Capability-aware memory copying between address spaces

Konrad Rafał Witaszczyk
nlh930@alumni.ku.dk
def@FreeBSD.org

University of Copenhagen

October 10, 2019

Communication between processes and the kernel

- Processes communicate with the kernel using the system call interface to perform privileged operations;
- They pass information in CPU registers or memory that include values or pointers to objects in user space;
- Objects to be passed are stored in separate virtual address spaces;
- In order to access a user-space object, the kernel needs to copy it to the kernel space;
- In order to allow a process to access a kernel-space object, the kernel needs to copy it to the user space;
- The system call interface must copy objects each time information are passed between processes and the kernel.

User-space ABIs

Calling conventions and layouts of objects used in the system call interface depend on a user-space ABI.

FreeBSD supports multiple ABIs, in particular:

- Native ABI;
  Programs built for the same target as kernel.
- 32-bit ABI.
  Programs built for a 32-bit variant of a kernel target architecture (AMD64, MIPS, PowerPC).

Memory copying in FreeBSD

Currently kernel implements `copyin()` and `copyout()` functions to copy data between address spaces.

- `copyin()` copies arbitrary `len` bytes from a user-space address `uaddr` to a kernel-space address `kaddr`.

  ```
  int copyin(const void *uaddr, void *kaddr,
      size_t len);
  ```

- `copyout()` copies bytes in the opposite direction.

  ```
  int copyout(const void *kaddr, void *uaddr,
      size_t len);
  ```

Example: `jail()` syscall handler for the native ABI

```
int
sys_jail(struct thread *td, struct jail_args *uap)
{
  struct jail j;

  (...)

  error = copyin(uap->jail, &j, sizeof(struct jail));
  if (error)
    return (error);

  (...)

  return (kern_jail(td, &j));
}
```

Differences between ABIs

```
struct jail_args {
 char jail_l_[PADL_(struct jail *)];
 struct jail *jail;
 char jail_r_[PADR_(struct jail *)];
};
```

```
struct jail {
 uint32_t        version;
 char            *path;
 char            *hostname;
 char            *jailname;
 uint32_t        ip4s;
 uint32_t        ip6s;
 struct in_addr  *ip4;
 struct in6_addr *ip6;
};
```

```
struct freebsd32_jail_args {
 char jail_l_[PADL_(struct jail32 *)];
 struct jail32 *jail;
 char jail_r_[PADR_(struct jail32 *)];
};
```

```
struct jail32 {
 uint32_t version;
 uint32_t path;
 uint32_t hostname;
 uint32_t jailname;
 uint32_t ip4s;
 uint32_t ip6s;
 uint32_t ip4;
 uint32_t ip6;
};
```

Compatibility layers

Support for ABIs is provided via compatibility layers.

Each compatibility layer implements its own system call handlers that perform additional operations required to call native ABI kernel routines.

32-bit ABI:

```
struct jail j;
struct jail32 j32;

error = copyin(uap->jail, &j32, sizeof(struct jail32));
if (error)
  return (error);
CP(j32, j, version);
PTRIN_CP(j32, j, path);
PTRIN_CP(j32, j, hostname);
PTRIN_CP(j32, j, jailname);
CP(j32, j, ip4s);
CP(j32, j, ip6s);
PTRIN_CP(j32, j, ip4);
PTRIN_CP(j32, j, ip6);
(...)
return (kern_jail(td, &j));
```

ABI-independent type-aware copyinout API

During a course project at the University of Copenhagen, we implemented an API that allows to copy an object and translate it directly to a user-space or a kernel-space definition.

In order to copy in a jail object with the new API, we would use the same function call in both the native ABI and the 32-bit ABI:

```
copyin_jail(uap->jail, &j);
```

We used C copy functions that didn't allow code optimizations, CPU-specific features and reduced performance.

Capability-aware copyinout API

We decided to improve the initial implementation and add new features.

We would like to pass objects between address spaces with information what operations can be performed with their values.

```
struct foo {
 size_t len;
 int    *array;
};
```

$\longrightarrow$

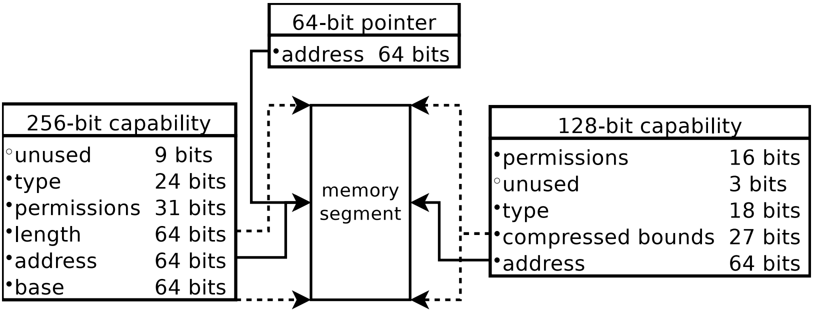
```
struct foo {
 size_t len;
 __uaddr_array(len) int * __capability array;
} __copyinout;
```

Capability Hardware Enhanced RISC Instructions (CHERI)

We imported the initial implementation to CheriBSD, FreeBSD adapted for the CHERI CPU.

