Rage Against the Sandbox: Bypassing Apple's iOS Security to Run Unsigned Code via SSH

Yuval Hanoch Hirschenbein Sadde

Abstract

Apple's Sandbox and code signing have rendered traditional security research tools and development workflows impractical on iPhones, forcing researchers to rely on full-system emulation or increasingly scarce jailbreak exploits. The Sandbox prohibits process creation, which shell programs need for job control. Code signing prevents execution of unsigned binaries.

In this work, we present novel techniques for bypassing iOS Sandbox's fork() restriction and code signing enforcement without exploiting vulnerabilities, and demonstrate their integration in TVM (Thread Virtual Machine), a practical framework for running unmodified programs on iPhones. TVM enables SSH servers and interactive shells on iPhones, with full support for fork() semantics and unsigned code execution.

We provide an in-depth technical analysis of our approach, which leverages multithreading, stack frame manipulation, and selective processor instruction emulation. This implementation demonstrates that the practical security boundaries of Apple's sandbox can be pushed significantly further than previously possible on non-jailbroken devices.

1. Introduction

Classic POSIX mechanisms such as isolated virtual address space per process and user/group driven filesystem permission schemes proved too weak to mitigate untrusted third-party software from compromising security. This led mobile device vendors to implement mechanisms to tightly control the attack surfaces untrusted software use with the operating system. There are currently 2 major operating systems for mobile devices, Google's Android and Apple's iOS. Although both depend on POSIX-compliant kernels (Linux for Android, XNU for iOS), their implementation differs substantially. Both operating systems were extended with a MAC (Mandatory Access Control) security subsystem that gates system call usage of userspace processes – preventing untrusted applications from performing privileged operations. SELinux and Sandbox are the two implementations of MAC subsystems for Android and iOS respectively. Apple's security boundaries are tighter than Android's, particularly regarding process creation. Using its MAC subsystem, Apple prevents third-party developers from creating processes via fork(). This limitation fundamentally breaks compatibility with standard POSIX programs that rely on multi-processing:

1. **Interactive shells and terminal emulators** which require process isolation for job control, command pipelines and process groups.
2. **Databases** such as Redis that implement efficient background persistence by utilizing fork()'s memory duplication to save data to the filesystem asynchronously.
3. **Web and SSH servers** relying on multi-processing to handle each client request in an isolated process.

Beyond preventing process creation, Apple has integrated an additional security protection dubbed Code Signing which restricts executable mapping of unsigned code by validating cryptographic signatures at the mmap() system call level. While this does not inherently break pre-signed applications, it precludes runtime code generation and dynamic code loading - eliminating entire classes of technologies including JIT compilers, plugin repositories, package managers, and development environments that compile and execute code on-device.

1.1. Existing Approaches

Several projects have attempted to provide POSIX-like environments on iOS despite sandbox restrictions. We examine two prominent approaches that represent different points in the design space.

iSH [1] implements a Linux userspace environment through x86 CPU emulation. It provides a kernel wrapper that intercepts system calls from emulated x86 processes and implements Linux subsystems including process management, Copy-On-Write memory, and a virtual filesystem with musl libc. The emulator runs unsigned ELF binaries by translating x86 instructions and maintaining

separate address spaces for each emulated process through an MMU layer.

While iSH successfully provides a Linux-like environment, its emulation approach has fundamental limitations. Programs running in iSH cannot interact with the native iOS system – system calls are handled by iSH's kernel wrapper rather than XNU. This isolation prevents integration with iOS APIs, native frameworks, or system services. Additionally, iSH's approach requires all code to execute through the x86 emulator, incurring performance overhead even for operations that could run natively.

a-Shell [2] takes a different approach: implementing shell functionality through thread-local storage rather than true processes. It replaces process-global state (stdout, environ) with thread-local equivalents and replaces usage of fork() with pthread_create(). This allows multiple "processes" to coexist in a single address space while appearing isolated.

Unlike iSH, a-Shell directly interacts with iOS - running ifconfig returns real system interfaces. However, this approach requires extensive source modifications to eliminate process-global variables and a custom LLVM toolchain for WebAssembly compilation. Standard compiled binaries cannot run in a-Shell, and fork() semantics remain unavailable. Programs must be specifically ported to the a-Shell environment.

Neither approach enables standard, natively-compiled POSIX programs to run with true fork()/execve() semantics while integrating with the host OS. iSH achieves fork() through emulation but sacrifices host integration and performance. a-Shell provides host integration but requires custom toolchains and cannot support standard binaries or true multi-processing semantics.

1.2. Our Approach

We present TVM (Thread Virtual Machine), a system that enables unmodified POSIX programs to run on iOS with full fork() and execve() semantics. TVM's key insight is selective emulation: unsigned user code runs in an emulated CPU and calls to signed code transition to the native CPU to increase performance and allow integration to the host operating system. This hybrid emulation technique avoids the complete isolation of full-system emulation (iSH) and non-standard compilation toolchain requirements (a-Shell).

TVM implements process semantics through thread-based virtualization combined with a custom dynamic linker that bypasses code signing restrictions by never mapping unsigned code as executable. Programs compiled with standard toolchains (GCC, Clang) can run without modification, accessing iOS frameworks and system services directly. We demonstrate TVM's robustness by supporting complex POSIX applications including SSH servers, shells, and development tools entirely within the iOS sandbox. TVM supports ARM64 and x86-64 architectures on both XNU-based systems (iOS, macOS) and Linux-based platforms (Android, desktop, servers, embedded).

1.3. Contribution

This work makes the following contributions:

- A thread-based process virtualization technique that implements fork() semantics using pthread_create() through Copy-On-Write memory management and stack frame manipulation, enabling both parent and child threads to resume execution from the same call site with proper dual-return behavior.
- A selective emulation architecture that executes platform code natively while emulating unsigned user code, maintaining both performance and host integration.
- A custom dynamic linker supporting ELF and Mach-O formats that enables standard binaries to run without custom compilation toolchains.

The remainder of this paper presents TVM's architecture (Section 2), implementation details (Section 3), discusses deployment and applications (Section 4), surveys related work on memory virtualization techniques (Section 5), and concludes (Section 6).

2. Basic VM Scheme

TVM implements POSIX multi-processing semantics within a single OS process through thread-based process virtualization. Each virtual process is represented by one or more native threads, and TVM ensures the illusion of independent file descriptor tables, signal handling, and process lifecycle. Rather than taking full control of the host process, TVM creates an isolated enclave of threads that operate as virtual processes, allowing itself to coexist with other unrelated threads such as UI workers for an iOS application.

TVM can be characterized as a partial virtual machine: it implements host OS features to provide virtual processes with isolation, while deliberately maintaining direct access to host OS services. This hybrid approach distinguishes TVM from full system emulators - virtual processes interact directly with iOS frameworks and system libraries while remaining isolated from each other.

The foundation of TVM is function interposition: system and library calls from virtual processes are redirected through TVM's virtualization layer before reaching the host OS. This enables TVM to implement process isolation through function overrides like dup(), fork(), exit(), waitpid(), and sigaction() without modifying the source code of programs running on top it.

This foundation is then extended with unsigned code execution capability through selective CPU emulation. Signed platform code (system libraries) and TVM infrastructure executes natively, while unsigned user binaries execute through a CPU emulator. The emulator is configured with passthrough addressing, where the emulator “sees” the host process address space and has access to the same memory, enabling seamless integration between emulated and native execution modes.

