Pointer Provenance in a Capability Architecture

Alfredo Mazzinghi
Dept. of Computer Science and Technology
University of Cambridge
alfredo.mazzinghi@cl.cam.ac.uk

Ripduman Sohan
Dept. of Computer Science and Technology
University of Cambridge
ripduman.sohan@cl.cam.ac.uk

Robert N.M. Watson
Dept. of Computer Science and Technology
University of Cambridge
robert.watson@cl.cam.ac.uk

Abstract

We design and implement a framework for tracking pointer provenance, using our CHERI fat-pointer capability architecture to facilitate analysis of security implications of program pointer flows in both user and privileged code, with minimal instrumentation. CHERI enforces pointer provenance validity at the architectural level, in the presence of complex pointer arithmetic and type casting. CHERI present new opportunities for provenance research: we discuss use cases and highlight lessons and open questions from our work.

Keywords provenance, pointer, CHERI

ACM Reference Format:
Alfredo Mazzinghi, Ripduman Sohan, and Robert N.M. Watson. 2018. Pointer Provenance in a Capability Architecture. In *Proceedings of Conference (TaPP'18).* ACM, New York, NY, USA, 5 pages.

1 Introduction

Spatial memory safety seeks to ensure that invalid memory accesses cannot occur, e.g., that a modification to a field of an object in memory does not change any memory location outside the address range allocated for the object. A large portion of security vulnerabilities arise from the lack of memory safety; whereas out-of-bound accesses to arrays or structures are allowed in C code on common commercial architectures, it is important to be able to reason about what represents a pointer in memory, which part of memory the pointer is actually pointing to, and whether it points to the intended object. On most common architectures, this is complex because integer data values can be arbitrarily interpreted as pointers; thus, there is no reliable way to distinguish data and pointers at the machine level. Moreover, pointers are offsets into a process address space and do not carry information about the pointed object (e.g. size or constant-ness).

The CHERI [6, 14] architecture attempts to solve the problem of memory safety in C while retaining high performance and compatibility with existing code bases. CHERI pointers are fundamentally different from other architectures. A CHERI pointer is an unforgeable token of authority (a *capability*) that grants access to a region of address space.

The CHERI architecture enforces *pointer-provenance validity*, ensuring that a valid pointer can be derived only from other existing valid pointers. Pointers carry additional information about how they can be used to access the referenced region: they have a notion of *base* and *bound* addresses in which they can be dereferenced, and they have *permissions* that limit the way they can be dereferenced (e.g., read-only).

Moreover CHERI builds *in-address space compartmentalisation* on top of its pointers: pairs of code and data capabilities grant access to a compartment and are used to perform function calls that cross a protection domain [11, 12]. Capability pointers are *monotonic*: a new valid pointer can be created only by restricting the access rights of another valid pointer.

Why pointer provenance?

CHERI implementations are required to track pointer provenance, because the architecture enforces pointer-provenance validity [13]. CHERI pointers are conceptually organised in a tree, with the root at the boot-time capabilities spanning the whole address space that the processor provides at platform reset. Internally, our CPU implementations do not build this tree; instead, they manage a tag bit to maintain whether a pointer is valid and can produce other valid pointers.

CHERI provides three instructions that generate a new pointer from another valid one[1]: *fromptr* converts a legacy C address to a capability, *setbounds* and *andperm* respectively shrink the memory bounds and clear access permission flags of a capability. By tracing these instructions, we can determine the parent-child relationship between two capabilities and reconstruct the provenance tree. Tracing load, store, call and capability arithmetic instructions, allows to follow the flow of capabilities as a program runs.

At any given time, the set of capabilities to which a protection domain has access determines which parts of address space it is allowed to access, and how. Understanding how programs manipulate capabilities is therefore critical to understand how effectively a program is using CHERI features and corroborate our understanding of the architecture. Pointer provenance enables the analysis of stack and heap allocation patterns, protection domain isolation and delegation of resources via capabilities.

