations. The shadow stack pointer and return address registers are always restored from the trap frame on the shadow stack, ensuring that they cannot be corrupted. The untrusted handler can modify the program counter only to increment it to the next instruction, for example when emulating a floating-point instruction on a system without a floating-point unit. Any other modification of the PC could violate control-flow, and is prohibited. If the untrusted handler attempts a prohibited modification, this is treated like any other violation, and execution is terminated.

DeTRAP includes a code scanner (see Section IV-H), which verifies that untrusted code, including the application's trap handler, does not include the trap return (`mret`) instruction. Because it is not safe for the untrusted handler to modify the shadow stack pointer, the application cannot use the trap handler to transfer control between threads, as in an interrupt-driven scheduler. If it were necessary to enable context switching, methodology similar to that of Kage [29] could be applied—the trusted code could save per-task state in a per-task shadow stack within the write-limited region and switch between them on-demand.

F. Forward-edge Control Flow Integrity

DeTRAP's debugger triggers enforce shadow stack integrity (Section IV-C), and DeTRAP ensures that trap handlers cannot corrupt the SSP (Section IV-E). However, to completely prevent the exploitation or misuse of the SSP, DeTRAP also must ensure that forward-edge control flow cannot transfer to the middle of function prologues and epilogues. Specifically, inter-function forward branches must jump to the first instruction of a function's prologue, and intra-function branches must jump to a valid location within the function. Additionally, on processors supporting variable-length instructions, such as RISC-V's "C" Compressed Instruction Extension [72], DeTRAP must ensure that branches jump to the beginning of an intended instruction, as there might be a coincidental and valid sequence of instructions that is an offset from the intended instructions that could subvert DeTRAP's security.

For intra-function control-flow integrity, specifically, indirect jumps from `switch` statements, LLVM—the compiler upon which DeTRAP is based—compiles them to use

```
1 .section .trusted.text
2 foo$trampoline:
3   sw   ra,  0(x3)
4   addi x3, x3, 4
5   j    foo$postjump
```

Listing 2: Return Address Save Trampoline

```
1 # Restore original stack pointer
2 addi sp, sp, FRMSIZE
3 # Load return address from shadow stack
4 lw   ra,  -4(x3)
5 # Decrement shadow stack pointer
6 addi x3, x3, -4
7 jr ra    # Original Function Return
```

Listing 3: DeTRAP Function Epilogue

a bounds-checked jumptable. For indirect function calls, DeTRAP uses LLVM's `icall-cfi` [19], [67], which provides type-based CFI, meeting more than the minimum requirements above (more details in Section V-C). A more fine-grained CFI could provide better protection against forward-edge threats, such as call-oriented [58] and function-reuse [35] attacks, but is unnecessary for reverse-edge protection.

Forward-CFI can prevent mismatches between function prologues and epilogues for most programs. However, misuses of `setjmp/longjmp` may disrupt the balance. Since `setjmp/longjmp` are infrequently used in programs for embedded systems, we provide our design for handling them in our technical report [55].

G. Shadow Stack Overflow and Underflow

DeTRAP uses a shadow stack pointer to read/write return addresses from/to the shadow stack, and keep it from being corrupted. Although the DeTRAP code scanner (Section IV-H), ensures that the SSP is only modified in trusted code and function epilogues (Section IV-D), it is also necessary to ensure that the SSP cannot underflow or overflow. While it would be possible to add bounds checks for overflow after each increment, we instead use an additional debug trigger that matches on writes to the last entry of the shadow stack. Because writes to the shadow stack are strictly incremental, this is sufficient to detect an overflow without any runtime penalties. Underflow would imply either corruption of the shadow stack pointer, which is checked for by the code scanner, or a violation of proper control flow (such as illegal execution of function prologue/epilogue code), which is handled by DeTRAP's CFI (Section IV-F). This also allows DeTRAP to avoid bounds checks for underflow.

H. Code Scanning

After compiling and linking a program, DeTRAP runs a code scanner on the generated executable and warns about any vulnerabilities the code scanner discovers. This code scanner provides two critical services. First, as all code runs in the processor's privileged mode, the code scanner ensures that the program does not use privileged instructions to bypass DeTRAP's protections. Second, the code scanner ensures that *all* native code (code generated by the DeTRAP compiler, assembly code written by hand, and code generated by other compilers) does not break DeTRAP's security guarantees.

1) External Code: External precompiled code and hand-written assembly must either use DeTRAP's return address handling (Section IV-D), handwritten or generated via DeTRAP's compiler, or consist only of functions that keep the return address in the `ra` register (e.g.: leaf functions). Otherwise, the code scanner will detect unsafe loading of the return address. It is the user's responsibility to ensure that any linked external code that is in the trusted code section does not violate DeTRAP's requirements.

2) Configuration Protection: Untrusted code must not modify the debug trigger or trap handler configurations as doing so could nullify DeTRAP's protections. The debug trigger and trap handler configurations are governed on RISC-V by Control and Status Registers (CSRs) [56], [73] configured via the `CSRR*` instructions [72] that perform a read-modify-write operation. The CSR to be modified is encoded as an immediate value embedded in the opcode. The code scanner assumes that any CSR instruction that is not an atomic bit set/clear instruction with a hard-coded zero input will modify its targeted CSR. If the code scanner finds a `CSRR` instruction that modifies CSRs governing debugging and trap handling, it rejects the program; all other CSR modifications are permitted.

