HeapSafe: Securing Unprotected Heaps in RISC-V

Asmit De and Swaroop Ghosh, *Senior Member, IEEE*

***Abstract*—RISC-V is a promising open-source architecture primarily targeted for embedded systems. Programs compiled using the RISC-V toolchain can run bare-metal on the system, and, as such, can be vulnerable to several memory corruption vulnerabilities. In this work, we present HeapSafe, a lightweight hardware assisted heap-buffer protection scheme to mitigate heap overflow and use-after-free vulnerabilities in a RISC-V SoC. The proposed scheme tags pointers associated with heap buffers with metadata indices and enforces tag propagation for commonly used pointer operations. The HeapSafe hardware is decoupled from the core and is designed as a configurable coprocessor and is responsible for validating the heap buffer accesses. Benchmark results show a 1.5X performance overhead and 1.59% area overhead, while being 22% faster than a software protection. We further implemented a HeapSafe-nb, an asynchronous validation design, which improves performance by 27% over the synchronous HeapSafe.**

***Index Terms*—Buffer overflow, Use after free, Heap, RISC-V**

I. Introduction

PROGRAMMING languages such as C, which are closer to the hardware, allow direct access to memory and IO to facilitate system and device level programming. Such languages are weakly typed and While being flexible and powerful, this also leads to a plethora of vulnerabilities, if not used with proper practices. C allows memory access using pointers, which are essentially memory addresses that can be referenced and de-referenced for data access. Pointers are an extremely valuable construct as they allow programmers to dynamically allocate memory regions on-demand based on the program's requirement. This saves space and also allows efficient allocation of resources in memory. However, this can also lead to several memory vulnerabilities. Unfortunately, even with decades of research, memory corruption vulnerabilities are still prevalent in modern systems [1], [2].

A commonly exploited memory corruption vulnerability is buffer overflow [3] which occurs when a data is written to a buffer if the size of the data is more than the size of the available buffer. Buffer overflows can occur in a process's stack, or in the heap. A buffer overflow in a process's address space can lead to several exploits such as Control-Flow Integrity violations, Data-Flow Integrity violations, Return-oriented-Programming attacks [4], [5], [6], [7], etc. Although stack-based buffer overflows are more common, heap buffer overflows can also occur [8], [9], [10], and it is more difficult to protect against.

A heap buffer overflow occurs when a pointer used to write data to the buffer goes beyond the allocated region of the heap and overwrites the critical data, potentially allowing an adversary to launch attacks, such as, write-what-where, malicious shellcode execution, etc. This is possible due to the lack of bounds checking (a technique that allows validation of a pointer's access bounds) in pointers. Unlike a stack-buffer overflow based attack, which occurs in tandem with the rolling/unrolling of the process's stack, heap-buffer overflows are arbitrary and can happen anywhere at any point during the program's execution. Stack-buffer based overflows, and consequently ROP attacks can be mitigated using techniques such as stack canaries [11], shadow stacks [12], etc. However such techniques are not applicable to heap-based buffer overflows, since the allocated memory is dynamic and does not follow the process's stack frame, and as such, there are no return addresses to protect. This makes heap-buffer overflows much harder to detect and prevent.

Several techniques have been explored in literature to mitigate heap buffer overflows. Pointer protection is a common technique that provides protection to pointers based on bounds checking of allocated memory [13], [14]. Such techniques associate additional metadata structures with the pointers and provide software runtimes to perform access validation. A similar metadata association approach has also been explored for objects instead of pointers [15]. A better alternate approach to pointer-based protection is low-fat pointers [16], where instead of using additional metadata structures with pointers, the authors have utilized the native pointer itself for storing the metadata, thereby preserving backwards compatibility. A few hardware based memory allocation and runtime monitoring approaches have also been explored in [17], [18]. Recently, there has been some developments on a secure and memory safe processor [19], [20], which utilizes the fat-pointer scheme implemented in the architecture pipeline. The primary difference between Shakti-T [19] and our work is that, Shakti-T is a processor with memory safety built into the pipeline, and hence is not scalable or customizable. The metadata field width is also fixed. Our implementation is a decoupled implementation which can be attached to any RISC-V core interfacing with the RoCCIO. Since our implementation is on a custom coprocessor, the design can be tuned and scaled

TABLE I
Qualitative comparison of Heap Protection Approaches

Type	Compatibility	Completeness	Performance	Area
Fat-pointer	↓	↑	↓	↓
Low-fat pointer	↑	↓	↑	↓
Object tagging	↑	↓	↓	↓
Hardware Allocators	↑	↓	↑	↑
HeapSafe	↑	↑	↑	↑

A. De and S. Ghosh are with the School of Electrical Engineering and Computer Science, The Pennsylvania State University, University Park, PA, 16828 USA e-mail: asmit@psu.edu, szg212@psu.edu

Manuscript received March 18, 2021; revised March 18, 2021.