Contributions

We present a pointer provenance analysis approach that takes advantage of unique features of CHERI and applies

TaPP'18, July 09–13, 2018, London, UK
2018.

[1] This is a simplification of the CHERI-MIPS ISA, for a more accurate description the reader should consult [13]

to both user and supervisor code (Section 2). We introduce interesting use cases enabled by our approach and implement a prototype to support our experiments (Section 3). We show that CHERI (and fat-pointer architectures in general) provides interesting opportunities for provenance analysis.

2 Approach

CHERI allows to reliably distinguish valid pointers and to determine bounds and access permissions on objects in memory at an architectural level. We can obtain the information needed for our analysis by inspecting the instructions executed by the processor, without additional instrumentation of the binaries. Instruction traces, generated by our FPGA softcore and CHERI QEMU, include every instruction executed, along with its results, exceptions, and memory accessed.

There are both advantages and disadvantages to using this low-level tracing approach: traces are large and do not scale well with the duration of code execution; however, they allow us to trace both user and privileged code written in any language that can be compiled for our architecture, while being completely transparent to the executed code. The scalability problem introduced by the size of instruction traces can be mitigated: our implementation attempts to process different chunks of the trace in parallel.

Our prototype is constituted of four main components (appendix fig. 1): the *trace source*, the *trace parser*, the *data model*, and the *model visitors*. A trace source, in our case QEMU or an FPGA, generates instruction traces in a custom binary format. The parser reconstruct the provenance tree of capabilities, records function and system calls, and extracts debug information and symbols from the traced binaries. The information collected by the parser is stored in a convenient intermediate representation, the data model. Debug information and symbols are used to map addresses to symbols, and help recognise interesting events in the data – such as capabilities returned by `malloc()`. The data model is accessed using the visitor interface and allows to retrieve information about the flow of pointers in the traced code. The visitor interface can be used to modify the model (e.g., to perform some optimisation steps) and to extract information that can be used later to support visualisation tools.

Our initial prototype is built for a CHERI implementation based on MIPS and focuses on the analysis of pointers in a single-threaded user-space program on a single-processor system. However, with few changes, our approach can support tracing across threads and different address spaces, and may be ported to other CHERI-like architectures, because the data model can be kept architecture independent, and only the parser needs to be adjusted.

2.1 Abstraction Model

Our data model consists of three graphs: the pointer provenance graph, the function call and system call graph, and a protection domain transition graph (appendix fig. 2). In the model, we use the number of CPU cycles from the beginning of the trace as a relative measure of time.

In the pointer provenance graph, vertices represent a new capability. When the parser encounters one of the instructions that create a capability, a new vertex in the graph is generated. Every vertex in the pointer graph is associated with a list storing all dereferences of the pointer, along with the address of the dereferencing instruction. A separate list keeps track of all memory locations where the pointer is stored and the time interval in which the memory location holds the pointer. Pointer graph vertices also store pointer creation and destruction times. The destruction time is set when the pointer is overwritten in the last memory location or CPU register in which it is contained.

The call graph is used to hold information about function and system calls. A vertex in the call graph represents a function call or a system call, with the associated stack frame. These vertices are associated with vertices in the provenance graph that represent the pointers visible in CPU registers at the time of the call and at the time of return.

The domain transition graph is used to keep track of protection-domain crossing. It is similar to the call graph, but holds domain-crossing function calls. Each vertex represents a domain-crossing call and is associated with a vertex in the call graph, which represents the domain entry function used.

The model does not use a portable format, such as PROV. We attempt to use a compact representation given the large amount of vertices in the graphs, however we did not fully optimise the data structures to favour experimentation.

2.2 Optimisations

It is possible to introduce some optimisations to lower the time required to parse the trace, the time to inspect the data model and reduce the size of the model.