3) Call and Return Verification: We designed DeTRAP so that *all* native code loaded on to the system follows DeTRAP's requirements. This includes code compiled by the DeTRAP compiler and *external code* such as hand-written assembly language code and library code compiled by other compilers e.g., a C standard library compiled by GCC. To this end, the DeTRAP code scanner performs the following checks on *all* native code linked into the final binary executable.

First, the code scanner verifies that all indirect branches, including those in assembly and precompiled code, are preceded by the appropriate CFI checks as Section IV-F describes. Second, the code scanner ensures that either the `ra` register has not been modified or that it has been spilled and reloaded from the shadow stack as Section IV-D describes. Additionally, the code scanner verifies that only function epilogue code modifies the shadow stack pointer register and that it does so only by decrementing the register by the correct amount (as shown in Listing 3). Third, the code scanner verifies that only trusted code uses the trap return instruction `mret` [73].

There are some functions that do not follow DeTRAP's conventions but are still safe to use, e.g., indirect jumps in the `memset()` function in `libc`. The code scanner permits a developer who has vetted such jumps to add them to a whitelist with their destinations. This allows the scanner to confirm that the functions otherwise meet DeTRAP's requirements.

V. Implementation

Our implementation is based on a purpose-built runtime and a modified version of Clang and LLVM [43] 15.0.7. We also enhanced the Rocket core [10] RISC-V processor to implement a more recent version of the debug trigger ISA [56] . As our benchmarks do not use it, we did not implement the `setjmp/longjmp` handling.

A. Rocket Core Modifications

The upstream Rocket breakpoint module is based on the 0.13 draft of the RISC-V Debug Support Specification. Implementations are free to support as little of the specification as they want, using write-any-read-legal (WARL) semantics such that a read-back of the configuration register will reflect only what is supported. Unfortunately, Rocket's breakpoint module implementation does not properly support combining both program counter (PC) and memory triggers into the same chain, even though its WARL read-back implies it should. This deviation from the specification is undocumented and

prevented DeTRAP from working properly. Because DeTRAP needs this functionality, we modified the implementation to properly support such a chain.

In the Rocket core pipeline, the breakpoint module effectively has two breakpoint units (BPUs) that eavesdrop on the outputs from each stage of execution. The PC BPU monitors the instruction fetch (IF) stage to determine if a PC trigger should fire; the MEM BPU checks the input of the memory (MEM) stage — the output of the execute (EXE) stage — for matches against memory read or write triggers. For chained triggers, all triggers in the chain must match during the same cycle for an exception to be raised. However, each instruction is executing in only a single pipeline stage at a time. Consequently, in any given cycle, the breakpoint module is examining the behavior of *different* instructions in the PC and MEM BPUs. What DeTRAP needs is to have the MEM stage generate a trap *if* the instruction matched a PC trigger when it was examined by the PC BPU a few cycles earlier.

We fixed this problem by adding pipeline registers to the outputs of the instruction decode (ID) and EXE stages to track whether individual triggers matched the instruction address. These PC-based "pretriggers" then feed back into the breakpoint module alongside the memory stage inputs and are combined with the memory triggers to ensure that mixed chains properly raise exceptions. Our evaluation in Section VII-F shows that this change uses negligible additional area and energy and brings the implementation into compliance with the specification.

B. Shadow Stack Implementation

We modified the Clang/LLVM compiler to implement the write-protected shadow stack described in Section IV-D. Our modification of function prologues and epilogues is based on Clang's ShadowCallStack [20]. Since we built DeTRAP before the RISC-V ABI designated `x3` as a platform register [16], [17] and ShadowCallStack adopted it as the shadow stack pointer register [41], our implementation uses `x18` as the shadow stack pointer register like the original RISC-V ShadowCallStack implementation., unlike the code shown in Listings 2 and 3. Our implementation writes a copy of the return address to both the shadow stack and the original untrusted stack; returns use the write-protected copy from the shadow stack. This implementation allows existing code that reads the return address from the stack, such as `__builtin_return_address`, to function without modification. DeTRAP, also like ShadowCallStack [20], merely uses the return address on the write-protected shadow stack on function return and does not check whether the return address restored from the shadow stack matches the return address on the original untrusted stack.

C. Forward-Edge CFI Implementation

To ease implementation of forward-edge control flow protection, we used Clang/LLVM's existing indirect function call checking `-fsanitize=cfi-icall` [19], [67]. At compile time, this CFI creates jumptable entries for each function that is address-taken, sorted by the function type signature; when taking the address of a function, it then substitutes the address of the jumptable entry instead. Indirect calls are rewritten to verify that the pointer is aligned with and in-bounds of those jumptable entries that match the expected type signature, and then the function is called via the jump table entry. This form of CFI exceeds the minimum requirements for forward-edge control flow identified in Section IV-F. If greater precision on forward-edge control flow is desired, an alternative DeTRAP implementation can use other forward-edge CFI mechanisms (e.g., label-based CFI [1] with a precise call graph).

D. `nospill` Attribute for CFI-Sensitive Data

CFI sensitive data is data that is used to check the destinations of indirect branches, including constant values used in CFI run-time checks and `switch` statement jump calculations, and the values of validated pointers. Previous work has noted that LLVM's forward-edge CFI implementation [19] may spill CFI sensitive data to the stack [22], [45], making the this data vulnerable to memory safety attacks. DeTRAP adds a new `nospill` attribute to LLVM IR, which will prevent a virtual register from being spilled to the stack. We discuss the details of `nospill` in our technical report [55].