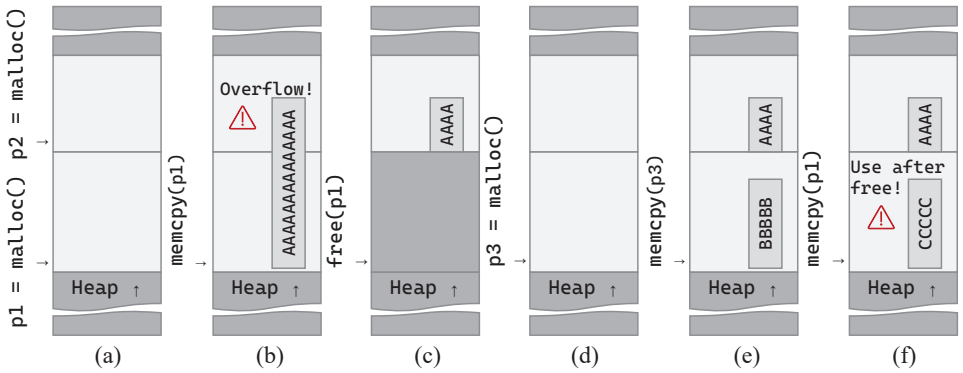


Fig. 1. (a) Dynamic memory allocation on a heap; (b) Buffer copy and resulting overflow; (c) Memory de-allocation; (d) New allocation in freed location; (e) New data copy to buffer; (f) Data corruption using dangling pointer (use-after-free).

according to needs, without modification of the core pipeline architecture. Table I provides a qualitative analysis of the different heap protection approaches.

RISC-V is a promising open source instruction set architecture (ISA) that can be adapted to SoC architectures targeting a varied range of applications such as, IoT devices, machine learning accelerators and even data-center microprocessors. It allows programs and applications to run bare-metal on the hardware for application specific scenarios. Such applications has complete access to the range of memory available in the hardware, and as such, can suffer from the same memory corruption vulnerabilities.

We propose HeapSafe, a hardware assisted heap protection engine built on the RocketChip SoC [21] running the RISC-V ISA. We leverage the Rocket Custom Coprocessor (RoCC) to design the HeapSafe module for protection from heap-buffer overflows and use-after-free attacks. Over a traditional software based approach, HeapSafe incurs no performance losses due to context switching or cache replacement, no manual secure memory management for bare-metal applications and no compiler dependent pointer analysis. HeapSafe is able to achieve high backwards compatibility and completeness, while retaining good performance.

II. Background

In this section, we explain two types of memory corruption attacks on the heap, against witch HeapSafe can be applied. We also describe the RISC-V RocketChip SoC platform and the Rocket Custom Coprocessor, which is our target implementation platform.

A. Heap attacks

A heap buffer is a memory space dynamically allocated on a process's heap. In the userspace, a heap is created using GNU C library (glibc) functions such as, `malloc()` or `calloc()` as shown in Fig. 1. The function returns the memory address of the first byte of the allocated space, and is used as the pointer to the heap. Data is written to the heap's allocated bytes with the pointer using glibc functions such as `memset()`, `memcpy()`, `strcpy()`. In the following paragraphs, we explain the basics of heap-based attacks using known vulnerabilities from the CWE database.

Buffer overflow (CWE-122): Fig. 2(a) shows a simple code demonstrating a buffer overflow vulnerability on the heap. Here, a heap buffer is allocated on the process's memory and a string from the command-line is copied to the buffer. However, it is easy to overflow the buffer if the size of the string is more than `SIZE`. This can potentially overwrite process data on the heap in other allocated buffers and may lead to memory exploits. The scope of such attacks is quite large, since, in real applications, a lot of complex constructs such as objects, structs, function pointers, etc. are allocated on the heap.

Use after free (CWE-416): This vulnerability occurs when, heap memory is reallocated after the data on a heap is freed. A previously leftover reference (dangling pointer) to that memory can potentially access newly created data from that heap location. Fig. 2(b) shows an example of this vulnerability. In this case, location pointed by `p1` gets freed if an error occurs. It is possible that when memory referred by `p2` is created, it is at the same freed location as `p1`. In such a situation, referring to `p1` later in the code can inadvertently leak or even corrupt data at that location.

B. RocketChip System and RoCC

The HeapSafe architecture is based on Rocket Chip (written in CHISEL), an open source parameterized system-on-chip (SoC) design generator. We use the RocketChip generator to generate synthesizable RTL for the standard Rocket Core SoC, a six-stage single-issue in-order pipeline processor that

```
#define SIZE 32

int main(int argc, char **argv)
{
  char *buffer;

  buf = (char*)malloc(SIZE);
  strcpy(buffer, argv[1]);

}
```

(a)

```
char* p1 = (char*)malloc(SIZE);
if (err) {
  abrt = 1;
  free(p1);
}
char* p2 = (char*)malloc(SIZE/2);
...
if (abrt) {
  logError("Aborted", p1);
}
```

(b)

Fig. 2. Vulnerable code showing (a) heap buffer overflow weakness; (b) use-after-free weakness.

