HWST128: Complete Memory Safety Accelerator on RISC-V with Metadata Compression

Hsu-Kang Dow
University of New South Wales
Sydney, Australia
h.dow@unsw.edu.au

Tuo Li*
University of New South Wales
Sydney, Australia
tuoli@unsw.edu.au

Sri Parameswaran
University of New South Wales
Sydney, Australia
sri.parameswaran@unsw.edu.au

ABSTRACT

Memory safety is paramount for secure systems. Pointer-based memory safety relies on additional information (metadata) to check validity when a pointer is dereferenced. Such operations on the metadata introduce significant performance overhead to the system. This paper presents HWST128, a system to reduce performance overhead by using hardware/software co-design. As a result, the system described achieves spatial and temporal safety by utilizing microarchitecture support, pointer analysis from the compiler, and metadata compression. HWST128 is the first complete solution for memory safety (spatial and temporal) on RISC-V. The system is implemented and tested on a Xilinx ZCU102 FPGA board with 1536 LUTs (+4.11%) and 112 FFs (+0.66%) on top of a Rocket Chip processor. HWST128 is 3.74× faster than the equivalent software-based safety system in the SPEC2006 benchmark suite while providing similar or better security coverage for the Juliet test suite.

CCS CONCEPTS

• **Security and privacy** → *Hardware security implementation*; **Embedded systems security**.

1 INTRODUCTION

Modern, complicated software is vulnerable to memory corruption [1]. Two out of the top three most dangerous software weaknesses are memory corruption related [2]. Memory unsafe languages, such as C and C++, do not restrict low-level pointer access. Therefore, a pointer can be dereferenced outside of its allocated space, causing out-of-bound accesses, known as spatial memory violation leading to memory corruption. Similarly, when a pointer is freed, other pointers inherited from the original pointer become dangling pointers. Adversaries can use a dangling pointer to access a memory location that no longer belongs to the original user, leading to temporal memory violations.

Memory corruption can be prevented by enforcing memory safety. Pointer-based memory safety algorithms [3–6] have been proposed to enforce memory safety. Pointer-based memory safety introduces additional information (known as metadata) to prevent the pointer from accessing invalid memory locations. Spatial memory safety is performed by introducing the *base* and the *bound* metadata to record the starting and ending addresses of a pointer (for example, the starting and ending address of an array) when a pointer is allocated. Then, a boundary check can be performed when the pointer is dereferenced. Such a check ensures the pointer will not dereference out of the original designated space, preventing spatial memory violation.

Temporal safety introduces two additional pieces of information to the metadata, *key* and the *lock*. First, a unique key is assigned to a pointer when allocated. Then, the *key* is stored in a lock_location. The address of the lock_location is called the lock. When a pointer is dereferenced, the *key* in the lock_location will be loaded first using the lock, then compared with the *key* held by the pointer. The *key* will be removed from the lock_location if a pointer is freed. Therefore, future usage of the old pointer will fail the key check when a dangling pointer is dereferenced.

However, the pointer-based memory safety algorithm introduces significant performance overhead due to the additional metadata operations. Prior research has been done to reduce the performance overhead introduced by the metadata operations. For example, Hardbound [4] utilized the sidecar register and tag cache to enforce spatial memory safety with low performance overhead (for x86). The open-source project, SHORE [7], adopted the idea of Hardbound and created a 128x64 bits shadow register file with in-pipeline metadata propagation, and then fused the checking process with the load/store instruction to further reduce the performance overhead (on RISC-V). Note, SHORE was limited to spatial memory safety only.

HWST128, the work described in this paper, provides both spatial and temporal safety for RISC-V architectures. A novel, configurable metadata compression technique has been proposed in HWST128 to fit the additional temporal metadata into a 128-bit shadow register file. A small TLB-like *keybuffer* is proposed in the microarchitecture to reduce the additional performance impact from temporal security operations. Thus, HWST128 achieves both spatial and temporal safety without additional hardware cost while with minimal impact on performance. HWST128 is the first complete solution for memory safety on RISC-V with hardware modifications and compiler support. The key contribution of HWST128 are as follows: 1) the first spatial and temporal safety accelerator on RISC-V; 2) a novel metadata compression scheme allowing for minimal hardware overhead; 3) a keybuffer to reduce the performance impact from temporal operations; and, 4) open-source tool-chain for the FPGA-ready RISC-V platform.[1] The rest of the paper is structured

*Tuo Li is the corresponding author.

