Code pointer protection using tagged memory on LowRISC

Matthew Toseland

January 24, 2023

1 Design decisions

1.1 Original memo

The original memo proposed tagging function pointers in structures, C++ virtual table pointers, and return addresses, with a "read-only" tag. Hence even if the attacker can write arbitrary memory locations, if he can't change the tags, he can't corrupt existing code pointers. Plus, before calling a code pointer we should check for the tag: This provides limited protection against changing the upstream pointer (e.g. object pointer) and use-after-free. free() should clear the tags on the data being freed, and memcpy()/memmove() should copy tags, since C code often copies structures involving code pointers.

The current implementation deviates from this on a number of points. The original proposal was not fleshed out in detail, and missed some important implementation difficulties affecting compatibility, security and performance. Furthermore, it makes assumptions about the hardware that are not necessarily true: Using a whole bit in the tag for "read only" may not make sense, for example, we want more configurability (see Lucas Sonnabend's work).

1.2 "Lazy" tags and cleanup

The first problem with this is "cleanup": The memo proposed that free() clears all the tags on the block of memory being freed. But apart from the performance cost, we cannot reliably and compatibly clear the tags when an object is freed, either in C or C++. Both support custom memory allocation (e.g. the placement new operator in C++), and techniques such as object pools are widely used. It is sometimes but not always possible to clear tags in C++ destructors. And even if we ignore source code compatibility issues, clearing the tags in free() may be relatively costly.

For most code, especially C, the solution is to use "lazy" tag behaviour: Writes are allowed, but the tag will be reset to 0. This indicates whether memory has been written to, or is still "clean".

We still check for the tag when dereferencing a code pointer, so the attacker may be able to corrupt a code pointer but can't exploit it. This means that the error is not detected immediately, and the attacker may be able to overwrite values with other values of the same tag; we should use a true read-only tag when possible. In particular it is used for constant spilled register values such as the return address. Also, checking the tag protects against attacks that change a pointer used to look up the function pointer.

1.3 Tagging sensitive pointers

The original memo proposed to tag only function pointers in structs and virtual table pointers in objects.

The current code tags any "sensitive" pointer, in C or C++ code, including virtual function table pointers, function pointers, and pointers to objects with virtual methods (or to tables of such pointers etc). This is similar to Code Pointer Integrity's "sensitive pointer" definition, except that it does not include void* and char* universal pointers (see later notes). We use different tags for different types to improve security, since an error (e.g. pointer overflow) only affects its own tag type.

This allows a maintainable implementation, as a middle-end pass on the LLVM IR (a "typed assembly language" used for writing optimisations), and it allows us to provide some protection for object-oriented C code, which is fairly common in practice.

Tags are implemented at load and store time: Loading a sensitive pointer checks for a tag, and storing to one writes a tag. We do not guarantee that we write a valid function pointer in the first place (unlike CPI). But we do guarantee that a function pointer read from memory was written to memory as a function pointer, even if it's part of a union, and casting anything else to a function pointer is undefined behaviour and detectable at compile time.

1.4 memmove() and memcpy()

C programs often use memmove() and memcpy() to copy data structures including code pointers. Hence it is necessary to copy the tags along with the data, as the original memo pointed out. But if the arguments are not 8-byte aligned, it is hard to see how we can meaningfully determine which tag belongs to which data, while providing consistent behaviour for programmers.

Also, copying tags allows attackers to bypass security protections such as tagged code pointers. Even though an attacker with access to memcpy() can only copy tagged code pointers rather than creating new ones, this may be sufficient to call arbitrary functions.

Hence we need a consistent contract for the programmer, but we also need to copy tags only when necessary. At run-time the only information available is whether the arguments are 64-bit aligned, so the libc functions check alignment and call a version with tags or without tags accordingly. These functions are also exposed to the programmer (in sys/platform/tag.h).

The compiler can sometimes choose which version to call, based on the size and alignment. This can be improved significantly by using type information as well, see Security Enhancements.

1.5 What and where are tags?

Are tags a separate address space? Are they part of the memory the data is stored in?

This matters for both optimisation and security, see later sections. Clearly tags are not a separate address space, as ordinary writes can and should affect tags and vice versa. But tags can be accessed separately to ordinary loads and stores. This causes problems for optimisation and thread safety. See later sections.

The current solution is "tagged registers": Registers contain a tag, and instructions can atomically load or store both the value of the register and the tag from a memory location. This allows e.g. loading a value atomically with its tag (vital to prevent exploitable race conditions when a tag represents a property of the value e.g. type), and more complex atomic operations.

Ordinary memory instructions which don't propagate tags still check whether the tag should cause a trap, and ordinary stores (including unaligned and smaller stores) set the tag to 0. This is important for protecting against buffer overflows (e.g. involving copying strings byte-wise). See Lucas's work for more details.

The compiler does not directly use tagged values, they exist only in memory. Tags are not propagated through move operations between registers, arithmetic etc, they are purely determined based on the type of the load or store.

1.6 Tag propagation?

We could represent tagged values in the IR and allow compiler passes to set and query tags on general values in registers. This could be used for implementing many of Lucas's use cases at compile time with run-time support, using operations on tagged registers. It is more flexible than fixed rules for tag propagation through the ALU, allowing us to take into account information known to the compiler such as types and semantics. But it would require more instructions, in particular an instruction to move a tagged value from one register to another, and solving a number of problems mentioned in the back-end section. We would also need to devise some suitable representation: Any 64-bit value? A structure? An opaque structure?

For control flow protection, tag propagation in registers is not particularly useful. We could set a tag when storing a function pointer from a label – even if it is then stored to an integer, it will still be tagged if it hasn't been changed. However this is of limited usefulness: C code that it would help has undefined behaviour anyway, and function pointers are usually accessed through other sensitive pointers – which are subject to pointer arithmetic and overflows, possibly while stored as integers!

1.7 Tags as types

The current implementation uses the tag (at least 2 bits) as a type. This is very flexible, and is used by the SAFE machine on which LFP was implemented. As explained in the Table 1, we can implement a wide range of security and integrity enhancements using 3 bits, and an even wider range using 4 bits. However, this is relatively costly, and we are mostly using tags on pointers, which have spare bits anyway. Not using the top bits in pointers for types allows them to be used for other purposes – by applications (can often be replaced with tag bits), or as capability bounds (see the section on memory safety later). An alternative approach, taken by this paper, would be to use one tag bit to indicate a pointer (maybe 2 bits to give read-only and invalid as well), and the top 16 bits (or less) of the pointer to store the type (assuming a 48-bit virtual address space). The main criticism of this is that the top bits can be put to better use in implementing LFP pointer bounds, see the later section on memory safety; storing a 4-bit type would reduce the precision considerably. Also it may require more checks on load and atomic operations on store. Even a 4-bit tag is only an overhead of 6% memory and hopefully much less than that in runtime. Note that tag lengths do not need to be powers of 2, but TODO it is not clear whether we gain any performance in caches from using 3 rather than 4 bits.

2 Implementation details

2.1 LLVM back-end and Spike

The initial tagged register instructions included LDCT, SDCT (load and store atomically both the value and tag), WRT (write a register's tag and value from other registers) and RDT (read a register's tag into another register).

The back-end exposes platform-specific intrinsic functions for atomically accessing tagged memory, e.g. atomically read a memory location, returning both the 64-bit value and the tag (as an anonymous structure). Note that intrinsics are just functions, they can't do external branches. An intrinsic could trap on a particular tag value but we would need to expand it into a branch in the back-end, which is more complex than in the middle-end.

The back-end is not aware of register tags and tagged values. For example, we expand the intrinsic function int_riscv_load_tagged (via custom lowering code) into LOAD_WITH_TAG $val, $tagreg, $addr, which atomically loads a tagged memory location and returns the value in one register and the tag in another. This is implemented as LDCT $val, $addr, 0; RDT $tagreg, $val. We do not need to clear the tag on $val as it will not be propagated by later operations (also in frame spilling, may change if we have tag propagation). Typical generated IR code would then check $tagreg with integer operations to see that the tag has the expected value, and then use $val in the rest of the program (usually this is a pointer). Register spilling between the LDCT and the RDT would not preserve the tag, hence we expand the pseudo-instruction after register allocation.

Tag-level instructions have been proposed, for example "branch if register tag equals immediate". If these are added, a peephole pass (before register allocation) can easily be added to take advantage, freeing up a register and saving at least one instruction. This will require new pseudo-instructions, which may have to both access memory and branch. However a table-based approach is cheaper. In some cases we do need to check multiple possible values for a tag (non-lazy, compatibility hacks).

Alternatively we could make the back-end aware of tagged registers and represent them in instruction selection. At the moment this does not seem necessary, and it requires an instruction to move tagged values between registers and a solution to register spilling of tagged values. It may be difficult since template-based instruction selection can't deal with multiple return values.