2.1. Entrypoint

TVM is initialized through a single tvm_init() call that establishes the calling thread as the initial virtual process - analogous to PID 1 in a POSIX system. All subsequent virtual processes are created as derivatives of this initial process through TVM's fork() implementation. The calling thread then enters its primary operation, which may be an SSH server, interactive shell, or any POSIX application that relies on fork() for its operation.

Example entrypoint for Dropbear SSH [3]:

```
int start_tvm() {
  tvm_init();
  char *args[] = {
    “dropbear”, “-F”, “-E”, “-P”, “2222”, NULL
  };
  execv(“dropbear”, args);
}
```

The execv() call goes through TVM, which implements loading and running the dropbear executable without the kernel, allowing the calling thread to open an SSH server in parallel to the rest of the threads running in the same real OS process.

It should be noted that other threads in the process are unaffected by this call and anything after that. This allows TVM to coexist with unrelated threads of an iOS application.

3. Implementation

The basic UNIX process isolation concept can be divided into a few subjects:

1. Virtual address space
2. Open file handles
3. Signal handlers

We’ll define a task struct, which represents a virtual process and its unique resources such as file descriptors table, signal handlers table and process ID:

```
typedef struct task {
  struct list_head tsk_list;
  pid_t tsk_pid;
  int tsk_result; // as returned by waitpid()
  int tsk_files[MAX_FILES];
  struct sigaction tsk_sighand[MAX_SIGNALS];
} task_t;
static pthread_mutex_t tasks_lock;
static struct task *main_task;
static __thread struct task *current;
```

In the above code snippet, we defined the task as a container of resources that will be used for translation. The tsk_files field is a mapping of task-specific file-descriptors to real file-descriptors opened with the host OS. For example, if we were to redirect stdout fd (1) to /dev/null in some task – the real file-descriptor of /dev/null would not be the file-descriptor number 1 as expected, as this is the file-descriptor used for the parent application’s stdout file-descriptor. Instead, it will be some arbitrary file-descriptor number that populates tsk_files[1]. Essentially, the implementation’s write() function would look like this:

```
ssize_t
write(int fd, const void *buf, size_t count) {
  if (fd >= MAX_FILES) {
    errno = EBADF;
    return -1;
  }
  int rfd = current->tsk_files[fd];
  return (
    ((ssize_t (*)(int, const void *, size_t))
    dlsym(RTLD_NEXT, "write"))(rfd, buf, count
  ));
}
```

The implementation performs a simple translation of file-descriptor numbers based on the file-descriptor table in the current task and then calls the operating system’s write() function, finding it using dlsym(). We assume all slots of tsk_files[MAX_FILES] are initialized to -1 when a task is created, thus calling write() with an unused file-descriptor will yield an EBADF error from the operating system’s write() implementation.

With this example in mind, the reader can more clearly envision the concept of resource translations between the VM and the host operating system.

Similarly to the write() example above, an overridden sigaction() function will register a signal handler with the current task's tsk_sighand field instead of with the host operating system - and a generic signal handler installed by the initial call to tvm_init(), invokes the correct handling logic depending on the thread that received the signal and its associated task struct.

Translating resources requires overriding implementations of hundreds of library function calls for the purposes of:

1. **File-descriptor translation**. Functions like read(), dprintf(), close(), dup() and the like should translate fd numbers from the task-specific file-descriptor tables.
2. **Relative path translation**. Path arguments should resolve relative paths with task-specific working directory.
3. **Process lifetime management**. Functions like exit() and waitpid() should use pthread_exit() and pthread_join() respectively.

Now that we understand the basic idea for how the VM will handle isolation of resources like files and signal handlers, we can move to virtual address space.

3.1. Copy-On-Write (COW)

The very basic semantic of fork() is that the entire virtual address space of a process is cloned into a new process - one process enters fork() and two return from it with identical call stacks. Operating systems do not tend to allow threads to have different virtual address spaces. 2 threads in the same process cannot access different physical addresses via the same virtual address. This makes it non-trivial to implement Copy-On-Write (COW) semantics like we could for file-descriptors. We considered four possible solutions for this issue:

1. **Signal-based page faults**. For each memory page, remove all access permissions with mprotect(PROT_NONE) and duplicate its content on fork(). When a SIGSEGV signal with SEGV_ACCERR code is delivered to a task, handle the memory access in a signal handler. Perform the load / store as indicated by EIP / PC register in the signal handler but on a different memory region, and restore execution at the next instruction. When only one task references a memory region, restore its access permissions so signal handlers are no longer triggered and addressing can resume to full efficiency. This method is very slow and inefficient but could theoretically work.
2. **CPU Emulator**. If all code runs in an emulator, we can also emulate the MMU and redirect an address accessed by a program to different virtual addresses in the host process depending on which task is currently running in the emulator. This method is still inefficient and requires us to be able to emulate platform code which can access non-trivial registers – adding complexity to the emulator implementation.
3. **Software Fault Isolation (SFI)**. A technique that separates multiple address ranges within a single address space. LFI [4] (Lightweight Fault Isolation), a specific implementation of SFI, does so by implementing indirect addressing. On each memory operation, the compiler ensures the upper bits of a pointer are replaced with a variable that can be changed for different contexts (for example, current thread). Addressing the same pointer leads to memory access in different regions of the address space. This method is highly efficient in runtime, especially when considering the previous methods. The downside is that a special compiler extension is needed to insert instructions before memory operations - we wish to support unmodified binaries on top of the implementation.
4. **Deep-Copy and Pointer Fixups**. If we duplicate the entire memory space on fork(), which is the anti-thesis of Copy-On-Write, and organize all copied ranges into a list of tuples of (old range start, new range start, range size) we can scan all newly duplicated memory ranges, scan for pointer values into old ranges and then fixup each pointer to an old range so it points to the same offset in the equivalent new range. This is a slow operation, especially with a lot of duplicated ranges - but the only slow part is the fork() operation itself. Once that operation is over, memory access efficiency is as good as Copy-On-Write.

We chose option 4 since it doesn't trade off the capability of supporting unmodified binaries and it only incurs a performance hit on fork(). We attempt to optimize the operation as much as possible by keeping the amount of memory needed to be scanned at minimum.

We can now envision how the fork() implementation should look like:

```
void child_func(task_t *);

pid_t fork() {
  // duplicate file and signal resources
  // but set unique process id
  task_t *child = duplicate_task(current);
  // new thread should know return address
  child->retaddr = /*current return address*/;
  pthread_create(child_func, child);
  // wait for child to copy mem
  condvar_wait();
  return child->pid;
}

void child_func(task_t *child) {
  current = child;
  // duplicate parent's memory and fixup
  // pointers to old memory ranges. Returns
  // new stack to be used when returning
  stack = deep_copy_memory(child->parent);
  // once mem copied, parent can continue
  condvar_signal();
  // sets return value register to 0
  set_stack_and_jump(stack, child->retaddr);
  // unreachable code
}
```

The main concern of fork() is starting the child, not much interesting logic in it. Since the child performs the memory deep-copy, we need the parent to wait for the child to finish copying – or the duplicated stack will be corrupted and not suitable to be used for returning from fork().

The call to deep_copy_memory() copies and fixes up all parent memory (optimizations to be elaborated later) including the data segments, heap and stack from its parent. This is a 2-step operation. First, we duplicate all relevant memory regions, organizing aside a list of tuples of (old range start, new range start, size). Second, iterate all new memory regions from that list of tuples and fixup any pointer that might point to any old range. This function conveniently returns an address to the duplicated stack, as if the fork() call stack frame was unwinded already.