DAC '22, July 10–14, 2022, San Francisco, CA, USA

ACM ISBN 978-1-4503-9142-9/22/07…$15.00
https://doi.org/10.1145/3489517.3530548

[1] https://github.com/Lycheus/HWST128

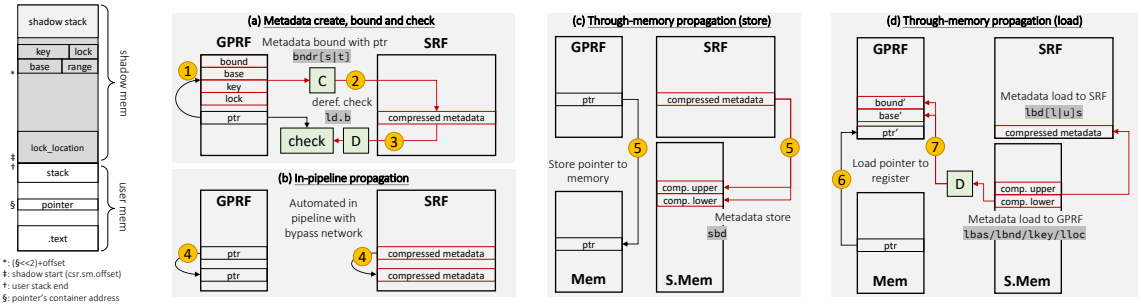

Figure 1: Memory layout of the linear-mapped shadow memory and metadata flow between general purpose register files (GPRF), shadow register file (SRF), user member (MEM) and shadow memory (S.Mem).

as follows. Section 2 provides a discussion of related work. Section 3 elaborates on our hardware/software/compression approach, and Section 4 describes the experimental setup. Section 5 provides results discussion, followed by Section 6, which concludes the paper.

2 RELATED WORK

Various algorithms exist to detect and protect from memory corruption. Tripwire-based [8], probability-based [9], and colouring-based [10] are widely used but have limitation in enforcing temporal and spatial memory safety. Capability-based methods [11] require a complete overhaul of the system memory design, limiting the ability to support legacy programs.

Pointer-based algorithms [3–6, 12–14] allow fine-grained protection for all memory accesses to memory. Three challenges present themselves when utilizing pointer-based algorithms. First of these is the compatibility of the programs. Pointer-based algorithms require the program to be instrumented with security operations. However, directly changing the source code of the program is undesirable. SoftboundCETS (SBCETS) [5, 6] utilizes the LLVM compiler [15] instrumentation and runtime library wrappers to maintain source compatibility (without the need to change the source code) and binary compatibility (without the need to recompile the libraries) [1].

The second is the memory space for metadata. Pointer-based memory safety algorithm needs the metadata for protection. Where to store the metadata becomes an issue. Works such as SHAKTI-MS [16] concatenated the metadata to the pointer, making what is known as a fat-pointer. This method will intrude upon the stack space during a function call, thus breaking binary compatibility. Another technique stores the metadata separately from the pointer [4, 5]. Thus, creating a disjoint shadow memory for metadata storage. Disjoint shadow memory can be made as a trie or a direct linear map of user memory space. The benefit of a shadow trie is the better utilization of the user address space. However, a linear-mapped shadow space is more hardware-friendly, simplifying the hardware design for better performance.

The third is the performance overhead introduced by the metadata operations. The approach to mitigate the performance overhead is to utilize hardware accelerators. WatchdogLite (WDL) [12] uses the x86 256-bit AVX instructions to handle the metadata. WDL provides spatial and temporal memory safety and has excellent performance. However, it is a closed source project and is simulation-based. BOGO [14] utilizes the IntelMPX [13] spatial memory protection extension and extends the protection to temporal safety with little additional cost. BOGO achieves this by scanning and nullifying pointers' boundaries when pointers are freed. However, BOGO can only provide partial temporal safety (use-after-free) instead of complete temporal safety (includes use-after-return). SHORE [7] is another hardware accelerator based on RISC-V, which is open source but limits the acceleration to spatial safety only. SHORE utilize a 128-bit shadow register file to handle the metadata in the pipeline. In contrast to the above works [7, 12, 14] HWST128 provides both spatial and temporal safety acceleration with compressed metadata (so a 128-bit shadow register file is sufficient) to reduce the storage overhead.