- RISC ISA extension;
- Allows to minimize privileges of a process through *capabilities* that are managed by hardware and software;
- Designed and implemented by the CTSRD research group from SRI International and the University of Cambridge;
- Prototype implemented as a BERI-based (MIPS) coprocessor on FPGA. BERI (Bluespec Extensible RISC Implementation) is an open source 64-bit pipelined RISC processor;
- CHERI ISAv7 from June 2019 introduces CHERI-RISC-V but also CHERI-x86-64;
- Arm wants to implement CHERI in their CPUs.

Pointer and CHERI capability abstractions



CHERI hardware-software stack

The CHERI project consists of several subprojects:

- LLVM that supports CHERI capabilities and instructions;
- QEMU that emulates CHERI;
- CheriBSD that supports the following ABIs:
  - Pure-capability ABI (CheriABI);
    A process uses only capabilities instead of virtual addresses.
  - Hybrid ABI (native ABI);
    A process partially uses capabilities.
  - Legacy ABI (freebsd32 and freebsd64).
    A process runs a program compiled for FreeBSD.

Current interface in CheriBSD

C pointers to user space have the type qualifier `__capability` that enforces capability usage instead of regular pointers.

Kernel implements `copyin()` and `copyout()` variants that are capability-aware:

```
int copyin_c(const void * __restrict __capability udaddr,
      void * _Nonnull __restrict kaddr, size_t len);

int copyout_c(const void * _Nonnull __restrict kaddr,
      void * __restrict __capability udaddr, size_t len);

int copyincap(const void * __restrict __capability udaddr,
      void * _Nonnull __restrict kaddr, size_t len);

int copyoutcap(const void * _Nonnull __restrict kaddr,
      void * __capability __restrict udaddr, size_t len);
```

`copy{in,out}_c()` don't copy capabilities.
`copy{in,out}cap()` copy capabilities.

Differences between ABIs (CheriBSD)

```
struct jail_args {
 char jail_l_[PADL_(struct jail *)];
 struct jail *jail;
 char jail_r_[PADR_(struct jail *)];
};
```

```
struct jail {
 uint32_t        version;
 char            *path;
 char            *hostname;
 char            *jailname;
 uint32_t        ip4s;
 uint32_t        ip6s;
 struct in_addr  *ip4;
 struct in6_addr *ip6;
};
```

```
struct cheriabi_jail_args {
  char jailp_l_[PADL_(struct jail_c *
    __capability)];
  struct jail_c * __capability jailp;
  char jailp_r_[PADR_(struct jail_c *
    __capability)];
};
```

```
struct jail_c {
  uint32_t       version;
  char * __capability path;
  char * __capability hostname;
  char * __capability jailname;
  uint32_t       ip4s;
  uint32_t       ip6s;
  struct in_addr *
    __capability ip4;
  struct in6_addr *
    __capability ip6;
};
```

jail() syscall handlers in CheriBSD

Native ABI:

```
struct jail j;
error = copyin(jail, &j, sizeof(struct jail));
if (error)
  return (error);
return (kern_jail(td, __USER_CAP_STR(j.path),
    __USER_CAP_STR(j.hostname), __USER_CAP_STR(j.jailname),
    __USER_CAP_ARRAY(j.ip4, j.ip4s), j.ip4s,
    __USER_CAP_ARRAY(j.ip6, j.ip6s), j.ip6s, UIO_USERSPACE));
```

Pure-capability ABI:

```
struct jail_c j;
error = copyincap(uap->jailp, &j, sizeof(j));
if (error != 0)
  return (error);
return (kern_jail(td, j.path, j.hostname, j.jailname,
    j.ip4, j.ip4s, j.ip6, j.ip6s, UIO_USERSPACE));
```

