Pointer Tagging for Memory Safety

Tony Chen (tonychen@microsoft.com)

David Chisnall (David.Chisnall@microsoft.com)

1 Summary

Memory safety attacks continue to be prevalent on computer systems in use today, as large amounts of unsafe C/C++ code continues to provide attackers with a large supply of buffer overrun, use after free and type confusion bugs. This paper proposes a fundamental instruction set architecture change to combat memory safety problems. The ISA change is mostly transparent to application code and typically only requires a recompilation of the application to gain the security benefits. The change involves having the CPU hold two extra tag bits to the side of each piece of 64-bit data to denote whether the data holds a code/data pointer or not. By doing this, we can prevent attackers from using 'data' to corrupt 'pointers' and cause undesired damage. We believe the proposed architecture change enables stronger control flow protection than shadow stack (Intel CET) plus Control Flow Guard (CFG) with less performance overhead. Thus, this ISA change not only enhances security, but does so while improving performance.

2 Overview

The fact that memory unsafe languages like C and C++ are used by all of the popular operating systems (e.g. Windows/Linux) as well as a significant portion of applications (e.g. Office) continues to cause critical security issues on a regular basis. While we have made some progress on this issue by restricting the control flow of the CPU through features such as CFG and CET, there is strong evidence emerging that attackers can still cause enormous damage just by modifying data pointers. A recent Data Oriented Programming (DOP) paper [1] shows that it is possible within some processes to build a Turing complete machine by just corrupting data pointers without subverting the natural control flow of the original program. In anticipation of attacks like this happening, we need to come up with more innovative ideas to prevent future memory safety attacks.

Of the many proposals on how to solve the memory safety problem, The CHERI proposal from the University of Cambridge appears to be one of the most promising solutions proposed in the literature. CHERI uses 128-bit fat pointers / capabilities to limit the range of memory that each pointer is allowed to access (and in what way), while also using an extra tag bit to protect the capability pointer from being accidentally or maliciously modified. The fat pointers give the system the extra information needed to perform bounds checks to avoid buffer overflows, but also incurs a performance penalty due to the increased size of pointers, as well as making the adoption of CHERI more complicated due to its ABI breaking nature. This paper proposes a solution that uses tag bits to "tag" and protect pointers but uses regular 64-bit pointers. This results in a solution that is not as secure as CHERI, but much easier for the software ecosystem to adopt due to it not being an ABI breaking change and only requiring C/C++

applications to be recompiled without source code changes in the common case. It should also result in less performance overhead compared to CHERI because the size of pointers does not change. Pointer tagging does incur a small overhead due to the need to maintain extra tag bits but the performance loss is expected to be less than other forms of anti-ROP and anti-JOP memory safety countermeasures that are deployed (or planned to be deployed) such as shadow stack (Intel CET) and Control Flow Guard (CFG). Thus, compared to a traditional system where shadow stack and CFG is used to prevent ROP and JOP, pointer tagging not only provides the same (or better) level of anti-ROP and anti-JOP protection, but also frustrates DOP attacks, and it does all of this while enhancing the performance of the system.

Typical control-flow-hijacking memory-safety attacks follow the pattern of attacking in 3 stages:

1. Find initial buffer overrun, use after free, type confusion or other memory safety bug that enables arbitrarily memory read/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) 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, a mechanism to increase the work factor for each stage of the attack, or prevent any one of the stages, will make a significant dent in the attackers ability to carry out memory safety attacks.

One of the observations about the three stages above is that all three stages rely heavily on corrupting pointers sitting in memory to point to a new location of the attacker's choosing. If we take away the attacker's ability to corrupt pointers, then the end to end attack becomes much harder even if the attacker can still find memory safety bugs. Based on this observation, the spirit of this proposal is to make corruption of pointers as hard as possible. We achieve this by storing (non-addressable) tag bits on the side of each 64-bit word in the computer to denote whether the data is merely regular data or it actually contains a pointer and, if so, what kind of pointer (code or data). Note that these tag bits would need to be present throughout the entire system including DRAM memory, the L1/L2/L3 cache and alongside of each general-purpose register. How to efficiently store and access the tag bits will be discussed in section 5.4, but there are ways to do this with very little or even zero performance loss.

The general spirit of this pointer tagging proposal is to make the HW aware of whether each 64-bit in the system is regular 'data', or whether it is a 'data pointer' or a 'code pointer'. This coarse-grained type information is implicitly available to the programmer and the C/C++ compiler, but was never explicitly maintained by the HW and enforced by the HW. The fact that HW does not maintain this information and does not enforce any rules based on it has enabled attackers to mix and match data and pointers in various ways to cause harm to the system.

To take advantage of this hardware feature, software will need to be recompiled by a compiler aware of this feature. We believe that little or no source code changes are needed by the developer. There will be

code changes needed in the OS (discussed in Section 5.5) but our primary goal is to require minimal source code change for applications.

The paper is organized as follows: We start by describing the proposal in more detail in section 3. Then, in section 4, we will do some security analysis about this proposal, and what the attacker needs to do to break this, and what counter actions we can take. After that, we will discuss some practical implementation and adoption issues in section 5. Finally, we will mention some related work and conclude our contribution.

This entire document is written with examples using the Intel/AMD 64-bit instruction set, but the concept could also be adopted for other ISAs such as ARM64 or RISC-V.

3 Details of Pointer Tagging

This section will describe the details of the pointer tagging proposal. We will introduce the concepts individually before describing how they compose.

3.1 Possible Tag Values

We are proposing using 2 extra tags bit on the side of each piece of 64-bits in the system to denote whether the 64-bits contain:

- 00 (**data**): memory contains regular data (or code) that cannot be used as a pointer.
- 01 (**pointer**): memory contains a data pointer that can be used as the address to access other pieces of memory.
- 10 (**code pointer**): memory contains a code pointer that can be used as a destination for indirect calls/jumps or returns.
- 11 (**protected data**): memory contains sensitive data that deserves to be tagged such that it becomes hard to maliciously modify by data corruption.

The 'code pointer' tag type is further subdivided into five subtypes which will be discussed in more detail in Section 3.7. Possible usages of 'protected data' will be discussed in Section 3.9.