The call to set_stack_and_jump() switches the stack pointer and performs an absolute jump to the original caller of fork(). Since the stack is switched to a duplicated one, execution should continue as normal.

We'll revisit stack deep copying and switching after we tackle the issue of optimizing the deep copying of the heap and the data segments.

3.2. Optimized Heap

There are many heap implementations. Every userspace thread in our target operating systems allocates memory from at least 2 heaps – the userspace heap (implemented by libc) A heap can be implemented as simple cache over mmap() or as a monstrosity of optimized pointer lookups – but for the purposes of our implementation, we care about something most heaps don't – duplication and fixup time.

It becomes immediately clear that we'll have to manage a different heap for each task. We'll use the term arena for a single instance of our heap linked to a single task. Each allocation and deallocation is performed on a specific arena. Each arena holds a list of slabs, where each slab describes size-class and a range of sequential memory allocated through mmap() which allocations of that size class will be supplied from. This heap's API looks like this:

```
arena_t *arena_create();
void arena_destroy(arena_t *ar);

void *arena_alloc(arena_t *ar, size_t size);
void arena_free(arena_t *ar, void *ptr);
void *arena_realloc(
  arena_t *ar, void *ptr, size_t size);

int arena_duplicate(
  arena_t *dst_ar, arena_t *src_ar);
```

Each task in our implementation holds a pointer to its own arena, which is created when the task is initialized. Main task's arena is initialized by the call to tvm_init(), which at that point is empty. Each subsequent call to fork() creates a new task and its new arena shall be duplicated from the parent task's arena using the arena_duplicate() function.

Our implementation's malloc() function looks like this:

```
void *malloc(size_t size) {
  return arena_alloc(current->tsk_ar, size);
}
```

Now that we understand how our heap is used, we'll to delve into how slabs are organized and implemented in each arena. Let's start by going over the initial problem we're trying to solve with our custom heap – memory duplication and fixup.

Let's assume duplicating a 0x4000 byte-sized page is fast. Our performance issue is the fact we need to iterate every 8-bytes in the newly duplicated page and check if it is a pointer in a list of tuples of (address, size). Granted, the list is typically not very long – but for a few 0x4000 pages it already becomes a time-consuming nested loop operation.

First, we can make sure that slabs are only allocated if they aren't empty. Slabs are destroyed once the last allocation in them is free()'d. This alone makes sure our arena never holds empty slabs, thus never scanning empty slabs.

Second, for each slab we bookkeep the minimum and maximum allocated space offset into the

mmap() range – allowing us to scan significantly less memory on fork(). When duplicating and fixing up pointers in the heap, we are minimizing the range that we are required to scan and fix during the fixup stage. This alone makes a very bold cut-off to the amount of data we have to scan for fixups.

Third, to manage the metadata that tells us what space in the slab is free and what is allocated, we'll use a bitmap where each bit indicates a range of size-class bytes within the slab mmap() range. 1 is allocated and 0 is free. The other common way of tracking this information is by using freelists, but since freelists are usually implemented as a header before each allocation – they will also be scanned when a slab is duplicated and fixed up. Here is an illustration for our slab representation:

```
struct slab {
  size_t   obj_size;
  size_t   slab_size;
  size_t   alloc_cnt;
  size_t   total_cnt;
  size_t   min_off;
  size_t   max_off;
  bitmap_t residents;
  char   data[0];
};

// `total_cnt==16`, `residents` is 16-bit wide
//
// residents: |0000110110010000|
//                 ^       ^
//              min_off   max_off
```

Fields min_off and max_off represent the range of memory we'll scan on fork(). To calculate the corresponding address of min_off, this formula can be used:

```
(intptr_t)slab + sizeof(slab) + min_off
```

When a slab is allocated, it holds the metadata and allocation pool in the same mmap() range. Other features such as caching or memory alignment can be implemented on top of this basic scheme.

In the end, we've created a heap that is optimized deep copying and fixing up memory on fork(). It isn't the most efficient heap in terms of allocation time, as we haven't implemented many mechanisms used by modern and efficient heap implementations like CPU caches – but it does the job, nevertheless. Arenas can store hold allocations not only allocated by its owning task, but also isolatated memory resources such as task-specific environment variables. This allows environment variable pointers passed between parent and child virtual processes to not break address space isolation.

3.3. Dynamic Linker

We have now covered how we're going to implement Copy-On-Write like semantics with fork(), and have demonstrated how we optimized the heap to help us do it efficiently on its memory. The remaining memory types to cover in this regard are the data segment of the program and the call stack. Unlike the heap and the stack – addresses in data segment are usually obtained by PC-relative offsets. Since executable and data segments of the same dynamic library are loaded in the same offsets from each other – the compiler can reference static variables with a fixed distance from the current opcode it generates.

To duplicate the data segments of a program, we need to duplicate all loadable segments of it. We'll implement our own dynamic linker alongside the operating system's dynamic linker. This dynamic linker has unique characteristics:

1. **Namespacing**. To isolate data segments, each task will have its own linker namespace with loadable segments. On fork(), this namespace will be duplicated – loading all segments for each library again and copying the contents of the data section.
2. **Special Symbol Resolution**. When a program imports a function such as write() or a global variable such as environ – we'll resolve these to our overridden functions and task-specific global variables.
3. **Non-executable mapping**. Map executable segments as readable instead. This allows our dynamic linker to work with code signing enforced and is required emulation to work. This will be elaborated on the CPU emulation section.

This linker's API looks like this:

```
linker_t *linker_create();
void linker_destroy(linker_t *lnk);

void *linker_dlopen(
  linker_t *lnk, const char *file, int flags);
void linker_dlclose(void *handle);
void *linker_dlsym(
  void *handle, const char *sym);

int linker_duplicate(
  linker_t *dst_linker, linker_t *src_linker);
```

Similarly to the heap, each task holds a pointer to its own linker namespace (linker_t). Since our linker performs the symbol resolution of imported functions, we can override dlsym() imports of

loaded libraries with our own implementation, which looks like this:

```
void *
private_dlsym(void *handle, const char *sym) {
  return linker_dlsym(
    current->tsk_linker, handle, sym);
}
```

Internally, linker_dlsym() searches for the symbol in its own namespace. If not found, override functions of TVM are returned instead. If still not found, the platform's implementation is returned via a call to the default dlsym() function the system provides. This behavior ensures loaded code is encapsulated inside the VM.

On fork(), we duplicate the namespace by reloading the executable and loaded libraries again – we deep-copy the contents of each parent data segment to its equivalent child data segment and fix pointers in them similarly as described for duplicated heap slabs.

Since we're targeting both Linux and XNU, we'll implement ELF and Mach-O file format support into our linker. While these are different formats, they are very similar in their design – Each Program Header (PT_*) or Dynamic Tag (DT_*) in ELF has some counterpart in Mach-O in the form of a Load Command (LC_*).

1. PT_LOAD - LC_SEGMENT_64
2. DT_NEEDED – LC_LOAD_DYLIB
3. DT_SONAME – LC_ID_DYLIB
4. DT_SYMTAB – LC_SYMTAB
5. DT_JMPREL – LC_DYSYMTAB
6. DT_REL – LC_DYLD_CHAINED_FIXUPS

The basic operation of any dynamic linker, in both Linux and XNU is as follows:

1. Reserve one contagious memory range for all loadable segments, then map each segment from the executable file in that range.
2. Repeat previous step for each unloaded dependency.
3. Iterate all newly loaded libraries and resolve relocations – symbol imports, rebase absolute pointers.

