2021 58th ACM/IEEE Design Automation Conference (DAC) | 978-1-6654-3274-0/21/$31.00 ©2021 IEEE | DOI: 10.1109/DAC18074.2021.9586293

SHORE: Hardware/Software Method for Memory Safety Acceleration on RISC-V

Hsu-Kang Dow, Tuo Li, William Miles, Sri Parameswaran
School of Computer Science and Engineering
The University of New South Wales
Sydney, Australia
{h.dow, tuoli, w.miles, sri.parameswaran}@unsw.edu.au

***Abstract*—Memory corruption vulnerabilities can lead to software attacks. Pointer-based memory safety protection has been shown as a promising solution covering both out-of-bounds and use-after-free errors. Software only approaches have significant performance overhead. Existing hardware/software implementations are largely limited to proprietary closed-source microprocessors, simulation-only studies or require changes to the input source code.**

In this paper, we present a novel hardware/software co-design methodology consisting of a RISC-V based processor extended with new instructions and microarchitecture enhancements, enabling faster memory safety checks. A compiler is instrumented to provide security operations taking into account the changes to the processor. The entire system is realized by enhancing a RISC-V Rocket-chip system-on-chip (SoC)[1]. The resultant processor SoC is implemented on an FPGA and evaluated with applications from SPEC 2006 (for generic applications), MiBench (for embedded applications), and Olden benchmark suites for performance. Our experiments show that the proposed approach achieves up to 3.79X speedup (average 2.6X) in comparison to the traditional software-based approach for SPEC2006 while possessing an overhead of 6.33% in terms of area. This speedup is better than the state-of-the-art approach. Our security coverage using the NIST Juliet test suite shows better coverage than the software only method.

I. Introduction

Memory corruption vulnerabilities can lead to a variety of software security attacks [1]. The latest Common Weakness Enumeration (CWE) study [2] states that three of the top five most dangerous software weaknesses are related to memory safety breaches. Memory corruption vulnerabilities are categorized into spatial and temporal errors. While dereferencing an out-of-bounds pointer (e.g., indexing beyond the bounds of an array) causes a spatial error, dereferencing a dangling pointer (e.g., "use after free"[2]) leads to a temporal error. Since low-level pointers are heavily utilized in C/C++, memory corruption vulnerabilities are a significant safety issue in C/C++ applications.

Pointer-based checking [3]–[7] has been studied as a method to ensure memory safety during the execution of a program. Pointer-based checking adds additional information (usually referred to as metadata) for each pointer into the program and enhances the program with the capability to use the metadata to check for spatial and temporal errors when dereferencing a pointer. For example, by using the lower and upper bounds of a pointer, one can detect spatial errors when the program dereferences this pointer during execution.

A pointer-based checking system typically performs three major types of operations: one, creating metadata when the program creates a pointer (called metadata initialization); two, performing checks when pointer dereference occurs (called dereference check); and, three, storing, moving, and modifying the metadata when pointers are moved or modified (called metadata propagation). For integrating these additional operations, the baseline system must be augmented in software and/or hardware incurring performance and/or hardware overhead.

[1]source code are available in GitHub https://github.com/Lycheus/SHORE
[2]https://cwe.mitre.org/data/definitions/416.html

978-1-6654-3274-0/21/$31.00 © 2021 IEEE

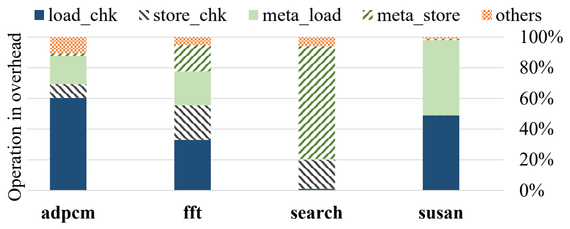


Fig. 1. **A demonstration on the distribution of the security operations in performance overhead after safety instrumentation.**

Existing pointer-based checking methods have limitations [8]. Software-implemented methods [5] modify compilers to realize pointer-based checking operations using native instructions available in the ISA (and requires no additional hardware). Consequently, these methods suffer from substantial slow-down (e.g., 140% in [5]). Hardware-implemented methods [4] have better performance than software-based but demand dramatic hardware changes, such as adding extra metadata (e.g., tag) to caches, and TLBs, for tracking the pointers in hardware, due to the lack of compiler support. In addition, most hardware-related methods are limited to a proof-of-concept study, where the system is implemented and evaluated on in-house simulators [9]. Some techniques also have compatibility issues where the source program to be checked has to be extensively modified and thus unsuitable for use with legacy code [10].

