CHERI-Lite for Memory Safety Exploit Mitigation

Tony Chen (tonychen@microsoft.com)

David Chisnall (David.Chisnall@scisemi.com)

1 Summary

Memory safety attacks remain a significant threat to modern computer systems, primarily due to the widespread use of unsafe C/C++ code, which is prone to buffer overruns, use-after-free, and type confusion vulnerabilities. This paper introduces a modification to the CHERI [9] architecture, termed "CHERI-Lite," which aims to mitigate these memory safety exploits. Unlike the original CHERI architecture that uses 128-bit pointers, CHERI-Lite retains 64-bit pointers, making it a more lightweight solution. The proposal involves adding an extra CHERI tag bit and utilizing the top 8 bits of the 64-bit pointer to store CHERI-like permission information. However, due to the limited number of bits available for permissions, CHERI-Lite does not include pointer bounds information, which means pointers can still be modified to point anywhere in the virtual address space. Consequently, while CHERI-Lite offers security benefits, it is not as robust as the full CHERI architecture.

CHERI-Lite should be viewed as memory safety exploit mitigation technology rather than an approach that guarantees memory safety. The security value of CHERI-Lite is to prevent memory safety attackers from directly fabricating pointers at will. Instead, with CHERI-Lite enabled, the memory safety attacker would need to find gadgets that add corrupted integer values into existing pointers and then try to use those pointers. Furthermore, these pointer modification gadgets must be naturally callable since the attacker would need to modify code pointers to perform ROP and JOP to call into unnaturally entry points, and the attacker is still in the process of finding a way to modify pointers. This means that bounds checking code at the beginning of functions cannot be skipped over which makes it much harder to find gadgets that are useful to the attacker. Unlike full CHERI, the architecture changes for CHERI-Lite is designed to be compatible with existing binary code and could enable large numbers of existing applications to gain some memory safety benefits without the need to even recompile.

This document describes the CHERI-Lite proposal and how it can be applied to the ARM64 architecture, although it could also be applied to other architectures. It assumes the reader is already familiar with CHERI and won't re-explain CHERI concepts. In section 2, we give an overview of the proposal and how we leverage the top 8 bits of the pointer to hold meaningful permission to enhance security as much as possible. We also explain how coarse grain partitions work and how it enables sealed pointers. In section 3, we discuss the actual instruction set changes needed to support CHERI-Lite. In section 4, we do a security analysis of this feature and why we believe it can still greatly enhance security and prevents exploitation of memory safety bugs despite not being able to enforce pointer bounds checking. In section 5, we talk about some practical deployment and implementation issues for CHERI-Lite and end with related work and conclusions.

2 Overview of CHERI-Lite

Of all the hardware architecture change proposals to solve memory safety, CHERI [9] is widely accepted to be the most secure and robust solutions proposed in the literature. However, it's been 15 years since the original CHERI paper from Cambridge University, and we still do not have any adoption of CHERI on any mainstream application cores used in computers/tablets/phones today. There are many reasons why adoption of CHERI has been held back, but the main reasons seem to be:

1. Because pointers double in size, the ABI of operating systems would need to change which constitutes the release of a new platform and the need to build up a new application ecosystem on the new platform.
2. Because the size of pointers is doubled, memory usage will increase, especially for applications that use pointers heavily.
3. Native code would need to be recompiled to support CHERI and there are no memory safety gains on existing binaries.
4. The amount of software code changes needed to support full CHERI is rather large and until convinced that improving memory safety is important, platform vendors rather invest the time and energy on other features that will have a better return on investment.

The motivation behind CHERI-Lite is to see if we can gain the resistance to pointer-injection of CHERI by retaining the tag bit to mark pointers as well as some permission bits but forgo the bounds checking and thus keep pointers at 64 bits. By doing so, we hopefully remove some of the reasons CHERI has not been used to hopefully gain adoption of CHERI-Lite. The authors would like to make it clear that we prefer the full adoption of CHERI. CHERI has much stronger memory safety guarantees, can support temporal memory safety, and enables compartments which has strong isolation guarantees and tons of useful applications. We are only proposing the CHERI-Lite idea because of our frustration that full CHERI has not been adopted and would rather see something significant to be done for memory safety instead of nothing being done.

CHERI-Lite uses the tag bit just like full CHERI and requires the tag bit to be maintained for the 64-bit X registers as well as in all of DRAM and DRAM caches. CHERI-Lite enforces the same basic principles of CHERI:

- A pointer (with tag bit set) is required to perform load, store, and instruction fetch.
- Instructions exist to remove a pointer's permissions, but there is no way to add permissions.
- Pointers can only be derived from another pointer and cannot be created out of thin air.
- Pointers lose their status as a pointer if the pointer is corrupted.

Since the ARM64 architecture already supports Top Byte Ignore (TBI) for addresses, we propose carving out the top 8 bits of a CHERI-Lite 64-bit pointer to hold permission information. There are many ways to use these 8 bits to store meaningful information, and we anticipate more ideas than what is proposed in this paper to emerge after more debate. But to get things started, this paper proposes the following usage of 5 of these 8 bits:

- Pointer Type (3 bits): These 3 bits describe what the 56-bit pointer can be used to do. While 3 bits can support up to 8 possible types, for now, we just propose the following 7 types:
    - ReadExecute: These pointers can be used for load instructions and instruction-fetch. They can be used as the destination for a branch instruction, but not a RET instruction.
    - ReadExecuteRET: These pointers can be used for load instructions and instruction-fetch. They are generated into the LR register during Branch and Link instructions. They can be used as a destination by the RET instruction, but not any branch instruction. The separation of forward and backward code pointers guarantees that they cannot be maliciously used in the other direction.
    - ReadWriteExecute: These pointers are the same as ReadExecute except that store instructions are also allowed on these types of pointers.
    - ReadWriteExecuteRET: These pointers are the same as ReadExecuteRET except that store instructions are also allowed on these types of pointers
    - ReadOnly: These pointers can only be used for load operations.
    - ReadWrite: These pointers can be used for both load and store operations.
    - Protected Data: The 56-bit value is not really a pointer but just protected data. The 56-bit value cannot be used as address for instruction fetch, nor load and store operations. The data is protected in the

sense that it requires a special instruction to set and get this data, and the tag bit will be lost if regular data instructions are used to manipulate this data.