Stack protection is implemented in the back-end: Registers spilled in the function prologue (especially the return address!) are tagged as read-only, and the tag is cleared in the epilogue. These values are not changed during the function. This is superior to stack canary based approaches because we tag the actual return address, not a special value near it on the stack.

memcpy()/memmove() are also implemented in the backend, as well as in glibc and newlib. The compiler converts small mem* operations into sequences of loads and stores. Hence we need to implement tag copying where appropriate. This happens in the early DAG generation stage of the back-end, which may cause a maintenance problem.

2.2 Tag code pointers pass

A new pass has been added to the middle-layer, operating on LLVM IR. This goes over every load and store in the program, and checks the type of the pointer being stored to or loaded from (this is usually hidden behind a bit-cast immediately before the memory instruction). A store to a sensitive pointer is changed to an intrinsic call to atomically store the pointer with an appropriate (fixed) tag. A load is changed to atomically load the pointer and the tag, and then check whether the tag is correct. If not, we call __llvm_riscv_check_tagged_failure, which by default executes a trap. This can be overridden at link time to give a meaningful error message, but must terminate the application.

The current code uses separate (lazy) tag values for function pointers, indirect function pointers (e.g. vtable pointers) and general sensitive pointers (e.g. pointers to objects with virtual functions). Our pass ensures that a value of a specific type can only be overwritten by a value of the same type, and only during an operation that legitimately writes that type (or a memcpy/memmove). Hence values loaded as a code pointer must have been written as a code pointer at some point. However, there are only 3 types, so this is a relatively weak guarantee (given overflows).

2.3 Tagging static code pointers

Sensitive constants and global variables must be tagged for the code to work correctly, e.g. C++ virtual function tables should be tagged.

Currently this is solved by setting the tags on constants and globals in an initialization function (this requires patching the simulator to not check for write access when setting a tag!). This also sets up the tagged registers' CSRs to trap on standard tag values (read only, invalid, write only).

A permanent solution might involve a new ELF section type and modifications to ld.so and possibly the kernel, to support tags in read-only sections (possibly including instructions).

2.4 Runtime libraries

memmove() and memcpy() are implemented by the compiler and by two different C libraries (newlib and a version of glibc). The libc versions of memcpy() and memmove() copy tags only if all the arguments are 8-byte aligned, including the length. The compiler attempts to determine whether this is likely to be true, and can often avoid the runtime overhead (and security risks) of this check and of copying tags. Ideally this would use the types of the arguments in a more tailored manner, e.g. copying an array of floats should not include tags.

The tag-copying and non-tag-copying versions are available as platform-specific libc extensions in sys/platform/tag.h, along with operations to store and load tags directly. LLVM sets the macros __TAGGED_MEMORY__ and __lowrisc__ to indicate that these are available.

2.5 Problems with RISCV-LLVM

Compiling C++ code with exception handling enabled fails because there is no support for user-mode exception handling yet (this appears to be an upstream hardware problem).

Currently the test scripts run the front-end, optimiser, back-end and GCC (for assembly and linking) separately. The Clang driver can run the whole process given solving some configuration issues, and this is necessary to integrate a -fsanitize= pass for tagging code pointers. TODO

2.6 Linux kernel and proxy kernel

Both the Linux kernel port and the proxy kernel now save tagged registers. When reusing a page, the kernel already clears it, which should clear the tags as well. Note that in supervisor mode the trap CSRs have no effect.

If we support "sticky" tags in future (which aren't cleared by normal writes), we will need to change the kernel.

Emulation of unaligned memory accesses may be incorrect: The kernel emulates these by fetching the words separately, but the tag checks are disabled in supervisor mode.

Changes may be needed in future to support loading tagged instructions, new ELF sections and mmap() with tags, see the Future Work section.

2.7 Spike, opcodes, etc

This is all implemented on top of Lucas's changes to support tagged registers, notably new instructions LDCT, SDCT, WRT and RDT. This affects various packages, see README.TagCodePointers in the RISCV-LLVM distribution (tag-code-pointers-tagged-registers-atomic branch) for details.

3 Security limitations

3.1 Background: Attacks

The basic attacks we are concerned with here:

Classic buffer overflow: Corrupt the stack, jump to the stack, execute shell code. Similar on heap. *(Non-executable stack fixes this, so does tagging the prologue spills)*

Return-Oriented-Programming: Since the heap and stack are non-executable, we find a convenient "gadget", a small stretch of code in the executable that forms a building block for a larger attack. We can set up arguments and return addresses on the stack and chain multiple such gadgets to perform arbitrary computation. This requires overwriting a return address or a function pointer. *(This appears to be fixed by tagging code pointers and return addresses!)*

Stack pointer corruption: Changing the location of the stack may allow ROP. On some calling conventions, calling a function with the wrong number of arguments will change the stack pointer. *(Appears not to be an issue, see below, TODO seek competent feedback!)*

Call-oriented programming: It may be possible to use valid functions as gadgets and hence execute arbitrary code, or use them to chain to (possibly restricted) return gadgets. *(This is the big question! Harder but may be possible sometimes. Discussed in detail later.)*

COOP: C++ version of call-oriented programming, does not require stack manipulation. Inject bogus (probably overlapping) objects, call their (not quite legal) virtual functions, chain them as gadgets, write memory, make system calls, do arbitrary computation. *(May be possible. Good cheapish mitigations are available.)*

Use-after-free: Find a dangling pointer bug: An object is deleted, but pointers to it remain. Delete the object and replace it with custom code, including a custom virtual table pointer (e.g. the memory allocator may reuse the space for efficiency). Call the method through the dangling pointer. *(Some variants of this still work, but harder. Full solutions include e.g. garbage collection.)*

Exploitable race conditions: It may be possible to bypass protections by changing a value before the tag is set etc. *(This is fixed by tagged registers.)* It

may also be possible to overwrite a spilled register holding a sensitive pointer. *(Could be fixed with full spill protection)*

All of these attacks are performed via bugs allowing access to memory, without necessarily being able to write tags. There is no protection against hardware bugs which allow access to the tags. *"Access to memory" is often misunderstood:* The attacker cannot directly overwrite tagged values (at least not without clearing the tag), but if there is a pointer overflow (e.g. in an array of the same type), he may be able to copy a sensitive pointer to another location. This is the basis for most remaining attacks.

Protecting return pointers, function pointers and virtual table pointers appears to prevent most of the above attacks. CPI claims to be able to prove this property, although it is more complex than the tagged memory changes and some of its implementations are broken. Also, many interesting targets such as web browsers have JITs, which are particularly difficult to secure.

One recent attack on IE was built on top of a use-after-free enabling the attacker to bypass ASLR and increment an arbitrary memory location (without returning its value!), from which they then built a full ROP exploit. The point is, realistic attackers are prepared to put a great deal of work into developing an exploit; making it harder may not make it hard enough! Complexity doesn't necessarily help much either – exploit compilers and scanners are used for ROP and COOP, and the latter uses a SAT solver.

3.1.1 COOP details

COOP involves injecting objects (which usually overlap) with valid vptr's (in the sense of being vptr's that are valid for some object, or in some versions of being an offset from a valid vptr). By calling a series of virtual functions, attackers can read and write memory, make system calls and do arbitrary computation, passing information through overlapped objects' fields.

This demonstrates why it is important to protect not just code pointers but also vptr's: The attacker cannot create tagged vtable pointers, so they can't create overlapping objects.

However, if there is an overflow involving placement new, or if the attacker can do memcpy with arbitrary parameters, he may still be able to set up a COOP exploit. Note that neither corrupting ordinary objects nor use-after-free is sufficient to do the COOP attack.

3.2 Gadgets and CFI

Defences based on Control Flow Integrity would ideally guarantee that execution follows the statically determined Control Flow Graph, but this is expensive (and tags don't help much). Practical defences often ensure that indirect function calls are made to the beginning of functions, and that "return" instructions always return to an instruction following a call.

Both call sites and entry points can be used as gadgets, but "call sites" or return addresses are generally easier to use. See e.g. here and here.

So there are three basic questions:

1. Does our stack protection prevent all variants of ROP using return gadgets?
2. Can the attacker call arbitrary functions?
3. Can pure call-oriented programming work?

There may be corner cases involving code unloading where ROP gadgets can be created. Tagging function entry points may be a solution for this case.

3.3 Stack protection

Stack protection prevents the attacker from overwriting the return address. We mark the return address itself as read only, rather than checking a canary. It also protects the stack pointer. Assuming the attacker doesn't already have access to an ROP gadget (in which case we've already lost), he should not be able to manufacture one either by changing the return address or moving the stack. On some calling conventions, passing the wrong number of arguments corrupts the stack pointer. This appears not to be the case on RISCV-LLVM: The callee saves the RA and SP in the function prologue, and these are tagged read-only. They are then reloaded in the epilogue. The attacks against CFI here rely heavily on this, although it also uses entry point gadgets.

