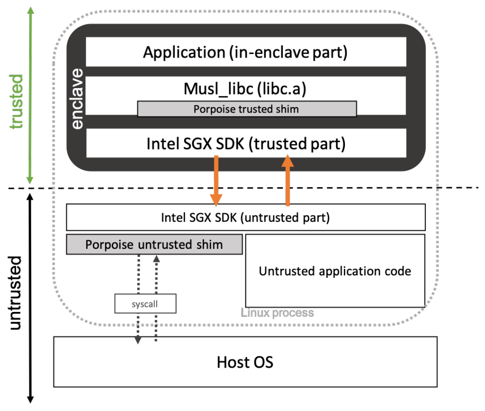
trusted
enclave
Application (in-enclave part)
Musl_libc (libc.a)
Porpoise trusted shim
Intel SGX SDK (trusted part)
Intel SGX SDK (untrusted part)
Porpoise untrusted shim
Untrusted application code
syscall
Linux process
untrusted
Host OS

Figure 3: Design of an enclave-based application that uses Porpoise. Porpoise consists of the trusted in-enclave shim, and an untrusted shim outside the enclave (shown in gray).

SUPPLEMENTARY MATERIAL

DESIGN AND IMPLEMENTATION OF PORPOISE

Our goal in this paper is to quantitatively evaluate the benefits and costs of the library OS, library wrapping and instruction wrapping approaches to port code to enclaves. While we could find open-source prototypes representing the library OS and library wrapping approaches, the source code for SCONE [4], which implements instruction wrapping, was not available (SCONE is the basis for a commercial offering from `scontain.com`). We therefore built our in-house prototype, called Porpoise, for our evaluation. We plan to make Porpoise's code available upon the paper's acceptance. In this section, we describe, in brief, the implementation of Porpoise and some key challenges we had to overcome in its implementation.

Porpoise is implemented as a set of modifications to `musl-libc` [18] version 1.1.9. As discussed in Section 3.3, we make the assumption that applications do not directly invoke the raw low-level instructions, but rather rely on libraries for doing so. `musl-libc` is an API-compatible implementation of the standard C library that is much more modular and easier to modify than its counterparts such as glibc.

Figure 3 depicts an enclave that uses Porpoise to support an enclave application. Porpoise relies on the Intel SGX SDK (version 2.7.1) [12] for various standard tasks such as enclave initialization, `ecalls` (calls to the enclave from outside), `ocalls` (calls from the enclave to the outside), maintaining the thread-control structures, the state save area and other structures that are part of the SGX's hardware/software interface. The hardware uses these structures to store the state of registers when it exits the enclave, so as to protect them from untrusted code outside the enclave. The Intel SGX SDK also has some untrusted support code outside the enclave that the untrusted code in the process uses to interact with the enclave. The core functionality of Porpoise is implemented in two shim layers: a *trusted shim* that runs within the enclave, and an *untrusted shim* that facilitates the interaction of the enclave with the rest of the user process.

The trusted shim layer is implemented as a set of wrappers around the evaluation of `syscall` instructions in `musl-libc`. The shim is responsible for marshaling data outside the enclave. It creates a copy of the system call's arguments to buffers in the untrusted shim layer, and transfer control to the untrusted shim layer. Correspondingly, the untrusted shim unmarshals the data outside the enclave, and performs the system call on behalf of the enclave. Once the system call returns, the untrusted shim returns control to the enclave with any return values from the system call stored in buffers outside the enclave. The trusted shim copies data back from the untrusted shim layer, and unmarshals the data for consumption by the enclave. Logically, our shim is structured as wrapper around occurrences of the `syscall` instruction in `musl-libc`, with a case analysis based on the system call number, to determine the number of arguments to be copied.

Because the trusted shim is the portion of the enclave that interacts with the untrusted world, it is also logically the place where filters that detect IAGO-style attacks can be implemented. Our Porpoise prototype currently only has wrappers for IAGO-style attacks largely similar to the file-system shield and network shield described in the SCONE paper [4].

Porpoise's trusted shim provides encryption by default for data that exits the enclave. Each piece of data that is not required for executing the system call is encrypted with keys managed by the trusted shim. However, we cannot encrypt all the arguments to the `syscall` instruction, *e.g.,* the system call number itself cannot be encrypted. Similarly, an enclave that interacts with the file system outside must be able to name the file, which must be sent in the clear (although the bytes sent to the file can be encrypted). For each system call, Porpoise's trusted shim encrypts the arguments that are not needed for the execution of the system call outside the enclave (*e.g.,* data blocks are not modified by the system call, and are therefore sent encrypted.

We note here that a number of papers have proposed oblivious file systems for enclaves [1, 17] that even hide the name of the file from the untrusted code. This is required to prevent the untrusted code from making inferences about the file accesses made from within the enclave. We do not consider these techniques to be within the scope of Porpoise for the purposes of the present paper, which is to evaluate the merits and costs of different ways of porting code to the enclave. However, Porpoise is extensible, and such algorithms can be incorporated within Porpoise as well.