32-bit ABI:

```
struct jail32 j32;
error = copyin(uap->jail, &j32, sizeof(struct jail32));
if (error)
  return (error);
return (kern_jail(td, __USER_CAP_STR(PTRIN(j32.path)),
    __USER_CAP_STR(PTRIN(j32.hostname)),
    __USER_CAP_STR(PTRIN(j32.jailname)),
    __USER_CAP_ARRAY(PTRIN(j32.ip4), j32.ip4s), j32.ip4s,
    __USER_CAP_ARRAY(PTRIN(j32.ip6), j32.ip6s), j32.ip6s,
    UIO_USERSPACE));
```

Goals

We state the following goals for our solution:

1. ABI-independent API;
2. Type-aware API;
3. Capability-aware API;
4. Ignored memory padding;
5. Support for new types should be easy to add;
6. Incremental adaptation;
7. Dynamic function declarations for kernel modules;
8. Assembly optimizations for CHERI that can be also used for other architectures in the future;
9. CPU-specific features (e.g. SMAP in AMD64);
10. Supports types used in the CheriABI compatibility layer.

The copyinout framework: annotations

Kernel implements the following annotations:

- Type annotations to indicate what functions should be generated:
  - `__copyin`;
  - `__copyout`;
  - `__copyinout`.
- Field annotations to indicate what capability should be stored in a field:
  - `__uaddr_array(field)`;
  - `__uaddr_bounded(field)`;
  - `__uaddr_code`;
  - `__uaddr_object`;
  - `__uaddr_unbounded`;
  - `__uaddr_str`.

The copyinout framework: ABI-specific copyinout table

FreeBSD and CheriBSD define a structure for each ABI with ABI-specific objects, e.g. a table of system call handlers.
We extend this structure with a table of copyinout function pointers.

We use the `SYSINIT()` mechanism to fill the table at compile time or when a kernel module is loaded.
Internally `SYSINIT()` is implemented using a technique called *Linker Set*.

The copyinout framework: kernel interfaces

We introduce macros to register and call copy functions:

- COPYINOUT_ALLOCATE(type);
- COPYINOUT_DECLARE(type);
- COPYINOUT_DEFINE(abi, type);
- COPYIN_CALL(type, uaddr, kaddr);
- COPYOUT_CALL(type, kaddr, uaddr).

Then for the jail type we can define:

```
#define copyin_jail(uaddr, kaddr) \
  COPYIN_CALL(jail, uaddr, kaddr)
#define copyout_jail(kaddr, uaddr) \
  COPYOUT_CALL(jail, kaddr, uaddr)
```

COPY{IN,OUT}_CALL() macros find a process structure of a currently running process and locate a function in a table with copy functions referenced indirectly by the process structure.

The copyinout framework: copyinout code generating tool

We implemented a C++ program using libclang from LLVM that can generate:

- Function prototypes;
- Function declarations;
- Function implementations in C or assembly.

The program parses ASTs generated by clang, finds annotated types and emits code.

The copyinout framework: test kernel module

We provide a kernel module that can be used to test generated implementations for private types as well as types from kernel.

The module registers copy functions for private types and registers `sysctl` nodes for each tested types:

- A sysctl node to copy in an object;
- A sysctl node to copy out previously copied in object.

We implement a program that for each tested type:

- Allocates an object;
- Passes the object to the kernel with the copyin sysctl;
- Calls the copyout sysctl;
- Compares if the copied out object is the same as the allocated one.

The copyinout framework: build system

The copyinout tool can be executed with a shell script that for each supported ABI:

1. Generates a user-space AST from a header file that includes all annotated types;
2. Generates a kernel-space AST from the same header file;
3. Runs `copyinout` with appropriate compiler flags and paths to ASTs;
4. Writes output to a source tree.

The above steps are performed for both a header file with kernel structures as well as a header file for the test kernel module.

In the future this script should be part of the build system.

Results: type generalization

1. We have only one type definition in kernel.

```
struct jail {
  uint32_t                                                  version;
  __uaddr_str char * __capability                           path;
  __uaddr_str char * __capability                           hostname;
  __uaddr_str char * __capability                           jailname;
  uint32_t                                                  ip4s;
  uint32_t                                                  ip6s;
  __uaddr_array(ip4s) struct in_addr * __capability  ip4;
  __uaddr_array(ip6s) struct in6_addr * __capability ip6;
} __copyinout;
```