3.2 Basic Hardware Changes

The core idea behind this proposal is for hardware to not only keep track of the 4 possible tag types for each piece of 64-bit data, but to also enforce that the tag types must be accurate when each piece of 64-bit data is being used as an address to access memory or is being used as the address to load code from. If these rules are violated, the system will fault and the process will most likely be terminated.

Using the Intel/AMD 64-bit instruction set as an example, the instruction set would change as follows:

- An instruction that accesses memory, such as MOV RAX, [RSI], would fault if the tag bits associated with RSI was not set to 'pointer'. The base for the EA (effective address) that is used to fetch data from memory must always have tag bits set to 'pointer' or the operation will fault.

After this operation, the tag bits associated with RAX would be set to whatever the tag bits were at the memory location pointed to by RSI. If this is an unaligned load, then the value will be data.

- The RIP register must be of type 'code pointer'. This means that instructions such as CALL [RCX] or JMP [RCX] (call or jump to the address in RCX) would fault if the tag state associated with RCX is not 'code pointer'. This also means that when the RET instruction pulls the return address from the stack, the tag state associated with that return address must be 'code pointer'. Note that this will naturally happen under normal circumstances, because CALL copies the RIP value onto the stack for the return address, so the tag bits for the return address on the stack is naturally a 'code pointer'. If, however, an attacker overwrote the return address on the stack with 'data', the RET instruction will fault.
- Consider the following sequence of code that calls a C++ virtual function:
    1. MOV RCX, {OBJECTS ADDRESS}
    2. MOV RAX, [RCX] // Load vtable pointer into RAX
    3. CALL [RAX+8] // call the second function in the vtable

    The tag bits associated with RCX should be 'pointer' after the first instruction. The tag bits associated with RAX after the second instruction should also be 'pointer'. At the memory location pointed to by RAX should be the vtable which is an array of 64 bit 'code pointer's. The final CALL will only succeed if the tag bits at location RAX+8 are 'code pointer'.

- The RSP register should typically be of the 'pointer' type. While assigning a 64-bit value into RSP that is not tagged as a 'pointer' will not immediately generate a fault. The system will fault once a PUSH/POP/CALL/RET instruction is executed because the value of RSP will be used as a pointer during those instructions. Note that since the value of RSP is typically copied into RBP in the prologue of functions, RBP will typically also have the tag bits set to 'pointer'.

3.3 Casting to a Different Tag Type

The instruction set is designed to naturally preserve the correct tag data for all pieces of memory. There will always be special cases where the rules need to be broken and the compiler wishes to "cast" the tag value to something different than it would naturally be. Examples of this include:

- When setting up a C++ vtable.
- When the OS loader performs fixups and sets up dispatch tables.
- When the OS prepares the initial RSP and RIP values for a process.

In order to support casting, 4 new instructions will need to be added to change the tag bit associated with a register to each of the possible tag types. For the purpose of discussion, we will name these 4 new instructions:

- CASTTODATA
- CASTTOPOINTER
- CASTTOCODEPOINTER
- CASTTOPROTECTEDDATA

Each of these instructions takes a register operand and changes the tag bits of that operand to the corresponding tag type. Note that if we make these 4 casting instructions occupy opcode space that is currently no-op, we can have the compiler generate code that gracefully works both on legacy hardware that doesn't support pointer tagging as well as new pointer tagging capable hardware.

3.4 Most Operations Default to Data

Most of the data residing in a system should be tagged simply as 'data'. This fact is enforced by the following rules:

- Whenever a register is cleared to 0 or an immediate value is assigned to it, the tag bits get set to 'data'.
- All types of pointers must be 64-bit aligned in memory. Any partial writes to memory where less than 64 bits are written will result in the tag bit associated with the full 64-bit value getting set to 'data'.
- Any output from MMX, SSE, AES, SHA or other special data-manipulation instructions should have the output tag set to 'data'. Note that some implementations of `memcpy` use vector instructions and vector scatter/gather uses vectors of pointers and so a subset of vector operations (loads and stores) may need to propagate tag values.
- Any I/O DMA into memory will result in the tags being set to 'data'.
- Any data read in through the IN instruction should have type 'data'.

3.5 The Pointer Provenance Model

A lot of C/C++ code casts pointers to integers and vice versa. In many cases, the C/C++ compiler can easily lose track of whether a 64-bit value is actually a pointer or merely data, for example when a generic data structure uses `void*` or `uintptr_t` as the type for data that may be pointers or integer values depending on the use of a data structure. The way this is handled on a pointer tagging system is to let the tag bit naturally carry the provenance/origin of the type of data and let it freely float through the system via the tag bits without casting it to different tag types using CASTTODATA or CASTTOPOINTER when the C/C++ code performs a cast.

Any value that originated from the stack via RBP/RSP or returned from a memory or heap allocation function would be correctly tagged as a 'pointer' initially. This pointer could then be cast to an integer by the C/C++ code, and the integer shifted and masked in various ways. Eventually, when this value is cast back into a pointer and used as a pointer, it will work because its origin was a pointer and thus will have the right tag type.

If the 64-bit value did not originate from a pointer and was merely constructed using 'data', and then was used as a pointer to access memory, the system would naturally fault since this typically represents a bug. There are some rare case where 'data' values truly need to be used to construct legitimate pointers (e.g. OS assigns the initial stack location into RSP, internal implementation of memory allocators). In such cases, the CASTTOPOINTER instruction can be used, but these cases should be rare, and should be done in a manner that makes it hard for attackers to leverage this code to fabricate pointers.

The pointer provenance model makes it such that CastToData or CastToPointer is not typically used when the C/C++ code casts values between integer and pointers. This not only has the benefit of not bloating the code size with this casting instructions, but also makes it much harder for attackers to find code gadgets that cast from 'data' to 'pointer' to fabricate pointers.