- Locked (1 bit): When this bit is set, the 56-bit pointer value is locked and cannot be modified through any ALU (e.g. add subtract) instructions. This bit can be set to prevent the movement of the pointer in cases where the compiler or loader sees no reason the target pointer should ever be modified to a different value. An example of a locked pointer could be the address taken of a simple data type that is not in an array. We believe many code pointers in memory could have the locked bit set. Note that the Locked bit is cleared when the pointer is loaded into the PC register.
- Sealed (1 bit): A sealed pointer cannot be modified or even used for load/store/instruction fetch. More details about how sealing works will be explained after we introduce the concept of “partitions”.

Note that this leaves 3 more bits that can be used for other purposes. One possible use for these 3 bits is to use them as tag bits for MTE. This enables CHERI-Lite to be used in conjunction with MTE. This will be discussed later in the doc.

With this CHERI-Lite design, we have a 56-bit virtual address space available to use. We can further enhance CHERI-Lite by breaking this large virtual address space into separate smaller virtual address spaces by using the high order bits of the address to denote which “partition” an address is in. For example, if we take the 8-bits between bit 48 through 55 of the address and use it as a “partition ID”, we can break the 56-bit address space into 256 separate partitions, with each partition having 48-bits of virtual address space. For the rest of this paper, we will discuss the partitioning feature assuming 8-bits are used for the partition ID. But just keep in mind that the number of bits used for partitioning could by other values (e.g. 4 or 12) or even be dynamic. We also recommend that the number of bits used for partitioning (as well as whether partitioning is enabled at all) be changeable on a per EL0 process level.

The rule that is enforced when this partition feature is enabled is that pointers cannot move across a partition boundary. In other words, when you add or subtract a value to a pointer, the pointer can never change the top 8-bits of the virtual address space. We can add a large (say 4GB) unused region of virtual address space at the beginning and end of each partition to guarantee that pointers cannot temporarily cross a partition boundary by mistake. This partition feature guarantees that a maliciously modified pointer in one partition can never be modified to reach into another partition. The sealing bit in the permission field can now be used due to the existence of partitions:

- A pointer can be sealed and unsealed if the 8-bit partition ID in the current PC matches the 8-bit partition ID of the pointer being sealed or unsealed.
- This enables a partition to seal and unseal its own pointers and then pass these sealed pointers to other partitions to use as an opaque token.
- Code running in other partitions would not be able to modify sealed pointers from another partition.

There are many use cases for this partitions and sealed pointers. We enumerate a few that we can think of here:

- Using the sealing feature, a heap manager can have its own partition to hold its code and heap related metadata but then use other partitions for the actual heap allocations. It can then place sealed pointers of the heap manager partition into the heap partitions in front of every allocation. By doing this, it makes heap metadata corruption by any buffer overruns or use after free bugs on the heap detectable. It also makes it such that linear overruns that corrupt the heap across allocations also corrupt the sealed pointer and thus become detectable by the heap manager once it tries to unseal that pointer.
- In EL1 and EL2, the memory manager that manipulates all the page tables can be placed in its own partition with all the page tables in that isolated partition. This guarantees that any memory safety issue in the other partitions can never be used to maliciously modify any page tables.
- The heap, stacks, and global variables can all be placed in their own partitions.

- The loader could spread all the sections of an image into different partitions to make it less likely that corruption in one partition to affect the other.
- The stack of each thread can be in different partitions. This prevents a stack vulnerability in one thread from affecting the stack in another thread if they happen to be in different partitions.
- The heap manager can place heap allocations of different sizes in its own partition, or at least break smaller and larger allocations into different partitions. This makes it such that a use after free bug for a certain heap pointer can only be used to corrupt other heap allocations that happen to be in the same partition.

Dynamic loaders need to be able to create pointers as code and data is being loaded, fixed up and placed into memory. This means the loader will need to generate pointers to other partitions. To address this need, we can designate some partition to be the loader partition and initialize the loader partition with the 256 ReadWriteExecute pointers for all the 256 partitions. This way, the loader partition would have the ability to create any type of pointer it wants in any partition it wants.

Higher exception levels could even withhold a subset of the 256 pointers to the loader and use that reserved subset of partitions for sealed pointers that the higher exception level wants to grant to the lower exception level for use as opaque handles. For example, EL1 could withhold the pointer to partition 0xFF from the EL0 loader partition and thus deny EL0 from running any code or creating any pointers in partition 0xFF. This enables EL1 services to use sealed pointers in partition 0xFF as opaque handles that can be given to EL0 to designate ownership of some EL1 resource (e.g. file handle). There would be no way for the EL0 process to unseal or modify such a pointer.

Since CHERI-Lite pointers are 64-bit and requires no bounds checking, we don't need to add new instructions to manipulate pointers. The original 64-bit instructions that add and subtract values to X registers can be used to also manipulate pointers. We do need to make some changes to the add and subtract instructions to make sure the upper 16 bits of a pointer is never modified. We will explain the details of the instruction set changes in the next section, but the important thing to note here is that existing instructions will continue to work when CHERI-Lite is enabled, and all the added CHERI-Lite instructions are optional and don't have to be used in EL0 to take advantage of the protection that the tag bit provides to prevent pointer corruption. This makes it such that a well behaved existing EL0 application might work with CHERI-Lite enabled without needing to recompile the code. This will be a huge adoption advantage for CHERI-Lite as it can enhance memory safety for existing applications already in market. More research is needed on what percentage of applications can work "as is" with CHERI-Lite enabled, and if they don't work, can we make additional ISA changes or even create a binary-to-binary translator to fix the problem. Note that this assumes that the current platform ABI already have all pointers aligned on 8-byte boundaries, but this should be true for most modern 64-bit platforms.

