Tagged memory and minion cores in the lowRISC SoC

December 2014

lowRISC project team
Computer Laboratory
University of Cambridge

lowRISC-MEMO 2014-001

This document is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License

Tagged memory and minion cores in the lowRISC SoC

Alex Bradbury, Gavin Ferris, and Robert Mullins

Introduction

The lowRISC project aims to produce a fully open-source System-on-Chip (SoC) implementing the open RISC-V instruction set architecture [15]. We plan to manufacture in volume and to distribute low cost development boards. The lowRISC platform intends to be open, secure, high performance, and flexible. We describe two features that will help support such goals, namely tagged memory and minion cores. We would like to start with a disclaimer of sorts – this document describes planned, unfinished and unpublished work. We are sharing our plans at such an early point in the open source spirit of "release early, release often" and to get as much feedback as possible. We are grateful to those who have already been so generous with their comments and ideas, which have been very helpful in further developing our thinking.

It would be grossly negligent to embark on a project like lowRISC without giving thought to the possibilities of improving upon the security features available in currently shipping processors. Tagged memory associates metadata with each memory location and can be used to implement fine-grained memory access restrictions. Attacks which hijack control flow can be prevented by using this protection to restrict writes to memory locations containing return addresses, function pointers, and vtable pointers. Importantly, we anticipate this can be implemented with a worst-case performance overhead of a few percent and a similarly low area cost. This fine-grained memory protection can be used automatically by the compiler, meaning improved security is available to existing programs without source code modifications. We intend to provide tagged memory alongside security features which are already commonly deployed such as secure boot, encrypted off-chip memory, and cryptographic accelerators.

The flexibility of many commercial SoCs is often limited by the provision of a fixed set of peripherals and limited or restricted documentation. Attempts to extend low-level I/O functionality by running code on the SoC's main processors is often both difficult and very inefficient. As a result, it is not uncommon for external microcontrollers and FPGAs to be added to fulfill the requirements of even relatively simple projects. We hope to overcome such issues and promote an open and flexible design through the addition of simple processors ("minions") with predictable timing and direct access to I/O. In addition to supporting software-defined peripherals, we envisage minion cores being used to filter I/O and run tasks with lower performance requirements in order to reduce energy usage.

The two features outlined actually represent the revival of two rather old ideas. The approach of dedicating cores to handle off-chip I/O goes back at least as far as the 1964 CDC6600 and has been adopted in various forms in commercial designs many times since. The provision of multiple core types representing different points on the power/performance spectrum is also a well established idea [9] and is exploited in many commercial SoCs today. Systems such as the IBM AS/400 featured tagged memory, and it has been used extensively in the research community to implement shadow memory or when investigating implementations of capability systems.

Tagged memory

Motivation

Computer systems are expected to not only perform their computations correctly and quickly, but also securely. We have been carefully considering the addition of new hardware features which may help to protect users of the lowRISC platform. Without the same concerns regarding legacy software faced by existing mainstream architectures, we are in an ideal position to introduce new changes to improve security. Despite this ability to make changes without concern for existing software, the addition we are pursuing could actually be added in a backwards compatible way to existing architectures. An analysis of the CVE database [13] showed the three most common exploit causes in the advisory database are cross site scripting, SQL injection, and buffer overflow. For some classes of attack, it isn't feasible for hardware designers to mitigate them directly. These are best approached through support of generalised security features like compartmentalisation through virtualisation. Stopping these attacks requires changes to development practices or programming language features, and the recent 'shellshock' [4] vulnerability is a good example of this. Memory safety issues on the other hand have frequently been addressed by changes to the hardware, compiler, and runtime.

The 'war in memory' [18] has evolved over the past several decades. Basic process isolation through the virtual memory system has been augmented by the widescale deployment of non-executable stack and heaps, address space layout randomisation, and stack canaries. Attackers are no longer able to inject and execute arbitrary shellcode, and the major mobile phone platforms execute installed applications in a sandbox environment that restricts access to dangerous libc functions. The foe's arsenal has evolved to include more sophisticated attacks which do not rely on code injection. In return-oriented-programming (ROP), arbitrary code execution is achieved by corrupting a code pointer (such as the return address) and chaining together the execution of multiple 'gadgets'. These small sequences of code in the original binary are used to implement the attack. In order to prevent such exploits, we must make it impossible for programming errors to allow code pointers to be overwritten.