It may be worth checking the tag when reloading spilled registers, even if the tags are read-only. This would provide some protection if the attacker somehow manages to change the stack pointer. However it might need a unique tag value, and would involve adding branches in frame spilling, so somewhat costly and difficult. It would be nearly free with a table-based approach. TODO.

TODO Request competent feedback! CFI etc wouldn't be needed if canaries worked! This is stronger than canaries, but it all falls down if return gadgets are possible?

TODO Be sure that we never use dangerous calling conventions!

3.4 Replacing function pointers

We assume the attacker can corrupt arbitrary memory but cannot arbitrarily change tags.

If the code casts ordinary pointers, integers etc to function pointers, the attacker can call an arbitrary location and do an ROP-like attack. In general this is undefined behaviour and can be detected at compile time, but it cannot be avoided in some cases, notably Just-In-Time compilation. Protecting universal pointers (void* and char*) may help here, but e.g. RISC-V instruction code might sensibly be written as integers and then cast to a function pointer. Full protection for JITs requires stronger measures such as sandboxing, which are out of scope and hopefully present already.

So assume that function pointers are only set from valid (read only!) function addresses and other function pointers.

Since we are using lazy tags, with the right overflow, the attacker can change any function pointer to the value of another function pointer, e.g. using an overflow in an array of function pointers.

We could possibly reduce the attack surface by having more tag types, e.g. different kinds of (indirect) function pointer etc (thus requiring a more specific kind of overflow).

Also, he may be able to change a pointer used to look up a function pointer. We tag all such "sensitive pointers", so again he will need a suitable overflow.

The other major attack vector is memmove()/memcpy() with chosen arguments, which also copies tags. We can reduce this vulnerability with a smarter compiler but not eliminate it. C++ code should not use memcpy()/memmove() to copy objects with virtual functions but may do so in practice, and this is supported by the current code (but not by proposed non-lazy extensions).

Using such an overflow in general will require knowledge about the location of functions and objects in the address space, so better ASLR protection would help.

In general, we conclude that with the right bugs an attacker can replace any function pointer with the value of any other function pointer, so can call any function that the source takes a pointer to.

3.5 Calling arbitrary valid functions

The attacker can call any function we put in a function pointer. Is this sufficient to do useful attacks?

Note that this is narrower than jump-oriented programming, which is in any case dramatically more difficult on RISC-V than on x86 due to fixed-length instruction encoding. It is also narrower than call-oriented programming in the broader sense of "ROP without returns": The attacker needs to find whole functions that he can use as gadgets.

This is definitely feasible with C++, but there are cheapish mitigations, see the section on COOP. For C, it is less clear, and depends on the code being exploited. Both these papers use return (call site) gadgets as well as function calls (entry point gadgets). However the first suggests that using entry point gadgets alone may be sufficient: They can be chained, it is just more difficult. We cut down the search space by forcing the attacker to use only those functions we have function pointers to. This includes every virtual function in C++ code, but not necessarily every library function.

In general, short of preventing overflows, it is not possible to rule out function gadgets being sufficient for arbitrary exploits. With full control over memory (but not tags), it is possible that both dispatcher and functional gadgets can be found, even at the level of entire functions, especially if the code being protected is large. In the case of JIT's, the attacker can manufacture new functions, though he won't have full control over them. *TODO Best way to find out is to scan real code for them!!*

3.6 Pure C++

COOP may still be feasible, even in pure C++ code, provided there is a suitable overflow in object allocation / placement new. It is much harder, in that an attacker cannot inject objects as a block of data, and in fact he can't even change the pointers to the objects without a suitable overflow.

There are several options for extending the code to prevent COOP entirely, see below.

Another relevant attack is a use-after-free: Find a pointer to a deleted object, allocate a string over the object that was just deleted, write a custom vptr. The simplest form of this attack is entirely defeated by tagging the vtable pointer. Allocating a new object of a different type may or may not allow a useful exploit. Most of the solutions to COOP also beat this, or separate pools for separate types (requires preventing custom memory allocation), or garbage collection.

3.7 Exceptions, longjmp, etc

Not currently implemented. Will need protection for jump addresses, but may also need stack unwinding code (e.g. to clear the read-only tags). Currently not implemented in hardware, and will be vulnerable when it is, but should be easily fixed.

GCC's indirect jump extension involving putting a jump address in a variable is supported by LLVM, and not covered here, and might be an attack vector. It is mostly used in interpreters, where it enables a significant performance gain. It might need a different tag to function pointers, or a non-lazy tag. The tag can be set only when a valid label is assigned. TODO

4 Extensions: Security enhancements

4.1 Tagging/marking function entry points (ROP)

In general, tagging code pointers is stronger than simple CFI defences, which are equivalent to tagging call sites and entry points. It also has different corner cases, for example recent Windows' control flow guard can be bypassed for JITs because it can't tag new generated functions; this would not be a problem with code pointer tagging, but there are other problems with JITs.

However, if code is unloaded and then the address space is reused, there may be (a limited number of) ROP gadgets available. Ideally the OS would guarantee that address space used for code is never reused for code, but if not we could tag function entry points (and possibly valid return points) to protect against this, and for defence in depth. If we are confident of our stack protection and the kernel/loader makes this guarantee it may be better to avoid the overhead.

We could easily tag function entry points and valid return points. The latter would need 2 tag bits since there are 2 instructions in a dword. The protection offered is equivalent to looking for a "call" instruction, but there isn't a "call" instruction on RISCV; in any case this is only necessary if stack protection is

bypassed somehow. However tag bits on instructions may be used for other purposes...

We would have to load the tag prior to doing an indirect call. Hence it will be loaded into the dcache before fetching the instruction, slowing calls slightly. The actual LLVM implementation might be messy given it may need to introduce a branch. This does not require an ABI change, but we only get full protection if we always check for the tags, so ideally we'd like GCC to set them as well as LLVM, even if it doesn't check for them.

It might be more efficient to implement this in hardware. On the other hand, a software implementation opens some interesting possibilities:

A more advanced form of this would involve a dword before any function, which would be the hash of the function signature (including the base class defining the interface in the C++ case). This might be tagged to prevent manufacturing it in a JIT (may not be necessary, JITs should/do obfuscate constants anyway). In general casting function pointers from one type to another in C is non-portable and may be unsafe; if we need to support it we could still use e.g. the number and sizes of arguments. For C++ we could use the fully qualified prototype, including the defining base class, and this would prevent COOP. Varargs and unchecked prototypes could still be supported, details depending on whether we know it's varargs in the header. Doing the same thing for function *return* would maybe avoid problems with tagged instructions as well as improving ROP protection, but would expand the code by 8-12 bytes per return address.

This requires an ABI change (it could be implemented with static linking, or system-wide). Hash collisions are probably not a serious concern here, but for static linking we could allocate a counter at link time.

This is a slightly more complex approximation to CFI. A full implementation would require more complex checks, and almost certainly static linking. Tags are too short to contain the required ID's. Tagging individual instructions as executable may or may not be useful.

4.2 Pure C++/COOP

We can prevent COOP entirely by either:

1. Forbidding custom memory allocation and placement new. We could refuse to compile code with placement new, and ignore new/delete operator overloading. We would still need to protect malloc/free from corruption.

2. Implementing function prototype checking as described in the previous section. This requires an ABI change or static linking (but could be implemented on the whole system level).

3. Implementing type checking using RTTI on a virtual function call. (existing LLVM passes: -fsanitize=vptr/cfi-vcall). The latter requires static

linking, but is <1% overhead. It is part of a wider CFI project for LLVM, but only provides partial protection at present.

4. Directly detecting overlapped objects. See below.

4.3 ASLR bypassing / information leak protection

Address space location randomisation is widely deployed but can often be bypassed, typically by using an overflow to read a pointer (as an integer or string). We may be able to improve matters by preventing reading pointers except when the compiler expects to read one.

This requires a tag value for an ordinary pointer. Writes may or may not be allowed for normal instructions, so this is not strictly non-lazy: Address space can be reused by custom allocation, as long as the code doesn't read the previous contents (which are clearly invalid). But reads are only allowed when the compiler expects a pointer. Obviously the level of protection provided will be reduced if the code routinely stores pointers as integers!

4.4 Non-lazy tags

Non-lazy tags would allow earlier detection of exploits and bugs, closer to the cause. This helps with debugging and identifying attacks. It could allow true read-only for C++ vtable pointers, improving protection in mixed C/C++ code (but this is of limited value if the attacker can change an object pointer instead!).