3 ISA changes for CHERI-Lite

This section will describe the details of the instruction set changes needed for CHERI-Lite. Note that when something goes wrong on a CHERI-Lite enabled system, we can either fault or we might choose to merely clear the tag bit of the affected pointer. The description below will make an educated guess on whether causing a fault or clearing the tag bit makes more sense, but it is merely listed as a suggestion, and each architecture that adopts CHERI-Lite might choose to implement it differently.

3.1 Basic ISA Changes

This section describes the ARM64 instruction set changes related to load/store/instruction fetch, and writing into special registers:

- During Load/Store instructions, the address used for the load/store must satisfy below conditions:
  - The tag bit must be set and must not be of protected data type

    - If two X registers are added to form the address, then exactly one of the registers must have the tag bit set, and the pointer permissions applied to the operation should be from the register with tag bit set.
    - The 3-bit pointer type field of the pointer must specify a type that allows the corresponding load/store operation.
    - The sealed bit of the pointer must be 0.
    - A fault will be triggered if any of above condition is not met.
- Any X register moved into the PC register through a Branch instruction must satisfy below conditions:
    - The tag bit must be set and must not be of protected data type
    - The 3-bit pointer type field must specify the ReadExecute or ReadWriteExecute type
    - The sealed bit must be 0.
    - A fault will be triggered if any of above condition is not met.
    - The lock bit will be cleared when pointer is moved into PC.
- Any X register moved into the PC register through a RET instruction must satisfy below conditions:
    - The tag bit must be set and must not be of protected data type
    - The 3-bit pointer type field must specify the ReadExecuteRET or ReadWriteExecuteRET type
    - The sealed bit must be 0.
    - A fault will be triggered if any of above condition is not met.
    - The lock bit will be cleared when pointer is moved into PC.
- The Branch and Link instruction will place a return pointer into the Link Register with the following attributes:
    - The pointer type should be ReadExecuteRET if current type in PC is ReadExecute and ReadWriteExecuteRET if current type in PC is ReadWriteExecute .
    - The Locked bit should be 1.
    - The Sealed bit should be 0.
- Any pointer moved into the SP register must satisfy below conditions:
    - The tag bit must be set and must not be of protected data type
    - The 3-bit pointer type field must specify a type that allows both load and store operations
    - The locked bit must be 0.
    - The sealed bit must be 0.
    - A fault will be triggered if any of above condition is not met.
- Any pointer moved into special register used to hold addresses (e.g. SP_ELX, TTBRX_ELX) must satisfy below conditions:
    - The tag bit must be set and must not be of protected data type
    - The 3-bit pointer type field must specify a type that allows load and store operations
    - The sealed bit must be 0.
    - A fault will be triggered if any of above condition is not met.
- If a store instruction writes less than 64-bits, or if the 64-bits are not 8-byte aligned, then the tag bit of all affected 8 byte aligned regions will be cleared.

3.2 ALU Operation Changes

This section describes the ARM64 instruction set changes related to ALU operations:

- Any operation that writes to a W register results in the tag bit of the X register being cleared. Future discussions below are all about operations on X registers.
- The add instruction is changed in the following way:
    - The operation is a pointer operation if any operand has the tag bit set, otherwise it is just a regular add operation on integers and remaining rules don’t apply.
    - If both operands are pointers, the destination tag bit is cleared and remaining rules don’t apply.

    - The top 8 bits of the operand with the tag bit set is copied to the destination X register with tag bit set
    - If the addition results in changes to the top 16 bits of the pointer, the destination tag bit is cleared.
    - The destination tag bit is cleared if pointer type is of protected data.
    - The destination tag bit is cleared if the Locked bit is set.
    - The destination tag bit is cleared if the Sealed bit is set.
- The subtract instruction is changed in the following way:
    - If both operands do not have the tag bit set, it is just a regular integer subtraction and remaining rules do not apply.
    - If the left operand is not a pointer, but the right operand is a pointer, the destination tag bit is cleared and remaining rules don't apply.
    - If both operands are pointers, the result of the operation is an integer with tag bit cleared, and the subtraction ignores the top 16 bits.
    - If the left operand is a pointer and the right is not, then following rules apply
    - The top 8 bits of the left operand is copied to the destination X register with the tag bit set
    - If the subtraction results in changes to the top 16 bits of the pointer, the tag bit is cleared.
    - The destination tag bit is cleared if applied to a pointer type of protected data.
    - The destination tag bit is cleared if the Locked bit is set.
    - The destination tag bit is cleared if the Sealed bit is set.
- The AND/ORR/XOR instructions are changed in the following way:
    - The operation is a pointer operation if any operand has the tag bit set, otherwise it is just a regular AND/ORR/XOR operation on integers and remaining rules don't apply.
    - If both operands are pointers, and the top 8 bits don't match, then it is just a regular AND/ORR/XOR operation on integers, destination tag bit is cleared and remaining rules don't apply.
    - The top 8 bits of the operand with the tag bit set is copied to the destination X register with tag bit set
    - If the result of the operation changes to the top 16 bits of the pointer, the destination tag bit is cleared.
    - The destination tag bit is cleared if applied to a pointer type of protected data.
    - The destination tag bit is cleared if the Locked bit is set.
    - The destination tag bit is cleared if the Sealed bit is set.
- Greater or less than (>, <, <=, >=) operations are changed in the following way:
    - Regular comparison takes place as long as one of the operands is not a pointer. Remaining conditions only apply if both operands are pointers.
    - The top 16 bits of the pointer are ignored in the comparison
    - If one operand is a pointer and the other is not, the operation should fail in some way (perhaps set an appropriate error flag bit).
- Equal and Not Equal (==, !=) operations are changed in the following way:
    - Comparison can take place when the 2 operands are both pointers or both not pointers. If one is a pointer and one is not a pointer, the operation should result in not equal without comparing the actual values (NULL comparison must be supported). Remaining conditions only apply if both operands are pointers.
    - The top 8 bits of the pointer are ignored in the comparison
- Other ALU operations such as multiply, divide, shift, etc are not valid operations on pointers and simply carry out the original ALU operation on all 64 bits and always set the destination tag bit to 0.