In practice, our dynamic linker supports something most dynamic linkers do not – we can load iOS and Android native code into the same program running on either of those operating systems if a POSIX compliant program that doesn't rely on variadic argument functions. Since these libraries are compiled to the same architecture and these operating systems have the same calling convention for simple non-variadic and non-floating-point C functions. But even basic functions like printf() can't be used due to ABI differences. This anecdote shows how much control we already have in the environment we created.

We are now able to load binaries from the file system. This poses a few minor constraints on the binaries executed through our implementation:

1. Static compiled binaries are not supported – we rely on overriding all relevant operating system's functions to have external code link to our implementation seamlessly.
2. Position-Independent-Executable compilation mode is required – this is the default case for most compilers. We are going to use the linker to map multiple executables in the same process. Non-PIE executable use fixed addresses for loading, which would cause collisions between two virtual processes running two Non-PIE programs.
3. The main() symbol must be present in the binary. We cannot call the _start entrypoint ourselves. This theoretically can be dealt with but requires overriding specific platform-dependent functions which for now, we prefer to skip and prioritize moving on.

Overall, our defined implementation for a linker is complete – it supports loading and relocating executable files and libraries. The only thing left is to initialize thread-local storage and run constructor/initializer functions, details we leave to the reader.

Now that we have implemented a custom linker, we have full control over the heap and data segments. The last missing piece is the call stack.

3.4. Call Stack & fork()

We are at the last step of our userspace fork() implementation. As earlier established, Copy-On-Write semantics is the non-trivial part. We've implemented a custom heap and dynamic linker to enable duplication and pointer fixups on the heap and data segments. Now we'll deal with the call stack and the fork() implementation itself – which are tightly coupled.

The API of pthread_create() used for creating new threads and the API of fork() used for creating new processes is very different and has direct implications to how the call stack is handled. Observe the following code illustration:

```
void *thr(void *) {
  // thread code on new stack
  // 0x00 .       .
  //      |       |
  //      |-------|
  //      |  thr  | <- current frame
  //      |-------|
  // 0xFF |_entry_| <- stack bottom
}

void func() {
  pthread_t p;
  pthread_create(&p, NULL, thr, NULL);

  fork();
  // parent and child code on duplicated stacks
  // 0x00 .      .
  //      |      |
  //      |------|
  //      | func | <- current frame
  //      |------|
  //       caller|
  //      |------|
  //      |  ... |
  // 0xFF '      '
}
```

Our goal is to have the thread start routine adjust its stack so critical frames are copied from the parent task's stack to the child task, fixup pointers to parent stack just like heap and data segments, and return back to the caller of fork() with the copied stack frames to imitate the same API of fork().

While by now it's easy to replicate the same technique used for heap and data segments, the complex part is rather the stack switching and relinking the stack frames. It isn't a straightforward process and requires non-trivial work to do correctly. Since our implementation targets 2 different architectures, with 2 different ABIs – we prioritize implementing the call stack switch with as minimal architecture-specific code as possible.

Our primitives for switching stack are setjmp() and longjmp() functions. These POSIX compliant functions deal with saving and restoring CPU state on a jmp_buf object. Here is an example usage of setjmp() and longjmp():

```
#include <stdio.h>

jmp_buf errjmp;

void work() {
  while (-1 != open("/dev/null", O_WRONLY))
    ;
  longjmp(errjmp, 1); // error
}

void main() {
  if (!setjmp(errjmp))
    work();
  printf("error\n");
  return 1;
}
```

In the above example, errjmp is a global variable initialized at program start. When an error occurs it's used to exit the program. Function setjmp() returns with 0 when first called to save execution context to the jmp_buf argument, and non-0 value when returning due to a longjmp() using that jmp_buf variable. One important constraint is that once setjmp() is called in a certain function – longjmp() shouldn't be called if the function that called setjmp() on the same jmp_buf has already returned. This constraint makes sense, since if the original context-saving function returned, its call frame has already cleaned up and the stack variables, return address and saved frame pointers are corrupted. This implies a very crucial rule we'll take advance of – whenever a longjmp() is being performed – the stack frames are unwinded to a previously saved frame on the stack. Here is an illustration of the stack between setjmp() and longjmp() calls in the previous example:

```
0x00 .      .
     |      |
     |------|
     | work | >-. longjmp from this frame
     |------|   |
     | main | <-' to this frame
     |------|
0xFF |_libc_| <- stack bottom
```

A reader well acquainted with operating systems and low-level programming can infer jmp_buf contains an array of CPU critical registers including the return address, stack and frame pointers, and the instruction pointer / program counter – and when looking at Glibc's and Apple's source code for setjmp() and longjmp() that is indeed the case.

We'll now demonstrate a technical solution to fork(). Before the parent thread's thread creates the child thread, it calls setjmp(). We'll fixup jmpbuf register values in the child start routine so longjmp() switches stack pointer and return address, returning from fork(). In the start routine we copy relevant frames from the parent's call stack to the child's call stack and relink the earliest frame copied to the caller of the start routine's frame as illustrated below:

```
       PARENT              CHILD
0x00  .       .           .       .
      |       |           |       |
      |       |    ,---> |-------|
      |-------| ---'     | fork  |
      | fork  |           |-------|
      |-------|           |caller |
      |caller | memcpy()  |-------|
      |-------|           |  ...  |
      |  ...  |           |-------|
      |-------|           | begin |
      | begin |    ,---> |-------|.
      |-------| ---'     | start | | relink
      |  ...  |           |-------|'
0xFF  '       '           |_entry_|
```

Frame “begin” marks the earliest relevant frame to copy from the parent stack to the child stack. For both x86-64 and AArch64, each stack frame holds a 16-byte prologue that contains an 8 byte pointer to the previous frame and an 8-byte return address pointer. When initially executing code in the thread’s start routine, we obtain the current frame pointer with __builtin_frame_address(0). Relinking the “begin” frame to the child’s entry frame as illustrated above requires overriding “begin” frame’s prologue previous frame value to the value we queried at the beginning of the thread’s start routine. We’ll also override the return address in “begin” frame’s prologue using the __builtin_return_address(0) in the beginning of the fork() call in the parent.

Note that by relinking “begin” to “entry”, we are essentially skipping the start routine frame. That’s okay, and in fact we can completely overwrite the start frame in the process because its longer relevant after the final call to longjmp().

It is by sheer tyranny of luck that stack frame prologues in x86-64 and AArch64 architectures’ ABI for both Linux (Android) and Darwin (MacOS, iOS) are designed exactly the same – making this whole part of our implementation architecture-agnostic.

After the parent calls setjmp() and pthread_create(), it waits for the child’s thread completion signal before returning as the child needs to safely copy stack frames that will otherwise cleanup. This keeps the parent’s stack dormant and valid while the child copies frames from it.

While the child modifies its stack own stack, copying frames from the parent, it needs to use a temporary stack – as we cannot modify a stack we’re currently using. Switching stacks is as simple as calling setjmp(), adjusting the saved registers in jmp_buf pointing to the old stack to be pointed to the new stack, and performing a longjmp() call on that jmp_buf. This means we’re actually switching stacks twice in the child start routine. First switch is to a temporary stack while modifying the original stack, second switch from the temporary stack to the original stack with stack frames copied from the parent.