A naïve implementation of the parser iterates the trace sequentially; however, the information in traces is local to a relatively small number of instruction entries. The full content of the registers can be recovered from the most recently executed exception handler, where all registers are saved; the remainder of the data is available in the trace entries for each instruction (address, updated register content, memory accessed etc.). Because of this locality property it is possible to parse the trace using a MapReduce approach: each thread produces a subset of the provenance and call graphs, which are merged to generate the final model. The merge process is also necessary to verify the consistency of the state of memory and registers at subgraph boundary – specifically, which memory locations are expected to hold pointers and which pointers are expected to be in CPU registers.

In some cases the compiler creates temporary capabilities, for instance when reducing bounds and permissions of a capability, a *setbounds* instruction is followed by an *andperm*. The capability returned by *setbounds* is only used as input to *andperm*, therefore this sequence of instruction is logically a

single pointer creation. The parser normally produces two vertices in the graph, one for each new capability, however we can merge these two operations into a single vertex representing the resulting final pointer. This helps save space and reduce the amount of vertices in the provenance graph.

Optimisations of the dereference and memory location lists have not been implemented in our prototype yet, and are left for future work. The list of pointer dereferences and locations where pointers are stored in memory should provide fast access to the elements; e.g., a radix tree could be used for memory locations. Entries in the dereference list may also be coalesced into a single entry representing the dereference of a range of addresses, which is a common pattern for memory copying and zeroing operations. This can be accomplished by checking whether the last dereference was adjacent to the current one and updating the number of bytes accessed in last dereference.

2.3 Limitations

Interactions among different threads, different processes, and in general different address-spaces are tricky to handle. Our approach enables this case with minimal changes, although we leave the implementation of a prototype that solves this problem for future work. Thread support influences only the call and domain transition graphs, because the address space where the pointers are valid does not change. It is sufficient to detect when context switching occurs and to build a different call and domain transition graph for each thread. Multiple processes can be handled by replicating the current model for each process.

The parser needs changes to be able to detect context switches and address space switches, as well as properly handle shared physical memory pages, because capabilities in such pages may leak to other address spaces. The former can be done by having the kernel emit markers for context switches in the trace, or detecting and analysing calls to the context switch routine in the kernel. Handling physical memory sharing is generally harder, and requires keeping track of the TLB (Translation Look-aside Buffer). In MIPS this is visible from traces, because the TLB is software-managed; in other architectures, the trace producer may need to emit special trace entries for this type of events.

The approach proposed is able to observe only a single execution path. It cannot make claims about every possible program execution. This limitation is acceptable, given our goal to aid understanding and debugging CHERI features. Although our work is restricted to CHERI, existing solutions, such as Intel PT and ARM CoreSight, show that there is a growing interest in fine-grained hardware tracing. Our analysis technique suggests that similar solutions may benefit from providing insight on pointer provenance.

3 Use cases

We identify some use cases that are representative of how the model can be used to reason about high-level properties of traced code. Our work focuses on security-related considerations, although the technique presented can be used for other purposes as well. We developed a prototype tool [1] that is currently being used to measure bounds effectiveness in small C programs compiled for CHERI, such as openssl.

Pointer lifetime. We define pointer lifetime as the interval between pointer creation and the time when the tag bit is unset on the last copy of the pointer, either in memory or in a CPU register. From a security perspective, it is interesting to detect whether pointers are used, or could be used, after they are freed. The destruction time stored for provenance graph vertices, allows to determine whether a pointer remains in memory after being passed to `free()`. In particular, we are able to determine whether some pointers are left behind (e.g., in the stack after a stack frame is deallocated), and could be exploited by a potential attacker.

Reachable memory. At any point in a CHERI program, the memory that can be accessed is the transitive closure of the set of capabilities immediately available in CPU registers. This analysis is useful to reason about compartment *isolation* and object *delegation*. We can detect the set of private objects by observing pointer sub-trees, where all pointers are never accessible outside a compartment. Delegation is detected by observing capabilities exchanged across a domain boundary.

Debugging security violations. When a security violation occurs (i.e., by dereferencing a pointer outside bounds or use-after-free), the provenance graph can be used to find the origin of that pointer and determine in which function it has been created.