3.6 Performing Arithmetic on Pointers

Due to the provenance model, arithmetic operations must be allowed on 'pointer's because it is very common in C/C++ code to cast between integers and pointers and perform arithmetic on either. This does not; however, mean that arithmetic needs to be allowed on 'code pointer's. Legitimate use cases where math is performed on code pointers should be very rare. As such, the default behavior on performing arithmetic operations (e.g. add, sub, shift, and, or, xor, multiply, divide, etc…) on a 'code pointer' should be to clear the tag to 'data' (or fault). In the extremely rare case where arithmetic is truly needed on a code pointer, the compiler must cast the result back to a code pointer using CastToCodePointer if needed. Again, this should rarely happen, and the code will need to be authored in a manner that makes it extremely hard for attackers to leverage the code to fabricate code pointers.

We will also disallow arithmetic operations on 'protected data' to prevent such data from being maliciously modified. The exact use of 'protected data' will be explained in Section 3.9, but if arithmetic were allowed on 'protected data' then it becomes much easier to maliciously modify 'protected data' because any piece of code that performs arithmetic on a 64-bit value could be used as a gadget to modify 'protected data'. By not allowing arithmetic on protected data, the tag would get cleared back to 'data' when protected data is accidentally provided to such a sequence. This would enable the owner of the 'protected data' to detect the tag change and fault. Again, in the rare case where protected data really needs to be modified, the CastToProtectedData instruction is still available.

3.7 Code Pointers

Code Pointers are used to carry pointers to callable functions or return addresses. There is a strong desire to separate forward code pointers and return code pointers so that one cannot be maliciously used for the other. Also, many code pointers are stored in a manner where copying the pointer does not really need to be allowed. Example of such cases include the return address on stacks, C++ vtable entries, and the global dispatch table (the PLT GOT on ELF platforms). As such, we would also like to be able to construct "Unclonable Code Pointers", where `memcpy` cannot be used to maliciously copy pointers from one location to another. We add special instructions to allow the OS to swap data from memory without breaking this guarantee in userspace code.

The way unclonable code pointers is implemented deserves some discussion. We propose implementing unclonable code pointers as "store once" pointers. These pointers will initially be constructed in a register from where it is allowed to be stored into memory once. The value stored into memory is modified to a different state where the one-time opportunity to store the value to memory has been consumed. When this value is read back into a register, the value is still tagged as a unclonable code pointer where the value can still be used by an indirect call or jump. If the value is stored back into memory again (perhaps by a malicious `memcpy` call); however, the tag associated with the unclonable code pointer gets cleared back to 'data'.

Note that it is also possible to apply the “Store Once” idea to data pointers to create unclonable data pointers. We do not believe that there is a significant benefit to allowing unclonable data pointers because data pointers typically need to be copied around in memory. Note that the “Store Once” model could be adapted for systems such as CHERI, as a variation of the linear capabilities model.

This requires five different types of code pointers:

- Forward code pointer
- Forward unclonable code pointer
- Forward unclonable code pointer before store
- Reverse unclonable code pointer
- Reverse unclonable code pointer before store

The pointers with “before store” in the name represent the state where the code pointer has been converted to the ‘store once’ state in a register but has not been stored into memory. The unclonable pointers without “before store” in the name represent the state where the one-time opportunity to store to memory has already been consumed.

At this point, the reader must be wondering: How are we going to encode these 5 different states given that ‘code pointer’s only occupy one tag type combination? We see 2 possible solutions. One obvious solution is to expand to use 3 tag bits instead of 2 to accommodate the extra states. We do not advocate this solution because it adds considerably more overhead from tag storage and movement. The other, preferred, solution is to leverage the fact that the high order bits of a code pointer are rarely (if ever) used in any meaningful fashion. If we take away the most significant 3 bits from the code pointer, we still have a 61-bit virtual address space for code which is more than enough for current generation of computers. The top 3 bits of the code pointer can be manipulated and managed along with the 2 tag bits as code pointers flow through the system and are loaded/stored in memory. We can think of these 3 bits as extra tag bits that come for free without any storage overhead or performance penalty.

Note however that taking away high order bits only works for ‘code pointer’, as ‘data’, ‘pointer’, and ‘protected data’ need to use all 64-bits of the value. In particular, interpreters and dynamic language implementations often uses *NaN boxing*, a technique where either a double-precision floating point number or a 48-bit pointer can be stored as a tagged union by using the fact that there are a number of invalid Not-a-Number (NaN) representations in the IEEE 754 specification.

Let’s discuss in more detail how these code pointer states could be used by the AMD64 architecture. We can simply modify the behavior of the CALL instruction to put the ‘reverse unclonable code pointer’ tag type on the return address as it is being placed on the stack and the have the RET instruction insist that the return address pulled from the stack must have a ‘reverse unclonable code pointer’ tag type or else the operation will fault. For the cases where we are setting up forward unclonable code pointers such as C++ vtables or global dispatch tables, we will need to cast a register into the “Forward unclonable code pointer before store” state, which gets converted to the “forward unclonable code pointer” state when the register is actually saved out into memory as the vtable or dispatch table. As such, we will need to add the instruction CASTTOUNCLONABLECODEPOINTER to enable the creation of registers in the “Forward

unclonable code pointer before store" state. Indirect call and jump instructions (e.g. CALL [RAX+8]) will fault if the tag type is not some kind of forward code pointer. Code that should only perform an indirect branch/call through an unclonable code pointer should add an extra instruction FAULTIFNOTUNCLONABLECODEPOINTER before the indirect branch/call to guarantee that the branch target is indeed stored as an unclonable code pointer. Both new instructions should also occupy the no-op opcode space on legacy systems to make the code function correctly on both new and legacy systems. For symmetry and practical use, we should also add the following instructions in the no-op space:

- FAULTIFNOTCODEPOINTER (any type of code pointer)
- FAULTIFNOTPOINTER
- FAULTIFNOTDATA
- FAULTIFNOTPROTECTEDDATA

Note that in normal usage the AMD64 architecture does not need the "Reverse unclonable code pointer before Store" state because the CALL instruction directly places the return address on the stack in the "reverse unclonable code pointer" state. Because of this, we only need to steal 2 extra high order bits from the top of the code pointer. However, in other architectures where a link register is present, such a state is needed. For example, during an ARM64 branch-and-link instruction, the return address would be placed in the link register in the "reverse unclonable code pointer before store" state. When the link register is spilled, it would be converted to the "reverse unclonable code pointer" state. Note that for architectures that do not have a dedicated return instruction, we would need an explicit assertion to check the state.