3 METHODOLOGY

Pointer-based memory safety relies on additional information (metadata) of pointers to check whether a pointer is legal when it is dereferenced. However, the **creation**, **movement**, and **utilization** of metadata introduces significant performance overheads to the system. Additionally, the original code requires instrumentation. Automating this instrumentation significantly reduces the onus on the programmer and allows for legacy code to be instrumented easily. Therefore, HWST128 aims to reduce the impact on performance and the user by the utilization of hardware/software co-design. As a result, HWST128 achieves temporal and spatial memory safety with low overhead by utilizing microarchitecture support, pointer analysis from the compiler, and metadata compression.

Threat model and assumptions — HWST128 hardware is based on RISC-V Rocket Chip project [17], where the baseline processor is assumed to be trusted and no underlying hardware bugs [18] leads to memory corruption. Furthermore, HWST128 assumes no vector instructions are used for memory accesses, and the adversary cannot corrupt the metadata by using non-memory-safety related attacks or illegitimately obtain higher (root) privilege. As HWST128 relies on the pointer analysis of SBCETS, we assume that SBCETS instrumentation and function wrappers are covered for all the libraries used when complete coverage is required.

3.1 Metadata for memory safety

Metadata is created for spatial and temporal protection. Metadata for spatial memory protection contains two parts, *base* and *bound*. When a pointer is created, the corresponding base metadata points at the start and bound metadata points at the end of the allocated memory. When the pointer is dereferenced, the dereferenced address will be checked against the *base* and the *bound*. This action will ensure that memory access will not be out-of-bound, enabling spatial safety.

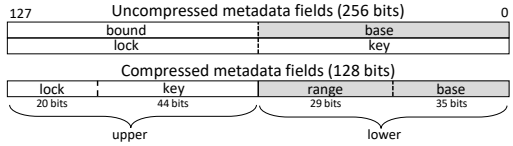

Figure 2: Metadata fields.

Metadata for temporal protection also contains two parts, the *key* and the *lock*. A unique key is assigned to the pointer and saved in a lock location when a pointer is created. The address of the lock location will become the lock metadata. The *key* and *lock* will bind to the pointer. If the pointer is later freed, then the *key* in the lock location will be erased. When a pointer is dereferenced, the *key* stored in the lock location will be checked against the *key* held by the pointer. If the pointer has been freed or reassigned previously, the *key* held by the pointer will not match the *key* stored in the lock location. Thus, the use-after-free (heap) and use-after-return (stack) temporal attack is detected by checking the *key*, enabling temporal safety.

3.2 Metadata Flow

In pointer-based memory safety, each pointer will come with four pieces of metadata (*base/bound/key/lock*). These metadata require additional space to store. When a pointer is in the general-purpose register file (GPRF), the metadata are also in registers for dereferencing checking. The metadata will occupy the registers and cause register spilling, which causes performance degradation. Therefore, a shadow register file (SRF) is proposed to mitigate this problem. An SRF is an additional register file that resides next to the GPRF as shown in Fig. 3 decode stage. The SRF has a one-to-one relationship with the GPRF, which mean that for each register in GPRF, there will be a corresponding shadow register to store the metadata. In HWST128, the SRF is 128-bit to contain the four metadata.

When the metadata is bound with the pointer, if the pointer moves from the GPRF to memory, the corresponding metadata also needs to move from register to memory, namely, through-memory propagation. Therefore, additional space needs to be allocated in memory to store the additional metadata. This additional space is called shadow memory (S.Mem). Because each 64-bit pointer requires 128-bit metadata, thus, two-third of the user memory address must be reserved for creating a linear-mapped shadow memory (LMSM), as shown on the left-most side of Fig. 1.

Metadata creation and binding — When a pointer is allocated (Fig. 1-a1), the starting and ending address of the allocated space of the pointer (*base/bound*) and a unique *key* with a pointer (*lock*) pointing to a pre-allocated memory (lock_location) will be created in the GPRF. Next, the metadata (*base/bound/key/lock*) will bind to the pointer (Fig. 1-a2) in the corresponding SRF, using bounding instructions (bndr[s/t]- instructions created for memory safety) with spatial and temporal metadata compression in (C) of Fig. 1-a1. The binding instructions compresses the metadata and places them in the SRF – see Section 3.3 for detail.

