Application Data Integrity (ADI)

SPARC M7 processor adds the Application Data Integrity (ADI) feature. ADI allows a task to set version tags on any subset of its address space. Once ADI is enabled and version tags are set for ranges of address space of a task, the processor will compare the tag in pointers to memory in these ranges to the version set by the application previously. Access to memory is granted only if the tag in given pointer matches the tag set by the application. In case of mismatch, processor raises an exception.

Following steps must be taken by a task to enable ADI fully:

1. Set the user mode PSTATE.mcde bit. This acts as master switch for the task's entire address space to enable/disable ADI for the task.
2. Set TTE.mcd bit on any TLB entries that correspond to the range of addresses ADI is being enabled on. MMU checks the version tag only on the pages that have TTE.mcd bit set.
3. Set the version tag for virtual addresses using stxa instruction and one of the MCD specific ASIs. Each stxa instruction sets the given tag for one ADI block size number of bytes. This step must be repeated for entire page to set tags for entire page.

ADI block size for the platform is provided by the hypervisor to kernel in machine description tables. Hypervisor also provides the number of top bits in the virtual address that specify the version tag. Once version tag has been set for a memory location, the tag is stored in the physical memory and the same tag must be present in the ADI version tag bits of the virtual address being presented to the MMU. For example on SPARC M7 processor, MMU uses bits 63-60 for version tags and ADI block size is same as cacheline size which is 64 bytes. A task that sets ADI version to, say 10, on a range of memory, must access that memory using virtual addresses that contain 0xa in bits 63-60.

ADI is enabled on a set of pages using mprotect() with PROT_ADI flag. When ADI is enabled on a set of pages by a task for the first time, kernel sets the PSTATE.mcde bit fot the task. Version tags for memory addresses are set with an stxa instruction on the addresses using ASI_MCD_PRIMARY or ASI_MCD_ST_BLKINIT_PRIMARY. ADI block size is provided by the hypervisor to the kernel. Kernel returns the value of ADI block size to userspace using auxiliary vector along with other ADI info. Following auxiliary vectors are provided by the kernel:

AT_ADI_BLKSZ	ADI block size. This is the granularity and alignment, in bytes, of ADI versioning.
AT_ADI_NBITS	Number of ADI version bits in the VA

IMPORTANT NOTES

- Version tag values of 0x0 and 0xf are reserved. These values match any tag in virtual address and never generate a mismatch exception.
- Version tags are set on virtual addresses from userspace even though tags are stored in physical memory. Tags are set on a physical page after it has been allocated to a task and a pte has been created for it.
- When a task frees a memory page it had set version tags on, the page goes back to free page pool. When this page is re-allocated to a task, kernel clears the page using block initialization ASI which clears the version tags as well for the page. If a page allocated to a task is freed and allocated back to the same task, old version tags set by the task on that page will no longer be present.
- ADI tag mismatches are not detected for non-faulting loads.
- Kernel does not set any tags for user pages and it is entirely a task's responsibility to set any version tags. Kernel does ensure the version tags are preserved if a page is swapped out to the disk and swapped back in. It also preserves that version tags if a page is migrated.
- ADI works for any size pages. A userspace task need not be aware of page size when using ADI. It can simply select a virtual address range, enable ADI on the range using mprotect() and set version tags for the entire range. mprotect() ensures range is aligned to page size and is a multiple of page size.
- ADI tags can only be set on writable memory. For example, ADI tags can not be set on read-only mappings.

ADI related traps

With ADI enabled, following new traps may occur:

Disrupting memory corruption

When a store accesses a memory localtion that has TTE.mcd=1, the task is running with ADI enabled (PSTATE.mcde=1), and the ADI tag in the address used (bits 63:60) does not match the tag set on the corresponding cacheline, a memory corruption trap occurs. By default, it is a disrupting trap and is sent to the hypervisor first. Hypervisor creates a sun4v error report and sends a resumable error (TT=0x7e) trap to the kernel. The kernel sends a SIGSEGV to the task that resulted in this trap with the following info:

```
siginfo.si_signo = SIGSEGV;
siginfo.errno = 0;
siginfo.si_code = SEGV_ADIDERR;
siginfo.si_addr = addr; /* PC where first mismatch occurred */
siginfo.si_trapno = 0;
```

Precise memory corruption

When a store accesses a memory location that has TTE.mcd=1, the task is running with ADI enabled (PSTATE.mcde=1), and the ADI tag in the address used (bits 63:60) does not match the tag set on the corresponding cacheline, a memory corruption trap occurs. If MCD precise exception is enabled (MCDPERR=1), a precise exception is sent to the kernel with TT=0x1a. The kernel sends a SIGSEGV to the task that resulted in this trap with the following info:

```
siginfo.si_signo = SIGSEGV;
siginfo.errno = 0;
siginfo.si_code = SEGV_ADIPERR;
siginfo.si_addr = addr; /* address that caused trap */
siginfo.si_trapno = 0;
```

NOTE:

ADI tag mismatch on a load always results in precise trap.