There are still some unusual scenarios where an unclonable code pointer in memory legitimately needs to be copied elsewhere. The most common such case is when an interrupt occurs when a register is holding an unclonable code pointer that hasn't been stored to memory yet. The interrupt forces the system to save the registers to memory, but at the end of the interrupt, when the memory contents is restored back to the register, the one-time memory store was already consumed by the interrupt register spill so the code behaves incorrectly later on. There are many ways to address this problem, but the simplest way is to set all the registers containing unclonable code pointers to the "before store" state whenever it is restored from an interrupt. This might unnecessarily set some unclonable code pointers to the "before store" register state when in fact it should be the "after memory store" register state, but we do not believe that this is useful for an attacker because they would have to precisely time the delivery of an interrupt. This may be possible if an attacker can schedule high-precision timers.

Another possible alternative would be to add dedicated instructions similar to XSAVE and XRESTORE that save and restore the register state faithfully without exhibiting the state modification behavior associated with the one-time memory store. When doing this, we again need to be careful that this does not become copy gadget for unclonable code pointers.

3.8 Tag preservation and Memcpy

On a system with pointer tagging enabled, the attacker can no longer directly use corrupted 'data' as a pointer (of any type). The attacker will need to resort to finding naturally callable gadgets that can

create pointers based on attacker supplied 'data'. While this may be possible, it may not be enough. The attacker will also need to find `memcpy` gadgets that copy the minted pointers to the target location. Thus, decreasing the number of attacker-callable `memcpy` gadgets is desirable.

The default behavior of `memcpy` should be to copy and preserve the tag for everything except unclonable code pointers. The generic version of `memcpy` would carry this behavior. However, we can also create another version of `memcpy` that can only be used to copy 100% pure 'data'.

In cases where the compiler can guarantee that the data being copied is only of the 'data' type and does not contain any pointers, the compiler can choose to use the data-only `memcpy` to perform the memory copying. The generic version of `memcpy` will need to be used in other cases. This should shrink the gadget space of code that contains calls to the generic version of `memcpy`, and maximize usage of the data-only `memcpy` such that attackers are less likely to find callable gadgets that can copy pointers. This effort may not be worthwhile if enough tag-preserving `memcpy` gadgets exist for an attack to succeed. More study is needed in this area to determine to what extent this could limit the gadget space.

Separately, we have the problem that there is no single standard `memcpy` implementation and some of them might not necessarily preserve tag bits. Different programs use integer, SSE, or AVX loads and stores, or REP MOVSB sequences. Unfortunately, a lot of programs also contain inline versions of `memcpy` that are either written by hand or (more commonly) inserted by the compiler. This is particularly common for small known-size copy operations, which may be replaced by a single load / store instruction pair.

Copies that use 64-bit aligned general-purpose registers will automatically preserve tags. If tags are not preserved through vector registers then some optimized `memcpy` implementations will not be possible in recompiled code, which may present a significant performance penalty (for example, a 256-bit structure copy cannot be copied by the compiler as a single AVX load and store). The REP MOVSB sequence is also problematic as specified because any single-byte load or store must reset tags to the data state and this instruction is a repeated load and store of a single byte. To make this work, we would require that this instruction copies all 64-bit sized and aligned chunks atomically while preserving tags.

3.9 Protected Data Tag for High Value Data

In this section, we discuss possible use cases for the extra 'Protected Data' tag type. The spirit of this tag type is to use it to protect data residing in memory that is high value that we wish to prevent attackers from easily modifying (without generating a fault) via corruption.

One of the most obvious use cases for 'Protected Data' is to protect high value kernel data structures. The page tables stand out as a prime example. In such a use case, the hardware page table walker can enforce that the tag bits for any page table data must be set as 'Protected Data' and fault if any page table data is tagged incorrectly. The memory manager that populates the page table will need to use the CASTTOPROTECTEDDATA instruction whenever storing any page table entries. With this change, it becomes much harder for a supervisor mode attacker to corrupt the page table as he/she would need to find a callable gadget that can fabricate arbitrary protected data values. A similar construct could be used for other important in memory data structures that are read and interpreted by hardware.

Another possible use case for 'Protected Data' is to protect critical 'data' fields in C/C++ structures/classes. A classic example is the `ByteArray` class illustrated below:

```
class ByteArray {
      _ProtectInMemory int64 arrayLength
      BYTE* byteArrayPointer
      …
}
```

If an attacker corrupts the `arrayLength` field to a large number as it resides in memory, then bounds-checked accesses to this array are now able to access most of memory.

To combat this problem, we could add support in the compiler to SAL-annotate such critical fields to denote the desire to prevent the field from being modified while residing in memory. In the above example, the annotation '_ProtectInMemory' denotes this. A field that is declared this way instructs the compile to handle the storage and retrieval of `arrayLength` from memory in the following way:

1. The value should be CastToProtectedData before it is stored in memory
2. When we read the value back from memory, we should FaultIfNotProtectedData
3. If arithmetic needs to be performed on the value afterwards, the compiler must explicitly cast the value to data and back around the arithmetic operations.

This change requires annotation in the source code, which we have been trying to avoid since it adds work for the developer. It nevertheless gives us a powerful tool to counter attackers once attackers start finding new ways to subvert a system with pointer tagging turned on. Note that this compiler feature does not necessarily need to be deployed immediately. Defenses based on this tag can be deployed in response to new attack patterns.

It is also worth noting that large numbers of protected data use cases may introduce gadgets that an attacker can use for manipulating protected data. As such, each use should be carefully considered.

4 Security Analysis of Pointer Tagging

In this section, we present some analysis of the security of pointer tagging and why we believe it will be effective at frustrating the memory safety attacker. Over time, as we improve compilers and remove gadgets that attackers find, we believe pointer tagging will enable us to completely prevent memory safety attacks in many cases. The security analysis of pointer tagging is an area where we believe more research is needed, and the authors of this paper would greatly appreciate any feedback on the security effectiveness/weakness of pointer tagging, and whether any part of our analysis below is incorrect or missing something important.