3.3 Added CHERI-Lite Instructions

This section describes the new ARM64 instructions needed for CHERI-Lite functionality:

- Instructions will be added to clear the tag bit of an X register, but no instructions will exist to set the tag bit.

- Instructions to clear all the tag bits in a region of memory.
- Instructions will be added to test the tag bit of an X register.
- Instructions will be added to decrease permissions on a tagged pointer in the X register, but no instructions will exist to add permissions.
- Instructions will be added to Lock a pointer in an X register (setting Locked bit to 1), but no instruction will exist to change the Locked bit back to 0.
- Instructions to seal and unseal a pointer. Sealing and unsealing is only possible if:
    - The partition ID of the current PC is equal to the partition ID of the pointer being sealed or unsealed.
- Instructions will be added for Protected Data type values:
    - Convert a 56-bit integer in an X register into a tagged protected data type. This instruction doesn't really break the rule: "pointers can't be created out of thin air" because the protected data type isn't really a pointer (although tag bit is set), since it can't be used for load/store or instruction fetch.
    - Extract the 56-bit integer value of a Protected data type value into another X register, it also checks whether the tag bit is set and fails if it is not set.
    - Modify the 56-bit integer in an existing Protected data type value, it also checks whether the tag bit is set and fails if it is not set.
- Instructions needed by EL1 or higher for managing tag bits during memory paging.

3.4 Other Instruction Changes

This section describes some other miscellaneous ARM64 instruction set changes needed for CHERI-Lite. This set is by no means complete. We are sure other changes not mentioned here are also needed.

- The CPY (MOPS_MEMCPY) instruction should copy the tag bit if all the 8-byte aligned region is copied, but clear the tag bit if only a portion of the 8-byte aligned region was written. It also needs to confirm that the X register for the source and destination address satisfies:
    - The tag bit is set
    - The permission allows load operations for source address and store operations for destination address
    - The sealed bit is not set
- The SET (MOPS_MEMCPY) instruction should clear all the tag bits on the destination.
- DC ZVA instruction should also clear all tag bits in the cache line.
- Any DMA into memory from peripherals should result in the tags being cleared.

3.5 Tag clearing during `Memcpy`

The default behavior of `memcpy` should be to copy the tag bits along with the memory contents during the copy. However, a separate version of memcpy should also exist that clears the destination region of all tag bits. Such a version is useful whenever the system is copying data that it knows does not contain any pointers. For example, the JavaScript TypedArray type supports Set and Split operations that need to use memcpy, but since TypedArray elements should never contain pointers, the tag clearing version of memcpy can be used for Set and Split. Compilers are also typically aware of the type of data that is being copied and whether it contains any pointers. Whenever no pointers are present, the tag clearing version of memcpy should be used. This helps reduce the number of tag preserving memcpy gadgets that could be used by the attacker. This special version of memcpy can be constructed by simply concatenating the CPY instruction with instructions that clears all the tag bits for the destination region of memory.

4 Security Strength of CHERI-Lite

In this section, we present some analysis of the security of CHERI-Lite and why we believe it will be effective at frustrating the memory safety attacker. Over time, as we leverage partitions, improve compilers, and remove gadgets that are useful to attackers, we believe CHERI-Lite will be able to prevent most end to end memory safety exploits. This section aims to do some analysis and explain why the security value of CHERI-Lite is significant and worth the ISA change.

In order to discuss the security effectiveness of CHERI-Lite, we need to first analyze how memory safety exploit attacks are typically pulled off. The typical memory-safety attack follows the pattern of attacking in 3 stages:

1. Find initial buffer overrun, use after free, type confusion or other memory safety vulnerability that enables attacker-controlled memory write on a small region of memory (crude memory corruption).
2. Find a way to use the results of stage 1 to enable arbitrary read/write on all memory in the address space (precise memory corruption).
3. Leverage the result of stage 2 to strategically overwrite memory in many locations to perform a return-oriented (ROP), jump-oriented (JOP), or data-oriented (DOP) [1] attack on the system to achieve the actual goal of the attack.

All 3 of the above stages are critical to the attacker. As such, any mechanism to increase the amount of work needed for each stage of the attack, will make a significant dent in the attacker's ability to carry out memory safety attacks. One of the observations about the three stages above is that stage 3 relies on corrupting pointers in memory. All ROP/JOP/DOP attacks require modification of pointers of some type and if we can make it hard for attackers to modify pointers in stage 2, end to end exploits become extremely hard. In the following subsections, we will discuss how CHERI-Lite affects each stage of the attack.

To better analyze the security of CHERI-Lite, we need break up stage 2 of the attack into 2 substages:

a) Enable arbitrary read/write on all data (non-pointer) memory in the address space
b) Enable arbitrary read/write on all pointers in the address space

The distinction between data and pointer is very relevant on a CHERI-Lite system, but irrelevant on traditional computer architecture. As you will see in following subsections, CHERI-Lite makes stage 2a of the attack harder than before, and stage 2b even harder. Also note that until the attacker achieves stage 2b, the attacker cannot perform any ROP/JOP/DOP attacks on the system which means that in the process of performing stage 2a and 2b of the attack, ROP/JOP/DOP cannot be used.

4.1 How CHERI-Lite Affects Stage 1 of Attack

While CHERI-Lite does not prevent typical buffer overruns, type confusion and use after free bugs, it does prevent Double Free vulnerabilities from messing up the heap metadata assuming the heap manager uses a dedicated partition, and sealed pointers are used to link each allocation to its metadata in the heap manager partition. Because heap metadata is stored in a separate partition other than the heap allocation itself, it cannot be easily corrupted. Also, since the sealed pointer pointing back to metadata will get erased after the first free, the second free should be easily detectable by the heap manager.

Even though CHERI-Lite does not prevent buffer overruns, type confusion and use after free bugs from overwriting memory, it does make the memory corruption in stage 1 clear the tag bits of any pointers that was corrupted. The benefits of this tag bit clearing won't materialize until stage 2.