Measure reachable ROP gadgets. By inspecting the memory referenced by executable pointers, it is possible to determine at a given point in the trace how many ROP gadgets are available to an attacker. A separate analysis of the binaries that are traced is necessary to look for gadgets.

Measure bounds effectiveness. We can reason about how well CHERI pointers are protecting access in the presence of different allocator policies for setting bounds or compiler options. We observe whether the size of the bounds in the traced code is reduced, while the code continues to function as expected. If some large pointers disappear in favour of pointers with tighter bounds, some parts of the address space that are not required and could potentially be used maliciously are no longer accessible.

Protection enforcement consistency. We can use ELF metadata and DWARF debug information to check whether the bounds enforced at run-time are consistent with the size of data structures and type qualifiers of variables.

4 Open Questions

With our work, we hope to raise the interest of the community towards the research opportunities for provenance analysis offered by capability architectures.

Scalability.

There are different ways to address the scalability issue presented by the size of instruction traces. Avoiding storing the trace would probably offer better performance if we only care about the intermediate representation and never look back in the instruction trace. Interactive inspection of the data would require optimisations in the model, to allow fast and possibly parallel search operations. Our simple model works for tracing fairly small programs and non-interactive data analysis. The instruction trace format may be improved to let the hardware perform some form of compression of the trace entries. Visualisation techniques are also a challenge: it is hard to display our provenance data in a way that exposes meaningful properties of the code.

OS Level Support. The OS may help detecting certain events. It is worth enabling the kernel to pause and restart tracing with thread granularity to simplify single-thread tracing. Adding no-op instructions to emit process and thread identifiers during context switching, and page table information may be useful to simplify tracking multiple processes.

5 Related Work

Memarian et al. [9] suggest that the C language standard provides a provenance interpretation for pointers. Our approach benefits from how CHERI maps this concept into its provenance validity enforcement.

Large execution traces are used by omniscient debuggers such as TOD [10] and ODB [8] to step backward in time and recover the root cause of bugs more easily. While TOD and ODB focus on Java, the technique is generic and there are commercial solutions such as TimeMachine and UndoDB that work for native code. Our traces could be used to extend such debuggers with pointer provenance information. However, we are interested only in pointer manipulations, for which our data model needs to store less information.

TAEDS [15] is a trace-based framework for tracking Java data structure evolution history. Our work could support a similar analysis technique for native code and determine data structure bounds and visibility.

Provenance tracking techniques have been used to support different security-related analyses, such as tracking the origin of null pointers [3], detect dangling pointers [4], identify leaks of sensitive data [2, 7]. Hardware techniques have been proposed to detect pointer corruption attacks [5]. Our approach attempts to be more generic and could be used to support similar types of analysis.

6 Conclusion

We present a methodology that takes advantage of architectural features of a capability architecture for provenance applications. We describe how raw data from instruction traces can be organised to support reasoning and practical measurement of high-level security-related properties of CHERI codebases. Despite limitations of our technique, we show that a capability architecture that enforces pointer provenance validity is valuable to enable provenance analysis at a low level, with minimal code instrumentation.

Acknowledgments

This work was supported by a research contract from Google, Inc. This work was also supported by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-10-C-0237 ("CTSRD"). The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

References

[1] 2018. cheriplot Github repository. (2018). https://github.com/CTSRD-CHERI/cheriplot

[2] Steven Arzt et al. 2014. FlowDroid. *ACM SIGPLAN Notices* 49, 6 (jun 2014), 259–269.

[3] Michael D. Bond et al. 2007. Tracking bad apples. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications - OOPSLA '07*, Vol. 42. ACM Press, New York, New York, USA, 405.

[4] Juan Caballero et al. 2012. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. *ISSTA* (2012), 133.

[5] S Chen et al. 2005. Defeating memory corruption attacks via pointer taintedness detection. *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on* (2005), 378–387.

[6] David Chisnall et al. 2015. Beyond the PDP-11: Architectural Support for a Memory-Safe C Abstract Machine. *SIGARCH Comput. Archit. News* 43, 1 (March 2015), 117–130.