The fundamental value pointer tagging brings, is the ability to prevent (or at least make much harder) the malicious modification of a pointer in memory to an attacker desired value. Without pointer tagging, corruption of pointers can be easily achieved by any memory write primitive. With pointer tagging, the attacker must also find a way to have that memory location also get updated with the correct

corresponding pointer tag type. While we can still see some ways the attacker might be able to do this using existing gadgets in the code base, there is no question that the process is now much more complicated than before, and if we can over time remove the gadgets that the attackers need, we will have a means to prevent the attacker from modifying pointers all together.

Note that pointer tagging is only useful in preventing the malicious modification of pointers. Malicious modification of pure data is still possible even with pointer tagging. This means that pure 'data' corruption attacks will still be possible with pointer tagging. However, the amount of damage that the attacker can pull off with a data only attack will be much less compared to the ability to execute Turing-complete code through ROP/JOP/DOP which all require the corruption of pointers.

4.1 How Pointer Tagging Affects the Three Stages of Attack

To analyze the security properties of pointer tagging, let us revisit the three stages required for most memory safety attacks. The three stages are:

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

How Pointer Tagging Affects Stage One

Pointer tagging prevents some buffer overrun, type confusion and use after free bugs from being exploitable even at this very first stage. Any bug where a 'data' value that gets written by one object that is mistakenly used as a 'pointer' by another object will naturally be defeated by pointer tagging due to the fault generated when the data is eventually used as a pointer. More research is needed to determine how effective this is against stopping stage one of a memory safety attack, but it should be clear that it is able to at least block some stage one attacks.

How Pointer Tagging Affects Stage Two

In stage two, the attacker tries to find a way to use the bug found in stage one to enable arbitrary read/write on all memory in the address space. This is typically accomplished by either corrupting the base, displacement or extent of a subsequent memory access

The base of memory access is typically a pointer itself. By corrupting this base pointer to an attacker-controlled value, the attacker gains the ability to read/write 'data' anywhere in memory. Pointer tagging either prevents this attack or significantly increases the work needed for this attack because a simple data overwrite of the pointer is not possible, the attacker must find a sequence of naturally callable gadgets to materialize a desired pointer value at the desired location.

Alternatively, it is also possible for an attacker to gain arbitrary memory access through modifying the displacement or extent of a memory access. An example of an extent modification is the `ByteArray`

class illustrated in section 3.9. By corrupting the `arrayLength` field in the `ByteArray` class, the attacker can gain read/write access of 'data' to most of memory. We hope that features such as `_ProtectInMemory` can be used to block modification of the displacement and extent of data structures like `ByteArray`. This; however, is not a transparent mitigation and depends on code auditing and modification which involves more work. However, such fixes should be possible without API changes and pointer tagging at least makes it possible to fix problems like this without modifying any API interfaces.

How Pointer Tagging Affects Stage 3 of Attack

Assuming the attacker achieves arbitrary read of all memory and arbitrary write of 'data' on all memory, the attacker now wants to perform the final stage to perform a return-oriented, jump-oriented, or data-oriented attack on the system to achieve the actual goal of executing their payload. Current known variants of these attacks generally involve the attacker being able to corrupt pointers. Pointer tagging adds significant work for the attacker as they need to search for gadgets that enable them to corrupt the various type of pointers that they need to corrupt. In some cases where the naturally callable gadget space is limited, it might not be possible to find all the needed gadgets which in effect blocks the attack all together.

Also note that as a consequence of pointer tagging, once a zero day attack is discovered, we not only can fix the original stage-one code defect, but we can also remove the gadgets used in stage two and three as part of the fix. This puts the attacker in the awkward position where each new exploit will require time spent to discover new defects and new gadgets needed for all three stages. In the current world without pointer tagging, the attacker merely needs to find another stage one code defect, there was never any good way to prevent stage two or stage three of the attack from happening. But now, pointer tagging enables this, which makes the attackers life significantly harder.

4.2 Possible Ways to Modify Pointers with Pointer Tagging

As discussed in the previous subsection, pointer tagging can still be defeated if the attacker can find the appropriate gadgets to maliciously modify pointers. In this subsection we attempt to enumerate some possible ways maliciously pointer modification can still be achieved. We start with the following assumptions on what the attacker can do to begin with:

- Read all of memory regardless of tag type.
- Write attacker chosen 'data' values anywhere in memory.

Note that these assumptions assume the attacker has already achieved the first 2 stages of a memory safety attack, even though pointer tagging should make it harder for attackers to get to this stage. But for the sake of this discussion, let's just assume the attacker already achieved stage 2 of the attack and see how far the attacker can get in terms of pointer modification.

With the above assumptions, let's discuss some ways that an attacker could maliciously modify 'pointer's, 'code pointer's and 'protected data'. Note that this list is by no means complete and attackers will come up with new exploit techniques, but note that the attacker typically needs to achieve the goal of modifying the pointer at a specific memory location to a specific value. Merely modifying a specific

pointer to some different value or modifying some pointer to a specific value is usually not good enough. The attacker wants to control both the exact location of the pointer as well as the exact pointer value to corrupt to.

We have identified several possible ways in which an attacker may be able to bypass pointer tagging, for example:

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;
   ```

   An attacker that is able to corrupt data values `Offset1` and `Offset2` in above code would be able to write an arbitrary 'pointer' value at an arbitrary location. Note that this type of gadget can only be used to modify pointers if `Ptr2` was a pointer.
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 any pointer in memory to any value. This is due to the unfortunate fact that pointers can still be manipulated by code that was intended to manipulate data, because arithmetic is allowed on pointers. Note; however, that this type of gadget cannot be used to modify code pointers or protected data as arithmetic is not allowed on these tag types. Also note that this attack is also possible even if Data2 is a small constant (say 1). Even though in such a case, the attacker will need to repeatedly call the gadget many times to modify the pointer to the desired value.

3. If an attacker finds an object with a code defect (e.g. use after free), as well as a pointer fabrication gadget, then with such a defect combination the attacker can place a victim object in the freed memory location, and write an attacker chosen pointer value into an attacker-controlled location within the victim object. Note however that this type of attack can only be applied to objects on the heap and does not work for the stack, unless another `memcpy` gadget is found to copy the pointer from the heap to the stack.
4. The attacker can also break up the problem into 2 separate tasks:
   a. Find a naturally callable gadget that fabricates a pointer, code pointer, or protected data.
   b. Find a naturally callable gadget that lets the attacker perform a tag preserving `memcpy` from an attacker chosen source to an attacker chosen destination. Note that both the source and destination location would need to be controlled by 'data' in memory in order for the `memcpy` to be useful. An example for such a gadget could be:

      ```
      *(PtrDest + DestOffset) = *(PtrSrc + SrcOffset)
      ```

The assumption is that DestOffset and SrcOffset are attacker corruptible 'data' values. PtrDest and PtrSrc are 8 byte aligned pointers to 8 bytes to make it such that the tag bits are also copied in the assignment.

Note that in all the above cases, the following still holds:

- An attack on one tag type (e.g. 'pointer') will not necessarily work on another tag type. As such, if the attacker needs to modify both a pointer and a code pointer to achieve their goal, the amount of work needed to find all the needed gadgets increases.
- Unclonable code pointers should be considered as another tag type where it is even harder to find fabrication gadgets, and almost impossible to find `memcpy` gadgets.
- When the term "naturally callable gadget" is used in the above descriptions, it means that the attacker must be able to make the CPU execute the gadget purely based on 'data' only corruption. Remember, the attacker hasn't found a way to modify code pointers yet, so they cannot make the system execute into unnatural ROP or JOP gadgets yet.

While we believe that many of the defects/gadgets described above exists in common code bases, we nevertheless increase the work factor for attackers by making them search for these gadgets. The open question we would appreciate research in is finding out how much of an increase this work is and whether the increase in work is durable. We also have the ability to modify compilers/tools to detect such gadget sequences in emitted code and either warn developers about them, or possibly generating code that does not have such gadgets. When such gadgets are discovered by attackers or by our own compiler, we can either change the code to use tagged pointer values to mitigate 'data' corruption, or we can leverage the `_ProtectInMemory` compiler feature to harden the existing code and remove the gadget, without the need to break any API contracts. The hope is that over time, we can remove the presence of these gadgets such that it becomes practically infeasible to modify pointers on a machine with pointer tagging active.

Whether there are other possible gadgets that can be used to fabricate or clone pointers is a area where we would appreciate any feedback or insight. Also, how prevalent these gadgets are in the existing code base is something that is interesting to analyze, and any information would be appreciated.

4.3 Security Properties of Pointer Tagging We Hope to Achieve

This section lists the most critical security properties of pointer tagging that we hope to achieve that would justify the value of pointer tagging. The authors of this paper would appreciate any feedback on whether these security properties can really be upheld or not:

1. Attacker cannot fabricate 'code pointers': Code pointers should come into existence at code module load time as well as when PC relative addressing is used during "address taken" operations of code functions. We are assuming that in all these cases, either the CASTTOCODEPOINTER instruction is not used at all, or even when it is used, it is done in a manner and point in time where is it is impossible for an attacker to use it to fabricate 'code pointers' of attacker chosen value via a naturally callable gadget. Note that some operating systems support

loading new code and setting up new code pointers purely in with user mode code without strong verification in supervisor mode of the newly loaded code and pointers. On operating systems like this, it will be possible to construct fabricate code pointers if the attacker already achieved stage 2 of an attack when the new code is loading. We view this as a shortcoming in the operating system that needs to be addressed to block off this attack venue.

2. Attacker cannot fabricate or clone 'unclonable code pointers': Since ‘unclonable code pointers’ cannot be copied using memcpy gadgets, the only known way to fabricate or clone an 'unclonable code pointer' is to find a naturally callable gadget that contains the CASTTOUNCLONABLECODEPOINTER instruction. However, the CASTTOUNCLONABLECODEPOINTER instruction should only be used at code module load time when C++ vtables or global dispatch tables are being setup and during exception handling. The hope is that we can structure all the code containing CASTTOUNCLONABLECODEPOINTER instructions to never be naturally callable by an attacker to fabricate or copy unclonable code pointers.

3. It should be non-trivial for an attacker to fabricate 'pointers': It should not be trivial to find naturally callable gadgets to fabricate 'pointers' or modify ‘pointers’ to attacker desired value. The boundary between what is trivial versus non-trivial is hard to define, but we are simply defining it to mean that it should take a good hacker some genuine effort (e.g. at least a day) to find such gadgets in the address space it wishes to attack.

4. It should be non-trivial for attacker to memcpy 'pointers': It should not be trivial to find naturally callable gadgets to copy 'pointers' with attacker-controlled source and destination location. It should take a good hacker some genuine effort (e.g. at least a day) to find such gadgets in the address space it wishes to attack.

5. We have the ability to use compilers and tool chain improvements to detect pointer fabrication and memcpy gadgets over time and increase the amount of effort needed by an attacker to find pointer fabrication and memcpy gadgets.

4.4 Comparison of Pointer Tagging with Existing Anti-ROP Anti-JOP Tech

One of the arguments for pointer tagging is that it can be used to replace existing already deployed anti-ROP and anti-JOP mitigations with less performance overhead and possible even better security properties. As such, in this subsection, we will discuss how pointer tagging compares with Shadow Stack (Intel CET), ARM’s authenticated pointers, and Control Flow Guard (CFG).

Comparison with Shadow Stack (CET)

Intel’s CET prevents ROP attacks using a shadow stack where the return address is redundantly stored. Pointer tagging achieves similar protection without the need to maintain a separate stack and the extra memory cycles and cache space needed to access disjoint memory locations to redundantly store and load the return address. This is of course under the assumption that attackers cannot fabricate or clone return addresses which is stored as an ‘unclonable code pointer’. This is one of the basic security properties we want to uphold with pointer tagging.