Metadata usage — When the pointer is dereferenced (Fig. 1-a3) with the bounded load/store instruction, the compressed metadata will be decompressed without it going to the GPRF (D). The address of the pointer is checked against the *base* and the *bound*. Finally, the lock address of the pointer will be dereferenced to load the *key* and compared against the *key* held by the pointer to check if they are identical with the temporal check instruction (tchk). If all checks are passed, then the pointer is dereferenced.

Metadata in-pipeline propagation — The metadata of a pointer is bound in a corresponding register, e.g., if the pointer is in $R_n, the metadata of the pointer will be bound to $SRF_n. Therefore, when the pointer is moved to a different register, the corresponding shadow register's metadata will be transferred to another shadow register (Fig. 1-b4). The register file level of metadata propagation is handled in the hardware pipeline and requires no additional instruction for the movement.

Metadata through-memory propagation — When a pointer is stored into the memory (Fig. 1-c5), the corresponding metadata that binds with the pointer in the SRF will be stored into the LMSM with the metadata store instruction (sbd[l/u]). The target shadow address for the metadata is calculated using a preset offset in a control status register (CSR) set at the beginning of a program. The calculation of the target address in LMSM for a given pointer is shown in Eq. 1, where $Addr_{LMSM}$ is the target address to the linear-mapped shadow memory for metadata, $Addr_{ptr\_container}$ is the address of the container of the pointer, and CSR_{offset} is the preset offset in CSR.

$$Addr_{LMSM} = (Addr_{ptr\_container} \ll 2) + CSR_{offset} \quad (1)$$

Similarly, when the pointer is loaded back from memory, the corresponding metadata is loaded from the shadow memory. The metadata can be loaded (with created instructions lbd[l/u]s) to one shadow register (Fig. 1-d6) or multiple general-purpose registers (lbas/lbnd/lkey/lloc) (Fig. 1-d7). The destination depends on whether it is a user program (check instrumented in code) or third-party linked libraries (check instrumented in a function wrapper), e.g., glibc. A decompression process will be applied before the metadata is loaded back from S.Mem to GPRF.

3.3 Metadata Compression

Our work is based on the RISC-V RV64 instruction set, using a 64-bit address and data path. Initially, 64-bits are allocated for each metadata field (*base/bound/key/lock*). The four 64-bit metadata requires four additional loads and stores when the pointer is transferred between register and memory. However, 256-bit for the metadata is not necessary, as this data can be compressed as shown below.

An example of metadata fields layout is shown in Fig. 2, at the top of the figure is the uncompressed metadata, each of the metadata will occupy 64 bits, and each load/store can only store one field at a time. However, for a system with 256 gigabytes of memory, the user pointer does not exceed more than 38 bits of virtual addressing. Furthermore, the bit width of the base address can be reduced further with address alignment, as memory access in RISC-V RV64 is 8-byte aligned. The alignment can save an additional three bits in the base metadata Therefore, the base metadata can be compressed into 35 bits. A similar approach can be applied to the bound metadata.

A new type of metadata called "range" can be created by subtracting the address of *base* from the *bound* as shown in Eq. 2. The bit width of *range* is determined by the largest object of the program (Eq. 4). In our tests, the range bit needs to be at least 25 bits to pass the SPEC2006. Similarly, for SPEC2006, support is needed for up to one million unique pointers. Thus, the *lock* requires 20 bits to point to one million entries. The rest of the 44 bits are assigned to the *key* metadata The symbols used in Eq. 2-6 are as follows:

$BIT_{[base|range|lock|key]}$ is the bit width of the (base/range/lock/key) address; and, $Addr_{[base|bound]}$ is the address of the base and bound.

$$range = Addr_{bound} - Addr_{base} \quad (2)$$

$$BIT_{base} = \left\lceil log_2(\frac{memory\ size}{3}) \right\rceil - 3 \quad (3)$$

$$BIT_{range} = log_2(max(\{range_1, ..., range_n\})) - 3 \quad (4)$$

$$BIT_{lock} = log_2(lock\ entries) \quad (5)$$

$$BIT_{key} = 128 - BIT_{base} - BIT_{range} - BIT_{lock} \quad (6)$$

After compression, the 256-bit metadata is compressed into 128 bits and fits into a 128-bit shadow register file (SRF). Due to the nature of the 64-bit memory alignment of the RV64. The compressed 128 bits of metadata is split into upper and lower sections, and each section contains 64 bits. Therefore, our design's metadata layout for the general-purpose application uses 35 bits for the *base* and 29 bits for the *range* as the lower section of the compressed metadata. The *lock* field is 20 bits, and the *key* field is 44 bits as the upper section of the compressed metadata. The calculation of the bit width for each compressed metadata field is shown in Eq. 3-6.