4.2 How CHERI-Lite Affects Stage 2a of Attack

In stage 2a, the attacker tries to find a way to use the vulnerability found in stage one to enable arbitrary read/write on all data memory in the address space. In a CHERI-Lite world, to perform an arbitrary memory read write, you need to perform a store operation at the address of your choosing, which means you need to find a way to create a ReadWrite or ReadWriteExecute type pointer that points to your desired address. Since tag bits are cleared when memory is corrupted, the attacker will not be able to directly materialize such a pointer from the memory corruption in stage 1. The more likely scenario is to corrupt an integer in stage 1 and find a naturally callable gadget that adds that corrupted integer into an existing ReadWrite pointer. Code to do this does exist and the following code illustrates such an example:

```
class ByteArray {
  private:
    uint8_t* ArrayPtr;
    uint32_t ArrayLength;

  public:
    ByteArray(uint32_t length) {
        ArrayPtr = new uint8_t[length];
        ArrayLength = length;
    }

    void SetByte(uint32_t index, uint8_t value) {
        if (index < ArrayLength) ArrayPtr[index] = value;
    }
    …
}
```

In this example, a ByteArray class is used to abstract a byte array as well as access to the bytes in the array. If an attacker through some code defect in stage 1 was able to modify the ArrayLength field of one of these ByteArray objects, and was able to make SetByte calls on that object, then the attacker would have unfettered access to 4GB of memory starting from the ArrayPtr location. It's quite possible to find code like this that could be leveraged by an attacker, but once discovered, the code can be changed to address the problem.

If the developer knows of a constant upper bound on the number of bytes in a ByteArray, the developer could change the code for SetByte as follows:

```
    void SetByte(uint32_t index, uint8_t value) {
        if (index < ArrayLength && index < MAX_BYTEARRAY_SIZE)) ArrayPtr[index] = value;
    }
```

This would greatly limit the range of memory that could be corrupted using the ByteArray structure. If the size of ByteArray is truly dynamic and no constant upper bound exists, the code could then be changed to the following to remedy the problem:

```
class ByteArray {
  private:
    uint8_t* ArrayPtr;
    uint8_t* ArrayPtrBound;

  public:
    ByteArray(uint32_t length) {
        ArrayPtr = new uint8_t[length];
        ArrayPtrBound = ArrayPtr + length;
    }

    void SetByte(uint32_t index, uint8_t value) {
        if (ArrayPtr + index < ArrayPtrBound) ArrayPtr[index] = value;
    }
    …
}
```

As you can see, we have replaced a corruptible piece of data (ArrayLength) that was used for bounds checking with a pointer (ArrayPtrBound) which loses the tag bit once it is corrupted. If ArrayPtrBound is corrupted, it loses its status as a pointer. As such, when `ArrayPtr + index < ArrayPtrBound` is tested, a fault will be raised as a pointer cannot be tested to be less than an integer. Note that once code like above ByteArray example is fixed, it is removed from possible gadgets that can be used for stage 2a forever. Assuming only a reasonably small set of stage 2a gadgets like this exist in the code base, once they have all been discovered and fixed like above, attackers will have a hard time pulling off stage 2a of the attack.

Let's summarize the additional hurdles an attacker faces for stage 2a compared to a system without CHERI-Lite:

- You cannot simply corrupt a pointer to point at the location you want to read/write. You can only corrupt integers and then hope to find a naturally callable gadget that adds that integer to a valid pointer and then hope that the gadget also allows you to read/write into that pointer location.
- The crude memory corruption achieved is stage 1 must be able to corrupt the specific integer that is added to the pointer for the gadget that you found.
- The gadget may not necessarily allow you to modify all the memory you need to modify. In the above ByteArray example ArrayLength was uint32_t, but if it was uint16_t, then the attacker can only corrupt 64K Bytes of memory. Also, the attacker can only corrupt memory above ArrayPtr, but not memory below it.
- The gadget must be naturally callable without using any ROP or JOP techniques, since the attacker still hasn't found a way to corrupt pointers yet (until stage 2b is complete), the attacker is restricted to only using naturally callable gadgets.
- The pointer that is corrupted must be of the correct type that is needed to achieve the attack.
- If partitions are used, the attacker must find a gadget that can be used to corrupt pointers to the partition it wants. To corrupt pointers in another partition would likely require finding another gadget that allows modification of pointers in the other partition.

4.3 How CHERI-Lite Affects Stage 2b of Attack

Assuming the attacker achieves stage 2a and is able to corrupt data anywhere in memory, the attacker now wants to move to stage 2b of the attack. What the attacker will need to do in stage 2b is find a way to place a pointer of the attackers choosing into a location of the attackers choosing. Note that this is even harder than what needs to be done for stage 2a. In stage 2a, the attacker just needed to create a pointer to point at the location of the corruption, but the corruption value was data that could easily be fabricated. For stage 2b, the value that the attacker wants to write to that location is another pointer, which means the attacker needs to also find a way to fabricate the pointer that it intends to write into the target location. Note that the attacker needs to control both the exact location of the pointer as well as the exact pointer value to corrupt to.

Let's look at the possible gadgets that the attacker would need to find to achieve the stage 2b goal of placing a pointer of the attacker's choosing into a location of the attacker's choosing. Note that this list below is by no means complete.

1. If an attacker finds a naturally callable gadgets that directly places an attacker-chosen pointer value at attacker chosen memory location. The following code sequence would be an example of such a gadget:
   ```
   Ptr = Ptr + Offset1;
   Ptr2 = Ptr2 + Offset2;
   Ptr->Field = Ptr2;
   ```
   If the attacker that is able to corrupt data values `Offset1` and `Offset2` in above code, then attacker would be able to write an arbitrary pointer value at an arbitrary location. While it might be possible to find code

sequences like this, it should be very rare to find code sequences like this that are naturally callable and perform no forms of maximal bounds checking on Offset1 and Offset2 whatsoever. As long as some reasonable form of maximal bounds checking is done on offset1 and offset2, then this gadget cannot be used to achieve stage 2b. Also note, that once gadgets like this are found and some form of bounds checking is added to fix the problem, we have permanently removed this gadget from attacker use in the future.