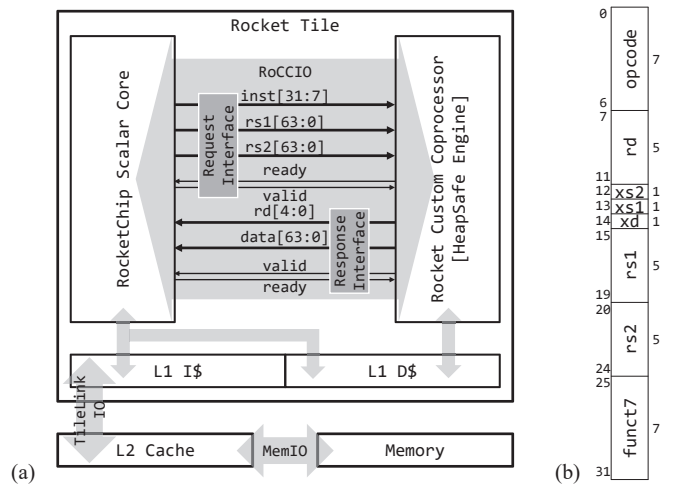


Fig. 3. (a) RocketChip SoC architecture with HeapSafe implemented RoCC; (b) RoCC instruction encoding.

executes the 64-bit scalar RISC-V ISA (Fig. 3(a)). The Rocket Tile consists of the scalar core, the L1 caches, and the Rocket Custom Coprocessor (RoCC). The RoCC is a user-defined accelerator for the core which communicates with core over the RoCCIO interface using a set of custom instructions.

RoCC Instructions: The 32-bit RoCC instructions extend the RISC-V ISA and are encoded as shown in Fig. 3(b). The four custom instructions supported by Rocket Chip are *custom0-3*, each having a different opcode. The *xs1*, *xs2*, and *xd* bits control read/write of the core registers by the RoCC instruction. If *xs1* is 1, the 64-bit value in the register specified by *rs1* is passed to the RoCC. Similarly, *xs2* bit controls the read of register specified by *rs2*. If *xd* bit is 1 and *rd* is not 0, the core will wait for a value to be returned by the coprocessor over RoCCIO after issuing the instruction to the coprocessor. The value is then written to the register specified by *rd*. If the *xd* is 0 or *rd* is 0, the core will not wait for a value from RoCC. The opcode field specifies the custom instruction for the RoCC, and the *funct7* field further specifies a user-defined function implemented in the RoCC. The RoCC is responsible for signaling illegal instructions to the core.

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

III. HeapSafe Implementation for RISC-V

In this section we describe the implementation of the HeapSafe engine, the associated HeapSafe library and scope of protection. We also explain the usage of HeapSafe in standard C programs for heap buffer protection.

A. HeapSafe protection scope

In this work, we aim to protect dynamically allocated buffers on the heap that are accessed using pointers derived from the allocation pointer. We enforce metadata propagation between pointers, and the system is able to trace the correct metadata for all pointer arithmetic operations. HeapSafe is able to protect against buffer overflow on heap, and also prevent inadvertent use-after-free accesses. Any program targeted for the RISC-V system can be updated to use the safe heap functions from the HeapSafe library. Each pointer used to allocate a heap buffer will be converted to a *safe_pointer* by the HeapSafe library. Any other pointers derived from the *safe_pointer* is also tagged as a *safe_pointer*. The tag is an identifier for the pointer that is encoded in the higher order bits of the pointer. We enforce tag propagation between pointers for all pointer assignments and pointer arithmetic operations. This allows us to propagate pointer metadata information across pointers. The program is compiled by including the HeapSafe library and while running on the core, the HeapSafe hardware is responsible for storing and validating heap metadata.

TABLE II
HeapSafe Tag Propagation

Case	Code
Memory allocation	`safe_ptr = safe_malloc(size);`
Assignment	`safe_ptr2 = safe_ptr1;`
Pointer Arithmetic	`safe_ptr2 = safe_ptr1 + offset;` `safe_ptr2 = safe_ptr1 - offset;`
Type cast	`safe_ptr2 = (type*) safe_ptr1;`

Since we are reusing the same pointer to store the tag, referencing pointers is trivial, without having to process the pointer information. This also allows us to easily enforce tag propagation in the following scenarios (Table II):

(a) *Memory allocation:* When allocating a heap buffer, a new *tag* is generated. The *safe_pointer* is created using the *tag* and the base pointer (*raw_pointer*).

(b) *Pointer assignment:* During a pointer assignment, the *tag* from the original *safe_pointer* is propagated to the new *safe_pointer* alongside the *raw_pointer* value.

(c) *Pointer arithmetic:* During a pointer arithmetic operation such as array access at a specific index, the new *safe_pointer* created by adding/subtracting the offset receives the same *tag* as the original *safe_pointer*.

(d) *Pointer type conversion:* When a *safe_pointer* is cast to a new type, the *tag* is propagated to the *safe_pointer* of the new type.