To overcome these limitations, this paper proposes a hardware/-software methodology, named SHORE, for realizing cost-efficient pointer-based checking functionality. SHORE reduces the performance overhead with minimal increase in hardware. Fig. 1 illustrates the security operations overhead distribution from the preliminary studies. The major overhead on pointer-based solutions is from the metadata load/store (meta_load and meta_store) and dereference checks (load_chk and store_chk). By characterizing the behavior of the pointer-based solution, the following solutions are implemented into the SHORE: 1) to perform fast deference check, SHORE fuses the dereference check with load/store operations, allowing the processor to perform load/store and dereference checks in a single instruction; 2) to reduce the overhead of metadata propagation, SHORE augments the processor microarchitecture enabling the processor to automatically perform in-pipeline metadata propagation, simultaneously when the pointer is moved or modified; and, 3) compiler support for pointer analysis and instrumenting SHORE's new instructions (ISA extension) into the program.

Following the existing studies [5], we also focus on spatial memory safety in this study, given that the temporal check shares a similar mechanism and can be extended on top of the spatial check system. Leveraging the openness and flexibility of RISC-V ISA (Rocket-core), in this study, we create a SHORE processor by extending a RISC-V Rocket-core processor (with SHORE ISA extensions and

TABLE I
COMPARISON CHART TO POINTER-BASED RELATED WORKS

Research	Instrumentation method	Additional hardware	Type of shadow memory	Hardware Propagation	Protection	Implementation Experiment platform
Hardbound	Hardware	Tag + TLB Cache Shadow register	Disjoint linear mapped	Y	S	Simics simulator
Softbound	Compiler	NO	Disjoint trie	N	S	Software
WatchdogLite	Hardware/Compiler	AVX2 (wide mode)	Disjoint linear mapped	N	S/T	In-house simulator
IntelMPX	Hardware/Compiler	ISA extension	Disjoint trie	N	S	ASIC
Shakti-MS	Hardware/Compiler	ISA extension	Fat-pointer	N	S/T	FPGA
SHORE (This work)	Hardware/Compiler	ISA extension Shadow register	Disjoint linear mapped	Y	S/T[3]	FPGA

microarchitecture augmentation) and the SHORE compiler by significantly augmenting the RISC-V LLVM [11]). To evaluate the overhead of the SHORE approach, the SHORE processor was implemented on an FPGA and was tested with SPEC2006 benchmarks as well as other embedded benchmarks. In addition, for testing security coverage, Juliet NIST [12] test cases were employed. **The key contributions of this paper are as follows:**

- a RISC-V based processor is extended with novel ISA and microarchitecture extensions to provide efficient memory safety;
- software instrumentation is performed using compiler-based software analysis and transformations to enhance input programs to exploit hardware extensions of the processor maximally; and,
- by combining the above hardware and software, for the first time this paper provides a fully automatic hardware/software design methodology to instrument C programs with pointer-based checking functionality, and an extended RISC-V based processor capable of executing these programs.

The rest of the paper is structured as follows. Section 2 provides a discussion of related work. Section 3 elaborates our hardware/software approach. Section 4 gives the experiment and results. Section 5 provides further discussion followed by Section 6, which concludes the paper.

II. Related Work

There are many memory protection solutions described in the literature [3]–[7], [9], [10], [13], [14]. While methods such as probability-based checking [13] and trip-wire based checking [7] are widely used, they are limited in their capabilities. Others provide greater memory safety, such as type-safe system [3], capability-systems [15] and pointer-based system protection [4]–[6], [9], [10]. In this paper, we focus on the pointer-based memory safety.

In pointer-based protection, pointers are annotated with extra boundary information (metadata). Hardbound [4] is a hardware approach to manage metadata, which requires additional tag-cache and TLB to track pointers and its metadata. Softbound [5] and its temporal safety counterpart CETS [6] use compiler instrumentation to insert runtime functions to perform safety operations. WatchdogLite [9] further replaces runtime functions with instructions and utilize AVX vector operations to speed up performance for Intel processors. IntelMPX [16] is the commercial version of the pointer-based protection implemented by Intel, and BOGO [14] further extended the system to cover temporal safety. As for RISC-V platform, Shakti-MS [10] is the only RISC-V implementation using pointer-based protection.

Table I gives an overall comparison to pointer-based protection related to our work. In comparison, Hardbound relies on an additional tag cache and TLB to track the register with pointers and metadata, which introduces considerable size overhead. In SHORE, the pointer analysis in the compiler is used to eliminate tracking hardware.

Softbound and its temporal safety counterpart CETS utilize compiler instrumentation to inline security operations in the form of software functions. The benefit of using compiler instrumentation