E. Code Scanner

We implemented the DeTRAP code scanner, discussed in Section IV-H, using LLVM's MC disassembler library. The scanner first identifies all reachable code by examining the symbol table for all functions, including forward-edge CFI jumptable entries (see Section V-C) and `switch` jumptable destination pointers. It also reads the section headers to be able to distinguish between trusted and untrusted code.

Next, inspired by the static analyzer of Jalyoan, et. al. [38], the scanner traces all possible execution paths, generating a directed graph of basic blocks. The input value for return instructions (`jr ra`) is checked to ensure that it is either unmodified since the preceding `call`, or was loaded from the shadow stack.

To handle connecting basic blocks that are linked via indirect branches, the scanner checks the instructions leading up to the jump to ensure that the destination is either statically known (e.g., a long jump, which first requires an `auipc` [add upper immediate to PC] instruction before the `jr` [jump to offset from register] jump) or is loaded from (`switch`) or checked against (indirect call) a jumptable, and the results are used when connecting the basic blocks. Whitelists were added for handwritten assembly that performs safe indirect jumps that do not rely on jumptables (e.g., newlib's RISC-V `memset`, which includes jumps whose offset is directly computed, rather than being loaded from a table).

While tracing the discovered instructions in untrusted code, the scanner checks for corruption of the shadow stack pointer, and other dangerous instructions (see Section IV-H).

F. Runtime

Applications are linked to a custom runtime that contains trusted code, including startup code necessary to implement

TABLE II: Generated System-on-Chip Configuration

Core	rv32imafdc at 50 MHz
Branch Target Buffer	28-entry
Branch History Table	512-entry
Return Address Stack	6-entry
Breakpoints	8, address match only
Phys Mem Protection	8 regions, 4 byte granularity
Cache line	64 bytes
L1 Data	64 KiB, 4-way
L1 Code	16 KiB, 2-way
L2 (Shared Inclusive)	256 KiB, 8-way 5 MSHRs
On-board DDR3	256 MiB ×16 at 333 MHz CL5

DeTRAP. Standard library support is provided by the RISC-V newlib port [51] based on revision `83d4bf`, with compiler support routines from compiler-rt 15.0.7. The runtime also includes code for tracking and reporting the outputs of performance counters via the serial port.

VI. Security Benefit

To examine DeTRAP's security, we examined its effectiveness against RiscyROP [21]: the most sophisticated attack against RISC-V of which we know. RiscyROP found that, compared to x86 and ARM32, it is more challenging to find useful gadgets on RISC-V, due to multiple factors such as differences in calling conventions. However, it also found that one can launch powerful code-reuse attacks to call arbitrary functions with attacker-controlled arguments. RiscyROP mainly exploits two types of gadgets: (1) those that load a return address from the stack and return to that address (called stack-based jump), and (2) those that jump to an attacker-controlled register (called jump-to-register), which corresponds to indirect function calls. RiscyROP analyzed `libc` and several applications and reported that the majority of gadgets are stack-based jumps,[2] which are also used multiple times in its proof-of-concept attack. DeTRAP provides RAI, thus preventing stack-based jump gadgets from being exploited. As a result, DeTRAP can thwart the attack demonstrated by RiscyROP. Additionally, RiscyROP's jump-to-register gadgets can be exploited to target arbitrary locations, while DeTRAP restricts those gadgets to only target the beginning of a group of functions using CFI (Section IV-F), which also mitigates Jump-Oriented Programming [12]. Overall, DeTRAP significantly reduces the control-flow hijacking attack surface for RISC-V.

VII. Performance Evaluation

To evaluate DeTRAP's performance, we used the Chipyard [5] System-on-Chip (SoC) framework version 1.6.2 to generate verilog for a full system with our modified Rocket core [10] RISC-V implementation. We ran our design on a Digilent Arty A7-100T Development Board [28] and used Xilinx Vivado 2021.2 to synthesize and implement the verilog to run on the on-board XC7A100TCSG324-1 FPGA.

[2]Figure 3 of RiscyROP [21] shows the distribution of gadgets, but the paper does not summarize the statistics.

We configured the SoC to be similar to the SiFive Freedom E310 [63] Arty (an E31 [61] core implementation for Arty A7 development boards) and FE310 [62] SoC. To support large applications that require more memory than available on the FPGA, such as those in CoreMark Pro [32], we changed the memory system to be backed by the 256 MiB on-board DRAM, and use a 256 KiB shared inclusive L2 cache sized to fit in the remaining FPGA SRAM. The L1 code and data caches are 16 KiB and 64 KiB, respectively, corresponding to the sizes of the code and data tightly-integrated memories on the (F)E310. The L1 data cache is 4-way set associative (the same as the L1 data caches on an ARM M7 [7] or M55 [9]). Like in the E310, our L1 code cache is 2-way set associative but lacks the tightly-integrated memory (ITIM) functionality. To evaluate benchmarks that use hardware floating-point instructions, we added a 64-bit FPU to the core (the FPU is an optional feature on E31 cores [61]). We also increased the number of additional event counters from two to the ISA maximum of 29 for enhanced data collection. Table II shows the full configuration.