We believe that memory tagging and the fine-grained memory protection it provides gives a meaningful increase in security at a low cost. A straightforward application of this feature is to protect return addresses, vtable pointers and other code pointers on the stack and heap in order to prevent code hijacking attacks. However we believe one of the advantages of tagged memory (and the reason we propose offering 2-bit tags) is that it can also be used for efficient implementation of a range of other tools, analyses, and security policies.

The tagged memory solution

In a tagged memory system every memory location is augmented with one or more tag bits that are accessed in parallel with the data. The simplest scheme to meet our goal of protecting code pointers, like the return address, would associate a single tag bit with each word in memory. Tagged memory can be approximated in software by maintaining a 'shadow memory', where loads and stores are instrumented with additional code to update this externally stored metadata appropriately. We aim to add hardware support for tags to maintain very low performance overheads.

In order to protect against corruption of the return address, the compiler is modified to generate instructions to mark the stack location where it is stored as read-only using an appropriate tag. In the event of a buffer overflow, any write to that memory location will fail and an exception will be raised. The vtable pointer inside C++ objects would also be marked read-only in the same way. This can be implemented as an automatic compiler transformation and is not subject to issues like false positives seen with more complex schemes such as dynamic information flow tracking. Using tags in this way protects against stray writes (e.g. due to overflow of an array), but is not alone sufficient to prevent attacks which exploit a use-after-free. A use-after-free bug is where a pointer remains in use after the underlying memory object has been freed. In the presence of such a bug, when the object is freed the tags marking the vtable as read-only would be cleared and the attacker may be able to manipulate memory allocation so as to cause their desired address to be placed at the location of the vtable pointer. Protecting against this simply requires checking that the tag is still present when loading a memory location used for storing a code pointer or vtable. With such an addition, the adversary is unable to hijack control flow to a specific location as long as the program does not allow them to tag arbitrary values with the same tag being used to protect control flow. A final caveat is that in the presence of a use-after-free bug, an object of a different type with compatible layout may be allocated in place of the previous object with a different but appropriately tagged vtable pointer. This gives some limited opportunity for exploitation, though is restricted to executing methods defined in such objects. Existing software techniques which put allocations of different types in separate pools can be used to totally prevent this class of attacks [1].

The bare minimum changes required for a tagged memory system to work are:

- Instructions to get and set memory tags.
- A modified compiler which will tag every code pointer and verify the tag is still present when loading it. The function prologue and epilogue will set and clear the tag on the return address.
- Modifications to the memory allocator to clear the tags upon freeing an allocation.
- Modifications to libc so that memcpy and memmove also copy the tags.
- The compiler and C++ runtime must be updated so that the presence of the tag on the vtable pointer is always checked upon virtual function call.
- The kernel virtual memory system must be updated to persist the appropriate tag bits when moving a page to secondary storage.

Implementation

As with many things in computer architecture, although the basic idea is simple there is a wide array of possible implementation choices with subtly different trade-offs. Provided there are only a small number of tag bits, tagged memory can be implemented efficiently by providing a tag cache at the memory interface to logically extend its width. L2 and L1 cache lines are extended to hold the tag bits associated with that line. The costs are a small increase in the L1 and L2 size and the extra area required to implement the tag cache. This cache can be small (e.g. 8KiB) and still have a very high hit rate. This implementation option requires only fairly small changes to the core and, for 64-bit words with 2-bit tags would give a 1/32 (∼3.2%) storage overhead and similarly small performance overhead provided the tag cache has a high hit rate. Once implemented, we will explore in detail whether 2-bit tags offers the best trade-off in terms of functionality versus overhead.

The choice of a 2-bit over a single-bit scheme increases overheads by 2x, but in return allows up to three distinct tags to be used. Every 64-bit word has a 2-bit tag associated with it. You can use these two bits arbitrarily, but there is also the option of having some of the mappings trigger an exception. The CPU can be configured such that the tags have the following meanings:

- 00: No effect
- 01: Exception on read

- 10: Exception on write
- 11: Exception on read or write

The possibility of monitoring an arbitrary address allows for infinite hardware watchpoints, a valuable feature for a wide range of software analyses [7]. This can be used in order to implement a more complicated software-supported tagging or shadow memory scheme with lower performance overhead, as accesses to particular parts of memory can be instrumented without rewriting all load or store operations.