Aside from these four cases, storing and retrieving *safe_pointers* from memory are trivial and same as storing and retrieving normal pointers from memory. The *tag* in the *safe_pointer* is retained throughout the store and retrieve operations. Passing *safe_pointers* as function arguments and returning *safe_pointers* from functions are also same as with normal pointers, and the*tag* is retained in the process. Two other cases that need special mention are the null pointer and manual pointer creation from integer values. In both cases, the *tag* is set to 0. The *tag* 0 is also indicates that the pointer is not protected and any safe heap operations using the HeapSafe library with the pointer will result in an error.

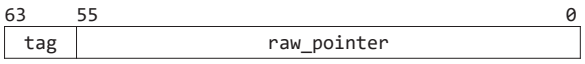


Fig. 4. *safe_pointer* bit allocation scheme.

B. HeapSafe library

The HeapSafe protection engine is accompanied by a library containing safe implementations of critical heap buffer functions such as `safe_malloc()`, `safe_copy()`, `safe_free()` and `safe_read()` / `safe_write()`, that utilize the HeapSafe hardware. We describe the operation of these helper functions below.

1) ***`safe_malloc()`****:* This function allocates a buffer in the process's heap similar to `malloc()` in the GNU C library. The allocated memory is referred to by the address of the first byte of the heap called the *raw_pointer*. We create a *safe_pointer* from the *raw_pointer* by using the top most significant byte as a tag reference. The bit allocation is shown in Fig. 4. We assign the *tag* from a static list of available tags, which are local to the process. Since we are using 1 byte to represent the *tag*, we can use HeapSafe for a maximum of 255 simultaneous heap allocations in the process. We exclude 0 as a *tag* to maintain compatibility with pointers that are not using the HeapSafe engine. Furthermore, this also excludes the higher order 256 byte memory region in the address space, however, this allocation scheme is sufficient for standard RISC-V applications running bare-metal. For RISC-V systems with user-space and kernel-space separation in memory, HeapSafe is able to protect user-space processes only. It is to be noted that, even though we have used 1 byte for representing the *tag*, this bit allocation is customizable in the library. While compiling the HeapSafe library, the bit allocation for the *tag* can be set as required to match with the HeapSafe hardware (details in Section V-E).

After allocating the *tag*, we send a custom RoCC instruction `HS_STORE` to the HeapSafe hardware to write the pointer metadata. We send the *safe_pointer* and the size of the allocated buffer encoded in the RoCC instruction.

The `HS_STORE` instruction is crafted as follows: The *opcode* is set to *custom0* (b'0001011), *rs1* is set to the register containing the *safe_pointer*, *rs2* is set to the register containing the size of the heap buffer, *xs1* and *xs2* fields are set to 1, and *funct7* is set to *hs_store* (b'0000000). The instruction is non-blocking, and the program proceeds without waiting for a response from the HeapSafe engine.

2) ***`safe_copy()`****:* This function enables a safe copy operation from a source buffer to the destination buffer which guarantees that the buffer will not be overflowed. To perform the copy operation, we check the pointer being used to refer to the destination heap buffer. We send the pointer with the `HS_VALIDATE` instruction to the HeapSafe engine to perform an out-of-bounds validation.

The `HS_VALIDATE` instruction is crafted as follows: The *opcode* is set to *custom0* (b'0001011), *rs1* is set to the register containing the pointer, *rd* is set to a register to receive the validation outcome, *xs1* and *xd* fields are set to 1, and *funct7* is set to *hs_validate* (b'0000001).

After performing the out-of-bounds validation, the HeapSafe engine returns a 0 or 1 indicating in-bounds or out-of-bounds respectively. The `safe_copy()` function can then proceed or halt based on the validation outcome.

3) ***`safe_free()`****:* This function is complementary to `safe_malloc()` which allows a clean de-allocation of the memory space and metadata removal from the HeapSafe engine. We first parse the *safe_pointer* to extract the *raw_pointer*. We then send the *safe_pointer* to the HeapSafe engine with `HS_FREE` instruction to perform the metadata removal.

The `HS_FREE` instruction is crafted as follows: The *opcode* is set to *custom0* (b'0001011), *rs1* is set to the register containing the *safe_pointer*, *xs1* is set to 1 and the *funct7* field is set to *hs_free* (b'0000011). The instruction is non-blocking, so the program continues to execute on the core without waiting for a response from the HeapSafe engine. After sending the `HS_FREE` instruction, the memory de-allocation is performed normally, similar to `free()` in glibc.

4) ***`safe_read()` / `safe_write()`****:* By re-purposing the MSB bits of the original pointer to store the *tag*, we get the benefit of enforcing easy tag propagation. However, it precludes us from performing pointer de-referencing to read/write data in the standard way. This is because, the *safe_pointer* by itself is not a valid memory address due to the inclusion of the *tag* bits, and de-referencing in the usual way, e.g., `data = *safe_ptr;` will raise a memory access exception. To circumvent this issue, we have also provided `safe_read()` and `safe_write()` functions, that can safely extract the *raw_pointer* from the *safe_pointer*. It then sends a `HS_VALIDATE` instruction to HeapSafe engine to validate the read/write access, and then performs the de-referencing to read/write data in memory based on the *raw_pointer*:

```
addr = extractRawPointer(safe_ptr);
data = *addr; // For read
*addr = data; // For write
```

C. HeapSafe hardware

The HeapSafe engine is a custom designed accelerator that is decoupled from the processor core and connected over the RoCCIO interface. The engine consists of a metadata parser, a metadata table, and a validation engine as shown in Fig. 5.