A. Build Configuration

We used our modified Clang/LLVM toolchain and runtime (see Sections V-C and V-F) to build the evaluated benchmarks. We compiled all code with `-O2` optimizations, link-time optimization, and linker relaxation enabled. We compiled benchmarks to use hardware floating-point instructions and the floating-point ABI. Except for any file-specific or benchmark-specific flags, all compiler and linker options, including those for optimization and sanitizers, are common across all sources (including compiler-rt). We compared this baseline to a binary that additionally enables DeTRAP protections.

B. Code Layout Effects on Performance

When evaluating benchmark performance, we observed run-to-run performance variations of less than 0.1%. However, as also seen in previous work [50], changes in memory layout impacted performance by 1% or more. For example, a build with all DeTRAP protections enabled could execute faster than one with none of its protections, even though the DeTRAP build executes more instructions.

To reduce the chance that fortuitous memory layouts make our approach faster, we compiled each benchmark with 100 different layouts: one layout generated by the default settings in the compiler and linker and 99 pseudorandom layouts provided by LLD's `--shuffle-sections` option [66]. We then report results from the fastest layout of each build. The fastest layouts of each build are compared to each other even though they are likely to have been linked with different `--shuffle-sections` values. When evaluating generated code sizes, we also use the sizes for the fastest build.

C. Benchmark Suites

We evaluated several benchmarks. **CoreMark-Pro** [32] benchmarks are from revision `4832cc`. We set the iteration count for each benchmark to the smallest value that would still

run for at least 10 seconds. We modified the zip benchmark to use pre-computed sample data, rather than generating it on-chip during the untimed initialization phase. **Embench** [53] benchmarks are from revision `d9b30c`. We left the iteration counts unmodified. The number of iterations of each benchmark was based on their scaling factors divided by our processor's speed. **BEEBS** [52] benchmarks were obtained from its git repository [11], commit `049ded`. We set each benchmark's iteration count so that it would run for at least 1 second or 100 iterations, whichever was longer. **CoreMark** [31] is from revision `b24e397` and used unmodified.

For some programs in BEEBS and Embench, the compiler was able to optimize away the entire benchmark's computation. In several cases, constant propagation allowed the compiler to calculate the benchmark's result at compile-time, so the compiler transformed the benchmark to simply output the precomputed result. For other benchmarks, the result was never used, so the compiler removed the computation altogether as unnecessary. Alternately, the compiler determined that each iteration performed an identical computation, and so emitted code that only performed the computation once regardless of how many iterations were requested. Affected Embench benchmarks were identified by manually examining the generated native code for benchmarks that ran for less than one second. For BEEBS benchmarks, we ran each benchmark with varying iteration counts, using the instruction retire count from each run to establish how many instructions were run per iteration. We then manually examined the generated native code for benchmarks that ran fewer than 100 instructions per iteration. Once identified, we added empty inline assembly statements to prevent these optimizations from removing the benchmark's core computation. Inputs were marked as "written" at the beginning of each iteration and outputs as "read". Embench benchmarks modified to prevent these optimizations were: cubic, st, statemate, and tarfind. BEEBS benchmarks modified were: aha-compress, bs, crc, crc32, cubic, fibcall, frac, janne, lcdnum, nbody, newlib-exp, newlib-mod, ns, qurt, sglib-queue and whetstone. Floating-point benchmarks that performed verification against expected values – ludcmp, matmult, nbody, st, and stb_perlin – were modified to allow for a small difference in the expected result due to floating-point rounding differences. We also fixed out-of-bounds array accesses in select and qsort, corrected duff to use the correct source and destination arrays, and modified function parameter types in sha256 to work properly with LLVM's indirect function call type checking.

D. Execution Times

Table III shows runtime performance for the CoreMark-Pro, Embench, and CoreMark benchmarks. Due to limited space, Table III only shows a statistical summary of the 80 individual benchmarks in BEEBS; full results from BEEBS can be found in our technical report [55]. Across all benchmarks we evaluated, the relative DeTRAP performance ranged from 0.991× (0.9% *faster*) to 1.201× (20.1% slower). The geometric mean across the 112 individual benchmarks was 1.011× (1.1% overhead).

TABLE III: Execution Times

Benchmark	-O2 (s)	DeTRAP (×)	Benchmark	-O2 (s)	DeTRAP (×)
CoreMark-Pro					
cjpeg	10.51	1.003	**parser**	11.74	1.006
core	191.1	1.031	**radix**	10.47	1.000
linear	12.20	1.000	**sha**	10.18	1.006
loops	51.34	1.002	**zip**	10.53	1.000
nnet	49.36	1.001			
min	10.18	1.000			
max	191.1	1.031			
geomean		1.005			
Embench					
aes	3.723	1.000	**picojpeg**	4.326	1.012
crc32	2.961	1.000	**primecount**	12.45	1.000
cubic	2.077	1.017	**qrduino**	3.332	1.001
edn	6.902	1.000	**sglib**	3.285	1.011
huffbench	2.872	0.999	**sha256**	3.834	1.035
matmult-int	2.758	1.055	**slre**	3.464	1.013
md5sum	2.347	0.998	**st**	0.184	1.000
minver	0.518	1.000	**statemate**	0.216	1.018
mont64	5.716	1.006	**tarfind**	1.586	1.001
nbody	0.180	1.000	**ud**	3.834	1.002
nsichneu	3.097	1.016	**wikisort**	0.399	1.133
min	0.180	0.998			
max	12.45	1.133			
geomean		1.014			
BEEBS (Summary)			**CoreMark**		
min	1.015	0.991			
max	1.489	1.201	**coremark**	10.40	1.019
geomean		1.010			

Comparing to Related Work. For the subset of BEEBS benchmarks reported by Silhouette [75], DeTRAP performance ranged from 0.991× to 1.131×, with a geometric mean of 1.008×. By comparison, Silhouette's performance overhead on these benchmarks was between 1.001× and 1.510×, averaging 1.102× — DeTRAP is 8.5% faster. Silhouette also evaluated CoreMark Pro, with performance between 1.001× and 1.049×, averaging 1.010× — DeTRAP is 0.5% faster. Although we did not evaluate against most benchmarks evaluated by µRAI [4], we did run the CoreMark benchmark [31]. DeTRAP's overhead on CoreMark is 1.9%, while µRAI's overhead is 8.1% — DeTRAP is 5.7% faster than µRAI.