[7] Adam P Fuchs, Avik Chaudhuri, and Jeffrey Foster. 2010. SCanDroid : Automated Security Certification of Android Applications. *Read* 10, November (2010), 328.

[8] Bil Lewis. 2003. Debugging Backwards in Time. (oct 2003). arXiv:cs/0310016

[9] Kayvan Memarian et al. 2016. Into the depths of C: elaborating the de facto standards. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2016*, Vol. 51. ACM Press, New York, New York, USA, 1–15.

[10] Guillaume Pothier, Éric Tanter, and José Piquer. 2007. Scalable omniscient debugging. *ACM SIGPLAN Notices* 42, 10 (oct 2007), 535.

[11] Robert N.M. Watson et al. 2015. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *Proceedings - IEEE Symposium on Security and Privacy*, Vol. 2015-July. 20–37.

[12] R. N. M. Watson et al. 2016. Fast Protection-Domain Crossing in the CHERI Capability-System Architecture. *IEEE Micro* 36, 5 (Sept 2016), 38–49.

[13] Robert N. M. Watson et al. 2017. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (version 6).* Technical Report UCAM-CL-TR-907.

[14] Jonathan Woodruff et al. 2014. The CHERI capability model: Revisiting RISC in an age of risk. In *Proceedings - International Symposium on Computer Architecture*. 457–468.

[15] Xiao Xiao, Jinguo Zhou, and Charles Zhang. 2011. Tracking data structures for postmortem analysis. *Proceeding of the 33rd international conference on Software engineering - ICSE '11* (2011), 896.

A Appendix

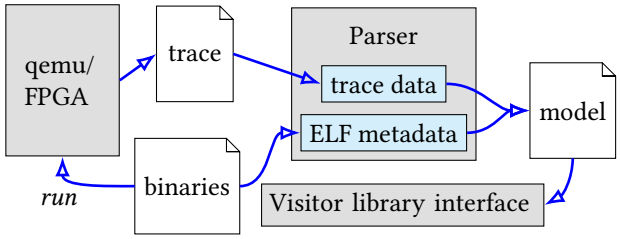

Figure 1. System architecture

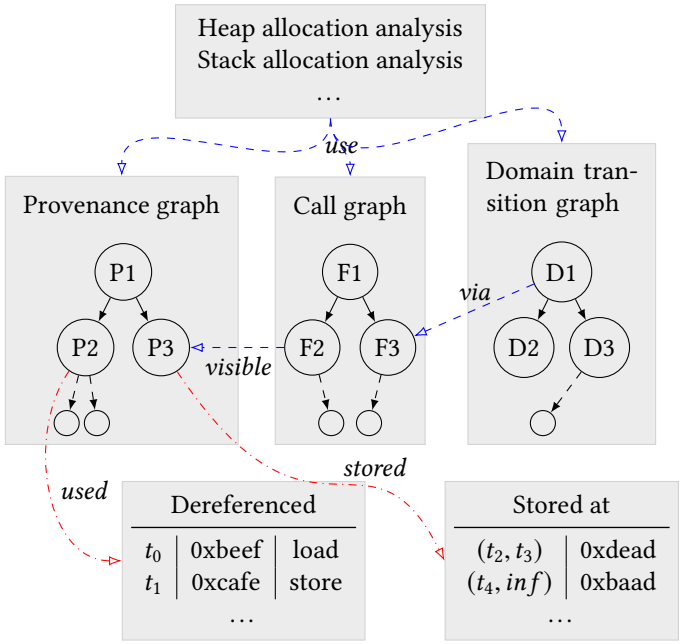

Figure 2. Model data structure. Domain transition graph entries are associated to call graph vertices that represent the domain crossing function call. Call graph entries are associated to pointer arguments, return values, and all the pointers visible by the function. Each pointer graph vertex is associated to two tables. The *Dereferenced* table records the time, address and kind of pointer dereferences. The *Stored at* table records where and how long a pointer have been stored at a given location. Additional analysis steps are built on top of this model.