The compression and decompression of the metadata are done in hardware. The bit width for each metadata is set within a 24-bit CSR at the beginning of the program. The bndrs instruction will compress the 128 bits *base* and *bound* metadata into 64 bits compressed spatial metadata and will bind it into the SRF. The bndrt instruction will do the temporal part of compression to the metadata and will bind to SRF. The sbdl instruction will store the lower 64-bit SRF to the lower shadow memory and the sbdu to the upper store. For metadata load, there are two sets of loading from the S.Mem. The lbdls and lbdus will load the metadata from S.Mem directly to the SRF without decompression, benefiting memory transfer functions such as memcpy(). The metadata of a pointer can be copied from SRF to SRF without passing through GPRF, thus avoiding register spilling.

3.4 Compiler Instrumentation

The compiler is based on LLVM with the RISC-V backend. The pointer analysis is augmented from SBCETS instrumentation to support RISC-V. The compiler compiles the source code into an intermediate representation (IR). Then, the pointer analysis generates the metadata depending on whether the pointer is statically or dynamically allocated. The *base* and the *bound* metadata can be set at initialization if a pointer is statically allocated. The temporal metadata, *key* and *lock* will be generated with instrumented runtime functions to find the available lock_location in shadow memory then assigns a unique key to the pointer. If the pointer is dynamically allocated, the metadata information can be found when memory allocation functions, such as malloc(), are called. The memory allocation functions are wrapped with wrapper functions. The metadata are created and bound to the pointer in the wrapper function. When a pointer is freed, the *key* stored in the lock_location will be set to zero. Therefore, any previous pointer with the same lock_location will be invalidated. The lock_location will be free to use in the future. However, the new allocation will have a different unique key that prevents access from invalid pointers.

In shadow trie design, the lock_location memory resides in user memory (heap). However, if the number of allocations does not exceed the code size of the program times two, then we can map

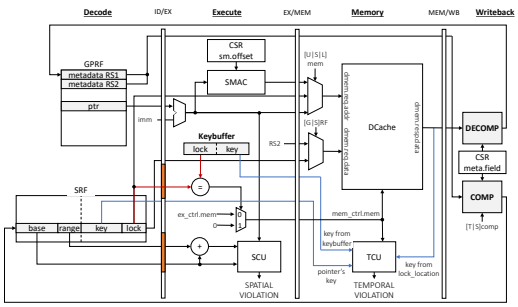

Figure 3: Overview of the hardware pipeline and memory security modules.

the lock_location to the beginning of the shadow memory. The technique is valid because the beginning of the shadow memory is correlated with the program's .text section, which is plain text without pointers (initialized and uninitialized const pointers reside in .bss and .data). Therefore, this technique can save user memory and applies to all the embedded workloads in MiBench and Olden benchmarks. SPEC benchmarks require more lock_locations than embedded workloads. Thus, the lock_locations are pre-allocated in the heap memory.

3.5 HWST128 Hardware Design

The hardware design of HWST128 is based on SHORE [7]. SHORE is a 5-stage pipeline in-order spatial memory safety accelerator based on the Rocket Chip with 128-bit SRF. HWST128 inherited the pipeline from SHORE, then extended the instruction set to support temporal safety. However, the original SHORE architecture does not handle temporal metadata in hardware. Therefore, we augmented the SHORE pipeline to support metadata compression and decompression to fit the temporal metadata into the SHORE pipeline.

Fig. 3 illustrates the pipeline modification of HWST128 from SHORE. The metadata is now compressed before writing to the SRF in the writeback stage. The compression bit width for each metadata field is set in the HWST128 CSRs. The compression module (COMP) and decompress module (DECOMP) will compress and decompress the metadata accordingly. The shadow memory address calculator (SMAC) will calculate the target address to shadow memory using the pointer's container address and the offset set in the HWST128 CSRs when loading or storing the metadata. The spatial boundary check can be performed at the execution stage when the target address of the bounded load/store instruction is calculated. First, the lower section of the SRF will be decompressed into *base* and *bound* values. Then, sent into the spatial check unit (SCU) with the target address. If the target address is out-of-bound, a spatial violation trap will be evoked.