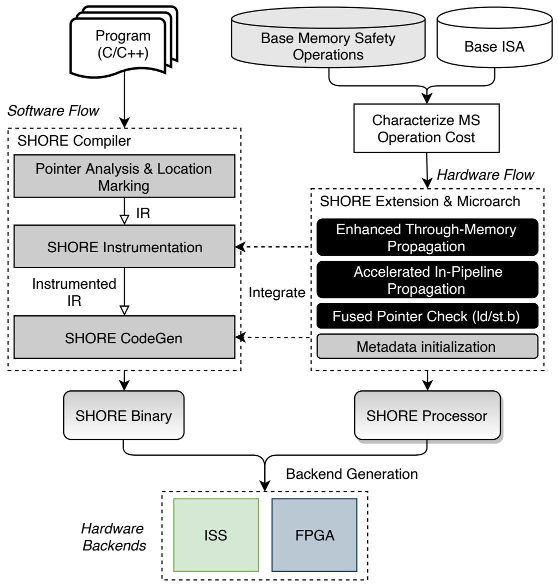

Program (C/C++)
Base Memory Safety Operations
Base ISA
Characterize MS Operation Cost
Software Flow
SHORE Compiler
Pointer Analysis & Location Marking
IR
SHORE Instrumentation
Instrumented IR
SHORE CodeGen
Integrate
Hardware Flow
SHORE Extension & Microarch
Enhanced Through-Memory Propagation
Accelerated In-Pipeline Propagation
Fused Pointer Check (ld/st.b)
Metadata initialization
SHORE Binary
SHORE Processor
Backend Generation
Hardware Backends
ISS
FPGA

Fig. 2. **Overview of the hardware/software co-design of SHORE.**

is that it does not require the user to change the source code and recompile third-party libraries (source and binary compatibility). However, the software-based security operations incur high-performance overhead. To reduce the performance overhead, we fuse the dereference checking into the load/store instruction of the RISC-V and propagate the metadata inside the hardware to speed up the operations.

This is similar to WatchdogLite, but WatchdogLite is a proof-of-concept implementation on the x86 platform and relies on Intel's special AVX extension making it less generic. WatchdogLite can only run their experiments in their in-house simulator. IntelMPX is using a trie structure to store the metadata in a disjoint shadow memory. The trie structure requires additional time to search, which is less straightforward to the linear mapped shadow memory used by Hardbound, WatchdogLite, and us. Shakti-MS [10] uses fat-pointers and stack frame cookies, which will alter the memory layout of the program stack and break the compatibility rendering it less useful. In contrast, we use disjoint shadow memory, which does not intrude user memory space. Moreover, we provide a set of instructions to access the shadow memory in hardware for improved performance.

III. The SHORE Approach

A. Motivation and Design Goals

The main mechanisms involved in pointer-based memory safety are the **creation, utilization, and movement** of the metadata. These

[3]The temporal safety can be enforced without hardware acceleration by enabling CETS instrumentation in the compiler. However, this is beyond the scope of this paper.

TABLE II
ISA EXTENSION FOR MEMORY SAFETY ACCELERATION

Instruction	Format	Description
bndr	rd, rs1, rs2	store the value of rs1 to base shadow register of rd store the value of rs2 to bound shadow register of rd
lbd[l/u]	rd, offset(rs1)	load the metadata of corresponding shadow memory of address in rs1 to rd
sbd[l/u]	rs2, offset(rs1)	store corresponding shadow register of rs2 to the corresponding shadow memory of address in rs2
l[b/h/w/d/bu/hu/wu].b	rd, offset(rs1)	bounded load, compare base bound in shadow register of rs1 against the pointer in rs1
s[b/h/w/d].b	rs2, offset(rs1)	bounded store, compare base bound in shadow register of rs1 against the pointer in rs1

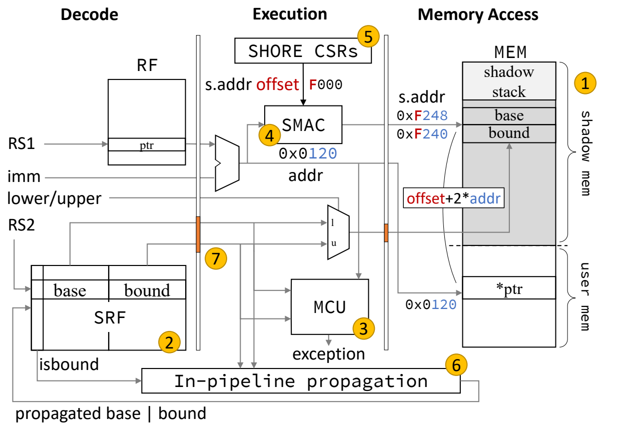
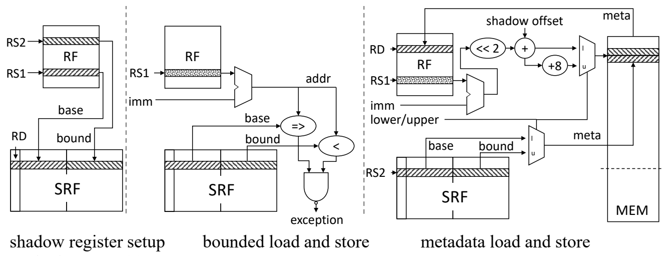


Fig. 3. **SHORE pipeline and shadow memory layout.**



shadow register setup
bndr **[SRF]** ***(a)***

bounded load and store
(load/store).b **[MCU]** ***(b)***

metadata load and store
sbd & lbd **[SMAC]** ***(c)***

Fig. 4. **Hardware design of the SHORE instructions.**

metadata-related operations introduce high-performance overhead to the program.