Results: simplified system call handlers

2. Compatibility layers are much simpler.

```
int
sys_jail(struct thread *td,
  struct jail_args *uap)
{
  uint32_t version;
  int error;
  struct jail j;
  void * __capability jail =
    __USER_CAP_UNBOUND(uap->jailp);

  error = copyin(jail, &version,
    sizeof(version));
  if (error)
    return (error);

  switch (version) {
  (...)
  case 2:
    error = copyin_jail(jail, &j);
    if (error)
      return (error);
    break;

  default:
    return (EINVAL);
  }
  return (kern_jail(td, &j));
}
```

```
int
freebsd32_jail(struct thread *td,
  struct freebsd32_jail_args *uap)
{
  struct jail_args args;

  args.jailp = uap->jailp;

  return (sys_jail(td, &args));
}
```

Results: reduced differences between projects

3. Kernel prototypes in CheriBSD are the same as in FreeBSD.

```
/* kern_jail() prototype in FreeBSD. */
int kern_jail(struct thread *td, struct jail *j);

/* kern_jail() prototype in CheriBSD without copyinout changes. */
int kern_jail(struct thread *td, const char * __capability path,
      const char * __capability hostname,
      const char * __capability jailname,
      struct in_addr * __capability ip4, size_t ip4s,
      struct in6_addr * __capability ip6, size_t ip6s,
      enum uio_seg ipseg);

/* kern_jail() prototype in CheriBSD with copyinout changes. */
int kern_jail(struct thread *td, struct jail *j);
```

Results: automatic capability constructions

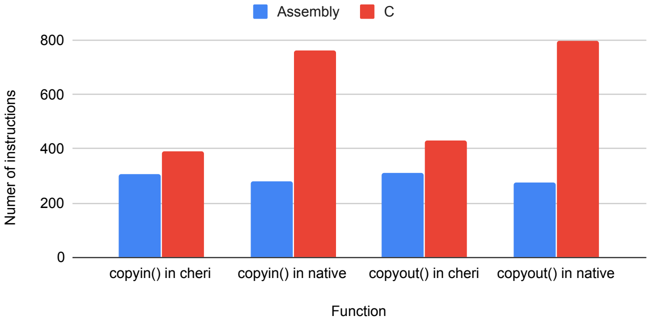
4. Capabilities are automatically created with type annotations.

```c
struct struct_with_array_uaddr {

  size_t  sa_len;

  __uaddr_array(sa_len) struct foo *
    __capability sa_uaddr;

} __copyinout;
```

```
LEAF(native_copyin_struct_with_array_
uaddr)
  cgetbase  t0, $c3
  blt t0, zero, _C_LABEL(copyerr)
  nop
  GET_CPU_PCPU(v1)
  PTR_L t0, PC_CURPCB(v1)
  PTR_LA  t1, copyerr

  PTR_S t1, U_PCB_ONFAULT(t0)
  cld v0, zero, 0($c3)
  PTR_S zero, U_PCB_ONFAULT(t0)
  csd v0, zero, 0($c4)

  PTR_S t1, U_PCB_ONFAULT(t0)
  cld v0, zero, 8($c3)
  cfromptr  $c5, $ddc, v0
  cld v0, zero, 0($c3)
  li  a0, 96
  multu a0, v0
  mflo  v0
  csetbounds  $c5, $c5, v0
  PTR_S zero, U_PCB_ONFAULT(t0)
  csc $c5, zero, 16($c4)

  j ra
  move  v0, zero
END(native_copyin_struct_with_array_
uaddr)
```

Results: performance improvements

5. Performance is not reduced.



Instructions used for `copyin()`/`copyout()` with the `jail` type.

	Assembly		C		Upstream	
ABI	Average	σ	Average	σ	Average	σ
Native	212,700.72	3,699.11	213,704.49	6,648.29	212,898.20	6,390.38
CheriABI	212,904.46	4,060.39	213,520.06	4,941.34	213,603.25	6,802.62

Kernel instructions used for the `jail` syscall.

Contributions

- CTSRD-CHERI/cheribsd:
  Correctly initialize iovec lengths in kern_jail().
- CTSRD-CHERI/llvm-project:
  Clang crashes when used with source annotations.
- CTSRD-CHERI/llvm-project:
  Clang crashes when used with capabilities and source annotations.
- CTSRD-CHERI/cheribuild:
  Update cheribuild.py usage in README.

Future work

- System call interface improvements;
- `ioctl(2)` interface improvements;
- Compatibility layers for emulated platforms;
- Compiler optimizations.

Conclusion

- We improved the previous copyinout implementation;
- We do not support complex types, e.g. anonymous structures, that are required to upstream the project;
- However, the rest of the goals were achieved and we have usable implementation;
- We received positive feedback from the community;
- We hope the framework will be adapted by both projects after further improvements and reviews;
- Collaborations in open source projects can lead to very interesting research.

Thank you for your attention!