The HeapSafe engine receives commands over the RoCCIO request interface. The Cmd Decoder decodes the RoCC instruction to read the *opcode*, the *rs1* an *rs2* data fields, and the *funct7* function field, and asserts the required control signals. In our implementation the opcode field is always decoded to *custom0*. The *rs1* and *rs2* data fields contains pointer metadata as requires for a specific function. The *funct7* field is decoded to functions such as, *hs_store*, *hs_validate* and *hs_free*.

***hs_store*:** This function indicates a heap buffer creation and instructs the HeapSafe engine to store the associated metadata which is received in the form of the *safe_pointer* on the rs1 field and the size on the rs2 field. The metadata parser processes the *safe_pointer* and extracts the tag and the *raw_pointer* values based on the specified bit encoding.

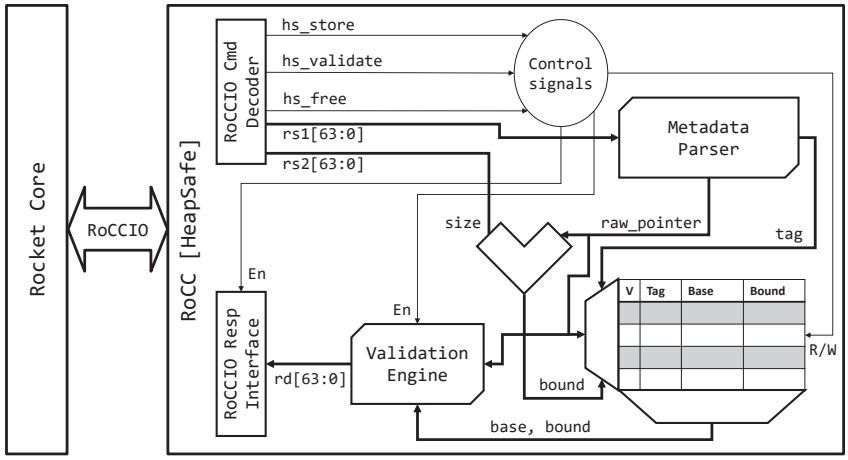


Fig. 5. HeapSafe architecture in RoCC.

The metadata table is implemented as a hardware content-addressable memory for parallel search and fast lookup. The table consists of three fields for storing the metadata - (i) Tag, (ii) Base, and (iii) Bound. Each row in the table is designed as a vector of the three 64-bit wide fields. In addition to the metadata, each row in the table also contains a valid bit to indicate the validity of the metadata. The size of the table is customizable as part of the design and can be set while instantiating the hardware. In our implementation for testing, we have used a table consisting of 256 rows, allowing metadata storage for a maximum of 256 heap buffers.

A write signal is automatically issued on decoding the *hs_store* as the function, which writes the pointer metadata at an available location in the metadata table. These locations are indicated by the valid bit set to 0. The parsed tag is stored in the Tag field and the raw_pointer is stored in the Base field. Instead of storing the size of the heap buffer, we pre-compute the bound address value as:

$$Bound[Tag] = raw\_pointer + size \tag{1}$$

Since the metadata store instruction is non-blocking, we improve performance by saving the calculated bound address value in the Bound field. Finally, the valid bit is set to 1 to mark the row as active.

***hs_validate*:** This function indicates a heap buffer write operation on the core and instructs the HeafSafe engine to validate the pointer's access bounds. The pointer being used to access the heap is received on the *rs1* field. The metadata parser processes the pointer on the *rs1* field and extracts the current *tag* and the *raw_pointer* (memory address) values.

A read signal is issued on decoding the *hs_validate* as the function, which performs a parallel search of the current *tag* on the *Tag* field of the metadata table. Once a tag match is found, the *Base* and *Bound* fields are read. The access bounds for the current pointer (*ptr*) is validated as:

$$isOOB = (ptr < Base) \parallel (ptr \geq Bound) \tag{2}$$

The out-of-bound signal (*isOOB*) is asserted when the current pointer (memory address) is either less than the lower bound (base), or is greater than or equal to the upper bound of the heap buffer. The value of the *isOOB* signal is held at 0 if the current pointer is within bounds. Since the validation is a blocking operation by default, having the upper bound address of the heap buffer in the metadata table speeds up the validation time. The value of *isOOB* signal is placed on the *rd* field of the response interface and is sent back to HeapSafe library to take the required action - (i) proceed or (ii) terminate.

***hs_free*:** This function indicates a heap buffer de-allocation on the core, and instructs the HeapSafe engine to clear its corresponding metadata. However, instead of removing or zero-izing the metadata from the table, we set the valid bit for the row to 0 to invalidate the metadata entry and mark it as available for future use. The *hs_free* operation is non-blocking and the program on the core continues to run without waiting for a response from the HeapSafe engine. Invalidating metadata entries allow us to mitigate inadvertent use-after-free vulnerabilities, since the *tag* for the dangling pointer will be invalidated after free.