2. If an attacker finds a naturally callable gadget that has the code sequence:
   ```
   *(Ptr + Data1) += Data2
   ```
   Then by corrupting `Data1` and `Data2`, the attacker can modify an existing pointer at location Ptr + Data1 to another pointer value. This type of gadget is less powerful than the previous gadget since it requires the destination address to already have a pointer to begin with since the gadget only adds Data2 to the destination and if it was not a pointer to begin with, it cannot be turned into a pointer. Again, while it seems like such gadgets might be possible to find, the rare thing to find is naturally callable code like this that doesn't perform any maximal bounds checking on Data1 before dereferencing *(Ptr + Data1).

3. The attacker first finds some naturally callable gadget to modify an existing pointer and leaves the pointer somewhere in memory, then finds another naturally callable gadget that lets the attacker perform a `memcpy` from an attacker chosen source to an attacker chosen destination to copy the modified pointer to the desired location. Note that both the source and destination location would need to be controlled by data in memory for the `memcpy` to be useful.

   An example for such a memcpy gadget could be a QwordArray class like the ByteArray class illustrated earlier. Since such an array accesses 8 bytes at a time, it would be able to move a pointer with its tag bit. The solution to this problem would be to also fix up QwordArray to either perform tighter bounds checking using a constant value or modify the code to use a pointer as the bounds check to prevent corrupting the bounds. Another option would be to add code to clear the tag bit during set and get operations on the array elements to avoid access to QwordArray being used to move pointers.

   The attacker could also just simply call memcpy (assuming it is naturally callable) to perform the copy, but that isn't necessarily easier, as the attacker would need to prepare 2 valid pointers pointing to the source and destination of the memcpy and place them each in the right parameter passing register before calling memcpy.

Note that in above descriptions, the term "naturally callable gadget" is used to represent a sequence of code that the attacker can make the CPU execute into naturally. Remember, the attacker cannot inject code pointers where they want until stage 2b is achieved, so the attacker cannot yet make the system execute into unnatural code entry points that bypass the initial parameter checking. So, the assumption that until the attacker achieves stage 2b, they have no code to work with besides naturally callable gadgets is still true. Again, let's summarize the additional hurdles an attacker faces for stage 2b compared to a system without CHERI-Lite:

- Since you cannot directly corrupt pointers, you need to find naturally callable gadgets that lets you add a corrupted integer to a valid pointer, and you need to do this for both the pointer value you wish to write as well as the pointer location you wish to write to.
- The gadget must be naturally callable without using any ROP or JOP techniques. Thus, the attacker cannot bypass typical bounds checking at the beginning of a function.
- The gadget must either have no reasonable bounds checking or only have bounds that are corruptible.
- The gadget must allow enough modification to each pointer to reach the target pointer value attacker wants.
- The pointer that is modified must be of the correct type that is needed to achieve attack.
- If partitions are used, the complexity of the attack just increased significantly, as pointers can only be modified to move within its own partition. This means that the naturally callable gadget that the attacker finds must be

one that corrupts pointers for the exact right partition that the attack wants. And remember, the attacker needs to materialize 2 pointers, one for the pointer value to corrupt to, and one for the address of the corruption, and if these 2 pointers are in different partitions, it makes the attack much harder.

We believe that gadgets that help the attacker achieve the goals of stage 2b will be non-trivial to find if they exist at all. When such gadgets are discovered, we can change the code to not exhibit such attacker desired functionality by adding appropriate bounds checking or tag clearing. The hope is that over time, we can remove the presence of most (if not all) of these gadgets in the common code base such that it becomes practically infeasible for attackers to maliciously modify pointers on a machine with CHERI-Lite active. If we can achieve this, then we believe that we will have made the end-to-end exploitation of computer systems orders of magnitude harder than they currently are.

4.4 How CHERI-Lite Affects Stage 3 of Attack

Once the attacker achieves stage 2b of the attack, the attacker can proceed to stage 3 to corrupt all the pointers it needs for the final ROP/JOP/DOP attack. For all practical purposes, CHERI-Lite does not materially affect the complexity of stage 3 of the attack once stage 2b has been achieved. Its main value still lies in the prevention/complication of stage 2a and 2b of the attack.

To summarize the value of CHERI-Lite: While it doesn't do much for Stage 1 of the attack, it adds significant work for Stage 2. Currently, without CHERI-Lite, attackers typically spend a lot of time looking for vulnerabilities to achieve stage 1 crude memory corruption. But after that, stage 2 and stage 3 end up being relatively simpler. With CHERI-Lite, stage 2 becomes non-trivial that requires lots of work to find gadgets to complete, and there is no guarantee they can find naturally callable gadgets to do all that they need to do. Even worse for attackers, over time, we can likely build tools to search and find such gadgets and eliminate them altogether.

4.5 Preventing Direct Data Manipulation (DDM) Attacks on Critical Data

Even if we prevent the malicious modification of pointers in stage 2b, one major problem still present is the fact that pure data could still be easily modified by an attacker after achieving stage 1 or stage 2a of an attack. This type of attack is referred to as Direct Data Manipulation (DDM) attacks in the literature. While the amount of damage that can be done by DDM attacks is usually less than what could be done if pointers can be modified, there are still cases where a DDM attack could still be catastrophic. A classic example is the safemode flag used to denote whether VBScript in a browser can operate in God Mode. Even though this safemode flag is just data, the consequences of this flag being maliciously modified is severe.

Cases like this are where the CHERI-Lite "protected data" type is useful. The way to secure the safemode flag is to store the safemode flag as a protected data type pointer in CHERI-Lite. Any code that sets the safemode flag uses the new CHERI-Lite instruction to set protected data values and reading the flag should use the instruction to get protected data. The instruction to get the protected data should fail if the tag bit for the protected data has been cleared due to corruption. All these code changes can be automatically done by the compiler. The compiler would need to support annotating variables that need to be protected and generating the code described above whenever that variable is accessed.

4.6 Compiler Enhancements to Improve Security

Over time, code generated by compilers for CHERI-Lite can do several things to boost the security of CHERI-Lite and make memory safety exploitation with CHERI-Lite even harder. Below are some of the compiler changes we can think of:

1. Lock any newly created pointer if there is no need to ever move this pointer to a different value using add/subtract/and/orr/xor operations. Short immediate displacement using immediate value in code is OK.