Note that in general data with non-lazy tags is accessed by bypassing the usual protections that apply to normal reads and writes. That is, a typical non-lazy tag value causes normal writes and possibly normal reads to fail, but it can be bypassed by code that expects that tag value. See the section on implementation.

Using non-lazy tags for everything would be a useful compiler option. It will not be compatible with code that uses custom storage allocation, and hence may need source changes. It also requires changes to free() to clear the tags, and non-lazy protection for malloc/free internal structures, including preventing freeing unused locations, possibly requiring a dedicated tag value for malloc (otherwise an attacker could clear the tags and then set them). Obvious extensions improve protection against information leaks, uninitialised memory access etc.

Non-lazy behaviour could also be turned on only for specific objects (e.g. C++ objects with virtual functions). In this case we need extra tag values, e.g. an extra tag bit.

4.4.1 Implementation of non-lazy tag operations

When writing a value with a non-lazy tag, e.g. a register spill, we may need to overwrite an existing value with the same tag. But the tag prevents writing. Clearing the tag first with STAG or SDCT is inefficient, may allow race conditions, and probably shouldn't be allowed. TODO

So to implement non-lazy operations efficiently, correctly and without race conditions, we need to be able to override the tag protection for individual operations. E.g. when writing a sensitive pointer, we need to be able to overwrite an existing sensitive pointer, while ordinary memory operations on the same address (such as a buffer overflow) should fault.

In general we want different, configurable behaviour for ordinary stores and loads vs for individual instructions which we know at compile time are accessing sensitive pointers. For example, we could have some tag values be overrideable on a per-instruction basis, and some reserved by the kernel for specific uses.

Loading a non-lazy tagged pointer would check for the specified tag value (which may not be readable by a normal load), fault if it was set to any other value, and return the data.

Store is more complex: When writing a code pointer, we may be overwriting an existing code pointer or the memory location may not have been initialised. We need to allow the write if it would normally be allowed, or if it is the same as the tag type we are writing. And then we need to write the data with the tag, all atomically.

4.4.2 Non-lazy tags (C++)

C++ objects (other than Plain Old Data C-compatible structs) have definite lifetimes, even with custom storage management, so we can mark and un-mark memory, regardless of how it is used by the application. Hence we can use non-lazy (i.e. read-only) tags. These must still have distinct values for e.g. function pointer vs indirect function pointer / vtable pointer. We could also use read-only tags for private sensitive non-address-taken fields and 64-bit constant member values.

This requires modifying the Clang front-end, at least to give enough metadata for a middle-layer pass to run reliably: Each constructor sets one or more vtable pointers, tagging them as read-only. In the delete call, the vtable pointers are unmarked. One complication is that each constructor overwrites previous vtable pointer values, so we need to overwrite it, possibly atomically checking that it is equal to the expected value of the parent vtable pointer, or using another tag value.

It also relies on the correct destructor being called, which is only guaranteed with well-behaved code. We should require -Wdelete-incomplete -Wdelete-non-virtual-dtor -Werror, which should result in failure to build code that deletes objects without virtual destructors through base class pointers. This is undefined behaviour which can lead to possibly exploitable crashing bugs involving freeing memory that hasn't been allocated, but which may work in practice by luck or in the absence of multiple inheritance. We could also check for it at run-time: If we are calling a non-virtual destructor, compare the vptr(s) to the expected vptr(s) and trap if it is different.

4.4.3 C++: Overlapped object detection

A further extension would be to prevent COOP (even in the presence of custom storage allocation) by detecting overlapping objects with virtual functions: Allocate an entire tag bit to indicate whether a dword is allocated to a C++ object with virtual functions. This bit must be preserved by write operations etc. Set the tag bit when allocating an object, atomically checking that it is not already set (this occurs in the "new" memory allocation, not the constructors, to support embedded objects). Clear it on deletion. TODO Can we avoid this for objects without placement new?

This is a relatively expensive countermeasure, depending on the typical size of objects. It might be cheaper to disable custom memory allocation, or check range structures.

However, it may be possible to combine the object allocation bit with the non-lazy bit. Valid writes to a sensitive pointer would ignore the lazy bit, even if a pointer is passed to another function. We could ensure that the non-lazy versions of each pointer tag behave the same as the lazy versions – except that the data cannot be overwritten by normal stores.

Since we write the tags for the whole object anyway, we could write type tags before we write any data, and check for them on storing. This would prevent writing sensitive pointers to locations not intended for them, which may help with spraying or attacks on memory safety. However we would need to be able to identify uninitialised pointers, by initialising them or more tag values, and it might be messy given that we need to set the tags in "new", not in the actual constructors.

4.4.4 Non-lazy tags (C)

For C code (with custom memory allocation), this is much harder. It may be possible to set tags when memory is used by a structure, especially if the code works with -fstrict-aliasing, but since there is no explicit deallocation this may break frequently. Clearing tags in free() only works if the program doesn't use custom memory allocation, and it probably isn't feasible to detect this at compile time for C. Non-lazy tags would thus be a useful compile time option, but not by default.

4.5 Memory safety

The definitive solution to the remaining attack surface would be some form of memory safety. It may be possible to implement memory safety only for sensitive pointers, since a non-sensitive pointer can't replace a sensitive pointer due to code pointer tagging. This is exactly what CPI does, at a cost of ~ 10% runtime and 17% memory according to the follow-up paper, but we can improve on that.

Using lightweight capabilities for sensitive pointers, such as Low Fat Pointers, would ensure memory safety at the level of malloc object allocations. Combined

with tag based pointer protection, this would ensure that 1) an attacker can only replace a code pointer with another code pointer and 2) he can only do so within the same memory object. Hence e.g. he can replace code pointers on objects in the same pool, but not in other objects.

In general a single memory allocation contains either a single object, which may contain complex sub-structure, or a pool of objects of the same type. Clearly this massively reduces the attack surface. However, it may be possible to chain a series of vulnerabilities involving different objects.

So ideally we'd like to be able to further restrict pointers to address only the single element of a structure that they are based on, e.g. in an LLVM IR pass operating on GetElementPointer operations, enforcing the C pointer semantics. There may be limitations with variable length arrays at the end of structures.

LFP allows arbitrary narrowing of pointer bounds, but the narrowed bounds are not precise. This may be acceptable in practice, but it may be possible to use a series of imprecise overlapping LFP's to "roll the pointer" from one end of a memory allocation to another, e.g. if each LFP represents an array within an object, inside a larger pool allocated as a single malloc object. This attack might be limited by alignment issues.

4.5.1 Asynchronous verification on minion cores

Many capability schemes can be implemented on a minion core. E.g. when we do pointer arithmetic, including memory accesses, on a sensitive pointer, we send the original pointer, the new pointer and metadata from the pointer (type, approximate bounds etc), to a minion core, which asynchronously verifies them, possibly doing memory accesses (but hopefully the data will be derivable from the pointer or cached already). As with Lucas's proposal, which was based on earlier work by TODO, we only need to synchronize with the minion(s) when doing a dangerous operation: A system call, or storing to memory shared between processes. However we may need to synchronize with more than one minion if the process is multi-threaded.

LFP cannot be implemented on a minion as the bounds change during pointer arithmetic. However it is easy to construct similar schemes with fixed bounds bits, which could be checked on a minion. For example, we could use a bit to specify whether the segment, is aligned on a 2B or 2B-1 boundary. We can then recover the start address with a 1-bit XOR, and apply 5-bit upper and lower bounds offsets. Arguably LFP and similar schemes are so cheap (with hardware support) that they could be implemented on the main core. If they are implemented on the minion we might need to do pointer narrowing on the main core, or not include the bounds and let the pointer be updated later by the minion or on synchronization.

A more flexible option which preserves the ability to do arbitrary narrowing, without needing to declare ahead of time what memory is used for what, is to use the address of the pointer itself to look up the bounds in a hashtable kept in shadow memory owned by the minion core. When an address storing a pointer is overwritten, even by an ordinary store, we would need to tell the minion; this

can be implemented by trapping asynchronously to the minion.

Most pointer bounds are likely to fit in an LFP-like scheme, so using LFP-like bounds when possible and storing bounds by pointer when not possible, and checking both of them on the minion, may be an efficient and precise solution. Especially if we can minimise the number of such cases by padding objects.

Schemes using minion cores are not necessarily better than the alternative; they improve average-case runtime (especially if there are relatively few sensitive pointer operations), at the cost of silicon, energy and possibly memory contention.

4.5.2 Maximising use of LFP-like encoding

Obviously we would like to use an LFP-like encoding as much as possible and only store bounds in memory when absolutely necessary. So we would like our sensitive sub-objects to fit exactly in an LFP as often as possible. So ideally we would like to manufacture precise protection from imprecise protection, by having a "buffer zone" on either side of a sensitive array large enough that regardless of the actual object alignment, we can always construct an LFP that contains only the sensitive array.