D. HeapSafe usage in C programs

We demonstrate a basic use of HeapSafe in a simple C program (Fig. 6a). Let us consider a function that receives a lowercase string data, converts each character to uppercase and stores to a buffer allocated on the heap. The function then returns the pointer to the heap buffer storing the uppercase string.

In this program, the `upper` buffer is vulnerable to overflow, and hence, let us modify the code to use HeapSafe to protect the buffer. We assume that `lower` buffer doesn't need to be protected. We perform the modification as follows:

We first replace the `malloc()` function call with `safe_malloc()`, that returns a tagged `safe_pointer` to `upper`. When iterating over the characters from the source buffer, after converting to lowercase, we need to de-reference the `upper` pointer to store the uppercase character. We modify the de-referencing code with the `safe_write()` function that performs the correct write to memory operation. Towards

```
char* convert_case(char *lower)
{
  char *upper;
  upper = malloc(SIZE);
  while (*lower != '\0') {
    char u = *lower - 32;
    *upper = u;
    upper++;
    lower++;
  }
  return upper;
}
```

(a)

```
char* convert_case(char *lower)
{
  char *upper;
  upper = safe_malloc(SIZE);
  while (*lower != '\0') {
    char u = *lower - 32;
    safe_write(upper, u);
    upper++;
    lower++;
  }
  return upper;
}
```

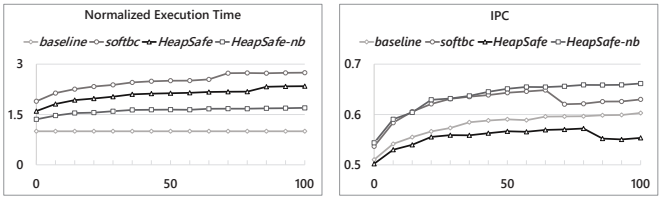
(b)

Fig. 6. (a) Source code with heap buffer overflow vulnerability; (b) HeapSafe protected code to prevent overflow.

the end of the loop, we update the destination heap pointer (`upper`) by increasing the pointer by 1. The pointer arithmetic propagates the original tag for `upper` to the new `upper`. The HeapSafe protected code is listed in Fig. 6b.

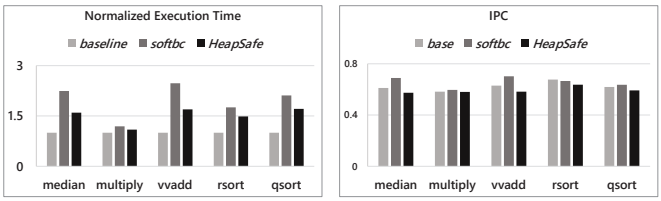
IV. Evaluation

We evaluated HeapSafe by generating a RocketChip SoC design config with the HeapSafe module. We tested the HeapSafe security architecture in the C++ cycle accurate emulator built from the config. The hardware architecture of HeapSafe is coded in CHISEL and synthesizable verilog is generated using the RocketChip generator. To evaluate performance, we created sample workloads that perform multiple buffer copy operations on the process's stack and heap. We compiled three versions of the code: (i) *baseline* with no protection, (ii) *softbc* with in-process software-based bounds checking, and (iii) *HeapSafe* with our HeapSafe library and protections. We swept the workload balance between the stack and the heap and evaluated the trend as shown in Fig. 7. We note that *HeapSafe*'s execution time overhead is low and similar to *softbc* when there are more stack workloads; however, as the heap workload increases compared to the stack workload, *HeapSafe* tends to perform better. At around 75% heap workload, *HeapSafe* performs 20% faster than *softbc*. However, instructions-per-cycle (IPC) suffers in *HeapSafe* compared to *softbc*, since *HeapSafe* runs less instructions in total. We have also implemented a fully non-blocking version of HeapSafe (details in Section V-A) which outperforms *softbc* in both execution time and IPC at the cost of delayed heap corruption detection. At high heap workloads, *HeapSafe-nb* is 38% faster than *softbc*. We estimated the area of HeapSafe by generating a E300 Arty FPGA bitstream and found it to have a nominal 1.59% overhead (number of cells) over the default configuration. We further updated the RISC-V ISA test benchmarks with *HeapSafe* and *softbc* and evaluated their performance (Fig. 8). *HeapSafe* incurs a 1.5X overhead over *baseline* on average, while being 22.4% faster than *softbc*. Average IPC is slightly low at 0.59 compared to 0.62/0.65 (*baseline*/*softbc*). A qualitative evaluation of HeapSafe has been shown in Table I as well.



(a) (b)

Fig. 7. (a) Execution time trend normalized to baseline; (b) IPC trend. X-axis represents the percentage of buffer copy operations occurring on the heap.



(a) (b)

Fig. 8. (a) Execution time trend normalized to baseline; (b) IPC trend. X-axis represents the percentage of buffer copy operations occurring on the heap.

