ARMlock: Hardware-based Fault Isolation for ARM

Yajin Zhou*
North Carolina State University
yajin_zhou@ncsu.edu

Xiaoguang Wang*
Xi'an Jiaotong University
xgwang@cs.fsu.edu

Yue Chen
Florida State University
ychen@cs.fsu.edu

Zhi Wang
Florida State University
zwang@cs.fsu.edu

ABSTRACT

Software fault isolation (SFI) is an effective mechanism to confine untrusted modules inside isolated domains to protect their host applications. Since its debut, researchers have proposed different SFI systems for many purposes such as safe execution of untrusted native browser plugins. However, most of these systems focus on the x86 architecture. In recent years, ARM has become the dominant architecture for mobile devices and gains in popularity in data centers. Hence there is a compelling need for an efficient SFI system for the ARM architecture. Unfortunately, existing systems either have prohibitively high performance overhead or place various limitations on the memory layout and instructions of untrusted modules.

In this paper, we propose ARMlock, a hardware-based fault isolation for ARM. It uniquely leverages the *memory domain* support in ARM processors to create multiple sandboxes. Memory accesses by the untrusted module (including read, write, and execution) are strictly confined by the hardware, and instructions running inside the sandbox execute at the same speed as those outside it. ARMlock imposes virtually no structural constraints on untrusted modules. For example, they can use self-modifying code, receive exceptions, and make system calls. Moreover, system calls can be interposed by ARMlock to enforce the policies set by the host. We have implemented a prototype of ARMlock for Linux that supports the popular ARMv6 and ARMv7 sub-architecture. Our security assessment and performance measurement show that ARMlock is practical, effective, and efficient.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Access controls, Information flow controls*

Keywords

SFI; ARMlock; Fault Isolation; DACR

*The bulk of this work was completed when the first two authors were visiting Florida State University.

CCS'14, November 3–7, 2014, Scottsdale, Arizona, USA.

ACM 978-1-4503-2957-6/14/11 ...$15.00.
http://dx.doi.org/10.1145/2660267.2660344 .

1. INTRODUCTION

Software fault isolation (SFI [43]) is a mechanism to effectively isolate untrusted modules in a host application. It creates a logically separated area called sandbox, or fault domain, in the host's address space, and strictly confines the untrusted module into this area so that it cannot directly access other domains' memory or execute their instructions (a domain is either the host or a sandbox). Data could only be exchanged via explicit cross-domain communications in a fashion similar to Remote Procedure Call [28]. SFI is particularly useful to isolate untrusted code such as third-party browser plugins downloaded from Internet, or benign code that handles untrusted and potentially malicious inputs. For example, many popular open-source libraries that have been embedded in hundreds of thousands of programs are found to contain vulnerabilities that may be exploited to compromise their host applications, such as libpng (CVE-2012-5470, CVE-2012-3425, etc.), libtiff (CVE-2013-4244, CVE-2013-4243, etc.), gzip (CVE-2012-1461, CVE-2012-1460, etc.), and many others [25]. By isolating these modules in separate domains, SFI can confine the damages and thus protect the host application and the operating system.

Since its introduction [43], there has been a long stream of research to improve or apply SFI to protect low-level software security [8, 9, 20, 21, 23, 33, 43, 46, 47]. For example, Native Client (NaCl) relies on SFI to safely run untrusted native plugins in a web browser, bringing performance and safety to browser plugins [33, 46]. Many of these systems focus on the x86 architecture, the most prevalent CPU architecture until recently. In the past few years, ARM has become the other dominating CPU architecture due to the huge popularity of ARM-based mobile devices. For example, Google has activated more than 750 million Android-based devices by March 2013 [3]. The vast majority of them are powered by the ARM processor. Moreover, ARM processors are increasingly being deployed in data centers because of the improved performance and superior power efficiency [4]. However, only a few systems have been proposed to enable SFI for the ARM architecture, including NaCl for ARM [33] and ARMor [49]. Both systems are based on binary rewriting: NaCl for ARM reserves the lower 1GB address space of the host application for the untrusted module, and instruments the module to confine its memory and instruction references to this range; ARMor, a "fully-verified" SFI for ARM, guarantees memory safety and control flow integrity [2] by inserting guards before potentially dangerous operations. Although it is easy to deploy since no kernel modification is required, binary rewriting could cause relatively high performance overhead or undesirable trade-offs (e.g., unchecked memory reads [21]). For example, ARMor has as high as 2× performance overhead for some computation-intensive tasks such as the string search.

In this paper, we propose ARMlock, a hardware-based fault isolation system for the ARM architecture. ARMlock leverages an often-overlooked hardware feature in the commodity ARM processors called *memory domain* to efficiently establish multiple sandboxes. Specifically, memory in a process can be assigned to one of the sixteen domains, whose access rights are determined by the domain access control register (DACR). ARMlock assigns different domain IDs to the host and the sandboxes, and updates DACR when entering or leaving a sandbox so that only the currently running domain is accessible. As such, instruction execution and memory references by the untrusted module are strictly locked down to its sandbox by the processor's memory management hardware. ARMlock imposes no limitations on the instructions that could be executed by an untrusted module. The untrusted module can also make its own system calls using the `svc` instruction. Those system calls are interposed by ARMlock to enforce the policies set by the host. Particularly, the host can instruct ARMlock to ignore system calls from the module or selectively allow a subset of them (e.g., to read from/write to an existing socket but not to create new ones).