The temporal check requires performing a load to fetch the *key* in lock_location before comparing it to the *key* held by the pointer in SRF. Hence, the temporal check cannot be merged with the load/store instruction as the spatial check has two memory accesses involved in a single instruction. A keybuffer is introduced to accelerate the access speed of the *key* in the *lock*. The keybuffer will keep a record of the most recent *key* loaded from the *lock*. The *lock* will be compared with the pointer's lock when the temporal check instruction (tchk) is executed. If the *lock* is identical, the

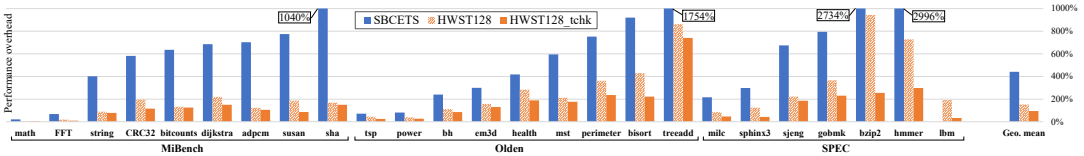

Figure 4: Performance overhead of MiBench, Olden and SPEC2006

key in the keybuffer will be used instead of the *key* from memory. The bypass of loading lock_location can be done by modifying the valid signal in the DCache module of the SHORE. The keybuffer will be cleared whenever a pointer has been freed to ensure the keybuffer will always hold the latest temporal metadata. In the end, the *key* from the *lock* and the *key* from the pointer's SRF will be sent to the temporal check unit (TCU) to be compared. If the *key* is mismatched, a temporal violation trap will be evoked to the kernel.

4 EXPERIMENTAL SETUP

The hardware platform is based on the Rocket Chip project [17] from the RISC-V foundation. The HWST128 RISC-V processor (RV64GC) is compiled from CHISEL into Verilog, then synthesized with Xilinx Vivado 2017. The FPGA target is the Xilinx Zynq UltraScale+ MPSoC ZCU102 FPGA board. The HWST128 compiler is based on LLVM 8.0 with our ISA extended RISC-V.

SPEC CPU2006 [19] is used as generic benchmarks. MiBench and Olden as the embedded benchmarks for performance evaluation. First, the sources of the programs are compiled into executables with the spatial and temporal memory safety instrumentation. Then, the executables are executed on the ZCU102 FPGA. The cycle count is gathered by enabling the proxy kernel's "-s" flag to report the total number of cycles of each execution. All performance benchmarks are compiled and linked without compiler optimization.

The memory safety test cases are from the NIST Juliet suite [20] for security coverage evaluation. There are 7074 spatial related cases and 1292 related temporal cases. The spatial attack subcategories are stack-based buffer overflow (CWE121), heap-based buffer overflow (CWE122), buffer underwrite (CWE124), buffer overread (CWE126), and buffer underread (CWE127). The temporal attack subcategories are double-free (CWE415), use-after-free (CWE416), null pointer dereference (CWE476), null dereference from return (CWE690), and free pointer not-at-start-of-the-pointer (CWE761). The security workload is run on the RISC-V instruction set simulator (SPIKE) [21]. The SPIKE simulator is augmented with the HWST128 security operation hardware and metadata compression.

5 RESULT AND DISCUSSION

Performance is measured using cycle count for each workload running on the RISC-V FPGA platform. The performance overhead (perf.oh) is used to observe the impact of the memory safety algorithm. The calculation of the perf.oh is shown in the Eq. 7. The *perf.oh(%)* is the performance overhead in percentage. The $instrumented_{cycle}$ and $baseline_{cycle}$ are the cycle count of program with and without security operation instrumentation. Therefore,

$$perf.oh(\%) = \left(\frac{instrumented_{cycle}}{baseline_{cycle}} - 1\right)x100. \quad (7)$$

5.1 Performance overhead

Fig. 4 shows the perf.oh of the MiBench/Olden/SPEC benchmarks on three different memory safety algorithms running on the RISC-V FPGA platform. The SBCETS is the perf.oh of the SoftboundCETS, a pure software solution to enforce spatial and temporal memory safety. The HWST128_tchk is the perf.oh with the hardware instruction tchk, which utilizes the keybuffer to perform the temporal checking. The HWST128 is similar to the spatial part of the HWST128_tchk but without the tchk instruction support. Therefore, HWST128 uses the software method to load the key from the lock_location to perform temporal security operations. The rightmost bar is the geometric mean (mean) of all the perf.oh. The pure software memory safety algorithm performs poorly in the RISC-V platform. The mean of the perf.oh using SBCETS is around 441.45%. When the metadata propagation is realized in HWST128 perf.oh reduces to 152.91%. The perf.oh can be further reduced to 94.89% with *tchk* instruction to perform the temporal key check with the assistance of the keybuffer.