V. Discussions

A. HeapSafe design improvements

HeapSafe implements fine-grained approach for instant detection of heap overflow at the cost of some performance penalties. However, such high security guarantees are not needed in less critical systems. We propose the following alternate flavors of HeapSafe system design:

1) Non-blocking validation: The validation mechanism with the `HS_VALIDATE` instruction can be made non-blocking to avoid wait by the core for a response from HeapSafe. In this design, HeapSafe will asynchronously raise an exception on the core when it detects an out-of-bounds error (instead of sending the value of the *isOOB* signal on the response interface). An exception handler is implemented in the HeapSafe library will terminate the application in such cases. This approach will improve performance by 27.6% (Fig. 7(a)) at the cost of slight delay in attack detection.

2) Compiler support: HeapSafe hardware is currently complemented by the HeapSafe library to replace unsafe heap operations with safe variants. Replacement of unsafe heap operations with safe variants using library can be automated by adding compiler support. In this design, we compile the source program to be protected using LLVM/Clang for RISC-V. The LLVM generates an intermediate representation (IR) of the source code. An LLVM compiler pass is written to parse the IR to scan for heap pointers and replace them with the tagged *safe_pointers*. The custom HeapSafe instructions are also inserted for the required operations to communicate with the HeapSafe engine. The IR is then compiled to generate the ELF binary to run on the system. This improvement is more design friendly to the source code programmer since the code does not need to be explicitly updated to use the HeapSafe engine.

3) Top byte ignore: Pointer de-referencing requires additional library functions to perform the correct pointer extrac-

tion in HeapSafe. This can be avoided if the core is set to ignore the top byte for any memory address in the user-space. This can be achieved by conditionally masking the top byte of the address in the address decoder in the core pipeline. This will guarantee that any tagged *safe_pointer* is seen as a normal *raw_pointer* in the pipeline, and all load/store operations will automatically be performed using the *raw_pointer*.

4) Multi-process support: Our current HeapSafe implementation is targeted to protect a single process, or a process running bare-metal on the system. However, due to the decoupled coprocessor based design, HeapSafe can be scaled up to support protection of multiple processes simultaneously. RISC-V cores view each running process as a hardware thread (hart). In the scaled up implementation, multiple instances of the HeapSafe coprocessor is instantiated in the same tile along with the core. Each instance of HeapSafe engine is associated with a `hartId`. When a process using the HeapSafe library is running on the core, the system hardware selects the specific HeapSafe engine to use with the hart for that process.

B. Security of HeapSafe hardware

In order for HeapSafe to guarantee protection, it needs to ensure that the HeapSafe hardware is not compromised. Hence, we need to ensure the integrity of the RoCC HeapSafe engine, so that no malicious code can overwrite metadata entries in the metadata table. This is guaranteed to some extent by appropriately setting the RocketChip configuration to run RoCC operations in machine (M) mode only, while rest of the code runs in user (S) mode. This prevents any malicious code to run the RoCC instructions while in user mode. The security of the hardware can be further improved by running the HeapSafe library functions in machine mode only. This requires some additional mediation logic in the application code utilizing traps that requires a switch to machine mode from user mode when calling a HeapSafe function, and then exit to user mode after returning from the function. This can mitigate code-reuse attacks that might try to run the HeapSafe library functions maliciously.

C. PMP vs. HeapSafe

The RISC-V architecture provides some basic memory protection as part of the ISA. There are 16 Physical Memory Protection (PMP) registers in the base architecture, which can be utilized to perform access control on different memory regions. The PMP registers allow machine (M) mode to specify which memory regions are available during user (U) mode operations. This is an easy and low-cost way to implement memory protection in user mode for simple systems. However, due to the limited number of registers available, it imposes a restriction on the number of regions it can protect. Hence this is not scalable to more complex applications. Furthermore, PMP protected regions need to be contiguous in physical memory, it can lead to memory fragmentation.

In contrast, HeapSafe can be applied in a more granular manner to individual pointers pointing to memory locations. The number of regions to be protected is not restricted by the core architecture, but set by the configurable HeapSafe engine. Thus HeapSafe is more scalable and versatile than PMP.

```
class WithNHeapSafe(n: Int) extends Config((site, here, up) => {
  case BuildRoCC => List.tabulate(n)( i =>
    (p: Parameters) => {
      val heapsafe = LazyModule(new HeapSafe(OpcodeSet.custom0,
                                             mtSize = 256,
                                             hartId = i)(p))
      heapsafe
})})
```

Fig. 9. RocketChip config class for configurable HeapSafe module generation.

D. Backwards compatibility

Our HeapSafe implementation is fully backwards compatible with standard unprotected pointers. This allows the source code programmer to use a mix of protected and unprotected pointers. The programmer can opt to use HeapSafe protected pointers only for security critical heap regions. Although this is less secure, it improves the performance since the program is not being slowed down due to unnecessary validations.