Additional uses

The primary motivation that led us down the path of tagged memory is the desire to protect regions of memory at a finer granularity than a single page. However, support for tagged memory actually has a much wider range of potential uses. Fine-grained memory protection can eliminate the need for canaries to detect overflow of buffers on the stack or heap. Marking the word that would have been used as a canary as read-only provides a greater level of protection with lower overhead, with no need to generate a secret. It could also be used to mark data on the stack or heap as read-only, though the caveat mentioned previously in this document is important – if an adversary can insert arbitrary data tagged in the same way as protected code pointers, this could be used to exploit a use-after-free bug by constructing a fake vtable pointer.

Hardware-supported memory tagging can be used as a building block for a more efficient implementation of analyses such as those provided by AddressSanitizer/ThreadSanitizer/MemorySanitizer. The exception handler can be used to invoke software hooks to perform more complex validation of accesses. If two bits per word are insufficient, the exception handler can be used to update arbitrary shadow memory data stored in memory. To ensure the tagged memory system can outperform dynamic binary instrumentation for these uses, we will investigate low-cost user-level interrupts. Tagging can also be useful even in memory-safe languages which already protect against control-flow hijacking. For instance, they could be used to mark pointers for efficient garbage collection.

The tag bits could be applied to instructions in order to (with some additional hardware support) provide simple control-flow integrity checks, i.e. indicate valid targets of indirect branches and thus reduce the number of potential gadgets within a binary. Frame cookies could be used to ensure that control flow reaches the target of indirect branches by first going through the appropriate function entry-point [14]. Of course, if code pointers are protected and their tag bits verified then there is no possibility for an attacker to hijack control flow in the first place.

There are a number of potential uses of tags for parallel programming. The ability to trap on a read or write to a certain memory location would be useful to assist a software transactional memory implementation. Another possibility is to use the tags to provide locks on each word, or with the addition of new instructions to provide full/empty bits for efficient fine-grained memory synchronisation.

Related work

A large number of previous works have looked at associating metadata with each memory location and considered various schemes involving it. For instance, dynamic information flow tracking [17] aims to provide protection by using the metadata to track whether the source of a given piece of data is trusted or not. Other implementation schemes have been proposed, such as special cache lines in the L2 which are marked as holding security tags [2]. Alternatively, a separate L1 and L2 cache for protection metadata can be added and accessed in parallel with the data caches [17]. When the number of bits per word are low, we believe simply copying the tags into the L1 and L2 by extending the cache lines offers the lowest overhead implementation option. Other works have proposed reconfigurable hardware to help implement metadata updates and arbitrary analyses [5].

Future work

We are now in the process of implementing this proposed scheme and investigating some of the parameters in the design space. We intend to experiment with the size of the tag cache and explore the effects of changing the number of bits per word in order to quantify their costs. Although we will be looking at a number of uses of tagged memory ourselves, we expect that the research and developer community at large will explore use-cases that we did not anticipate. The aim is therefore to produce a flexible implementation rather than one dedicated to a single purpose.

Minion cores

Figure 1 shows a strawman design for the first test chip, which we include to provoke discussion. The system contains two superscalar RISC-V cores and a larger number of smaller RISC-V cores or "minions". The minion cores have direct access to external I/O pins through a thin layer of logic or "I/O shim" which gives hardware support for basic tasks such as shifting data in or out efficiently.

We foresee minion cores fulfilling a number of different roles while helping to reduce the overall complexity of the SoC. We expect initial test chips will contain 4-8 minion cores, but could imagine this number increasing to provide a general-purpose parallel compute resource.

- Minions will enable the creation of software-defined I/O interfaces. The aim is to add significant flexibility to the design. The inefficiencies of crude bit-banging implementations will be avoided by providing additional hardware support in the I/O shim to reduce the number of low-level operations and the need to constantly poll inputs. Specialised ISA extensions will also aid the minions, e.g. by providing low-cost precise timing support. The ideal division of support between ISA and shim is currently under investigation.

- The minion cores may be used to pre-process data to reduce the work that must be done by the main cores, providing a higher-level interface. This could also involve adding support for quality-of-service controls or command validation according to a security policy in order to better support efficient access to a device from a virtualized guest or from user-level code. Minions can also be employed to monitor I/O and only wake the main cores when necessary. Appropriate power domains will be provided to ensure that the monitoring of I/Os will consume minimal power. One minion core could have responsibility for booting or waking up the rest of the SoC, having its own power island. This is much like the power management core included in many SoCs, except the RISC-V minion is programmable, documented, and exposed to the chip user in the same way as all other on-chip resources. It also has the advantage of potentially having access to the full memory and peripheral space of the machine.