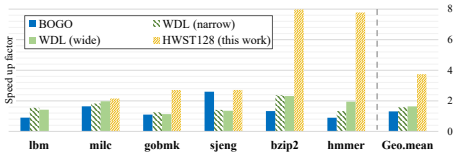

Figure 5: Speedup factor comparison between hardware acceleration methods

Performance of three spatial and temporal memory safety acceleration works, BOGO [14], the WatchdogLite (WDL) [12] and HWST128 (this work) are compared together. However, these three works are implemented on different architectures. Therefore, a direct comparison of the perf.oh is meaningless. Thus, we use the speedup factor to compare different architecture. Speedup factor is calculated using the speed (clock cycles) of the pure software memory safety SBCETS [5, 6] on each architecture as the dividend. The cycles of the hardware acceleration technique as the divisor to calculate how much improvement is seen by each technique. The calculation for speedup is shown in Eq. 8. The $SoftboundCETS_{cycle}$ and $Hardware\_Acceleration_{cycle}$ are the cycle count of program instrumented with SBCETS and hardware acceleration methods. Therefore,

$$speedup(\times) = \frac{SoftboundCETS_{cycle}}{Hardware\_Acceleration_{cycle}}. \quad (8)$$

The speedup factor of each technique is shown in Fig. 5. BOGO is based on IntelMPX [13] spatial protection acceleration with additional shadow trie support and sets the bound to zero when a pointer is freed, thus achieving partial temporal protection. The mean of IntelMPX speedup is around 1.52× faster than the baseline SBCETS in the prior study [22]. However, the additional temporal operations in BOGO impact the performance and reduce the speedup to 1.31×. WDL has two modes. The narrow mode uses the scalar operation to handle the metadata propagation, while the wide mode uses the AVX instruction to accelerate the metadata operations in vectors. WatchdogLite performs better than IntelMPX,

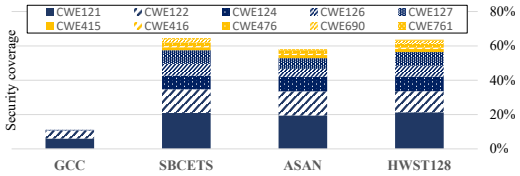

Figure 6: Security coverage of GCC, AddressSanitizer (ASAN), SoftboundCETS (SBCETS), and HWST128 on NIST Juliet suite

especially in SPEC bzip2 and hmmer. The mean speedup of WDL narrow and wide are 1.58× and 1.64×, respectively. Similar to the finding in BOGO, only six out of nineteen SPEC CPU2006 workloads instrumented with SBCETS can finish on x86 without error. The lbm in SPEC cannot finish on our RISC-V FPGA platform with SBCETS due to insufficient onboard memory.

In our experiments, the speedup of bzip and hmmer are high on the RISC-V platform. We run tests with CETS [6] instrumentation only (software temporal safety part of the SoftboundCETS) and found out that bzip, hmmer have reduced performance when compared to other SPEC workloads. However, the performance reduction in our temporal protection instrumentation with hardware acceleration is much less. The speedup in HWST128 of bzip and hmmer are 7.98× and 7.78×, respectively, when compared to SBCETS. The bzip and hmmer cases show how the hardware acceleration for temporal checking in our design can help to speedup the program. On average over number of benchmarks, HWST128 has a mean speedup of 3.74×.

5.2 Security coverage

For the security coverage, four protection/detection algorithms, including AddressSanitizer (ASAN) [8], SBCETS [5, 6], and HWST128 (this work) are evaluated with NIST Juliet suite on their published architecture. All test cases are compiled without optimization (-O0) as compiler optimization might remove attack vectors in Juliet test cases. The memory violation detection is done by parsing the output of the test case to observe if any violation is detected. The security coverage will be the total detected cases divided by the total number of Juliet cases (8366 cases). ASAN does not strictly enforce memory safety. However, ASAN can be used to detect various spatial and temporal violations, and therefore is listed in the evaluation. The default GCC-8.2.0-x86 is also included as a baseline reference for generic compiler protection.