We achieve backwards compatibility by assuming a *tag* value of 0, since pointers in user-level code has the MSB bits as 0. Since this is inherent to the design, we simply exclude 0 as a *tag* for HeapSafe pointers. If we encounter a pointer with its *tag* bits as 0, we treat the pointer as a non-protected pointer and exclude it from validation.

E. HeapSafe system generation

Due to the flexibility in HeapSafe's design configuration, we can customize the size of the metadata table to be generated on the hardware. We can also create multiple instances of the HeapSafe coprocessor to support simultaneous multi-process protection. The code in Fig. 9 demonstrates the easy configurability of HeapSafe design.

When generating a system configuration with HeapSafe, we can set the parameter `n` to specify the number of HeapSafe modules to instantiate. We associate a `hartId` to each HeapSafe module. The size of the metadata table can be set through the `mtSize` parameter. In this config, it is set to 256. In the HeapSafe module implementation, we calculate the bit allocation scheme for the tag in the *safe_pointer* as $\log_2 mtSize$, e.g., if `mtSize = 256`, we set the tag bit allocation to the MSB 8 bits in the *safe_pointer*.

During the SoC generation, the HeapSafe configuration changes are independent from the core configurations and the rest of the SoC configuration. This provides an advantage that the HeapSafe module can be hooked up to any RISC-V core configuration implementing the RoCCIO interface. As long as the application running on the core is compiled including the HeapSafe library, protection can be provided by the HeapSafe hardware.

VI. Conclusion

We presented HeapSafe, a customizable and lightweight heap protection hardware engine for the RISC-V ISA. We ensured heap integrity using tagged pointers and enforced metadata propagation for common pointer operations. HeapSafe improved performance over traditional software approaches. The design allows easy configuration and scalability of the security architecture implementation.

REFERENCES

[1] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *2013 IEEE Symposium on Security and Privacy*, pp. 48–62, IEEE, 2013.

[2] V. Van der Veen, L. Cavallaro, H. Bos, *et al.*, "Memory errors: The past, the present, and the future," in *International Workshop on Recent Advances in Intrusion Detection*, pp. 86–106, Springer, 2012.

[3] J. Deckard, *Buffer overflow attacks: detect, exploit, prevent*. Elsevier, 2005.

[4] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, pp. 1–40, 2009.

[5] W. Wu, Y. Chen, X. Xing, and W. Zou, "{KEPLER}: Facilitating control-flow hijacking primitive evaluation for linux kernel vulnerabilities," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pp. 1187–1204, 2019.

[6] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *Proceedings of the 7th symposium on Operating systems design and implementation*, pp. 147–160, 2006.

[7] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 15, no. 1, pp. 1–34, 2012.

[8] M. Daniel, J. Honoroff, and C. Miller, "Engineering heap overflow exploits with javascript.," *WOOT*, vol. 8, pp. 1–6, 2008.

[9] N. Huang, S. Huang, and C. Chang, "Analysis to heap overflow exploit in linux with symbolic execution," in *IOP Conference Series: Earth and Environmental Science*, vol. 252, p. 042100, IOP Publishing, 2019.

[10] B. Hawkes, "Attacking the vista heap," *Blackhat USA.(Aug. 2008)*, 2008.

[11] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks.," in *USENIX security symposium*, vol. 98, pp. 63–78, San Antonio, TX, 1998.

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

[13] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "Ccured: Type-safe retrofitting of legacy software," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 27, no. 3, pp. 477–526, 2005.

[14] D. Grossman, M. Hicks, T. Jim, and G. Morrisett, "Cyclone: A type-safe dialect of c," *C/C++ Users Journal*, vol. 23, no. 1, pp. 112–139, 2005.

[15] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors.," in *USENIX Security Symposium*, pp. 51–66, 2009.

[16] G. J. Duck and R. H. Yap, "Heap bounds protection with low fat pointers," in *Proceedings of the 25th International Conference on Compiler Construction*, pp. 132–142, 2016.

[17] O. Arias, D. Sullivan, and Y. Jin, "Ha2lloc: Hardware-assisted secure allocator," in *Proceedings of the Hardware and Architectural Support for Security and Privacy*, pp. 1–7, 2017.

[18] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha, "Secure embedded processing through hardware-assisted run-time monitoring," in *Design, Automation and Test in Europe*, pp. 178–183, IEEE, 2005.

[19] A. Menon, S. Murugan, C. Rebeiro, N. Gala, and K. Veezhinathan, "Shakti-t: A risc-v processor with light weight security extensions," in *Proceedings of the Hardware and Architectural Support for Security and Privacy*, HASP '17, (New York, NY, USA), Association for Computing Machinery, 2017.

[20] S. Das, R. H. Unnithan, A. Menon, C. Rebeiro, and K. Veezhinathan, "Shakti-ms: A risc-v processor for memory safety in c," in *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES 2019, (New York, NY, USA), p. 19–32, Association for Computing Machinery, 2019.

[21] Y. Lee, "Risc-v rocket chip soc generator in chisel," in *Online slides:" https://riscv. org/wp-content/uploads/2015/01/riscv-rocket-chip-generator-workshop-jan2015. pdf*, 2015.