MCD disabled

When a task has not enabled ADI and attempts to set ADI version on a memory address, processor sends an MCD disabled trap. This trap is handled by hypervisor first and the hypervisor vectors this trap through to the kernel as Data Access Exception trap with fault type set to 0xa (invalid ASI). When this occurs, the kernel sends the task SIGSEGV signal with following info:

```
siginfo.si_signo = SIGSEGV;
siginfo.errno = 0;
siginfo.si_code = SEGV_ACCADI;
siginfo.si_addr = addr; /* address that caused trap */
siginfo.si_trapno = 0;
```

Sample program to use ADI

Following sample program is meant to illustrate how to use the ADI functionality:

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <elf.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/mman.h>
#include <asm/asi.h>

#ifndef AT_ADI_BLKSZ
#define AT_ADI_BLKSZ  48
#endif
#ifndef AT_ADI_NBITS
#define AT_ADI_NBITS  49
#endif

#ifndef PROT_ADI
#define PROT_ADI      0x10
#endif

#define BUFFER_SIZE     32*1024*1024UL

main(int argc, char* argv[], char* envp[])
{
        unsigned long i, mcde, adi_blksz, adi_nbits;
        char *shmaddr, *tmp_addr, *end, *veraddr, *clraddr;
        int shmid, version;
      Elf64_auxv_t *auxv;

      adi_blksz = 0;

      while(*envp++ != NULL);
      for (auxv = (Elf64_auxv_t *)envp; auxv->a_type != AT_NULL; auxv++) {
              switch (auxv->a_type) {
              case AT_ADI_BLKSZ:
                      adi_blksz = auxv->a_un.a_val;
                      break;
              case AT_ADI_NBITS:
                      adi_nbits = auxv->a_un.a_val;
                      break;
              }
      }
      if (adi_blksz == 0) {
              fprintf(stderr, "Oops! ADI is not supported\n");
              exit(1);
      }

      printf("ADI capabilities:\n");
      printf("\tBlock size = %ld\n", adi_blksz);
      printf("\tNumber of bits = %ld\n", adi_nbits);

        if ((shmid = shmget(2, BUFFER_SIZE,
                                IPC_CREAT | SHM_R | SHM_W)) < 0) {
                perror("shmget failed");
                exit(1);
        }

        shmaddr = shmat(shmid, NULL, 0);
        if (shmaddr == (char *)-1) {
                perror("shm attach failed");
                shmctl(shmid, IPC_RMID, NULL);
                exit(1);
        }

      if (mprotect(shmaddr, BUFFER_SIZE, PROT_READ|PROT_WRITE|PROT_ADI)) {
              perror("mprotect failed");
              goto err_out;
      }

        /* Set the ADI version tag on the shm segment
         */
        version = 10;
        tmp_addr = shmaddr;
        end = shmaddr + BUFFER_SIZE;
        while (tmp_addr < end) {
                asm volatile(
                        "stxa %1, [%0]0x90\n\t"
                        :
                        : "r" (tmp_addr), "r" (version));
                tmp_addr += adi_blksz;
        }
      asm volatile("membar #Sync\n\t");

        /* Create a versioned address from the normal address by placing
       * version tag in the upper adi_nbits bits
         */
        tmp_addr = (void *) ((unsigned long)shmaddr << adi_nbits);
        tmp_addr = (void *) ((unsigned long)tmp_addr >> adi_nbits);
        veraddr = (void *) (((unsigned long)version << (64-adi_nbits))
                        | (unsigned long)tmp_addr);

        printf("Starting the writes:\n");
        for (i = 0; i < BUFFER_SIZE; i++) {
                veraddr[i] = (char)(i);
                if (!(i % (1024 * 1024)))
                        printf(".");
        }
        printf("\n");

        printf("Verifying data...");
      fflush(stdout);
        for (i = 0; i < BUFFER_SIZE; i++)
                if (veraddr[i] != (char)i)
                        printf("\nIndex %lu mismatched\n", i);
        printf("Done.\n");

        /* Disable ADI and clean up
         */
      if (mprotect(shmaddr, BUFFER_SIZE, PROT_READ|PROT_WRITE)) {
              perror("mprotect failed");
              goto err_out;
      }

        if (shmdt((const void *)shmaddr) != 0)
                perror("Detach failure");
        shmctl(shmid, IPC_RMID, NULL);

        exit(0);

err_out:
        if (shmdt((const void *)shmaddr) != 0)
                perror("Detach failure");
```

```
            shmctl(shmid, IPC_RMID, NULL);
            exit(1);
    }
```