The operations of **creating** pointer metadata are added by compilers. The created metadata is stored into an augmented register file called Shadow Register File (SRF) instead of inside a general-purpose register file (RF). The SRF helps reduce the register pressure incurred by the additional metadata. To **use** the metadata more efficiently, we fuse the dereference-check with the load/store instructions (see section III-C bounded load/store). The bounded load/store will automatically use the metadata stored in the SRF to perform the check without consuming additional cycles. **Movement** of the metadata has to be considered for two separate cases: one, when moving metadata within the core (in-pipeline propagation); and two, when moving the metadata between the core and the main memory (through-memory propagation).

In the first case (**use**), the datapath is augmented to propagate the metadata in shadow registers when pointer arithmetic is performed. The in-pipeline propagation eliminates the movement of metadata which requires additional cycles. For the second case (**movement**), main memory has to be enhanced. When a pointer is stored into memory, the metadata of the pointer also must be stored in a spatially separated memory. This design is to ensure the metadata will not intrude in the user memory layout. The spatially separated memory is formed as a linear mapped shadow memory. The shadow memory is a mirrored address of the user program. When a user stores a pointer to the memory, the associated metadata (base/bound) is also stored to the linear mapped address above the user memory (elaborated in Fig. 3-1). However, the cost of storing/loading is tripled (additional load/store of the base and bound). Therefore, we create hardware instructions to enhance the metadata load/store operations.

B. Approach Overview

Fig. 2 depicts the overview of the SHORE methodology, which combines the hardware (on the right) and software (left) flows. Using the base processor, base memory safety operations were first profiled and analyzed to extract hotspots (similar to Fig. 1). Based on hotspots, the hardware flow starts to create SHORE ISA to improve the performance of memory safety operations and reduce overhead. As shown in Fig. 2, SHORE's hardware flow includes ISA extension and microarchitecture augmentation, which aims at improving through-memory propagation (metadata load/store instructions), in-pipeline propagation (SRF and propagation circuitry), and the fused check operation (bounded load/store instructions). In addition, SHORE ISA provides essential operations such as metadata initialization (bndr instruction). By augmenting the microarchitecture of the processor, the resultant processor (SHORE processor) is created.

On the software side, the compiler is modified to include pointer analysis and identification of the locations for instrumentation (Step 1) and instrumentation (Step 2) on the program's intermediate representation (IR), by utilizing SHORE extensions. The instrumented IR is passed to the modified code-generation (Step 3, CodeGen) stage, which includes the added SHORE instructions into the program's binary. The hardware backend supported in our design framework includes an open-source instruction-set simulator (ISS) and an implementation on the FPGA. In our implementation, the base compiler is LLVM and the base processor is RISC-V Rocket-core (included in Rocket-chip SoC).

C. SHORE ISA Extension and Hardware

Register extension for metadata: RF is augmented with an SRF (Fig. 3-2) and is addressed by the same address bus as the RF. Thus, each register in the RF is associated with space in the SRF to hold the associated metadata. The SRF is a 32-entry double-word register file (32 entries of 129 bits – 64 bits for the bound value and 64 bits for the base value and 1 bit for *isbound* flag to indicate the register contains bounded pointer).

Setup shadow register *(bndr)*: When a pointer is created, the calculated base and bound values initially reside in RF. To associate the base and bound values to the pointer, the *bndr* instruction is introduced, (listed in the first row of Table II), to store the metadata to the associated shadow registers in SRF. For example, *"bndr RD, RS1, RS"* will store the value of register RS1 to the *base* shadow register of RD, and RS2 to the *bound* shadow register of RD as shown in Fig. 4(a). The bndr instruction will also set the *isbound* to TRUE to enable metadata propagation.

Metadata load and store *(sbdl / sbdu / lbdl / lbdu)*: The *metadata (store,load) x (lower,upper)* instructions, listed in the second and third and rows of Table II, accelerate shadow memory accesses. When storing a pointer to the memory, the metadata of the pointer will also be stored in the linear mapped shadow memory using the *sbdl/sbdu* instructions (the **'l' and 'u'** suffix is to identify the storing is for base or bound). Shadow Memory Address Calculator (SMAC) unit calculates the address of the shadow memory. The SMAC executes with the Arithmetic Logic Unit (ALU) in the processor's execution stage (Fig. 3-4). It takes the source register outputs and offset to calculate the shadow memory address for the following MEM stage. The calculation of the shadow address is given by Eq. 1 and hardware implementation shown in Fig. 4(c).

$$shadow\_address = shadow\_offset + 2 * (RS1 + imm) \quad (1)$$

The design requires the addition of three new 64-bits control registers (SHORE CSRs): one register for each of the three operating modes (machine, supervisor, and user). Within each CSR register, the highest bit is the enable flag of the memory safety protection. When the highest bit of CSR is set to zero, the memory safety mechanism will be bypassed (used for legacy libraries). The lower 63 bits of the CSR store the shadow memory offset address for SMAC as shown in Fig. 3-5.