- The provision of dedicated low-latency interconnect, such as register-mapped FIFOs, between the main cores and the minions would allow work to be off-loaded at a fine granularity. One application of using the minions in this helper-core role would be to implement more complex security policies or run-time checks. It may be worthwhile to support the communication of additional information (e.g. mispredicted branches, cache misses etc.) to aid such schemes.

- There are also security applications for minions, particularly in the case where each one has space in main memory which the application cores are not able to access. This pro-

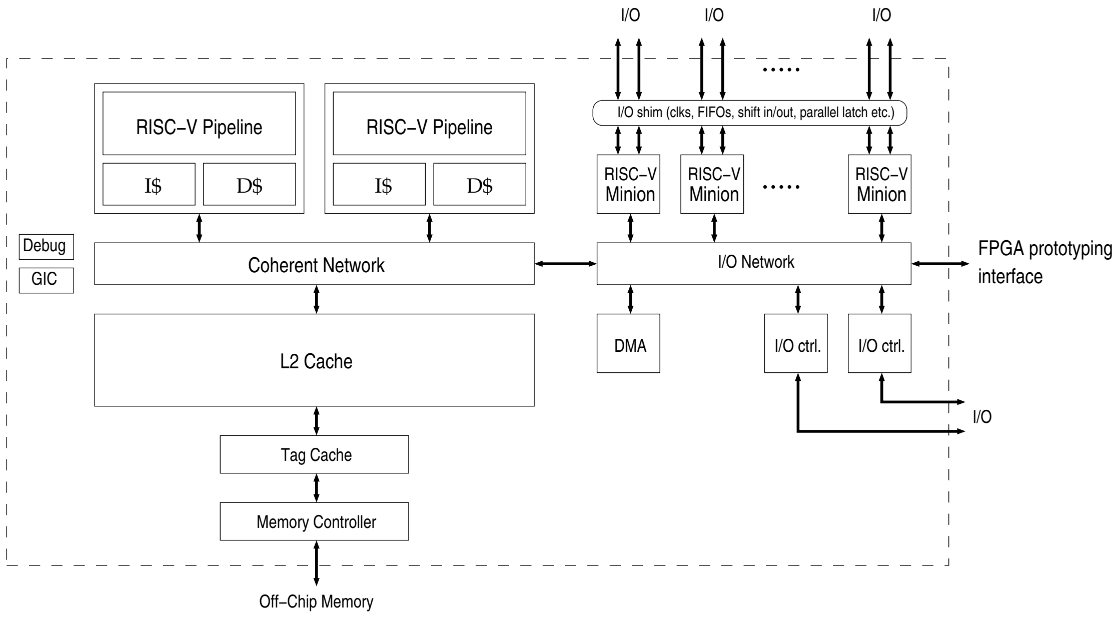

Figure 1: Strawman SoC for first test chip (2015). Two simple dual-issue RISC-V cores will act as the main processors. 4-8 minion cores will be provided. A direct connection to the I/O network aims to provide a simple way to prototype I/O devices and application-specific accelerators.

tection could be provided by the page table, or even through a small hard-coded memory allocation to each minion. The protected memory space could be used as a key-store, or to perform security-sensitive computations. Such hardware partitioning allows strong security guarantees with only a minimal software trusted computing base (TSB).

- Finally, minions may simply be used to run tasks that do not require the performance of the main cores. This can both reduce the main cores' load and help to reduce energy consumption and may be assisted by a low-latency communication mechanism between minion cores. Rather than adopting an additional ISA, lowRISC minions will implement the RISC-V integer base instruction set, with a small set of extensions to aid I/O and real-time related tasks.

Although we will first target relatively low-speed protocols such as SPI, I2C, I2C, or SDIO we are interested in pursuing implementations of more complex and higher speed protocols such as USB or Ethernet on minion cores. Moving even more functionality traditionally implemented using fixed-function hardware to programmable cores, such as the memory controller [3] would also be an interesting future direction.

Hardware support for I/O coherence will be provided. This will allow read and write requests from I/O devices to access data in the L1 and L2 caches if required, including support for coherent (L1/L2), non-coherent (L2) and uncached (straight to main memory) requests. We aim to include an IOMMU on the test chip if time permits. The I/O network includes a high speed off-chip interface for connection to an FPGA for prototyping purposes. With sufficient support in the coherency mechanism, this could also be used for building a system involving multiple coherent lowRISC SoCs.