The direct approach would be to insert padding into structures and objects. This is legal (more or less) for C++ objects with virtual pointers. It is problematic for C structures and the equivalent Plain Old Data objects in C++. However, the rules around POD are mainly useful for operations such as writing structures to disk, which are not usually appropriate for code pointers, so it may be worth a compiler option. The cost would be around 3%, but this applies to each object; if we have many layers of object embedding, it may be significant. We could add an __attribute__ to add the appropriate padding to structures.

4.5.3 Explicit storage allocation and non-lazy tags

Another option is to say that non-sensitive data (and even non-array sensitive pointers if we use another tag) can be included in the buffer zone. This only protects against "rolling the pointer" if we can use tags to ensure that sensitive pointers are only written to memory locations which are intended for sensitive pointers of the same tag class. This requires that space is explicitly allocated and we set types on each dword; see the section on non-lazy tags and object overlap detection. So again this probably only works well on C++ code, or at least well-behaved C code which always uses malloc/free.

Another option would be to give each sensitive array an ID, and include it in the pointer, and on the tag for the memory location. This is complex and expensive because of the need to support pointers to different sized objects and sub-objects, and not generally feasible without atomic cache-line-level tag operations. Various schemes of this kind are outlined in Appendix 2.

It may be more practical to store type information for ranges of memory in a minion's shadow memory as a hash or tree. This would allow deducing exact

bounds from the address and the compile-time type, either to check pointer arithmetic or on casting from a non-sensitive pointer.

"Type information" here is a tree: A given memory location may be part of several nested objects down to a single byte, and can be addressed at any of these levels. To save on memory accesses, we could compress the type information by storing the location and type of the top-level fixed-size object, and computing offsets from that. TODO Is the compile-time type sufficient? Do we want to store e.g. the depth of the node in the sub-object tree in the pointer?

4.5.4 Details

Pointer arithmetic is implemented in LLVM IR as GetElementPointer operations. Thus an IR pass could be implemented to make sensitive pointers into capabilities. Void pointers should be treated as sensitive.

Casting from a non-sensitive pointer to a sensitive one should be supported via an extension to the C library to get the bounds of the memory allocation containing a given pointer (e.g. by keeping a pointer to the first allocation on each page), but should generate a warning. If using LFP on its own, we could warn on structures that can't be protected precisely. On the other extreme, one advantage of schemes involving static memory allocation is we can deduce reasonable bounds based purely on the address and the compile-time pointer type.

This requires an ABI change, static linking or compatibility hacks for talking to legacy libraries (which will become the weakest link!), but it could be implemented system-wide.

A common objection to LFP is that applications already use the top bits in pointers for their own purposes. In particular, garbage collected interpreters tend to use the top bit to indicate a pointer. This only affects a minority of applications that can be changed to use a pointer tag.

Full mandatory capability support, e.g. allowing passing capabilities between processes as a protection mechanism, would require further hardware extensions, e.g. for immutable tagged values which can be passed around in registers and memory but can't be modified.

4.6 malloc()/free()

We need lazy tags because lots of code does custom memory management. We don't need to change malloc() or free() to prevent control flow hijacking, unless we use non-lazy tags. However we can relatively cheaply mitigate a number of other attacks including information leaks and some versions of use-after-free by tagging memory as invalid when it is freed and then marking it as write-only when it is allocated. Meaning that once it is written to it can be read from; the kernel may need to be aware of this when emulating unaligned writes, and we may need to fault on sub-word writes too, to set the value to zero. Hongyan's thesis combines marking unused memory as invalid with a reallocation buffer to minimise use-after-free, but this does not seem to be deployable without further

work due to memory usage. We should also protect internal memory allocator structures.

We need to write the tags for the whole block being freed/allocated. This may be costly, even if using STAL. TODO

4.7 AddressSanitizer etc.

More generally, we can cheaply detect use of uninitialised memory on the stack and the heap. There are expensive -fsanitize= options for this, but AddressSanitizer could probably be reimplemented cheaply with tagged memory. This may require kernel support for tagging new pages when lazily allocating pages (e.g. when expanding the stack).

Linear buffer overflows can be prevented by adding read-only "bumpers" between stack-allocated objects, within objects etc. See the section on memory safety for some more detail and another application.

4.8 General register spill protection

Protecting general register spills, and possibly non-address-taken stack-allocated values in general, could be implemented in the back-end. This would be helpful because sensitive values can be spilled to the stack during intermediate computations, and won't be protected by the load-store protection, so are vulnerable to a race condition. However, we would need to either use non-lazy tags and clear them explicitly on exiting the stack frame, or use lazy tags and check them on loading.

In any case we can't afford to use another register when spilling a register, so we need hardware support for setting a tag from an immediate (either in a register or on storing), and possibly checking a tag against an immediate. This is much easier/cheaper with a table-based solution.

If we were to implement tagged value propagation, we would need to store tagged values when spilling. We can store and load the values using SDCT/LDCT, but some of the tag values on the registers may have effects on memory behaviour. And we want spill protection, so we need to change the tag to ensure it isn't overwritten. One solution is to limit the range of compiler-propagated tag values, but using an entire tag bit to indicate a spill would be wasteful, unless it can be combined with other issues. It probably can; see the section on non-lazy tags. If this is combined with other use cases of memory tags, we need to 1) set some bits, 2) preserve some bits from the register and 3) preserve some bits from the existing memory value, without using another register.

Another implementation detail: This probably needs pseudo-instructions expanded after register allocation. RA and low level passes may expect a single instruction for a spill (see e.g. isStoreToStackSlot()). Note that prologue/epilogue spills occur after RA so probably can't use the same pseudo's. Also we should ideally avoid tagging the spilled registers in frame spilling.

4.9 Even more tag types?

What has been implemented is a crude form of type system.

Given that most attacks involve overflows, it might make sense to have more tag values, e.g. to distinguish between void(*p)() and void(**p)(), or even different kinds of functions. If there are no pointers of the latter type, an attacker cannot use an overflow to swap pointers of the first type. We could allocate a limited pool of types to maximise protection at link time, but this would be a significant amount of work and run-time/deployment cost for a limited improvement in security.

It might also be beneficial to have a tag type for C++ vtable pointers, as distinct from C/C++ indirect function pointers. This may require changes to metadata from the front-end.

4.10 Tagging ordinary pointers

It would be useful to have a tag type for an ordinary (non-sensitive) pointer. As explained above, this can be be "lazy", i.e. writable, and support custom storage allocation, and yet be non-readable to improve protection against ASLR bypassing.

This will provide less protection to code that stores pointers as integers, but unless memory accesses are aliased the code should still work correctly.

4.10.1 Garbage collection

We can use the same type for garbage collection: Tag all pointers, and don't free an allocated object until all pointers to it have been freed. This gives us temporal safety, i.e. prevents use-after-free. This works provided that the code does not include a full blown memory allocator; object pools are fine, we get GC at the malloc() level.

Garbage collection is not safe in general, if C code does things like storing pointers in integers, and xor'ing pointers (the former is common, and explicitly supported via "intptr_t"!). This is detectable at compile time. Also GC may cause unacceptable performance problems (pauses!) for some applications, and they may have their own (possibly copying) GC. So this would have to be an optional compiler flag for compatible code. LLVM already has run-time support for a garbage collector, although it would need to be adapted to recognise tagged pointers.

We could use a special tag type for redirects (as in SAFE) and therefore use copying collection, but this might be relatively costly, involving a branch or a trap.

It might be possible to support code using intptr_t with tag propagation, but this is problematic with arithmetic on integer values holding pointers.

4.11 Further improvements to memmove()/memcpy()

Currently the compiler only uses the non-tag-copying version of memmove()/memcpy() if the move is of a known size which is not a multiple of 64 bits. The tag copying versions can bypass tag protection, so ideally we'd like to use them as rarely as possible, based on the types of the arguments. E.g. void* is a universal pointer, so we should use the run-time version; if copying between two structures which include function pointers, we should use the tag-copying version; and if copying an array of floats (or integers shorter than a pointer), we should use the non-tag-copying version.

char* is a particularly interesting case as it often indicates strings, but is occasionally used as a universal pointer (this requires more casts though). Currently the LLVM IR can't distinguish between the two, but recent versions of LLVM, which incorporate CPI, have extra metadata for this; it is worth looking at how CPI handles this. TODO

Also in some cases we may be able to deduce the correct version without knowing the exact size, this is more of an optimisation.

Currently this is implemented in SelectionDAG, so it may need help to get accurate type information (perhaps SelectionDAG::getSrcValue()?).