Bounded load/store *(load.b/store.b)*: The bounded load/store instructions listed in the fourth and fifth rows Table II further accelerates the security operations. In pointer-based memory safety, checking happens when accessing the memory. By combining the checking with the load and store operations into a single instruction can further improve performance. These instructions are created in addition to the original load and store operations which are preserved in our system. Hence third-party libraries that are not instrumented by our compiler can still work with the generic load/store instruction. Metadata Checking Unit (MCU) is the checking unit we have introduced. MCU resides in the execution stage past the ALU, as shown in Fig. 3-3. The checks are done after the address calculations. When a load/store instruction is dispatched, the MCU takes the memory access-pending address and tests against the metadata that has been passed through the pipeline from the shadow register. If the address fails the check, then the unit raises an exception at the EX stage, which halts the pipeline and jumps to the exception handler as shown in Fig. 4(b).

In-Pipeline Base-Bound Propagation: In Fig. 3-6, for any operation that writes back to the register file, the corresponding metadata in the shadow register is propagated to the destination under the following conditions: if both sources or neither source is bounded then the destination register will be set to be unbounded; if one of the sources for the operation is bounded, then the bounds from the bounded source is propagated to the destination; immediate values are treated as unbounded sources. This is shown in short pseudocode in the following Listing 1.

```
e.g., [add rd, rs1, rs2]

rd <= rs1 + rs2;
if  (rs1.isbound == rs2.isbound){
  //both source register are unbound or bounded
  rd.shadow == NULL;
  //destination set to unbound
  rd.isbound == false;
} else if (rs1.isbound == true)
rd.shadow <= rs1.shadow;
else // rs2.isbound == true
rd.shadow <= rs2.shadow;
```

Listing 1. Pseudocode for in-pipeline propagation in ADD instr.

This propagation allows for pointer arithmetic to work as the programmer intended, without the compiler having to modify the metadata. To properly integrate this propagation into the pipeline, the base and bound values must be passed down to the pipeline in the same fashion as the register values as illustrated in Fig. 3-7. To ensure all the other parts of the safety system have access to correct control signals, the control signals must be routed through the pipeline and support bypassing (forwarding). This requires expanding the stage buffers to also hold the metadata, and to route the data between buffers and to units.

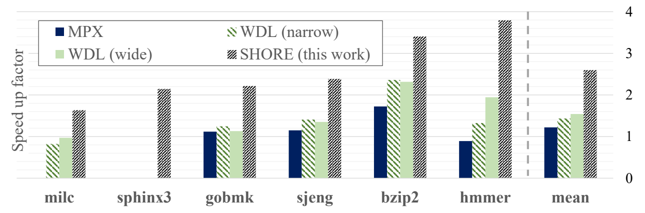


Fig. 5. **The speedup factor (higher the better) comparison between IntelMPX, WatchdogLite and SHORE (this work) in SPEC2006.**

D. SHORE Compiler Support

To support the hardware and enforced memory safety, several compiler instrumentations are performed in LLVM intermediate representation (IR). There are three goals for compiler instrumentation: one, bind the metadata to the shadow register; two, to perform through-memory bounds propagation; and three, to perform bounded load/store transformation.

Metadata binding to Shadow Register: To bind the associated metadata to the pointer when pointers are created statically, the compiler calculates the base and bound values and inlines an *"bndr"* instruction to bind the metadata to the shadow register corresponding to the register that contains the pointer. If a pointer is created at runtime, e.g., using *malloc()*, a function wrapper is created for pointer-creation functions and the shadow register is set up in the wrapper. The instrumentation is performed at the IR level.

Through-Memory Bounds Propagation: Using the above instrumentation, the metadata of pointers is calculated and loaded into shadow registers. From time to time, a pointer will be stored from a register into memory, thus the corresponding metadata data in the shadow register shall also be transfer into the memory. Our approach is to use instructions to invoke the transfer of metadata to and from memory. First, the compiler has to locate the load and store locations of the pointer in IR, then instrument the *lbd and sbd* instructions to store the metadata from the shadow register to the linear mapped shadow memory (or vice versa). In this way, the Through-Memory Bounds Propagation is achieved in an implicit style. This means only loading and storing a pointer to memory will be instrumented with a metadata load/store. Note that this is different from Hardbound [4], which does not differentiate between pointers and normal data in registers and therefore, explicitly loads and stores all metadata when memory operations occur.

Bounded Load/Store Transformation: This instrumentation is to locate memory dereference in the program. When memory safety is enforced, the compiler will substitute the protected load/store instruction into the bounded load/store. The bounded load/store will perform the checking in the pipeline with the corresponding metadata inside the shadow register as shown in Fig. 4(b).