Related work

Numerous microcontrollers and SoCs offer some form of programmable peripheral coprocessor. This provides the ability to create software-defined peripherals and reduce interrupt load on the main processor. Other interesting designs provide large numbers of simple cores or multithreaded processors with deterministic timing that can be dedicated to I/O related tasks, e.g. XMOS [11], GreenArrays [8] the Ubicom IP 3000 family of processors [6], or the Time Processor Unit (TPU) in the Motorola MC68332 [12].

A number of TI processors provide support for soft-peripherals in the form of Programmable Real-time Units (PRUs) [10]. For example, the BeagleBone Black development board is powered by a TI Sitara processor that provides 2 x 200MHz, 32-bit PRUs. Larger TI SoCs provide 4 PRUs. The PRUs provide deterministic real-time processing and direct, low-latency, access to I/Os. The PRUs are supported by a GPIO module that provides 28-bit shift in/out and 16-bit parallel latch on external signal support. Low-level interaction with the MII interfaces is also provided to support EtherNet/IP [16]. PRUs have their own ISA and can be programmed using TI's PRU C Compiler.

The idea of devoting cores to I/O is also adopted in low-cost multicore microcontroller designs. The NXP LPC4300 family marries an ARM Cortex-M4 processor with a Cortex-M0 for this purpose.

References

[1] P. Akritidis. "Cling: A Memory Allocator to Mitigate Dangling Pointers". In: *Proceedings of the 19th USENIX Conference on Security.* USENIX Security'10. Washington, DC: USENIX Association, 2010, pp. 12–12.

[2] D. Arora et al. "Architectural Support for Run-Time Validation of Program Data Properties". In: *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* 15.5 (May 2007), pp. 546–559.

[3] M. N. Bojnordi and E. Ipek. "PARDIS: A Programmable Memory Controller for the DDRx Interfacing Standards". In: *Proceedings of the 39th Annual International Symposium on Computer Architecture.* ISCA '12. Portland, Oregon: IEEE Computer Society, 2012, pp. 13–24.

[4] *CVE-2014-6271.* National Vulnerability Database. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-6271`. Sept. 2014.

[5] U. Dhawan et al. "PUMP: A Programmable Unit for Metadata Processing". In: *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy.* HASP '14. Minneapolis, Minnesota: ACM, 2014, 8:1–8:8.

[6] D. Foland. "Ubicom MASI - wireless network processor". In: *Proc. of the 15th HotChips Conference.* Aug. 2003.

[7] J. L. Greathouse et al. "A Case for Unlimited Watchpoints". In: *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems.* ASPLOS XVII. London, England, UK: ACM, 2012, pp. 159–172.

[8] GreenArrays. *Product brief: GreenArrays Architecture.* `http://www.greenarraychips.com/home/documents/greg/PB002-100822-GA-Arch.pdf`. 2010.

[9] R. Kumar et al. "Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction". In: *Proc. of IEEE/ACM Intl. Symp. on Microarchitecture (MICRO).* 2003.

[10] G. Martinez and G. Maur. *Programmable Real-Time Unit (PRU) Exending Functionality of Existing SoCs.* `http://www.ti.com/lit/wp/spry136a/spry136a.pdf`. 2011.

[11] D. May. “The XMOS Architecture and XS1 chips”. In: *IEEE Micro* 6 (Nov. 2014), pp. 28–37.

[12] *MC68332 Technical Summary*. `http://cache.freescale.com/files/microcontrollers/doc/data_sheet/MC68332TS.pdf`. 1996.

[13] S. Neuhaus and T. Zimmermann. “Security Trend Analysis with CVE Topic Models”. In: *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering*. ISSRE ’10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 111–120.

[14] K. Onarlioglu et al. “G-Free: Defeating Return-oriented Programming Through Gadget-less Binaries”. In: *Proceedings of the 26th Annual Computer Security Applications Conference*. ACSAC ’10. Austin, Texas: ACM, 2010, pp. 49–58.

[15] *RISC-V website*. `http://www.riscv.org`. 2014.

[16] V. Roy. *EtherNet/IP on TI’s Sitara AM335x processors*. `http://www.ti.com/lit/wp/spry249/spry249.pdf`. 2014.

[17] G. E. Suh et al. “Secure Program Execution via Dynamic Information Flow Tracking”. In: *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XI. Boston, MA, USA: ACM, 2004, pp. 85–96.

[18] L. Szekeres et al. “SoK: Eternal War in Memory”. In: *Security and Privacy (SP), 2013 IEEE Symposium on*. May 2013, pp. 48–62.