5 Suggested tag allocations (TODO no void*)

5.1 1 bit

Tag	Meaning
0	Normal data
1	Code pointer

Note no vtable protection! Even if we add metadata to the front-end, using the same tag for vtable pointers and function pointers is risky.

5.2 2 bits (Minimal functionality)

Tag	Meaning
0	Normal data
1	Code pointer
2	Indirect code pointer
3	Sensitive pointer *or* read-only/non-lazy

5.3 3 bits (Lots of functionality, use cases)

Tag	Meaning	Notes
0	Normal data	
1	Pointer	*E.g. for GC, maybe unreadable. Possibly lazy checked.*
2	Code pointer	*Lazy checked.*
3	Indirect code pointer	*Consider separate tag for vtable pointers. Lazy checked.*
4	Sensitive pointer	*Lazy checked.*
5	Read only	*E.g. debugger watchpoint, const values. NOT lazy.*
6	Write only	E.g. malloc/free information leak protection. NOT lazy.
7	Invalid	Spilled register, boundary, unallocated space, read watchpoint. NOT lazy. Note comments on spill protection

5.4 4 bits (Non-lazy support)

Bit 3 = non-lazy, set when an object is allocated with known contents where it will be deallocated. Bits 0-2: Similar to 3 bit case.

Because the last 3 values are non-lazy, we free up some tag values. E.g. we'd like to have a different code for invalid vs for a register spill, and for a vtable pointer (subject to frontend support) vs for an indirect code pointer, or even for malloc. A variable mask may be needed when checking tags on memory access. Also we may have some non-overrideable system tag values, see Lucas's proposal re sandboxing. We might want more values for void* or for different kinds of sensitive pointers (indirect jump targets, exceptions, or different kinds of code pointers); values might have to be allocated at link time. But with fixed values we can merge most alternative uses, at least security and integrity related ones, into this type scheme. However, concurrency-related uses don't fit.

6 Comparison with StackProtector, CPI/CPS/SafeStack, CFI etc.

The current code should be faster and safer than StackProtector as it does not need a canary and directly protects the return address.

CPS uses a private store to ensure that function pointers and vtable pointers can only be written to by operations which are intended to write such pointers. This is precisely what we are providing here (but we have several classes of sensitive pointers), and our implementation should be significantly faster in the worst case, at least once optimisation problems are overcome. CPS does more static analysis and only tags a function pointer if it is set to a valid location; we don't, but assigning anything else to one can be detected by existing compiler

flags. However, if we want to support more messy code, we could use tagged registers to propagate this information.

CPI protects a wider range of pointer types, providing full memory safety for sensitive pointers by tracking the object that every pointer is based on. CPI claims to provably prevent control flow hijacking. However both CPI and CPS rely on a private memory store. The initial x86-64 implementation used information hiding and was exploitable, and proposed solutions provide probabilistic protection or are costly (Intel hardware extensions may eventually be used once they are reintroduced). CPI costs 8% average and 40% worst case on x86 standard benchmarks.

Tagged memory could be used to implement such a store, or a minion core. We could do checks asynchronously on the minion, but updating pointer metadata would block. Also it might be possible to detect sensitivity of void* pointers at run time using tagged registers. An alternative but presumably equivalent approach would be to make all sensitive pointers into capabilities, see above.

Code pointer tagging provides better protection than simple CFI schemes but is weaker than full CFI, which is costly. See the discussions on ROP and gadgets.

7 Compatibility issues

7.1 Aliasing and the void* type hole

Aliasing, e.g. writing a sensitive pointer as an integer and then reading it as a pointer, will cause problems, and code which does this may break. However such code is generally technically illegal, and will not run correctly without -fno-strict-aliasing, because it interferes with important optimisations.

Code which stores sensitive pointers as void* and transparently casts them back to sensitive types should run correctly as long as it is consistent aliasing-wise, but will have reduced protection, just as with code which stores pointers as integers.

7.1.1 Strict aliasing rules

The C standard includes "strict aliasing" : A pointer to one type cannot usually overwrite a value of another type. This can be disabled for low-level code with -fno-strict-aliasing, but it enables important compiler optimisations and is on by default at -O2 and above in GCC.

Strict aliasing implies that a single memory location has a specific type, although it may change over time, and may not be easily determined in advance in some cases (e.g. variable length arrays). So if we read a sensitive pointer from a memory location, it should have been written as a sensitive pointer in legal C code. This is exactly what code pointer tagging guarantees. But there are some exceptions, explained below. We may even be able to set the type of memory locations in advance, although this can't be detected reliably in general.

Character pointers (char*) can alias any object and access its contents at the level of individual bytes. However, overwriting a code pointer via a character pointer and then calling it will likely result in undefined behaviour. And it's a common exploit. Code pointer tagging effectively prevents this, even though it may be legal sometimes. We support memmove/memcpy, the obvious common case where it is useful.

We support unions in the "tagged" sense: Writing a code pointer to a union and then reading it will work. Writing another element of the union, e.g. an integer or a non-sensitive pointer, and then reading it as a code pointer will break. Some versions of the C standard take this view but compilers generally support writing to a union as one type and reading it as another. However, with pointers this is not so clear-cut: Because of the union, the pointers can legally be aliased. But that does not tell the compiler that the object pointed to by the union is aliased! So preventing this is reasonable, and well-behaved code should not be broken by this.

Void pointers are "universal", i.e. can point to anything, and can be silently cast to/from anything. It is worth considering a tag type for void*. Char pointers can access individual bytes of other types, but in general casting them back to the other type violates aliasing rules. In practice char* is mainly used for strings but is still used as a universal pointer sometimes. TODO We should seriously consider a tag type for universal pointers. CPI protects both char* and void* but tries to detect non-sensitive char* string pointers statically.

This does not mean that a void pointer can be aliased to any other kind of pointer. In general, void pointers are not necessarily the same size or format as other pointers, so "void**" is a pointer to a void*, not a pointer to any pointer. Some C code might take the address of a pointer element in a structure, cast it to void**, and then use that to update the structure, e.g. in a "swap pointers" function. This is another violation of aliasing rules, but is supported by a configuration option EXPENSIVE_VOID_COMPAT_HACK. This allows reading a void* when expecting a sensitive pointer and vice versa, but does not allow aliasing function pointers to void*. It introduces many additional branches for each load, so is expensive with the current implementation, but would be cheap with a table-based approach. It should be a command line configuration option.

In conclusion, code that compiles without warnings with -fstrict-aliasing -Wstrict-aliasing should generally run correctly with code pointer protection. C code that needs -fno-strict-aliasing may or may not work correctly, and may need additional compile-time options e.g. turning off protection of sensitive pointers.

7.2 intptr_t etc ("integer hole")

Storing pointers (other than function pointers) as integers is legal, and is supported by the standard "intptr_t" typedef. Code which does this should work with reduced protection, but some proposed extensions will break, e.g. tagging ordinary pointers. Provided that the code still stores function pointers correctly,

we still provide protection against basic ROP-like gadget attacks.

Tag propagation would allow marking function pointer values, which can be passed around but should not be modified (so don't need to be propagated through ALU operations).

Propagating a "pointer" tag through registers and ALU operations is likely to be expensive for limited gain. See the proposal for DIFT taint tracking. We cannot usefully track the type of a pointer in an integer either: Low-level code may well take non-sensitive pointers, do arithmetic on them (as a void* or an integer), and cast the result to a sensitive pointer.

7.3 memcpy/memmove

Compatibility problems are possible with very low level C code. In general, copying a structure including sensitive pointers will work correctly because it will be 8-byte aligned. And calling the memcpy/memmove generically with void* pointers will check at run-time whether the arguments are aligned. However code that copies unaligned blocks of data (or blocks of unaligned length) that happen to include sensitive pointers may fail. Supporting such code would require changes to the memcpy/memmove variants.

TODO Occasionally the mem* optimisation passes may produce such code. This needs to be looked at.

7.4 Configurability (TODO)

As with StackProtector, some low level code that manipulates the stack directly may break. Stack protection should be configurable on a per-function level with an __attribute__ as it is with StackProtector (we should support the same attribute). Also we may want a compiler option for using lazy tags for spill protection (may need hardware support as registers may not be available).

The front-end should integrate the TagCodePointers pass as a -fsanitize= pass, and there should be options for how many tag values to use, spill protection, etc. This should improve compatibility.

Compiler options should be added to disable tagging sensitive pointers and indirect function pointers (to improve compatibility) and to emit an error on casting integers to code pointers (to guarantee that there is no "integer hole").

7.5 Binary compatibility (TODO)

In general we care mainly about source compatibility rather than binary compatibility. However, compiling everything in statically has a number of serious drawbacks; sometimes we'd like to link with dynamic libraries compiled with GCC without tag protection, even if it results in reduced protection (it will; real exploits usually target the unprotected library!).