Kernel memory space modification: To reserve the memory address for the disjoint linear mapped shadow memory, the kernel virtual memory space has to be modified. The kernel used here is the proxy kernel (pk) [17]. In pk, we set the address of the user stack top (highest address of user memory) to the third of the original address and aligned with the page size as shown in Fig. 3-1.

E. Eliminating Shadow Stack

When a pointer is passed into a function, the corresponding metadata of the pointer also must be passed into the function. An obvious method is to use an additional stack (shadow stack) in the memory, as demonstrated in [5], [9]. By taking advantage of the in-pipeline propagation hardware and compiler support in SHORE,

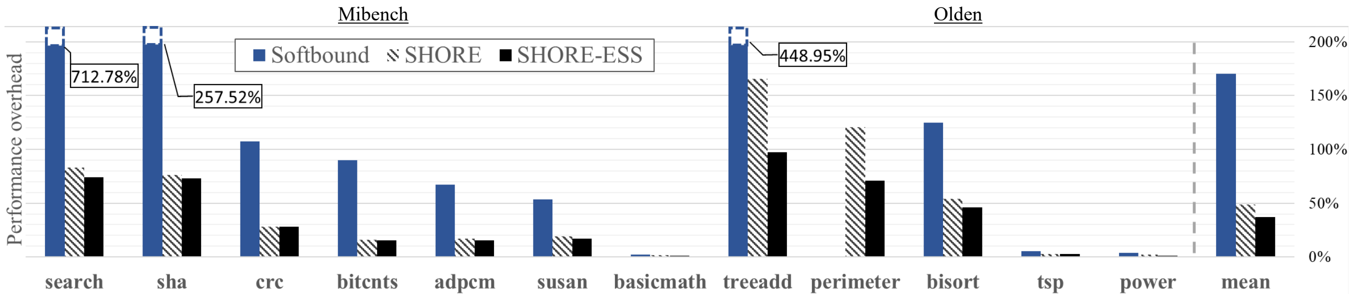


Fig. 6. **The performance overhead (lower the better) between Softbound, SHORE and SHORE-ESS.**

whenever there is a sufficient number of registers available (e.g., eight in RISC-V) for passing the function arguments (i.e., then no arguments are passed via the stack), SHORE can eliminate the shadow stack to further improve performance by reducing memory accesses while checking the pointer bounds. When the shadow stack is eliminated, pointer metadata will reside in the shadow register file and propagate automatically along with the pointer during function calls. This scheme where the shadow stack is not used is denoted as SHORE-ESS and is used for programs that have less than eight arguments in function calls. This is typically the case in embedded workloads.

IV. Experiment and Results

A. Experimental setup

For performance evaluation on the SHORE architecture, our hardware platform (shown in Table III) is based on the Rocket-chip project from RISC-V foundation [18]. The implementation of the SHORE RISC-V processor (RV64GC) is on the Xilinx ZCU102 FPGA board. There are two configurations of our hardware:

- **SHORE**: The baseline SHORE with shadow stack to support generic workload such as SPEC2006 [19].
- **SHORE-ESS**: Based on SHORE but improved for embedded workload, using shadow registers to replace the shadow stack. Passing of metadata for pointer arguments utilizes the shadow registers.

The software benchmarks in use are applications from SPEC CPU2006 [19] suite to simulate generic workloads, and MiBench and Olden [20], [21] suites for embedded workloads. For security coverage, spatial related test cases in NIST Juliet test suite [12] are used. There are total of 6814 spatial-related test cases from five subcategories. These subcategories are stack-based buffer overflow (CWE121, 2786 cases), heap-based buffer overflow (CWE122, 1684 cases), buffer underwrite (CWE124, 844 cases), buffer overread (CWE126, 656 cases), and buffer underread (CWE127, 844 cases).

B. Comparing to IntelMPX WatchdogLite in SPEC2006

Fig. 5 presents the speedup factor between IntelMPX (MPX) [16], WatchdogLite (WDL) [9] and SHORE when executing SPEC CPU2006 benchmarks. Both MPX and WDL are executed on x86 (MPX is on real hardware, while WDL is evaluated on a sample-based simulator), while our implementation is on the SHORE-RISC-V (executed on an FPGA). Speedup factor is used to show cross-platform comparison. The speedup factor is calculated using the equation in Eq. 2. The software-only approach (Softbound) is set as the baseline. The cycle numbers of Softbound is extracted for each of the corresponding works (e.g., MPX speedup is calculated using Softbound cycle number from [8]). The cycle numbers of the speedup calculation for MPX[4] is extracted from the GitHub repository of the paper [8]. The WDL cycle counts are extracted from Fig. 3 of [9], the wide mode of WDL uses the AVX2 accelerated version while the narrow mode uses scalar registers. The cycle numbers of SHORE is collected from our FPGA experimental platform and extracted from the performance counter in the SHORE hardware. For SPEC benchmarks there were no extensive results available for comparison on the RISC-V platform and thus had to be compared against the x86 platform.