E. Code Size

Embedded systems often have limited memory; keeping code size small is critical. We therefore evaluated DeTRAP's code size overheads by measuring the size of the code sections of each ELF executable. For each build of each benchmark, we measured size from the memory layout that had the smallest execution time. Due to limited space, we summarize the results in Table IV; full results can be seen in our technical report [55].

DeTRAP has code size overheads that average 4.5% across all benchmarks. Compared to Silhouette, which had a code size overhead of 8.9% on CoreMark Pro and 2.3% on a subset of BEEBS, DeTRAP's respective overheads are 7.9% and 6.7% — 1% better and 4.4% worse. However, we note that DeTRAP

TABLE IV: Relative Code Sizes

	-O2 (KiB)	DeTRAP (×)		-O2 (KiB)	DeTRAP (×)
CoreMark-Pro			**Embench**		
min	28.50	1.043	**min**	16.74	0.947
max	52.25	1.238	**max**	34.43	1.101
geomean		1.079	**geomean**		0.965
BEEBS			**CoreMark**		
min	7.254	0.951			
max	28.14	1.375	**coremark**	22.68	0.964
geomean		1.064			

instruments the standard library and board support code, which Silhouette does not. When we subtract the standard library code, Silhouette's average code size overheads become 16.5% on CoreMark Pro and 5.3% on BEEBS, making DeTRAP's overheads 7% better and 1.3% worse respectively.

Instead of evaluating code size, µRAI reports "Flash" utilization, which we understand to include code, read-only data, and initialized writable data. On CoreMark, µRAI has ∼40% Flash overhead, while DeTRAP's code, data, and rodata shows a 2.7% reduction in size — DeTRAP is ∼40% better.

F. Hardware Utilization

We evaluated the increased chip area (a proxy for manufacturing cost) needed to implement the required features for DeTRAP (see Section V-A). We used Chipyard [5] version 1.10.0's Hammer [46] VLSI design flow, utilizing OpenROAD [3] to implement the design for the Sky130 PDK [30].

The baseline design was a TinyRocketConfig with 4 debug trigger registers. We compared this to a modified 4-trigger design that meets DeTRAP's requirements. Our changes increase the unrouted pipeline core area (not counting cache or scratchpad) by 0.14%. For comparison, the 20 KiB of SRAM arrays used by the cache and scratchpad require 800% more area than a routed pipeline core.

VIII. Related Work

Memory management hardware has been used to mitigate memory safety attacks, even within single-address-space embedded applications. Kage [29] and Silhouette [75] utilize the ARM Memory Protection Unit (MPU) to create memory regions that only normal store instructions can modify; they then transform all untrusted store instructions into store-with-translation instructions that cannot write into these protected regions. They place shadow stacks and other security-critical data in these protected regions. uXOM [42] uses the same technique to implement execute-only memory.

RECFISH [69] also uses a MPU-protected shadow stack but requires a supervisor call to privileged code to push return addresses. IskiOS [34] leverages Intel PKU [37] to secure a shadow stack, temporarily enabling writes to the stack via a configuration register while writing a return address. CHERI [74] uses hardware-enforced capabilities, ensuring that new capabilities can be derived only from preexisting capabilities. Even if an application overwrites a return address, unless that write happened to be a valid code pointer, attempts to return to the corrupted address will fail. RetTag [70] adds pointer authentication instructions to the RISC-V ISA to authenticate return addresses. In contrast, DeTRAP makes no ISA modifications, requiring only the existing debug ISA.

µRAI [4] statically computes a complete call graph and encodes all jumps statically in read-only code jumptables. A dedicated register encodes the current location on the call tree, allowing the code to check against each possible return location, and return specifically to that location. If the register is corrupted, the jumptable lookup will not find a valid return address and fail. DeTRAP does not need to compute a complete callgraph and has less code memory overhead.

O-CFI [48] uses layout modification to ensure that all valid indirect targets have a known alignment, and clustering, which allows a bounds check to determine the validity of a branch target. The location of the table of valid bounds is randomized and saved only to a register, preventing leaks of the bounds lookup table's (BLT's) base address. However, O-CFI's protections will fail if an attacker can find the table, for example via a side channel attack of the BLT register saved to a kernel stack, or by scanning read-only memory locations for values that match a known valid indirect pointer (e.g. a return address spilled to the stack). Redactor [23] uses execute-only memory and statically generated trampolines with random memory and register layouts to prevent an attacker from reliably generating a usable gadget chain. Moreover, unlike DeTRAP, both O-CFI and Redactor must compute a reverse control-flow graph ahead of time, and their reverse CFI is not context-sensitive; an attacker can potentially redirect reverse control flow to the wrong caller, even without knowledge of the BLT's location. DeTRAP provides context-sensitive reverse CFI, even against omniscient attackers.