Calling third party libraries built without tagging may break in some cases: Passing a function to a library function, or returning one from a library function, will work fine. But if library code manipulates sensitive structures, it may not

set the tags. We should have a function-level __attribute__ to disable tag checking, but it would also make sense to define an attribute on structures defined by shared header files. This could be set by the front-end on shared header files under a specific prefix, and disabled by adding an opposite attribute in the file (inside an #ifdef __TAGGED_MEMORY__). This needs further research: How common is this anyway? And how would it be implemented? The problem may be fairly common with C++ libraries. And even if structures are defined by shared libraries, are they modified by them? Can we avoid this by only tagging function pointers? (Which still beats simple ROP!) Etc.

8 Performance limitations

In theory we should expect the performance cost of code pointer tagging to be minimal, since the L1 cache includes both the data and the tag for each 64-bit machine word, and the tags have to be fetched regardless. Setting tags may result in extra memory writes. Hierarchical tag caches will make un-tagged memory cheaper and therefore result in a larger performance hit for using tags.

Checking a tag on load currently involves two extra registers, and four extra instructions including a conditional branch. Even setting a tag involves at least two instructions and one register. Hardware changes may reduce both of these significantly (in particular, setting a tag from an immediate, and branching on comparing a register tag to an immediate). The registers are probably more of a problem than the instructions. The branch should have only a minor impact. At a higher level, e.g. with a rule-table implementation, loading a sensitive pointer involves checking the tag, and stores including normal sub-word stores become atomic operations (storing with non-lazy tags is a more complex atomic operation). These may be expensive on some cache architectures, but should work fine on the current Rocket LowRISC core. Loads do not become atomic operations, unless many of Lucas's use-cases.

At present the TagCodePointers sanitize pass is run by scripts rather than by the front-end driver, and if optimisation is enabled the sanitize pass runs at the end of the middle-end IR optimisation. Hence it does not prevent memory-allocated pointers being moved to registers.

If we run more passes after it we may need to adjust later passes to be aware of the new memory intrinsics. AFAICS this depends mainly on integrating it with the driver, how much the middle end can do with the generated branches, and particularly on how to deal with loop optimisations. TODO

There are concerns that some loop optimisations (Scalar Evolution Analysis) can create untyped variables and thus break type checking. So far I have not been able to demonstrate this problem. It is not clear whether this is a problem for store types (affecting tagging code pointers) or just a fake base address (possibly affecting memory safety extensions). TODO

It may be necessary to adjust the mem* optimisation pass to avoid generating unaligned copies involving sensitive variables.

In any case it is not possible to do meaningful benchmarks yet. There has

been some discussion of measuring miss rates in the functional simulator. It will be some time before there is actual hardware support including the cycle-accurate simulator and FPGAs. The next step will be to compile some real code... Ideally we would like to use standard benchmarks and compare to CPI, CFI (e.g. the partial implementation in LLVM which arguably complements tagging, e.g. try xalancbmc benchmark for code with lots of vcalls) etc.

8.1 Bulk access to tags

Several important operations (for extensions) involve accessing tags in bulk, e.g. storing / loading blocks of tagged memory to disk, malloc() and free() in the original proposal, tagging a page when allocating it etc. The current hardware provides an instruction STAL to set the tags on a whole L1 cache line at once (TODO performance implications? Does it load the whole cache line?)

9 Essential future work (TODO's for basic functionality)

Note this is rather low level code TODO's, dependant on the current hardware design, which may change radically e.g. with rule tables.

9.1 Remaining (re)design decisions

Do we need to fully support the void* type hole? Does real code cast void* to and from function pointers? Does it alias sensitive pointers to void** etc? Some support implemented, but slow.

Do we need to provide tag protection for void*, as CPI does, using another tag type? E.g. do we care about race conditions corrupting pointers between them being malloc'ed and being used for a function pointer table? There is a source level option for this.

If so, need to distinguish real universal pointers from strings! Needs more recent LLVM, see CPI.

Does real code store function pointers, sensitive pointers as intptr_t etc? Supporting function pointers as ints requires tag propagation (immutable value). Protecting integers storing sensitive pointers requires more complex propagation through arithmetic, e.g. DIFT. We could try to support this, or we could detect it and emit a warning.

Number of tag bits. If small, choose between sensitive pointers and read-only/non-lazy.

9.2 Third party review

The whole of the rest of the document, but more specifically:

- Stack protection.

- Performance difference between 3- and 4- bit tags.
- Cost of STAL (lower priority). Performance in general.

9.3 Bigger fixes

Correctness with optimisation enabled: Reproduce problems with loop optimisation UglyGEP's, and fix. Determine what the proper order of optimisation passes is and adapt the driver. Adapt memory-related passes that will run after the TagCodePointers pass to be aware of the operations (IntrReadArgMem allows reading any amount of memory).

Implement protection for exceptions and longjmp. This requires making exceptions work on RISCV-LLVM.

Sort out initialization properly *(essential but eventual)*

- ELF section, loader mods, possibly mmap() changes in the kernel?
- More complexities, see below

9.4 Compatibility and configurability

Integration: An -fsanitize= option has been implemented but crashes. More recent upstream code may be needed. Currently running the full compiler driver requires some config hacking.

Build and test some real code. Lots of questions can best be answered with real code (void*, libraries etc), try building a few random things or get some systematic data on behaviour?

Stack protection should be configurable on a per-function basis using the same attributes used by StackProtector.

Tag checking should be configurable on a per-function level using an __attribute__.

Ideally, it should be possible to define tag checking and setting (none, lazy, non-lazy etc) on a per-structure level, and a per-file level for #include's. The compiler should set this by default on included files in configurable locations, but allow overriding by libraries that do support tags (this introduces more boilerplate code...). See the section on compatibility. Needs further research – can an attribute apply to a large block of code, not just a single function?

Make the number of tags used configurable.

Do we need to support "appending" linkage in static code pointer tagging? Lots of other possible incompatibilities here.

See also Compatibility section.

9.5 Minor security enhancements (incomplete, see enhancements)

Protect the GCC indirect jump extension (either turn it off and use a jump table as before, or tag any variables holding jump targets; might need a different tag?). Test this.

Spill protection. Needs ability to set a tag from an immediate. Also need to either clear tags on leaving a stack frame or use lazy checks with a branch introduced post-RA (difficult and needs hardware support). May need pseudo-instructions, as storeRegToStackSlot callers assume this is a single instruction. Possible but sub-optimal without overrideable user-CSR or similar.

Lazy vs non-lazy fairly sensitive to hardware details e.g. somewhat different (cheaper!) with table-based approach.

Compiler option for using non-lazy tags everywhere even in C code. (Will break if they are using their own allocator).

Consider separate tag for vtable pointers (look at metadata).

See Security Enhancements section for many more possibilities.

9.6 Optimisation

Currently atomic tagged memory pseudo-instructions don't accept an offset, but LDCT/SDCT do. This typically costs an extra instruction. Fix this by passing in an offset. This will need changes to custom lowering. (Test e.g. VPtrTests/SubclassTest.opt.s. Might need a ComplexPattern, certainly need custom lowering).

Consider more peephole passes after instruction set is more final.

See also Performance section.

Use WRT with an immediate to optimise.

memcpy/memmove should use LDCT; SDCT directly. For glibc/newlib this will need inline assembler. For LLVM builtin mem* (in SelectionDAG), this may require deeper changes because of register spilling issues.

9.7 Cleanup

General cleanup, especially in TagCodePointers. Consistently do or don't use IRBuilder?

Tests: Integrate tests with the existing test automation. Implement more tests.

GCC as well as LLVM should set __TAGGED_MEMORY__ and __lowrisc__ when it is available.

Try to move platform specific mem* handling from SelectionDAG back into RISCV back-end. This will require upstream refactoring since SelectionDAG only tries the target-specific lowering after trying to lower a short mem* operation itself.

Polymorphic intrinsics: Make the new intrinsics take an arbitrary pointer, or an arbitrary 64-bit value, matching the correct output type. See e.g. int_sadd_with_overflow.

Make the intrinsics use char (i8) rather than i64 for tag values.

Spill protection: Consider using pseudo-instructions to avoid complicated stepping-over-instructions changes.

Tell LLVM to convert platform-specific libc extensions into intrinsics internally, this is preferable to inline assembler (although we need that anyway).

Clean up mess with CSRW/CSRR duplicated/over-specialised instructions. Problems with SelectionDAG automatic matching, but we don't need it, we can call them directly when needed from custom lowering and from the move reg to reg function. See commits around b1c343fe47790bb4fef68efa0770fb4aae83c595.