$$Speedup = \frac{Softbound_{cycle}}{HW\_technique_{cycle}} \tag{2}$$

Six cases from the SPEC CPU2006 benchmark suite are shown in Fig. 5. The mean value of speedup using the MPX is 1.22 times, WDL (narrow) 1.43 times, WDL (wide) is 1.54 times and SHORE is 2.60 times. Note, however, the underlying platforms are different. The SHORE speedup is better than the other three hardware implementation. WDL uses linear mapped shadow memory while MPX is using a trie. Both SHORE and MPX focus on spatial safety, while WDL handles both spatial and temporal safety. SHORE exhibits higher speedups for two reasons. One, the use of shadow registers and in-pipeline propagation to help reduce register conflicts; and two, the use of linear mapped shadow memory for faster metadata accesses.

C. Embedded workloads performance overhead

As stated in the experimental setup, there are two configurations for use within the SHORE memory safety system: one is SHORE for generic applications, and the other is SHORE-ESS which is created for embedded workloads. Here performance overhead is used for analysis since the same RISC-V architecture is utilized. The performance overhead is calculated by dividing the number of clock cycles taken by the memory safety enhanced program by the number of clock cycles taken by the native program (default compiler without third-party protection). The results are given as a percentage. The formula of the performance overhead is given in Eq. 3.

$$Perf.overhead = (\frac{Protection\_method_{cycle}}{native_{cycle}} - 1) \times 100 \tag{3}$$

The results showing the performance overhead is in Fig. 6. The mean values of the overhead for Softbound, SHORE, and SHORE-ESS are 170.37%, 48.83%, and 36.79% respectively. The first seven test cases are from Mibench and the last five are from Olden. Olden test cases are more pointer-intensive and pass pointers in function arguments more frequently. Therefore, SHORE-ESS is more efficient in the Olden workloads. Generally, SHORE-ESS which utilizes the

TABLE III
RISC-V System configuration

Attribute	Setting
ISA	GC (IMAFDC)
L1D-Cache	32KiB 8-way 64B
L1I-Cache	32KiB 8-way 64B
L1D-TLB	32 entries
L1I-TLB	32 entries
L2-TLB	128 entries

[4]IntelMPX is instrumented with the Intel ICC compiler

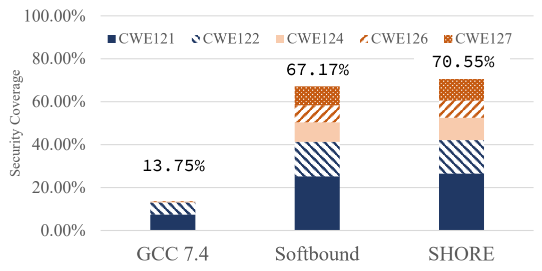


Fig. 7. **The security coverage (higher the better) of NIST Juliet test suite.**

shadow registers to pass metadata along with pointers in function arguments shows approximately 12.04% improvement (from 48.83% to 36.79%) in performance overhead, over the SHORE.

D. Hardware overhead

Table IV shows the overhead of the hardware cost in FPGA over a generic RISC-V Rocket-chip. The overhead of look-up tables (LUTs) is 6.33%, and flip-flops (FFs) is 4.73%. There is no additional cost in DSP and RAM blocks. The main contributor to the additional hardware cost is from shadow registers and the hardware support for bypassing (forwarding) as described in the In-pipeline Base-bound Propagation in Sec. III-C. The SHORE processors had 11% decrease in clock speed when compared to the baseline Rocket-chip. However, we believe this can be improved with further optimization.

TABLE IV
HARDWARE OVERHEAD OF THE SHORE RISC-V

Resource	Base Processor	SHORE	Overhead
LUTs	37346	39710	6.33%
FFs	16916	17716	4.73%

E. Security coverage

The NIST Juliet test suite is used for the security-oriented benchmark. Three compilers are used to demonstrate security coverage. The GCC 7.4 is the default compiler in the Ubuntu 18.04 Linux distribution. The original Softbound is based on LLVM 3.9 from the Softbound public source. The SHORE compiler is based on LLVM 8.0 with our ISA extended RISC-V. WDL toolchain is not publicly available and thus not available security coverage study. In-depth study of MPX coverage can be found in [8], [14].

Fig. 7 illustrates the coverage of the cases that can be detected under these three solutions. Out of the 6814 test cases, the GCC with built-in protection can detect 937 cases (13.75%). The Softbound can detect 4577 cases (67.17%), and the SHORE can detect 4807 (70.55%) cases. The cases that can be detected in Softbound but not in SHORE happen in the CWE124 buffer underwrite categories. The cases that can be detected in SHORE but not in Softbound falls into the categories containing allocation/declamation in a loop. In summary, the SHORE has higher security coverage (3.38%) than the Softbound method.