Pointer tagging is superior to CET in that it is also effective in protecting against RCE attacks in supervisor mode and hypervisor mode if you also use 'protected data' to mark and protect page tables. Preventing ROP like attacks in supervisor mode and hypervisor mode has historically been uninteresting because if the attacker is able to overwrite the stack in these modes, the attacker can just as easily modify page tables to add new code pages with attacker-controlled code. Pointer tagging gives us a low-cost way to protect page tables using 'protected data' that prevents malicious modification of page tables. With this, the attacker must resort to ROP attacks again, but the 'unclonable code pointers' used for return addresses prevents ROP attacks from working.

CET on the other hand is ineffective in supervisor mode and hypervisor mode not only because it provides no means to protect page tables, but even on systems where some other mechanism is used to prevent arbitrary code pages from being added (e.g. hypervisor enforced code integrity), CET is still ineffective from preventing ROP since the attacker can still modify the page table entries of shadow stack pages to temporarily enable writes to modify the shadow stack. Such a weakness does not exist for pointer tagging.

Comparison with Authenticated Pointers

Comparing pointer tagging to ARM64's authenticated pointers: Pointer tagging should be superior due to the fact that return addresses on the stack cannot be copied around and re-used as is possible with authenticated pointers. While authenticated pointers does tie the authenticated return address with the value of the stack pointer, it is still possible for the attacker to use alloca to force ROP gadgets into the right location on the stack to be copied out and re-used. This shortcoming is not present with pointer tagging since 'unclonable code pointers' can neither be fabricated nor cloned. Also, authenticated pointers only "probabilistically" prevent ROP attacks, whereas pointer tagging's ROP prevention is 100% deterministic.

Note that there are several ways in which pointer tagging can be used in conjunction with authenticated pointers. For example, we can have sensitive pointers be signed with their in-memory location (which would protect them against simple `memcpy` attacks) and also use the signature to protect the tag bits in a store-once pointer.

Comparison with Control Flow Guard (CFG)

Control Flow Guard (CFG) is a Microsoft compiler option to emit code that looks up an in-memory bitmap to determine if an indirect call/jump destination is valid or not before taking the call/jump. It is used to prevent JOP attacks by restricting indirect call/jump destinations to the beginning of valid functions instead of anywhere in the code space.

Pointer tagging should be superior than CFG in terms of performance (no lookup table needed) and comparable to CFG in terms of security value. Under the assumption that code pointers cannot be fabricated, pointer tagging is comparable to CFG since there is no way for an attacker to fabricate a code pointer to jump to anything but the beginning of a valid function. CFG can be programmed to block indirect calls to only the subset of functions that need to be indirectly called to, but pointer tagging can also enforce the destination code pointer to be an unclonable code pointer if the

FAULTIFNOTUNCLONABLECODEPOINTER instruction is used to limit the set of usable code pointers to only be unclonable code pointers.

However, putting aside the scope of what code pointers you can swap in to make the indirect call/jump go to a different location than originally intended, pointer tagging forces the attacker to do work (to find a memcpy gadget) to copy a cloneable code pointer, whereas in CFG, no extra work was needed to corrupt a code pointer to a different value.

5 Implementation Details and Options

In this section, we discuss some practical implementation details, adoption strategies, and some possible variations of tag bit usage.

5.1 Phased Deployment

To ease the adoption of pointer tagging, we need to be able to phase in the pointer tagging feature in a world where legacy hardware that does not support pointer tagging is still widely used, as well as supporting a world where applications/libraries/drivers that do not support pointer tagging simultaneously run on systems with applications/libraries/drivers that do support pointer tagging.

The first problem can be solved by making all the new user mode instructions added for Pointer Tagging occupy the no-op op code space on legacy hardware. This enables new code generated for HW that supports pointer tagging to gracefully work on legacy hardware. The effect of these new user mode instructions should be limited to manipulating the tag bits in different ways and possibly generating a fault based on the tag type.

The problem of enabling a system to support both apps and drivers that support and do not support pointer tagging is more tricky, but can also be solved. We assume that the application/library/driver binary contains a header indicating whether the code is pointer tagging aware or not. Furthermore, we assume that hardware supports enabling and disabling pointer tagging on a per hardware thread basis. When a thread is started for an application that supports pointer tagging the pointer tagging feature is enabled, and when a thread is started for an application that does not support pointer tagging, the feature is disabled. However, that alone is not enough. A DLL that does not support pointer tagging could get loaded into a new application or the application could call into a driver that does not support pointer tagging. Thus, we would also need to disable pointer tagging on any thread/process that transitions through any code that does not support pointer tagging.

By building systems like this, we enable seamless adoption of pointer tagging by gracefully supporting legacy applications/DLLs/drivers to be used without pointer tagging being enabled. But as more and more of the software ecosystem gets compiled by a “pointer tagging aware” compiler, a higher percentage of the threads on the system start running with pointer tagging enabled to gain the extra security benefits.

Note that the observed behavior of the system when pointer tagging is disabled is to simply not fault when a pointer tagging fault is detected. As for how the tag bits are handled when pointer tagging is

disabled is up to the hardware vendor to decide but it seems like it would be easier to let the tag bits propagate through the system normally as if pointer tagging was enabled, and simply suppress the faults if pointer tagging is disabled. This should simplify the micro architecture design as it makes the propagation of tag bits follow the same path regardless of whether pointer tagging is enabled or disabled.

5.2 Only Using 1 Tag Bit

In this paper, we proposed using 2 tag bits so that we can protect both code pointers and data pointers. It is possible to simplify the proposal and just use 1 tag bit to only protect code pointers and leave data pointers unprotected. This "1 tag bit" proposal will give the system strong control flow properties that can prevent ROP and JOP, but the system would still be suspectable to DOP attacks.

Given the value of protecting data pointers as well as supervisor page tables, we believe that the extra tag bit is worth the tradeoff. If we build a system that only prevents ROP and JOP with 1 tag bit, only to find that attackers all shift to DOP based attacks, it will be much harder to retrofit an additional tag bit into the system later to protect data pointers.

5.3 Using 2 Tag Bits Without Upper Code Pointer Bits

If taking away the 3 most significant bits from every code pointer presents a problem, we can still achieve most of the security properties proposed in this paper without using those bits or growing to 3 tag bits. But we can only do this if we are willing to not have separate states for forward and reverse code pointers, and not have 'protected data'. In this configuration, the 2 tag bits would be used to represent the following 4 tag types:

- Data
- Pointer
- Code Pointer