Work most closely related to ours has used debugging hardware to enforce non-discriminatory security policies that apply to all code; once configured, enabling access requires explicitly disabling the watchpoint. PicoXOM [60] provides execute-only code memory using watchpoint hardware and prevents reconfiguration of the memory-mapped watchpoint registers. Jang, et. al. [39] used watchpoints to allow an application to selectively lock and unlock regions via a system call; they also use watchpoints to prevent kernel access to user memory and to make kernel memory execute-only [40].

PHMon [27] adds an execution trace/monitor unit to the processor core. This monitor includes a small programmable unit that can perform actions in response to detected events. One use of the unit is to implement its own shadow stack, listening for call and return instructions to know when to push and pop addresses, interrupting the system if it detects a return to an address that does not match what it saved. Compared with DeTRAP, PHMon adds substantial hardware to the core, and because it only monitors a trace of completed execution, it can only throw a trap after instructions have been committed.

IX. Future Work

Several directions exist for future work. We can use debug triggers to solve other security challenges, such as protecting

additional control data [29] or isolating application components [36]. We can also explore whether improvements to debug triggers e.g., new data matching features or longer chain support, improves their utility for security enforcement.

X. Conclusion

We presented DeTRAP which provides embedded applications with return address integrity by utilizing RISC-V debug facilities, novel compiler transformations, and a trusted runtime to protect a shadow stack from corruption. DeTRAP's source code is available from https://github.com/URSec/DeTRAP. This work was supported by NSF Grants CNS 1652280 and CNS 2154322. The authors gratefully acknowledge Komail Dharsee for the idea of using debug triggers to implement a security policy.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information Systems Security*, vol. 13, pp. 4:1–4:40, November 2009.

[2] J. Afek and A. Sharabani, "Dangling Pointer: Smashing the Pointer for Fun and Profit," in *Black Hat USA*, 2007.

[3] T. Ajayi, V. A. Chhabria, M. Fogaça, S. Hashemi, A. Hosny, A. B. Kahng, M. Kim, J. Lee, U. Mallappa, M. Neseem, G. Pradipta, S. Reda, M. Saligane, S. S. Sapatnekar, C. Sechen, M. Shalan, W. Swartz, L. Wang, Z. Wang, M. Woo, and B. Xu, "Toward an open-source digital flow: First learnings from the openroad project," in *Proceedings of the 56th Annual Design Automation Conference 2019*, ser. DAC '19. New York, NY, USA: Association for Computing Machinery, 2019.

[4] N. S. Almakhdhub, A. A. Clements, S. Bagchi, and M. Payer, "μRAI: Securing Embedded Systems with Return Address Integrity," in *Proceedings of the Network and Distributed System Security (NDSS) Symposium*, San Diego, CA, USA, February 2020.

[5] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, P. Rigge, C. Schmidt, J. Wright, J. Zhao, Y. S. Shao, K. Asanović, and B. Nikolić, "Chipyard: Integrated design, simulation, and implementation framework for custom socs," *IEEE Micro*, vol. 40, no. 4, pp. 10–21, 2020.

[6] *ARMv7-M Architecture Reference Manual*, Arm Holdings, December 2014, DDI 0403E.b.

[7] *ARM Cortex-M7 Processor Technical Reference Manual*, Arm Limited, November 2018, DDI 0489.f.

[8] *ARMv8-M Architecture Reference Manual*, Arm Limited, October 2019, DDI 0553B.i.

[9] *ARM Cortex-M55 Processor Technical Reference Manual*, Arm Limited, September 2021, document 101051 version 0101-01.

[10] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The rocket chip generator," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[11] "BEEBS git repository." [Online]. Available: https://github.com/mageec/beebs

[12] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented Programming: A New Class of Code-reuse Attack," in *Proceedings of the 6th ACM Asia Conference on Computer & Communications Security (ASIACCS)*, Hong Kong, China, 2011, pp. 30–40.

[13] N. Burow, X. Zhang, and M. Payer, "Sok: Shining light on shadow stacks," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 985–999.

[14] N. Carlini and D. Wagner, "ROP is still dangerous: Breaking modern defenses," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 385–399.

[15] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data Attacks Are Realistic Threats," in *Proceedings of the 14th USENIX Security Symposium (SEC)*, Baltimore, MD, 2005, pp. 12–12.

[16] K. Cheng, *Introduce a new tag, `Tag_RISCV_x3_reg_usage`...* [Online]. Available: https://github.com/riscv-non-isa/riscv-elf-psabi-doc/pull/387

[17] ——, *Relax gp could be platform specific register...* [Online]. Available: https://github.com/riscv-non-isa/riscv-elf-psabi-doc/pull/371

[18] T.-C. Chiueh and F.-H. Hsu, "RAD: a compile-time solution to buffer overflow attacks," in *Proceedings 21st International Conference on Distributed Computing Systems*, 2001, pp. 409–417.

[19] Clang 13.0 Documentation, *Control Flow Integrity*. [Online]. Available: https://releases.llvm.org/13.0.1/tools/clang/docs/ControlFlowIntegrity.html

[20] ——, *ShadowCallStack*. [Online]. Available: https://releases.llvm.org/13.0.1/tools/clang/docs/ShadowCallStack.html

[21] T. Cloosters, D. Paaßen, J. Wang, O. Draissi, P. Jauernig, E. Stapf, L. Davi, and A.-R. Sadeghi, "RiscyROP: automated return-oriented programming attacks on risc-v and arm64," in *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*, ser. RAID '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 30–42.