V. DISCUSSION

Memory footprint — Pointer-based check approaches incur additional memory size to store the pointer metadata, despite using disjoint or inline pointer metadata (fat-pointer). Existing studies [9] measure the overhead of program memory footprint by counting the additional memory pages allocated for the metadata memory space (similar to our shadow memory). In our experiments, we pre-allocate a sufficiently large memory space for our shadow memory, in the RISC-V proxy kernel (PK), which is a lightweight proxy kernel. Hence, it is not viable for us to directly measure how many pages are allocated for shadow memory. However, because our shadow memory shares the same layout (linearly mapped mirrored the program's sections) with [9], we estimate our memory footprint overhead is similar, which is 56% on average.

Temporal safety — In this paper, similar to the prior arts [5], our SHORE approach at first focuses on spatial memory safety. The reason is: 1) spatial safety issues are the major portion of the memory safety issues; and, 2) temporal safety protection can be extended directly on top of the spatial safety system, for example, by extending the metadata to include keys and locks [6].

VI. CONCLUSION

In this paper, we presented the SHORE architecture, a hardware/-software co-design method based on a RISC-V based platform to accelerate pointer-based memory safety. SHORE is the fastest system which targets an open architecture that can enforce spatial memory safety without modification of the source code. In performance, the SHORE has 2.6X speedup, which is better than other hardware implementations such as IntelMPX and WatchdogLite with minimal hardware overhead. In security coverage, SHORE has 3.38% higher coverage in Juliet test suite than the Softbound. Overall, SHORE provides high-performance, low-cost and FPGA-ready hardware for RISC-V based processors for improved memory security.

ACKNOWLEDGEMENT

This research was supported by the Australian Research Council's Discovery Projects funding scheme (project DP190103916).

REFERENCES

[1] J. Pincus and B. Baker, "Beyond stack smashing: Recent advances in exploiting buffer overruns," *IEEE S & P*, vol. 2, no. 4, pp. 20–27, 2004.
[2] MITRE, "Cwe top 25 most dangerous software errors," 2020.
[3] G. C. Necula, McPeak, *et al.*, "Ccured: Type-safe retrofitting of legacy code," in *ACM SIGPLAN Notices*, vol. 37, pp. 128–139, ACM, 2002.
[4] J. Devietti *et al.*, "Hardbound: architectural support for spatial safety of the c programming language," in *ACM SIGARCH Computer Architecture News*, vol. 36, pp. 103–114, ACM, 2008.
[5] S. Nagarakatte, Zhao, *et al.*, "Softbound: Highly compatible and complete spatial memory safety for c," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 245–258, 2009.
[6] S. Nagarakatte, Zhao, *et al.*, "Cets: compiler enforced temporal safety for c," in *ACM Sigplan Notices*, vol. 45, pp. 31–40, ACM, 2010.
[7] K. Serebryany, D. Bruening, *et al.*, "Addresssanitizer: A fast address sanity checker," in *Proc. of the 2012 USENIX Conference on Annual Technical Conference*, p. 28, USENIX Association, 2012.
[8] Oleksenko *et al.*, "Intel mpx explained: A cross-layer analysis of the intel mpx system stack," *POMACS*, vol. 2, no. 2, p. 28, 2018.
[9] S. Nagarakatte *et al.*, "Watchdoglite: Hardware-accelerated compiler-based pointer checking," in *Proc. CGO*, 2014.
[10] S. Das, Unnithan, *et al.*, "Shakti-ms: a risc-v processor for memory safety in c," in *Proc. LCTES*, pp. 19–32, ACM, 2019.
[11] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis transformation," in *Proc. CGO*, 2004.
[12] T. Boland and P. E. Black, "Juliet 1. 1 c/c++ and java test suite," *Computer*, vol. 45, no. 10, pp. 88–90, 2012.
[13] C. Kil, Jun, *et al.*, "Address space layout permutation (aslp): Towards fine-grained randomization of commodity software," in *ACSAC'06*, pp. 339–348, IEEE, 2006.
[14] T. Zhang, D. Lee, and C. Jung, "Bogo: buy spatial memory safety, get temporal memory safety free," in *ASPLOS'19*, pp. 631–644, 2019.
[15] Woodruff *et al.*, "The cheri capability model: Revisiting risc in an age of risk," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pp. 457–468, IEEE, 2014.
[16] R. Ramakesavan, D. Zimmerman, and P. Singaravelu, "Intel memory protection extensions (intel mpx) enabling guide," 2015.
[17] Waterman *et al.*, "RISC-V proxy kernel and boot loader," 2018.
[18] K. Asanovic *et al.*, "The rocket chip generator," *EECS, UCB, Tech. Rep. UCB/EECS-2016-17*, 2016.
[19] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, p. 1–17, Sept. 2006.
[20] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proc. WWC-4*, pp. 3–14, IEEE, 2001.
[21] Carlisle *et al.*, *Olden: parallelizing programs with dynamic data structures on distributed-memory machines*. PhD thesis, Princeton, 1996.