Do we need to use a MemIntrinsic in SelectionDAG? Related issues e.g. SelectionDAG.getStore() does a lot of housekeeping (presumably related to dead store elimination etc?) that our memory ops don't.

Sanity check initialisation priorities. Some optimisations broken by non-65535 priorities, but we do need to set tags and CSRs before other init runs. Also set CSRs once on load, not once per module. Both go away if we solve the initialisation problem properly e.g. in the loader.

Port to more recent upstream LLVM/RISCV-LLVM (when it is stable enough).

9.8 Tagged code segment / mmap / paging

We should have a clean solution for loading tagged object code. Often loading involves memory mapping parts of a binary, so ideally we'd like to support memory mapping with tags. This would also enable swapping, which is important for usable LowRISC systems. This requires changes to libc and the kernel, and there are technical challenges:

Where do we store the tags? Do we store the tag memory separately on disk, as it is stored separately in actual memory? This will result in more seeking, and interact with memory allocation. Or do we try to store the tags close to the actual data, which might be very complex and waste space? This is beyond the scope of this project but will be needed for a usable LowRISC system, along with the kernel clearing tags on reusing memory pages etc.

There is a Cambridge Part 2 project proposal for CHERI that appears to include this. Note also that solving how to cleanly load tagged code is essential for Lucas's table-based tag checking/updating proposal since it uses tagged instructions extensively.

10 Appendix 1: Summary of (further) hardware recommendations

10.1 Basic functionality with mostly lazy tags (2 bits)

- Atomically set a tag when storing (with a mask if other schemes)
  - *E.g. Tagged registers, LDSC; masking needs more help*
- Atomically load and check the tag (with a mask if other schemes)
- All loads and stores trap on store to a location with a specific ("read only") tag
  - *E.g. current tagged registers CSRs*

- Ordinary stores must clear the tag (with a mask if other schemes)
  - *Current tagged registers code supports this*

10.2 Register spill protection

- Store with a tag provided by an immediate (possibly masked)
  - *E.g. WRT with an immediate, store with a tag immediate, etc.*
- Check against a tag provided by an immediate (lazy version, general spilling or SP change)
  - *E.g. branch on immediate (masked?) tag value*

These are both helpful in general but only vital for general register spill protection

10.3 ASLR protection and non-lazy support

- Load or store with a tag that would cause a trap for a "normal" load or store
  - *E.g. user CSRs vs supervisor CSRs*

10.4 Non-lazy support including non-lazy spills

- Fail on tag value not equal to that expected (which wouldn't trap on normal load/store)
- Store with tag but fail on complex tag checks on the existing tag
  - E.g. mask then check for ordinarily allowed values + one special value
  - E.g. mask then check for two possible values (either of which might be otherwise invalid)
  - *Without a mask, this could be done with a big immediate and trap CSRs*
  - *With a mask, this needs LR/SC or a table-based approach*
- Atomically check a value and tag and update to a new value and tag (C++ read-only vptr's)
  - *This is one way to solve problems with overwriting vptr's in constructors*
  - *E.g. LR/SC or CAS with tags*

10.5 Mixed tag use

- Atomically update tag on store with a mask (both with tags and normal stores)

10.6 Tag propagation

- Instruction to move a tagged value from one register to another
- Complex masking on storing a tagged value
  - Some bits from memory, some from register tag, some set to fixed values
  - Has to not need another register as happens in register spilling
  - E.g. 2 insn's: Mask the tag in the register then do a store which preserves some memory bits
- Simple AND masking on loading a tagged value
  - Again this has to happen without needing another register
- Tag propagation on integer arithmetic (!!)
  - *E.g. ALU tag propagation table; similar to DIFT*
  - Useful for compatibility with perverse code e.g. intptr_t (does not provide full protection!)

See also mixed tag use.

10.7 Instruction tags

- Check target has a specified tag on an indirect jump
  - ROP protection.
  - Compiler can specify whether it is a return or a call.
  - There are 2 instructions in a dword! We may need to indicate which of the two instructions is a valid return address in the instruction tag.
  - *Instruction tags may be used for other purposes. This could be a problem.*
  - *Meaning of tags may differ for instructions vs data, and instructions are generally readable and can have embedded data – some protection schemes use this.*

10.8 Miscellaneous: Stuff to think about

- Consider cache-line level tag operations
  - *Only if fast and atomic, probably not feasible?*
  - *Only needed for stuff that looks expensive/obscure*
- Consider setting tags in bulk cheaply, not necessarily involving the data
  - *Does STAL always load the cache line from memory?*
  - *All the same for a large area e.g. from a page - hierarchical tag cache?*
  - *Useful for malloc/free, write-then-read, AddressSanitizer etc.*
  - *Set varying individual values – e.g. mmap, C++ object initialisation*
- STAG should check the existing tag when writing
  - *Only necessary for sandboxing, not relevant here, but this appears to be a bug*
- Ordinary reads/writes, even unaligned and sub-word, should check tags and if appropriate clear
  - *Implemented but needs tests in unaligned case*
  - *Might need kernel changes since it is emulated*
- Consider how to implement write-after-read (write only → after writing becomes read/write)
  - *Trap on read → trap on unaligned write, subword write?*
  - *Useful for malloc()/free() information leak protection*
  - *If writing tags is always similar cost to writing data, then just zero it out.*

11 Appendix 2: Tagged memory and capabilities

Carter et al uses a tag bit to indicate capabilities, and requires them to be power of 2 size and power of 2 aligned (hence wasting 25% of memory on average!), but also provides code pointer protection. LFP avoids most of the overhead, and provides more precise narrowing.

If we have cache-line level atomic tag operations, and if large objects are always cache-line-aligned, we can implement memory safety using tag bits.

To provide malloc-level protection, a fat pointer would contain 6 bits for the "segment size" B, a 1 bit offset relative to the bit after the segment (so we don't have to allocate segments on power of 2 boundaries), and a 2 bit "object ID".

We have 7 more bits which can be used for permissions, validity, (probabilistic) temporal protection etc. The actual address, B and offset specify a 2^B-sized block of memory that the object must be contained in, which is enforced during pointer arithmetic. There can only be 3 objects of that size in the memory allocation, so a 2-bit object ID will uniquely specify the object, even if objects are deleted and added in between other objects. This requires 8 bits which are distributed across the cache line as 1 tag bit per dword.

Extending this to support (fixed!) sub-objects is not straightforward: A memory allocation consists of a tree of sub-objects, starting at the malloc level. It may include arrays. And we can declare pointers to any level of that tree – which we expect to be bounds checked. And at compile time we only have an approximation to the detailed type, e.g. a char* points to an array char[N].

We would like to allocate an ID to every dword, which when combined with the sub-segment (defined by the pointer value and the size and offset fields), uniquely identifies a sub-object. The sub-object may itself have sub-objects, so we need to identify higher-level sub-objects with a bit-mask or at most a range comparison.

For sub-objects of the same size we only need two versions (separated by a single alternating bit), but in general sub-objects at the same depth may be of wildly varying sizes, so we may have many sub-objects at the same tree level. We can introduce virtual nodes to ensure that the direct children of any node are all the same power-of-2 size, although this may limit the depth of the tree occasionally. Hence we only need 1 bit per level, and furthermore, the pointer only needs to include the sub-segment size, the 1 bit offset, the 1 bit OID, and the depth (the first three levels can indicate malloc-level OID's so we don't need that either). But the tag needs to include the full OID (for every level).

So the tag is 6 (segsize) + 2 (oid) + #levels bits.

The pointer needs 6 (subsegsize) + 1 (suboffset) + log2(#levels+3) bits.

E.g. for an 8 level object tree (with some restrictions on nesting of small objects), we need 2 bits per dword, and have 6 bits free on the pointer for permissions bits, temporal ID's etc.

Byte-level capability narrowing can be added by including 2 sub-objects in the cache line: A 6-bit offset indicates where the second begins, and the second sub-object's tree ID is specified by its depth. If a narrowed range is smaller than a cache line (64 bytes), we can specify it directly in the pointer: No object is larger than 2^{48} bytes so we use two bits from the segment size plus another 10 bits to specify the full range (although no spare bits e.g. read only).

Even with (linked) cache-line-level tag operations, malloc-level protection is 1 bit, and byte-level could be anything up to 4 bits depending on the tree depth. There are no extra memory accesses. Even if we have fast cache-line tag operations, this turns a sub-word store into an LR/SC pair with many simple instructions between the two. However, the proposal allows arbitrary narrowing, down to the byte level, provided that the tree of sub-objects is constructed statically (e.g. when creating a complex C++ object), or linearly (e.g. slicing a network buffer into a series of packets, which may then be sub-sliced). It does not provide arbitrary, possibly overlapping, run-time slicing (except for very

small slices).

Other schemes are probably cheaper in practice.