- Unclonable Code Pointer

Note that we technically still need to represent one more state which is the "Unclonable Code Pointer before Store" state. However, this particular state only exists in registers, and is never needed in the L1/L2/L3 cache or the memory subsystem. As such, it should be cost effective to support this extra tag bit state by having 3 tag bits on all registers, but only 2 tag bits everywhere else.

5.4 Implementation of Tag Bits

There are several possible ways to implement the additional tag bits. One possible solution is to use DDR5 memory's ability to store extra ECC bits on the side of each 64-bit piece of memory. Using ECC memory would enable this feature to incur almost no additional DRAM access and thus no performance loss, although the cost of building the system would increase.

An alternative efficient hardware mechanism for storing tags is described in [2]. This design describes a hierarchical tag cache, where tags are stored in physical memory and atomicity is managed by the memory controller. This design works with commodity DRAM at the expense of incurring a second DRAM read whenever there is a miss in the tag cache (for 0-5% of DRAM accesses, depending on the

workload). The advantage of the hierarchical tag design is that it allows large ranges of tags to be cleared very cheaply, significantly more cheaply than zeroing the memory. This would allow reverting a page of intermixed data and pointers to the data state without any DRAM traffic.

The same hierarchical tag cache mechanism can also be used to implement fast zeroing of memory itself, using a separate bit per cache line in the tag state to indicate that the line contains only zeroes. Cache lines full of zeros can then set the zero bit in the tag cache and avoid DRAM writes while zeroing large regions of DRAM (even avoiding a DRAM read when reloading them) by setting this bit at a higher level in a hierarchical tag cache. This would likely result in at least a 5% decrease in DRAM traffic with current workloads (more in .NET and similar environments) and would compose well with the general security desire to zero-initialize all heap allocations.

5.5 Operating Systems Changes

Any pointer tagging scheme requires some interactions from the operating system. These are a subset of the requirements from [3]. In particular, the OS must:

- Initialize all tags correctly on process start
- Preserve all in-register tag state on context switch.
- Preserve all in-register tag state on signal entry and return (specific to POSIX systems).
- Preserve all in-memory tags when swapping memory to disk.
- Strip all tags in data sent via IPC mechanisms and on any non-pointer values transferred between the kernel and userspace.
- Control tags in shared memory.

The first three requirements should be handled transparently for most tags, as long as appropriate new instructions are added to enable the kernel to faithfully preserve tag state. The initial values in the register state (for example the RIP and RSP values and pointers to kernel-initialized data structures) must have the correct tags set.

Stripping tags across the kernel boundary is relatively easy on a system with explicit copy operations for memory movement across the kernel boundary but will require some more invasive changes on Windows, where device drivers often interact with userspace memory unmediated.

The kernel must preserve tags when paging out memory. In the CHERI case in [3], the tags are stored in the structure that associates metadata with a swapped-out page. CHERI requires 32 bytes of tags per page, so this does not add a considerable amount of overhead. This scheme adds 128 bytes of tags per page and so the OS may also wish to swap out the tags.

The swapping problem is more complex with memory-mapped files (as opposed to anonymous memory) that contains pointers. Memory-mapped files may contain interior pointers. If they do, then it is a good idea to store the address at which the file must be mapped, as well as the tags, in the file somehow. *Extended attributes* can be used to store this information on POSIX file systems, *alternate data streams* can be used on NTFS. Transmitting files that contain pointers between computers is probably a bad idea.

The OS may also wish to prevent tags from being stored in shared memory, to prevent one process from accidentally storing valid pointers that can be referenced by other processes. This would require a hardware page permission that would strip tags (implicitly setting the value to 'data') on every store.

In addition, to support legacy processes that do not enable this functionality, the operating system must perform several other duties:

- The OS needs to tell the difference between legacy apps where pointer tagging cannot be turned on and newer apps where it can be turned on and enable/disable pointer tagging appropriately.
- The OS interface layer will also need to manually set the pointer tag on input pointer parameters when a legacy app calls into an OS interface with a pointer that does not have the tag set correctly. This is only needed if we wish to enable pointer tagging within the OS.

If we decide to tag page tables or other supervisor data structures as 'protected data', code that writes these data structures will need to do the appropriate tag casting.

Finally, the operating system will need to support tag introspection via any cross-process debugging interfaces that it exposes.

5.6 Performance of Pointer Tagging

The performance penalty associated with pointer tagging should be minimal as the hardware should be able to access the tag bits with zero added latency (if ECC memory is used) or very low added latency (if a hierarchical tag cache is used). The only other reason performance would suffer is due to the extra casting instructions that need to be inserted in special cases, but those instruction occurrences should be minimal and negligible. On the positive side, the performance loss associated with pointer tagging should be much less than the combined performance loss of CFG and shadow stack. This means that the net result of this feature ends up being a performance gain in addition to the extra security.

5.7 Reliability Improvement

This feature could end up not only being a security feature, but also a reliability feature as it could catch many type confusion bugs and pointer smashing bugs that have caused reliability and stability issues in the past.

6 Related work

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 similar to 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 a novel new idea on how to strengthen computer systems against memory safety attacks. The proposal involves having the CPU hold two extra tag bits to the side of each piece of 64-bit data to denote whether the 64-bits holds regular 'data', a data 'pointer', or a 'code pointer'. This coarse grain type information stored in the tag bits is implicitly available to the programmer and the C/C++ compiler, but was never explicitly maintained by the HW and enforced by the HW. The fact that HW does not maintain this coarse grain type information and does not enforce any rules based on this coarse grain type information has enabled attackers to mix and match data and pointers in various ways to cause harm to computer systems. By adding these extra tag bits, we can prevent attackers from using 'data' to corrupt various pointers to prevent ROP, JOP and DOP attacks. We believe the proposed architecture change could be a game changer in terms of making memory safety attacks much harder to pull off. If over time, we can eliminate all the bad gadgets that would enable attackers to fabricate and copy pointers, then we can completely protect pointers from attacker modification which would significantly change the security landscape. This will take time, but we believe it is achievable given that compilers can be modified to warn developers about the generation of these dangerous gadgets. Even if we fail to completely prevent attackers from modifying pointers with this feature, we will have still improved the system significantly with a better performing ROP/JOP prevention system while also making the attackers life more miserable by complicating the steps needed to pull off DOP attacks.

8 Bibliography

[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.