Fig. 6 shows the coverage rate in Juliet test cases. The SBCETS covers 5395 cases (64.49%), ASAN covers 4859 cases (58.08%), HWST covers 5323 cases (63.63%), and GCC covers 937 cases (11.20%). All memory violation detection algorithms can effectively pick up more than half of the memory violations in the tests. When comparing HWST128 to SBCETS, HWST128 has fewer cases coverage in CWE122 Heap_Based_Buffer_Overflow, which leads to 0.86% less coverage than SBCETS. When comparing HWST128 to ASAN, HWST128 is slightly better. The major difference comes from CWE690 NULL_Deref_From_Return, where ASAN cannot detect any of the cases in this category.

5.3 Hardware cost

The hardware cost of HWST128 compared to the baseline Rocket Chip uses 1536 more LUTs (+4.11%) and 112 FFs (+0.66%). In addition, the timing of the critical path increased from 5.26ns to 6.45ns. The additional latency is caused by the bypass network (forwarding) in the metadata propagation.

6 CONCLUSION

In this paper, we present HWST128, a hardware/software co-design method to accelerate spatial and temporal safety on RISC-V. A novel, configurable metadata compression technique to fit the additional temporal metadata into the shadow register file is shown, leading to a minimal increase of hardware cost of 4.11% in LUTs and 0.06% in FFs. The compressed metadata reduces the performance impact on in-pipeline and through memory propagation. Combined with a keybuffer, HWST128 achieved 3.74× speedup over the software solution of SBCETS. HWST128 provides comparable spatial and temporal safety coverage in Juliet test suite as SBCETS and ASAN. HWST128 is the first complete solution for memory safety on RISC-V, providing a high-performance, low-cost, and FPGA-ready platform for memory security.

ACKNOWLEDGEMENT

This research was supported by the Australian Research Council's Discovery Projects funding scheme (project DP190103916). We would like to thank Defence Science and Technology Group Australia for their support.

REFERENCES

[1] Szekeres *et al.*, "Sok: Eternal war in memory," in *2013 IEEE Symposium on Security and Privacy*, pp. 48–62, IEEE, 2013.
[2] MITRE, "Cwe top 25 most dangerous software errors," 2021.
[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] H. Dow, T. Li, W. Miles, and S. Parameswaran, "SHORE: hardware/software method for memory safety acceleration on RISC-V," in *Design Automation Conference, 2021*, pp. 289–294, IEEE, 2021.
[8] K. Serebryany, D. Bruening, *et al.*, "Addresssanitizer: A fast address sanity checker," in *USENIX'12*, p. 28, USENIX Association, 2012.
[9] 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.
[10] ARM, "Arm memory tagging extension whitepape," 2019.
[11] Woodruff *et al.*, "The cheri capability model: Revisiting risc in an age of risk," in *ISCA'14*, pp. 457–468, IEEE, 2014.
[12] S. Nagarakatte *et al.*, "Watchdoglite: Hardware-accelerated compiler-based pointer checking," in *Proc. CGO*, 2014.
[13] R. Ramakesavan, D. Zimmerman, and P. Singaravelu, "Intel memory protection extensions (intel mpx) enabling guide," 2015.
[14] T. Zhang, D. Lee, and C. Jung, "Bogo: Buy spatial memory safety, get temporal memory safety (almost) free," in *ASPLOS'19*, pp. 631–644, 2019.
[15] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis transformation," in *Proc. CGO*, 2004.
[16] S. Das, Unnithan, *et al.*, "Shakti-ms: a risc-v processor for memory safety in c," in *Proc. LCTES*, pp. 19–32, ACM, 2019.
[17] K. Asanovic *et al.*, "The rocket chip generator," *EECS, UCB, Tech. Rep. UCB/EECS-2016-17*, 2016.
[18] C. Trippel, Y. A. Manerkar, D. Lustig, M. Pellauer, and M. Martonosi, "Tricheck: Memory model verification at the trisection of software, hardware, and isa," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 119–133, 2017.
[19] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, p. 1–17, Sept. 2006.
[20] T. Boland and P. E. Black, "Juliet 1. 1 c/c++ and java test suite," *Computer*, vol. 45, no. 10, pp. 88–90, 2012.
[21] Waterman *et al.*, "Spike RISC-V ISA simulator," 2016.
[22] Oleksenko *et al.*, "Intel mpx explained: A cross-layer analysis of the intel mpx system stack," *POMACS*, vol. 2, no. 2, p. 28, 2018.