2. After ADR and ADRP instructions, remove execute permission from destination pointer if it is only needed to access global data and won't be needed for branch instructions.
3. Clear tag bits as much as possible whenever the compiler knows that the destination cannot be a pointer (e.g. QwordArray).
4. Clear tag bit on destination of memcpy whenever the data being copied cannot contain pointers.
5. Search for gadgets that are useful for stage 2 of attack and warn developers of their existence.

4.7 CHERI-Lite and other ARM64 Security Features

In this subsection, we discuss how CHERI-Lite affects the usage of other ARM64 security features:

Pointer Authentication Code (PAC)

Both PAC and CHERI-Lite use the top byte of a pointer, so they cannot coexist. We believe CHERI-Lite makes PAC redundant and is superior than PAC because:

1. There is no way to fabricate backward code pointers out of thin air in CHERI-Lite. It can only be generated using the Branch and Link instruction.
2. All backward code pointers are locked when generated, and thus cannot be modified to point to a different location.
3. CHERI-Lite control flow enforcement is deterministic and not probabilistic.
4. No need to depend on QARMA and the longevity of its crypto properties.

Branch Target Identification (BTI)

BTI marks where Branch and Jump instructions can land and thus prevents JOP attacks. CHERI-Lite also prevents JOP but does so by making it hard to generate the attacker desired code pointers needed to pass into a branch instruction. So both try to prevent JOP but solve the problem in different ways. As these 2 technologies do not conflict with each other, it does make sense to deploy them both to make it even harder to deploy JOP attacks. Over time, if we learn that CHERI-Lite already achieves the goal of preventing JOP, then BTI can be removed from code to shrink the code size.

Memory Tagging Extensions (MTE)

MTE's main goal is to mitigate some forms of stage 1 attacks such as heap buffer overruns, and use after free, while CHERI-Lite is about mitigating stage 2 of attacks. So, they complement each other very well, and it does make sense to deploy them both together. CHERI-Lite as defined in this doc currently only uses 5 of the top 8 bits in the address, so the remaining 3 bits could be used for MTE tag bits.

5 Deployment and Implementation Details

The Morello project has already created a CHERI based SoC, and confirmed large operating systems like BSD can be ported to support CHERI. CHERI-Lite should be even easier to support given that the size of pointers has not changed. In this section, we will discuss some practical implementation details, adoption strategies, and performance implications of CHERI-Lite. We will only discuss points that are specific to CHERI-Lite and will omit issues that have already been solved and well researched for full CHERI (e.g. efficient implementations of tag bit).

5.1 Deploying CHERI-Lite Per Exception Level

Systems that support CHERI-Lite should allow enablement and disablement of CHERI-Lite on a per exception level basis. For EL0, it can further be enabled and disabled on a per process basis. This enables CHERI-Lite to only be enabled on EL0 processes when the application has been verified to work correctly with CHERI-Lite or has been recompiled to support CHERI-Lite. Similarly, each other exception level (EL1, EL2, S-EL1, S-EL2…) can have CHERI-Lite enabled or disabled based on whether all the code at that exception level is ready to support CHERI-Lite.

The behavior of the system when it is operating at an exception level where CHERI-Lite is disabled is to NOT fault for any tag bit related reasons. But all the logic to update the tag bit can remain in place and operate the same way regardless of whether CHERI-Lite is enabled or not. This will make the micro architecture simpler as there are lots of small hardware changes related to clearing the tag bit and propagating the tag bit that can remain unchanged.

Given that pointer values could be passed across exception level boundary, extra code will need to be added on exception level boundaries (e.g. SVC/HVC/SMC calls) to make sure all the passed parameters that should have the tag bit set have it set, and (more importantly) all the parameters that shouldn't have it set are not set. This is especially true when it crosses between exception levels that do not enable CHERI-Lite and ones that do enable CHERI-Lite.

5.1 Partition ID Length

The number of high order address bits to carve out to use as the partition ID could probably support several different possible values (e.g. 0, 4, 8, 12) or even be dynamic. And this value could be different per exception level and even be different between different EL0 processes.

5.2 Making some new CHERI-Lite instructions occupy no-op space on legacy HW

There might be a subset of new CHERI-Lite instructions that is appropriate to map into the no-op op code space on legacy ARM64 hardware (e.g. convert RW pointer to RO pointer). If a new EL0 application compiled with CHERI-Lite only uses new instructions in this subset, then the application is compatible with existing ARM64 HW and can be deployed universally.

5.3 Performance of CHERI-Lite

The performance penalty associated with CHERI-Lite should be minimal as the tag bit is carried around with the 64-bit value in registers and caches. There is already a lot of research on how to manage CHERI tag bits efficiently and CHERI-Lite can leverage all that work. CHERI-Lite does have a higher tag bit overhead than full CHERI as a tag bit is needed for every 64 bits in memory, where as CHERI only needs a tag bit every 128 bits. However, the total memory overhead of CHERI-Lite should be smaller than CHERI due to the smaller size of pointers and less memory overhead for pointers. CHERI-Lite could also eliminate the need for other existing code flow enforcement features we have built in the past including SSP, PAC, BTI, SCS, and CFI. There is a good chance that we observe a net performance gain with CHERI-Lite once we disable all the other security measures that are redundant due to CHERI-Lite.

6 Related work

This paper is a follow up to a paper titled "Pointer Tagging for Memory Safety" [16] published in 2019. We changed the tagging concept to follow CHERI [9] much more closely and thus changed the name to CHERI-Lite. Because of the more stringent adoption of the CHERI concept that "Pointers cannot be fabricated out of thin air", the security analysis of the CHERI-Lite is much more complete and more robust than the previous arguments made in the pointer tagging paper.