Putting it all together, the order of operations in our implementation’s fork() function looks like this:

```
pid_t fork() {
  fork_arg_t *arg = alloc_fork_arg();
  arg->parent = current;
  arg->child = alloc_task_inherit(current);
  if (setjmp(arg->final_jmpbuf)) {
    // child task
    Current = arg->child;
    pthread_cond_signal(arg->cond);
    return 0;
  }
  // parent task
  pthread_create(&ntask->pthr, NULL,
                 tvm_entry, arg);
  pthread_cond_wait(arg->cond);
  return ntask->tsk_pid;
}
```

The heavy lifting occurs in the child’s start routine, tvm_entry(). Function alloc_task_inherit() allocates a new task struct and inherits resources and properties from the parent task (current variable).

Moving on, the child start routine begins by switching to a temporary stack:

```
void tvm_entry(fork_arg_t *arg) {
  uint64_t tmpfp, tmpsp, currsp, reg;
  arg->currfp = __builtin_frame_address(0);
  if (!setjmp(arg->tmp_jmpbuf)) {
    // switch to temp stack using tmp_jmpbuf
    currsp = arg->tmp_jmpbuf[SP];
    arg->tmpstack = mmap(STACKSIZE);
    // fp is first 16 bytes pushed to stack
    tmpfp = (long)arg->tmpstack+STACKSIZE-16;
    tmpsp = tmpfp - (currfp - currsp);
    // copy current frame to temp stack
    memcpy(tmpsp, currsp, currfp-currsp);
    // fixup jmp_buf to pointers to temp stack
    for (int i = 0; i < JMPBUF_CNT; i++) {
      reg = arg->tmp_jmpbuf[i];
      if (reg >= currsp && reg < currfp)
        arg->tmp_jmpbuf[i] -= currsp - tmpsp;
    }
    // switch to temp stack via jmpbuf
    longjmp(arg->tmp_jmpbuf, 1);
  }
  // code running on temporary stack
  tvm_entry_cont(arg);
}
```

In the above snippet, we’ve seen a hands-on example of how stack switching works with setjmp() and longjmp() for the first time. The above code implements the frame copying we discussed earlier, except we don’t relink any frame because this stack is temporary a return from tvm_entry() never occurs while using it. Once the thread runs on a temporary stack, we can safely modify its original stack and then switch back to it. The important takeaway from this first half part of the child’s start routine is how we used the jmp_buf as a primitive for switching stacks. The saved context remains the same except for the stack-based registers (sp and fp, at the very least) which are adjusted to the new stack with the newly copied frames. This is a crucial step to understand before we wrap up the call stack switching with the final step in tvm_entry_cont().

We've switched to a temporary stack using setjmp() and longjmp(), the child's stack is dormant and we can safely modify to it. With final_jmpbuf, a context saved in the fork() function with the parent's stack, we can switch stacks back and restore execution back in fork(). Except for copying parent frames and switching stacks, we have one extra step this time – duplicating all relevant memory ranges and fixing up pointers in them. We do memory fixups now its inherently intertwined with modifying the original stack. Stack frames copied from the parent contain pointers back to the same parent stack – such as pointers to previous stack frames. But the stack might also contain pointers to objects residing in the heap and the data segment. The heap and data segments might also contain pointers to themselves and the stack. We need to perform the pointer fixups on all ranges – stack, heap and data segment. This is quite of a boiling point for all the memory management issues we've talked about. Here is the implementation snippet:

```
void tvm_entry_cont(fork_arg_t *arg) {
  // copy stack frames and load to fixups
  uint64_t oldsp, oldfp, newsp, newfp, stacksz;
  oldsp = arg->final_jmpbuf[SP];
  oldfp = arg->parent->tsk_fork_fp;
  stacksz = oldfp - oldsp;
  newfp = arg->currfp; // from tvm_entry()
  newsp = arg->currfp - stacksz;
  arg->child->tsk_fork_fp = newfp;
  memcpy(newsp, oldsp, stacksz);
  addfixup(arg->fixups, newsp, oldsp, stacksz);
  // duplicate heap arena, register old and new
  // ranges with arg->fixup context
  arena_duplicate(
    arg->child->tsk_ar,
    arg->parent->tsk_ar,
    arg->fixups);
  // likewise, duplicate linker namespaces
  linker_duplicate(
    arg->child->tsk_linker,
    arg->parent->tsk_linker,
    arg->fixups);
  // load the final_jmpbuf to arg->fixups
  addfixup(arg->fixups, arg->final_jmpbuf,
           arg->final_jmpbuf, JMPBUF_CNT*8);
  // resolve fixups
  fixup_t *fx = arg->fixups.list;
  size_t fixupcnt = arg->fixups.cnt;
  for (int i = 0; i < fixupcnt; i++)
    for (uint64_t *p = fx[i].new;
         p+1 < fx[i].new + fx[i].size; p++) {
      for (int j = 0; j < fixupcnt; j++) {
        if (p >= fx[j].old &&
            p < fx[j].old + fx[j].size) {
          *p = *p + fx[j].new - fx[j].old;
          break;
        }
      }
    }
  }
  // switch to fixed up child stack at fork()
  longjmp(arg->final_jmpbuf, 1);
}
```

The above snippet is the fulcrum of this paper. The arg->fixups field is essentially the list of tuples of (old range, new range, size) we use to fixup pointers in ALL relevant memory regions. The algorithm towards the end of the above snippet is the raw pointer fixup logic. It takes the new range of each fixup tuple, checks if each 8-byte pointer in it pointers to any old range, and if it does – it applies the distance of the old range to its equivalent new range. The final_jmpbuf is also loaded into fixups list with the same new and old pointers, so it's only being fixed up inside but references to it remain the same.

After the longjmp() call, execution continues back in the fork() implementation provided earlier. In it, the child sets current to the new task and signals the parent for completion, letting both child and parent task return from fork().

An important detail here is that execution returns to our implementations fork() function, but since all unsigned executable segments were duplicated – the child returns to a different address. Same code as the parent, but duplicated to a different memory area. Our pointer fixup code iterated the copied stack frames and adjusted the return address from the fork() frame to the duplicated callside of fork().

The last thing required for supporting process creation is execve() in userspace. At this point, it's a pretty trivial implementation:

```
int execve(
  const char *file, char **argv, char **envp) {
  // calculate argc
  int argc = 0;
  for (; *argv; argc++, argv++)
    ;
  // create new linker namespace and load file
  linker_t *nlinker = linker_create();
  void *hnd = linker_dlopen(nlinker, file);
  void *p = linker_dlsym(nlinker, hnd, "main");
  // switch linker namespaces
  linker_destroy(current->tsk_linker);
  current->tsk_linker = nlinker;
  // jump to main symbol
  int ret = (((int (*)(int, char **, char **))
    p)(argc, argv, envp);
  exit(ret);
}
```

Additional cleanup of heap and stack are required, we leave these details for the reader. The famous Userland Exec [5] paper brilliantly explains all necessary details.

This section concludes our technique for thread-based process abstraction. We've achieved virtual process isolution in terms of resources and most importantly – memory. Copy-On-Write semantics in userspace are now fully implemented.

.

3.5. Special Files

We have achieved process isolation and now to have an SSH Server and a Shell we have one technical problem left to solve. We need a userspace implementation of TTY & PTY devices.

TTY (TeleTYpewriter) is a special character device that acts as a connection between some input/output terminal hardware and a terminal program. A TTY is usually opened as a controlling terminal, which means the "controlled" process is notified of special input via signals immediately as they are received by the operating system – such as Ctrl+C input causing a SIGINT signal to be delivered to the foreground process of a terminal session.