[22] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, M. Negro, C. Liebchen, M. Qunaibit, and A.-R. Sadeghi, "Losing control: On the effectiveness of control-flow integrity under stack attacks," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. Denver, CO: ACM, 2015, pp. 952–963.

[23] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical code randomization resilient to memory disclosure," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 763–780.

[24] J. Criswell, N. Dautenhahn, and V. Adve, "KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels," in *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, San Jose, CA, May 2014, pp. 292–307.

[25] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve, "Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation," in *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Istanbul, Turkey, 2015, pp. 191–206.

[26] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 401–416.

[27] L. Delshadtehrani, S. Canakci, B. Zhou, S. Eldridge, A. Joshi, and M. Egele, "PHMon: A programmable hardware monitor and its security use cases," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, August 2020, pp. 807–824.

[28] Digilent, *Arty A7 Reference Manual*. [Online]. Available: https://digilent.com/reference/programmable-logic/arty-a7/reference-manual

[29] Y. Du, Z. Shen, K. Dharsee, J. Zhou, R. J. Walls, and J. Criswell, "Holistic Control-Flow Protection on Real-Time Embedded Systems with Kage," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 2281–2298.

[30] R. T. Edwards, "Google/skywater and the promise of the open pdk," in *Workshop on Open-Source EDA Technology*, 2020.

[31] "CoreMark: An EEMBC benchmark." [Online]. Available: https://www.eembc.org/coremark

[32] "CoreMark-Pro: An EEMBC benchmark." [Online]. Available: https://www.eembc.org/coremark-pro

[33] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of Control: Overcoming Control-Flow Integrity," in *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, San Jose, CA, May 2014, pp. 575–589.

[34] S. Gravani, M. Hedayati, J. Criswell, and M. L. Scott, "Fast intra-kernel isolation and security with iskios," in *24th International Symposium on Research in Attacks, Intrusions and Defenses*, 2021, pp. 119–134.

[35] Y. Guo, L. Chen, and G. Shi, "Function-oriented programming: A new class of code reuse attack in c applications," in *2018 IEEE Conference on Communications and Network Security (CNS)*, 2018, pp. 1–9.

[36] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. L. Scott, K. Shen, and M. Marty, "Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries," in *2019 USENIX Annual Technical Conference*

(USENIX ATC 19). Renton, WA: USENIX Association, Jul. 2019, pp. 489–504.

[37] Intel Corp., "Intel 64 and IA-32 Architectures Software Developer's Manual," April 2021, 325384-074US.

[38] G.-A. Jaloyan, K. Markantonakis, R. N. Akram, D. Robin, K. Mayes, and D. Naccache, "Return-oriented programming on risc-v," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 471–480.

[39] J. Jang and B. B. Kang, "In-process memory isolation using hardware watchpoint," in *Proceedings of the 56th Annual Design Automation Conference 2019*, ser. DAC '19. New York, NY, USA: Association for Computing Machinery, 2019.

[40] ——, "Revisiting the ARM Debug Facility for OS Kernel Security," in *Proceedings of the 56th Annual Design Automation Conference 2019*, ser. DAC '19. New York, NY, USA: Association for Computing Machinery, 2019.

[41] P. Kirth, *D146463: [CodeGen][RISCV] Change Shadow Call Stack Register to X3*. [Online]. Available: https://reviews.llvm.org/D146463

[42] D. Kwon, J. Shin, G. Kim, B. Lee, Y. Cho, and Y. Paek, "*u*XOM: Efficient eXecute-Only Memory on ARM Cortex-M," in *Proceedings of the 28th USENIX Security Symposium*, ser. Security '19. Santa Clara, CA: USENIX Association, August 2019, pp. 231–247.

[43] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the 2nd International Symposium on Code Generation and Optimization*, ser. CGO '04. Palo Alto, CA: IEEE Computer Society, 2004.

[44] C. Lattner, A. D. Lenharth, and V. S. Adve, "Making context-sensitive points-to analysis with heap cloning practical for the real world," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, CA, USA, June 2007, pp. 278–289.

[45] C. Liebchen, "Clang control flow integrity (cfi) bypass techniques." [Online]. Available: https://github.com/0xcl/clang-cfi-bypass-techniques

[46] H. Liew, D. Grubb, J. Wright, C. Schmidt, N. Krzysztofowicz, A. Izraelevitz, E. Wang, K. Asanović, J. Bachrach, and B. Nikolić, "Hammer: a modular and reusable physical design flow tool: invited," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, ser. DAC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1335–1338.

[47] lowRISC contributors, "OpenTitan Security Model Specification." [Online]. Available: https://docs.opentitan.org/doc/security/specs/secure_boot/

[48] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz, "Opaque control-flow integrity," in *NDSS*, 2015.

[49] B. H. Møller, J. G. Søndergaard, K. S. Jensen, M. W. Pedersen, T. W. Bøgedal, A. Christensen, D. B. Poulsen, K. G. Larsen, R. R. Hansen, T. R. Jensen *et al.*, "Preliminary security analysis, formalisation, and verification of opentitan secure boot code," in *Nordic Conference on Secure IT Systems*. Springer, 2021, pp. 192–211.