Tagged architectures have a long history dating back to Burroughs' B5000 [4] and including some commercial systems such as the AS/400 series [5] from IBM. In recent years, they have been used to guarantee strong provenance properties in capability systems such as the M-Machine [6], Aries [7], Low-fat pointers [8], and CHERI [9]. These schemes provide very strong security guarantees at the expense of binary compatibility (and, often, with some restrictions on source compatibility). The scheme proposed in this paper provides weaker security guarantees in exchange for better performance and ease of deployment.

The Secure Bit [10] scheme aims to protect addresses by using a 1-bit information flow policy, implemented as a tag, to prevent any data injected from an external source from being used as an address. More recently, the lowRISC project has proposed a set of general-purpose tagged memory extensions [11]. These describe a variety of potential use cases including some that are like this proposal but leave the enforcement of any given policy entirely to software.

Software tagging schemes, such as Address Sanitizer [12], use a larger tag space ("shadow memory") to provide information about memory layout that can be used for bounds checking. These techniques provide different guarantees (some temporal and spatial memory safety, but no data-pointer type safety) at a considerably higher overhead. Intel MPX [13] provided a similar set of guarantees in hardware, with a higher overhead and a considerably larger tag space (256 bits of metadata for every 64 bits of pointer, though no tags for quarter pages that did not contain any tags).

The guarantees provided here are close to those intended by Code Pointer Integrity (CPI) [14], though with some additional protection on data pointers. The CPI scheme has been shown [15] to be vulnerable to side channels that leak secret data. Our scheme does not depend on secrets and so is not vulnerable to the same kind of attacks.

7 Conclusion

This paper proposes adopting the CHERI concept of tagging pointers, and only allowing tagged pointers to be used to specify the address of load, store, and instruction-fetch operations. However, we propose keeping the pointers at 64 bits and thus need to forgo the bounds checking on pointers. The top 8 bits of pointers are still used to store permission information, and more high order bits in the address space could be used to carve memory into partitions. The security value of CHERI-Lite is to prevent memory safety attackers from directly fabricating pointers at will. Instead, with CHERI-Lite enabled, the memory safety attacker would need to find gadgets that add corrupted integer values into pointers and then try to use those pointers. Furthermore, these gadgets must be naturally callable by the attacker since the attacker would need to modify code pointers to perform ROP and JOP in order to call into unnaturally entry points, and the attacker is still in the process of finding a way to modify pointers, so cannot use ROP or JOP yet. We believe that over time, we can remove the presence of most (if not all) of these naturally callable gadgets such that it becomes extremely hard for attackers to maliciously modify pointers on a machine with CHERI-Lite active. If we can achieve this, then we believe that we will have made the end-to-end exploitation of computer systems through memory safety bugs orders of magnitude harder than they currently are. Unlike full CHERI, the architecture change for CHERI-Lite is designed to be compatible with existing binary code and could enable large numbers of existing applications to gain some memory safety benefits without the need to even recompile. The performance overhead of CHERI-Lite is small and could provide better security value than most other code flow integrity security features that most likely have a higher performance overhead.

8 References

[1] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena and Z. Liang, "Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks," in *2016 IEEE Symposium on Security and Privacy (Oakland)*, 2016.

[2] A. Joannou, J. Woodruff, R. Kovacsics, S. W. Moore, A. Bradbury, H. Xia, R. N. M. Watson, D. Chisnall, M. Roe, B. Davis, E. Napierala, J. Baldwin, K. Gudka, P. G. Neumann, A. Mazzinghi, A. Richardson, S. Son and A. T. Markettos, "Efficient Tagged Memory," in *Proceedings of the 2017 IEEE 35th International Conference on Computer Design (ICCD)*, Boston, 2017.

[3] B. Davis, R. N. M. Watson, A. Richardson, P. G. Neumann, S. W. Moore, J. Baldwin, D. Chisnall, J. Clarke, N. W. Filardo, K. Gudka, A. Joannou, B. Laurie, A. T. Markettos, J. E. Maste, A. Mazzinghi, E. T. Napierala, M. R. Norton, M. Roe, P. Sewell, S. Son and J. Woodruff, "CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer

Privilege in the POSIX C Run-time Environment," in *In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*, Providence, 2019.

[4] A. J. W. Mayer, "The Architecture of the Burroughs B5000: 20 Years Later and Still Ahead of the Times?," *SIGARCH Computer Architecture News,* vol. 10, no. 4, pp. 3-10, 1982.

[5] B. E. Clark and M. J. Corrigan, "Application System/400 performance characteristics," *IBM Systems Journal,* vol. 28, no. 3, pp. 407-423, 1989.

[6] N. P. Carter, S. W. Keckler and W. J. and Dally, "Support for fast capability-based addressing," in *ACM SIGPLAN Notices*, 1994.

[7] J. Brown, J. P. Grossman, A. Huang and T. F. Knight Jr, "A capability representation with embedded address and nearly-exact object bounds," Project Aries Technical Memo 5, MIT, 2000.

[8] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight Jr and A. DeHon, "Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013.

[9] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton and M. Roe, "The CHERI capability model: Revisiting RISC in an age of risk," in *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on Computer Architecture*, Minneapolis, 2014.

[10] K. Piromsopa and R. Enbody, "Secure Bit: Transparent, Hardware Buffer-Overflow Protection," *IEEE Transactions on Dependable and Secure Computing,* vol. 3, pp. 365-376, November 2006.

[11] A. Bradbury and G. Ferris, *Tagged memory and minion cores,* lowRISC-MEMO 2014-001, 2014.

[12] K. Serebryany, D. Bruening, A. Potapenko and D. Vyukov, "AddressSanitizer: A Fast Address Sanity Checker," in *USENIX ATC 2012*, 2012.

[13] C. W. Otterstad, "A brief evaluation of Intel® MPX," in *9th Annual IEEE International Systems Conference (SysCon)*, 2015.

[14] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar and D. Song, "Code-Pointer Integrity," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Broomfield, CO, 2014.

[15] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard and H. Okhravi, "Missing the Point(er): On the Effectiveness of Code Pointer Integrity," in *2015 IEEE Symposium on Security and Privacy*, 2015.

[16] Tony Chen, David Chisnall, “Pointer Tagging for Memory Safety” in Microsoft Report MSR-TR-2019-17, July 2019