PTY (Pseudo-TTY) is a virtual TTY represented by 2 bidirectional and connected file-descriptor ends – master and slave. The slave end acts as a normal TTY while the master end acts as the input/output hardware that can send special input like Ctrl+C.

PTY is the backbone technology that allows GUI-driven operating systems to emulate terminals through programs like GNOME Shell or iTerm. Software can implement both ends of the terminal connection, feeding user input from a GUI application through the master end of a PTY and into the shell process that receives it through the slave end of the PTY. Likewise, it allows SSH servers to open interactive terminal sessions, feeding user input from a socket into a PTY's master end.

We must implement a userspace PTY driver because by design a process can have only one controlling terminal. Our sandboxed process cannot use the operating system's PTY driver if our implementation aims to support more than one active terminal session. Besides that, we have no control over which thread in the process receives a signal from the controlling terminal when the operating system's implementation is used.

PTY drivers are quite simple, whenever a character is being sent by one end and to the other – some special processing is made on the data, potentially consuming it. Instead of the other end read()'ing special characters representing Ctrl+C, a signal is delivered instead. If you ever need a reference implementation, you are encouraged to read XNU's [6] implementation of PTY – it's simpler than the Linux counterpart.

To implement such device in userspace, we could use the socketpair() system call to create 2 bidirectional anonymous UNIX domain sockets to act as the master and slave end of a PTY and on every write() operation, do some data-processing. For example, if the characters representing Ctrl+C are being written, send a SIGINT signal to an arbitrary thread of the controlled virtual process and drop the data (do not write() it).

There's a bigger issue hiding in here. When a task performs write() on a file-descriptor, how can we know if we need to perform special behavior or just pass it through to the operating system's write() with the translated file-descriptor number? Furthermore, each open TTY in an operating system is described by a character device inode on the filesystem, usually at /dev/ttyXX (where XX is a unique identifier for that TTY). It is also then required that the tty pathname be openable by processes willing to attach themselves to the TTY and make it their process' controlling terminal. These details aren't some mere hypothetical situations but what Dropbear SSH [3] and mksh [7] require to properly function.

We need some type of distinction between file-descriptor entries in the tsk_files field of each task. Also, we need awareness to special filenames when implementing open().

Instead of having tsk_files defined as an array of ints, we'll define it as an array of struct file * entries:

```
#define DEV_REG 0
#define DEV_TTY 1

struct file {
    int f_rfd; // real file-descriptor
    int f_type;
    void *f_data;
    struct fops *f_ops;
};
```

Each file entry will no longer only be represented by the real file-descriptor number but also by a type field. DEV_REG type is just a regular file with only file-descriptor translation IO behavior, its f_data and f_ops fields are set to NULL. When a TTY is opened, f_type is set to DEV_TTY, f_data set to our implementation's internal TTY struct and f_ops points to a structure containing function pointers to special IO implementations for our TTY. Each file type is implemented as a device with its own file operations.

Which means our write() implementation should now look like this:

```
ssize_t
write(int fd, const void *buf, size_t count) {
  if (fd >= MAX_FILES ||
      !current->tsk_files[fd]) {
    errno = EBADF;
    return -1;
  }
  struct file *file = current->tsk_files[fd];
  if (file->f_ops && file->f_ops->write)
    return file->f_ops->write(file, buf,
                              count);
  return (
    ((ssize_t (*)(int, const void *, size_t))
    dlsym(RTLD_NEXT, "write"))(
      file->f_rfd, buf, count
  ));
}
```

That file->f_ops->write function pointer points to a function that implements the special TTY data-processing we mentioned earlier.

Another important implementation to review is open(), which should now check if the desired pathname to open is related to any special file type such as TTY:

```
int
open(const char *path, int flag, mode_t mode) {
  struct file *file = NULL;
  // iterate device types and try to open each
  struct device *dev;
  list_for_each_entry(dev, &devices) {
    file = dev->fops->open(path, flag, mode);
    if (file)
      break;
  }
  // if no device can open path, regular open
  if (!file) {
    int rfd = ((
      (int (*)(const char *, int, mode_t))
      dlsym(RTLD_NEXT, “open”))(
        path, flag, mode)));
    if (-1 != rfd) {
      file = alloc_file();
      file->f_type = DEV_REG;
      file->f_rfd = rfd;
    }
  }
  // couldn’t open file, even with open()
  if (!file)
    return -1;
  // put file into an available fd in the task
  int nfd = task_assign_file(current, file);
  // too many open files in task
  if (-1 == nfd) {
    free_file(file);
    errno = EMFILE;
    return -1;
  }
  return nfd;
}
```

In the above code, we first tried to open a file with each registered device type and only once all device types couldn’t open the file we resorted to opening it with the operating system. Our implementation’s system call wrappers begin to resemble modern POSIX compliant operating system implementations.

This device subsystem of our implementation can be extended to other device types, such as:

1. **Virtual File System**. In case the sandboxed environment process has no writable directories to be used as $HOME, we could create an in-memory file-system of our own.
2. **Initramfs**. With our iOS application, we can ship an archive with precompiled unsigned binaries to be mounted at some top-level directory and have them readable by our tasks once we implement the emulator. We could even put the Dropbear, mksh and other dependencies we currently statically compile to the implementation.

With TTY device support, our implementation supports SSH servers, shells and shell utilities. Our userspace TTY driver is fully functional and supports raw mode. With a robust TTY driver, Vim, Nano and even a terminal-based Snake game can be used on top of our VM.

3.6. CPU Emulation of Unsigned Code

Our implementation is fully capable of loading programs without modifications and serving virtual process while keeping a sense of isolation – but we still can’t execute unsigned code. This limitation stems from the fact we can’t map memory with PROT_EXEC permissions on a sandboxed environment like an iOS application. Our solution is to allow unsigned binaries to run without their executable segments having EXECUTE memory privileges by using a CPU-instruction parsing emulator. The Unicorn [8] emulator supports both x86-64 and ARM64 instruction sets with a simple C API:

```
#include <unicorn/unicorn.h>
#include <setjmp.h>
int main() {
  // save context for emulated work
  jmp_buf jb;
  if (setjmp(jb))
    return work();
  // setup emulator
  uc_engine *uc;
  uc_open(UC_ARCH_ARM64, UC_MODE_ARM, &uc);
  uc_mem_map_ptr(
    uc, 0x4000, 0xFFFFFFFFFFFF8000,
    UC_PROT_ALL, 0x4000);
  uc_reg_write(uc, UC_ARM64_REG_PC, jb[PC]);
  uc_reg_write(uc, UC_ARM64_REG_SP, jb[SP]);
  uc_reg_write(uc, UC_ARM64_REG_LR, jb[LR]);
  uc_reg_write(uc, UC_ARM64_REG_FP, jb[FP]);
  uc_reg_write(uc, UC_ARM64_REG_X0, jb[X0]);
  // ^ do this for X0-X28 ...
  // make stack space for emulation, start it
  alloca(0x8000);
  return uc_emu_start(uc, jb[PC], 0, 0, 0);
}
```

The above snippet assumes the reader is familiar with setjmp() as explained in the fork() section. This example program "switches" to emulation mode by extracting registers from a saved jmp_buf and using them to resume execution on an emulated CPU on the same native thread. The key insight here is the uc_mem_map_ptr() call. This call nullifies Unicorn's MMU by mapping the entire address space of the host process into the same offset, essentially giving the emulator pass-through access to the host memory. The emulation proceeds to execute on the same stack as the native program and calls work(). This works because Unicorn has implemented their emulator to run in the same thread calling the uc_emu_start() function, and returns when emulation stops.