We structured our implementation of the trusted shim by creating a simple `send`/`recv` interface (as was also discussed in prior work [27]). Each argument of the system call is wrapped with a `send_user` call, whose API is as follows:

```
  send_user  (void *enclbuf, void *userbuf,
                  ssize_t size, int prot)
```

The first two arguments point to the in-enclave source buffer containing the data and the buffer within the untrusted shim to receive the data, respectively, `size` denotes the amount of data to be copied, and `prot` determines whether the data in the buffer should be encrypted on its way out. We use AES in CTR mode with 128 bit keys for encryption. A corresponding `recv_user` call obtains data on the return path.

Porpoise incorporates support for 145 system calls (of the total of 325 system calls in Linux-4.4.0-169). While we have plans to add support for the remaining system calls, prior work [35] indicates that system calls vary in terms of importance (based on their usage in real-world packages). Indeed, we did not encounter these system calls in any of the application benchmarks that we studied. We now discuss the technical challenges that Porpoise overcomes in the implementation of some system calls.

• *In-enclave threading*. Porpoise supports multi-threaded applications via the `pthreads` API. Although Intel SGX supports multiple threads of execution within the enclave, it does not allow thread creation from within the enclave. Rather, an application has to pre-declare the number of enclave threads that it would like to support, and the application creates this number of threads in untrusted code. Each of these threads can enter the enclave in fresh thread context, thereby creating the illusion of a multi-threaded enclave. Thus, each enclave thread will have an associated counterpart thread in the untrusted user-space process.

Porpoise's `pthread`-compatible threading model has to work within the constraints of SGX. The `pthread` library uses the `clone` and `arch_prctl` system calls to create new threads. The `clone` system call is used to create a thread, while the `arch_prctl` is used to modify the `%fs` and `%gs` registers, which point to structures that store the thread state. For instance, each thread has its thread-local structure (that we will call `thread_data`) that stores a pointer to the base of the thread's stack, the thread ID, signal mask, canaries, and so on. The `%fs` register is used to point to this structure of the currently executing thread. Both the `%fs` and `%gs` registers can only be modified when the processor is in supervisor mode, and they cannot be modified in user- or enclave-mode. They can be read irrespective of processor mode.

Within a traditional process, `pthread` uses the `arch_prctl` system call to set the `%fs` register to point to the `thread_data` of the thread that is currently executing. This `thread_data` structure resides in a memory location within the process's address space. The key difficulty with wrapping `arch_prctl` is that if the wrapper simply performs the equivalent operation outside the enclave within the user process, the resulting call will set the `%fs` register within the user-space. The kernel cannot access enclave memory, and therefore cannot change the pointer to the thread's state inside the enclave.

We address this problem as follows. As discussed earlier, the Intel SGX does not allow threads to be created within the enclave itself. Rather, the application must pre-declare the number of enclave threads it intends to use, and the enclave is initialized accordingly. Internally, the Intel SGX SDK maintains an in-enclave *thread control structure* for each thread . This is akin to the `thread_data` structure for traditional user-space threads, but is required for in-enclave bookkeeping of the thread.

Porpoise maintains a table that associates the in-enclave thread-control structure for each enclave thread with the corresponding `thread_data` structure of that enclave thread's user-space counterpart. As the wrapped `arch_prctl` call updates the pointer to the `thread_data` structure in the process (by modifying the `%fs` register), Porpoise identifies the corresponding in-enclave thread to which this `%fs` corresponds, and updates the thread-control structure to resume executing that thread during enclave entry (without exiting the enclave).

A related problem happens with other data structures of the `pthread` library, where the kernel directly modifies data structures. For example, the kernel modifies the `detach_state` data structure in the `pthread` library to denote the current state of a thread (*e.g.,* `EXITED`, `JOINABLE`, . . .). On an enclave-based system, this data structure cannot be stored within the enclave, because it will not be accessible to the kernel. Porpoise addresses this problem by maintaining two copies of the data structure: one within the enclave, and one in the user-space process. As the kernel modifies the data structure within the process, Porpoise modifies the corresponding copy within the enclave.

• `brk` *and dynamic memory allocation.* The current version of Intel SGX does not allow dynamic memory allocation within enclaves, although this has been proposed for future versions of SGX [39]. Instead, the Intel SGX SDK implements functionality that simulates the effect of dynamic memory allocation. It pre-allocates a certain amount of memory for use by the enclave, and maintains an internal break point to denote the top of the heap. This break point is modified by a "`malloc`" call that simply returns memory by modifying this break point. We redirect `brk` system calls to this implementation to offer the illusion of dynamic memory allocation for legacy applications.