[50] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Producing wrong data without doing anything obviously wrong!" in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIV. New York, NY, USA: Association for Computing Machinery, 2009, p. 265–276.

[51] "Risc-v port of newlib." [Online]. Available: https://github.com/riscv-collab/riscv-newlib

[52] J. Pallister, S. Hollis, and J. Bennett, "BEEBS: Open benchmarks for energy measurements on embedded platforms," *arXiv preprint arXiv:1308.5174*, August 2013.

[53] D. Patterson, J. Bennett, P. Dabbelt, C. Garlati, G. S. Madhusudan, and T. Mudge, "Embench™: An evolving benchmark suite for embedded iot computers from an academic-industrial cooperative: Towards the long overdue and deserved demise of dhrystone," in *RISC-V Workshop Zurich*, 2019.

[54] D. R. Piegdon and L. Pimenidis, "Hacking in physically addressable memory," in *Seminar of Advanced Exploitation Techniques, WS 2006/2007*, vol. 12, 2007.

[55] I. Richter, J. Zhou, and J. Criswell, "DeTRAP: RISC-V return address protection with debug triggers," *arXiv preprint arXiv:2408.17248*, August 2024.

[56] RISC-V Debug Subcommittee, *RISC-V Debug Support*, 9 February 2022, revision `b659d7`. [Online]. Available: https://github.com/riscv/riscv-debug-spec

[57] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-Oriented Programming: Systems, Languages, and Applications," *ACM Transactions on Information Systems Security (TISSEC)*, vol. 15, no. 1, pp. 2:1–2:34, Mar. 2012.

[58] A. Sadeghi, S. Niksefat, and M. Rostamipour, "Pure-call oriented programming (PCOP): chaining the gadgets using call instructions," *Journal of Computer Virology and Hacking Techniques*, vol. 14, no. 2, pp. 139–156, 2018.

[59] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen, "Adapting software fault isolation to contemporary CPU architectures," in *Proceedings of the 19th USENIX Security Symposium*, ser. Security'10. Washington, DC: USENIX Association, 2010, pp. 1–11.

[60] Z. Shen, K. Dharsee, and J. Criswell, "Fast execute-only memory for embedded systems," in *Proceedings of the 2020 IEEE Secure Development Conference*, ser. SecDev '20. Atlanta, GA: IEEE Computer Society, 2020, pp. 7–14.

[61] SiFive, Inc., *SiFive E31 Core Complex Manual*. [Online]. Available: https://sifive.cdn.prismic.io/sifive/c29f9c69-5254-4f9a-9e18-24ea73f34e81_e31_core_complex_manual_21G2.pdf

[62] ——, *SiFive FE310-G003 Manual*, version 1.0.1. [Online]. Available: https://sifive.cdn.prismic.io/sifive/3af39c59-6498-471e-9dab-5355a0d539eb_fe310-g003-manual.pdf

[63] ——, *SiFive Freedom E310 Arty FPGA Dev Kit Getting Started Guide*. [Online]. Available: https://www.sifive.com/documentation/freedom-soc/freedom-e300-arty-fpga-dev-kit-getting-started-guide/

[64] ——, *SiFive FU540-C000 Manual*, version v1p4. [Online]. Available: https://sifive.cdn.prismic.io/sifive/d3ed5cd0-6e74-46b2-a12d-72b06706513e_fu540-c000-manual-v1p4.pdf

[65] ——, *SiFive FU740-C000 Manual*, version v1p6. [Online]. Available: https://sifive.cdn.prismic.io/sifive/1a82e600-1f93-4f41-b2d8-86ed8b16acba_fu740-c000-manual-v1p6.pdf

[66] R. Song, *D74791 Add a –shuffle-sections=seed option to lld*. [Online]. Available: https://reviews.llvm.org/D74791

[67] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, U. Erlingsson, L. Lozano, and G. Pike, "Enforcing Forward-edge Control-flow Integrity in GCC & LLVM," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC'14, 2014, pp. 941–955.

[68] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning, "On the Expressiveness of Return-into-libc Attacks," in *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection (RAID)*, Menlo Park, CA, 2011, pp. 121–141.

[69] R. J. Walls, N. F. Brown, T. Le Baron, C. A. Shue, H. Okhravi, and B. Ward, "Control-flow integrity for real-time embedded systems," in *31st Conference on Real-Time Systems (ECRTS'19)*, July 2019.

[70] Y. Wang, J. Wu, T. Yue, Z. Ning, and F. Zhang, "RetTag: hardware-assisted return address integrity on risc-v," in *Proceedings of the 15th European Workshop on Systems Security*, ser. EuroSec '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 50–56.

[71] Z. Wang and X. Jiang, "HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity," in *Proceedings of the 31st IEEE Symposium on Security and Privacy (S&P)*, May 2010, pp. 380–395.

[72] A. Waterman, K. Asanović, and J. Hauser, Eds., *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213*. RISC-V Foundation, December 2019. [Online]. Available: https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf

[73] ——, *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20211203*. RISC-V International, December 2021. [Online]. Available: https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf

[74] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The CHERI Capability Model: Revisiting RISC in an Age of Risk," in *Proceeding of the 41st Annual International Symposium on Computer Architecture*, ser. ISCA '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 457–468.

[75] J. Zhou, Y. Du, Z. Shen, L. Ma, J. Criswell, and R. J. Walls, "Silhouette: Efficient protected shadow stacks for embedded systems," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1219–1236.