The above example uses a cool trick to allow the emulated CPU and real CPU share the same stack. It gives the emulated CPU a 0x8000 byte-sized slice of the stack, evident by the call to alloca(), before entering the emulator. This lets both CPUs run on different zones of the stack:

```
0xFFFF .      .
       |      |  | native
       |      |  |   stack area
0x8200 | uc_* | /
       |------|-
       |      | \
       | .... |  | emulator
       |      |  |   stack area
0x0200 | work | /
       |------|'
0x0100 | main |
       |------|
0x0000 |_libc_| <- stack bottom
```

In our implementation, instead of mapping the virtual address space as UC_PROT_ALL, we'll map it as UC_PROT_READ | UC_PROT_WRITE without UC_PROT_EXEC permission. This allows us to selectively control what code executes in the emulator, and what executes on the real CPU. Essentially, the only code we wish to execute in the emulator is code dynamically loaded via our linker's dlopen(). Whenever a library calls an external function not loaded with our dynamic linker, we'll stop emulation, extract arguments from the registers and stack, and then call that function from the native CPU before returning back to the emulator with the result passed in ARM64's X0 or x86-64's RAX register. Every time the emulator hits a memory region it can't execute due to the missing UC_PROT_EXEC permission, it's going to stop emulation and let us handle the permission error. This allows us to implement a page-fault mechanism where we incrementally map executable segments to the emulator if the memory violation address resides in a region loaded by our dynamic linker – or else treat it as a function call, run it on the native CPU and continue execution in the emulator at the return address.

We will define emulator_call(), a functions that calls another function in the emulator, with the details discussed above. Here's a code example with indicative comments:

```
uint64_t
emulator_call(
  uint64_t pc, uint64_t *params)
{
  // setup emulator
  uc_engine *uc;
  uc_open(UC_ARCH_ARM64, UC_MODE_ARM, &uc);
  uc_mem_map_ptr(
    uc, 0x4000, 0xFFFFFFFFFFFF8000,
    UC_PROT_READ | PROT_WRITE, 0x4000);
  // setup new stack with stack arguments
  uint8_t *estack_top = mmap(NULL,
    STACK_SIZE, PROT_READ|PROT_WRITE, ...);
  uint8_t *estack = estack_top + STACK_SIZE;
  // load params to UC and stack, modifies
  // `estack` to reflect pushed params
  params_to_uc_and_stack(params, uc, &estack);
  // write initial stack and return address
  // detect finish by fetch error on 0xDEAD
  uc_reg_write(uc, UC_ARM64_REG_SP, estack);
  uc_reg_write(uc, UC_ARM64_REG_LR, 0xDEAD);

  // run until finish
  while (1) {
    uc_err err = uc_emu_start(
      uc, pc, 0, 0, 0);
    pc = uc_reg_read(uc, UC_ARM64_REG_PC);
    // only instruction fetch prot error is ok
    if (UC_ERR_FETCH_PROT != err)
      abort(); // error
    // extract return value on finish, return
    if (pc == 0xDEAD)
      return uc_reg_read(uc, UC_ARM64_REG_X0);
    memset(&dli, 0, sizeof(dli);
    // check if pc loaded by linker
    linker_dladdr(
      current->tsk_linkns, pc, &dli);
    // found in linker, page in text segment
    if (dli.dli_fbase) {
      uc_mem_map_ptr(uc, dli.dli_textbase,
        dli.dli_textsize, UC_PROT_ALL,
        dli.dli_textbase);
      continue; // continue emulation
    }
    // not found in linker, run on host
    // params_from_uc() reads the stack too
    params_from_uc(params, uc);
    uint64_t ret = (((uint64_t (*)(
      uint64_t, uint64_t, ..., uint64_t))pc)(
      params[0], params[1], ..., params[N]);
    pc = uc_reg_read(uc, UC_ARM64_REG_LR);
    uc_reg_write(uc, UC_ARM64_REG_X0, ret);
    // restart emulation at return address
  }
  // unreachable
}
```

The above snippet is implemented for ARM64, but it can trivially be implemented to both ARM64 and x86_64 with a few if conditions. That whole function lets the implementation call unsigned code using Unicorn and let the unsigned code to switch back to the host if it calls signed code. This

essentially allows the emulated code to interact with our implementation and the operating system itself – closing the final important step in our implementation's solution to the sandbox problem.

The only thing left to do is connect the dots, use emulator_call() when calling a main function in execve() that was loaded using our dynamic linker with pc set to the function address and params pointing to an array of uint64_t parameters (argc, argv, envp).

A small modification to our implementation is required to support fork() as unsigned code requires fork() to return in emulation mode but that isn't much big of a deal considering we can redirect fork() usage through dynamic linker to another implementation of fork() the wraps our original fork() implementation with extra steps of saving / loading emulator context.

A few small modifications are also needed for all functions accepting a function argument such as pthread_create(), atfork(), sigaction() and even bsearch() and qsort(), as the function executed is required to run in the emulator as the real CPU cannot jump back into the given function pointer as its not mapped as executable.

In the end, we successfully solved the defined problem – a full implementation that not only supports thread-based process virtualization, but also CPU emulation of unsigned code that allows us to overcome Code Signing.

If an unsigned program that uses IOKit's API is executed on top of our implementation – the inner loop of the demonstrated emulator_call() code lets the emulator jump out to the host's CPU and perform the necessary system calls to interact with the desired IOKit driver. All of this logic occurs while full Sandboxing and Code Signing mechanisms are set in place.

4. Discussion

4.1. Architectural Resilience

TVM relies on three categories of standard POSIX APIs available to all iOS applications: pthread_create() for thread creation, mmap() with read-write permissions for memory allocation, and dlsym() for dynamic symbol resolution. These primitives are fundamental to many applications and their dependencies, including applications that implement threading libraries, memory allocators, plugin systems, and runtime analysis tools.

Even aggressive platform restrictions would not break TVM's core design. We examine potential API limitations and their workarounds:

1. **Signal handling**: If POSIX signal APIs were restricted, TVM could register Mach exception handlers directly. POSIX signals on Darwin are implemented as a layer over Mach exceptions - by receiving exception messages on dedicated Mach ports and handling them in a background thread, TVM could reimplement signal semantics without relying on the platform's sigaction() implementation.
2. **PThread API**: If threading APIs were unavailable, TVM could implement pre-emptive scheduling in userspace through asynchronous signaling - though such restrictions would break the broader application ecosystem.
3. **dlsym()**: If symbol resolution were restricted, TVM's custom dynamic linker could parse executable formats directly to locate and link platform functions.

TVM's architecture demonstrates resilience: the fundamental approach (thread-based virtualization with selective emulation) remains viable even if specific APIs become restricted. Restrictions severe enough to break TVM would necessarily impact legitimate applications relying on the same primitives.

4.2. App Store Policy Considerations

iOS applications can be distributed through the App Store, enterprise deployment, unofficial app stores and the TestFlight beta testing program. The App Store requires compliance with review guidelines specific to that distribution channel. This section examines App Store policies as they relate to one potential distribution method; TVM's functionality and alternative distribution options are independent of App Store approval. However, as the App Store represents the primary distribution channel for the vast majority of iOS users, these policies merit discussion.

In November 2020, Apple sent a notice of termination to iSH [9] and a-Shell [10] under App Store Review Guideline 2.5.2. The guideline states:

"Apps should be self-contained in their bundles, and may not read or write data outside the designated container area, nor may they download, install, or execute code which introduces or changes features or functionality of the app, including other apps." [11]

Both applications enable users to download and execute programs through CPU emulation or WebAssembly, in addition to supporting scripting languages. Following developer appeals, Apple approved both applications without requiring significant architectural changes.

TVM's architecture aligns with the design patterns that proved acceptable in the iSH and a-Shell cases. Like those applications, TVM ships all platform binaries within the application bundle rather than downloading them at runtime. User-provided code executes through CPU emulation, treating instructions as data rather than mapping executable code. The application functions as a self-contained development environment.

While approval decisions are made case-by-case, the continued availability of similar projects suggests that development environments and shell applications can comply with App Store review guidelines.

4.3. Applications

Security Research. TVM enables several workflows previously impractical on iOS devices. Security researchers can compile and execute code on target hardware without signing their code. This eliminates the need for network connectivity during development and avoids exposing research tools to external signing services - a consideration relevant for sensitive security research. The ability to run arbitrary POSIX programs facilitates systematic exploration of sandbox boundaries. Researchers can rapidly test hypotheses about accessible APIs, file system permissions, and inter-process communication channels without creating custom applications for each experiment.

Testing. TVM addresses practical development challenges. Continuous integration pipelines can execute shell scripts and command-line tools on actual hardware rather than simulators, improving test fidelity while maintaining automation workflows familiar to developers.

Container Deployment. TVM's POSIX environment enables container runtime implementation on iOS. OCI/Docker images could run unmodified, eliminating the need for iOS-specific porting of containerized applications.

4.4. Limitations

Security. Virtual processes share a common address space. The CPU emulator enforces memory isolation for emulated code through a virtualized MMU. However, when emulated code invokes native functions (system calls, library functions), it passes memory addresses as arguments. These native functions execute without MMU constraints and can access any address in the shared space, including memory belonging to other virtual processes. This boundary crossing between protected emulated code and unprotected native execution creates an exploitable gap in the isolation model.

Performance. CPU emulation incurs significant overhead compared to native execution. iOS Code Signing restrictions prevent JIT compilation, forcing instruction-by-instruction interpretation. Additionally, Copy-On-Write semantics requires immediate memory duplication rather than lazy copying, as userspace processes cannot trap page faults to implement deferred copying in a portable manner. This incurs slow process creation time.

5. Related Work

Library OS. Library OS implementations intercept and re-implement system calls in userspace. Container sandboxing systems like gVisor [12] provide additional security by limiting direct OS access, while confidential computing systems like Gramine [13] provide POSIX environments for applications running in non-Unix environments such as SGX enclaves. However, these systems typically either disallow fork() entirely or delegate it to the host kernel, avoiding userspace implementation of process semantics. While Library OSes address resource isolation challenges like TVM (file descriptor management, signal handling), they do not tackle memory duplication in shared address spaces - the core challenge TVM addresses.

Single-Address-Space Operating Systems. SASOS research addresses fork() implementation when processes share a common address space - the same challenge TVM faces. Early systems like Angel [14] and Mungi [15] relied on segment registers for address indirection, enabling fork() without pointer fixups. However, modern instruction sets no longer support this addressing mode. Recent work, uFork [16], implements fork() through explicit pointer fixups similar to TVM's approach, but leverages CHERI memory tagging for pointer validation - hardware unavailable on iOS devices. TVM achieves similar functionality using software-based pointer tracking and fixups, trading performance for broader platform compatibility.

Software Fault Isolation. SFI techniques offer an alternative approach to implementing fork() in shared address spaces. LFI [4] uses compiler extensions to represent pointers as 32-bit offsets within program-local segments, with upper address bits supplied from context registers. This enables fork() without pointer fixups: memory is copied between segments while pointers retain their offsets. TVM instead supports unmodified binaries from standard compilers, accepting the cost of explicit pointer fixups during fork() to maintain compatibility with existing POSIX programs.

iOS Shell Apps. As discussed in Section 1, iSH and a-Shell represent prior iOS shell implementations. TVM differs by combining native execution for signed code with selective emulation for unsigned binaries, enabling both host integration and arbitrary binary execution.

6. Conclusion

We presented TVM, a system that implements POSIX process semantics within sandbox constraints through thread-based virtualization and selective CPU emulation to overcome code signing enforcement for unsigned executables.

TVM's key contributions are techniques for: (1) implementing fork() using pthread_create() with resource isolation and Copy-On-Write semantics, and (2) a hybrid emulation architecture that switches between emulated and native execution modes when running unsigned and signed code respectively.

We demonstrate an implementation that allows unsigned programs to run in a fully sandboxed iOS application environment with Sandbox and Code Signing restrictions. This enables previously impractical workflows: security researchers can execute programs on physical devices without code signing, and experiment with real iPhone hardware with standard Unix tooling.

iOS security restrictions prevent applications from creating processes or executing unsigned code. Yet the standard APIs available to those same applications are sufficient to bypass both constraints from within. Security boundaries are only as strong as the primitives they permit.

References

[1] "iSH site," [Online]. Available: https://ish.app/.

[2] "a-Shell site," [Online]. Available: https://holzschu.github.io/a-Shell_iOS/.

[3] "Dropbear SSH site," [Online]. Available: https://matt.ucc.asn.au/dropbear/dropbear.html.

[4] Z. Yedidia, "Lightweight Fault Isolation: Practical, Efficient, and," 2024. [Online]. Available: https://www.scs.stanford.edu/~zyedidia/docs/papers/lfi.pdf.

[5] grugq, "The Design and Implementation of Userland Exec," [Online]. Available: https://grugq.github.io/docs/ul_exec.txt.

[6] Apple, "Github page of XNU," [Online]. Available: https://github.com/apple-oss-distributions/xnu.

[7] "MirBSD Korn Shell site," [Online]. Available: http://www.mirbsd.org/mksh.htm.

[8] "Unicorn site," [Online]. Available: https://www.unicorn-engine.org/.

[9] "About iSH's pending removal from the App Store," 8 11 2020. [Online]. Available: https://ish.app/blog/app-store-removal.

[10] @a_Shell_iOS, "Tweet from a-Shell developer confirming Apple termination notice," [Online]. Available: https://x.com/a_shell_ios/status/1325526061099196416.

[11] "Apple App Store Review Guidelines," [Online]. Available: https://developer.apple.com/app-store/review/guidelines/#software-requirements.

[12] "Wikipedia page of Google gVisor," [Online]. Available: https://en.wikipedia.org/wiki/GVisor.

[13] "Gramine documentation," [Online]. Available: https://gramine.readthedocs.io/en/latest/index.html.

[14] K. M. A. S. a. T. S. Tim Wilkinson, "Compiling for a 64-bit single address space architecture.," 1993. [Online]. Available: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=56a151c71e09dac552094a6940e6262bc28b642f.

[15] K. E. J. V. S. R. a. J. L. Gernot Heiser, "The Mungi Single-Address-Space Operating System," 1998. [Online]. Available: https://trustworthy.systems/publications/papers/Heiser_EVRL_98.pdf.

[16] H. L. a. P. O. John Alistair Kressel, "µFork: Supporting POSIX fork Within a Single-Address-Space OS," 2025. [Online]. Available: https://arxiv.org/html/2509.09439v2#bib.bib40.