By combining the hardware-based memory isolation and flexible system call interposition, ARMlock has the following three major advantages: *first*, instructions running inside the sandbox virtually have no performance overhead compared to those running outside it. ARM processors always have memory domain enabled. Running in a different domain does not affect the performance. *Second*, the host application and its untrusted modules usually are tightly coupled. Efficient domain switch is of vital importance for any SFI solution. Our experiment shows that ARMlock can perform more than 903,000 domain switches each second, or $1.1\mu s$ per domain switch, even on the low-end ARM processor of Raspberry Pi [29]. *Third*, by not restricting instruction layout or structure [21, 33, 46], ARMlock can readily support advanced features such as self-modifying code, just-in-time compiling (JIT), and exception delivery. Those features are difficult or even impossible to support in existing systems, but are useful nevertheless, for example, to isolate the JavaScript engine that uses JIT to compile frequently-used JavaScript code into native instructions.

Like any hardware-supported fault isolation systems, ARMlock requires certain kernel-level support to operate on the privileged hardware. To facilitate its deployment, ARMlock is designed as a kernel module (along with a user library) that can be loaded into the kernel on-demand. We have built a prototype of ARMlock for Linux. Its kernel module has less than 500 lines of source code. The main Linux kernel remains unchanged except for five lines of incompatible code we adjusted during our prototyping. As such, our prototype's addition to the trusted computing base is negligible. Our experiments show that ARMlock can effectively isolate vulnerable software components, and it's highly efficient with minimal performance overhead for the sandboxed code and fast domain switches.

The rest of this paper is organized as the following: we first describe the design and implementation of ARMlock in Section 2 and Section 3, then evaluate the system performance and effectiveness in Section 4. We discuss limitations and possible improvements in Section 5 and the related work in Section 6. Finally, we conclude the paper in Section 7.

2. SYSTEM DESIGN

2.1 Goals and Assumptions

ARMlock is a hardware-based fault isolation scheme for the ARM architecture. It is designed to securely isolate untrusted modules from the host application so that they can safely co-exist in a single address space. To achieve that, we have three design goals for ARMlock:

- *Strict isolation*: ARMlock needs to strictly isolate untrusted modules into their own domains. Specifically, memory references by an untrusted module, including *read*, write, and execution, can only target memory in the sandbox. Any attempt to escape from the sandbox should be prevented. Moreover, ARMlock does not limit instructions of the module, including the system call instruction. System calls made by the module should be interposed. Commodity OS kernels have a large attack surface that may impair the system security.

- *Performance*: our solution should incur negligible performance overhead when properly applied. Particularly, code running inside the sandbox should have no or minimal overhead when compared to that running natively. This prevents us from adopting binary rewriting because it will lead to constant performance overhead [9, 33, 49]. Moreover, untrusted modules are often tightly coupled with its host, and require frequent cross-domain communications. For example, tcpdump [40] has parsers for hundreds of network protocols. Many parsers were found to surfer from memory-based vulnerabilities (CVE-2007-3798, CVE-2005-1267, CVE-2005-1278, CVE-2005-1280 [25]). By sandboxing those parsers, we can significantly reduce tcpdump's attack surface. However, close interaction between the parsers and other parts of tcpdump requires ARMlock to minimize the domain switch overhead. We briefly considered to use page tables to isolate domains but vetoed it because switching page tables is an expensive operation as it may affect *TLB* (Translation Lookaside Buffer [41]) and *user-space cache* [38].

- *Compatibility*: our system should not impose restrictions on untrusted modules. Many existing SFI systems mandate memory layout and/or instruction structure of the modules [9, 21, 33, 46]. Such constraints may undermine its compatibility or harm the performance. For example, many SFI systems do not support self-modifying code and exception delivery. Moreover, to improve compatibility, ARMlock should be structured as a loadable kernel module, and avoid changing the base kernel, if at all possible.

Threat model: in this work, we assume a threat model similar to other SFI systems in which the kernel is trusted, and the host is benign but vulnerable. The goal is to protect the host application from the compromised or malicious modules by isolating them in separate domains. The host does not trust its modules, which could be vulnerable (e.g., protocol parsers in tcpdump) or simply malicious (e.g., browser plugins downloaded from Internet). Security of the host or the kernel themselves is a non-goal and considered out-of-scope. Various existing systems can be applied to enhance their security and be combined with ARMlock to provide defense in depth [16, 35].

2.2 Overall Design

In the rest of this section, we briefly introduce the background of ARM's virtual memory management and then give a high-level overview of the design of ARMlock, particularly the run-time environment for the sandbox.

ARMlock leverages the "memory domain" of the page table to create its sandboxes. 32-bit ARM processors support 2-level page tables. The first-level page table has 4096 entries, each mapping 1MB of memory for a total of 4GB address space. Each first-level

Type	Value	Description
No Access	00	No access permitted
Client	01	Permissions defined by page tables
Reserved	10	Reserved
Manager	11	No permissions check

Table 1: ARM domain access control

page table entry includes a 4-bit domain ID used as an index into the domain access control register (DACR). DACR is a 32-bit privileged register [1] and is only accessible in the privileged processor modes. It is divided into 16 two-bit fields, and each of which defines the access right for its associated domain. The four possible access rights for a domain are `No Access`, `Client`, `Reserved`, and `Manager` as shown in Table 1. That is, those fields allow us to (1) prohibit any access to the mapped memory – `No Access`, (2) ignore permission bits in the page table and allow unlimited access to the related memory – `Manager`, (3) or let the page table to determine the access right – `Client`. `Client` is the default setting. In Linux, domain 0, 1, and 2 are reserved for the kernel, user-space, and device memory, respectively. They all have the access right of `Client`. As such, ARMlock for Linux can simultaneously support 13 sandboxes for each process. Few applications require that many sandboxes, and more can be supported when necessary by adjusting the page table, as long as these sandboxes are not active at the same time. ARM's memory domain is particularly suitable for SFI because changes to DACR are *instantly* put into effect without affecting the TLB. In Section 5, we discuss how the x86 architecture might be extended to support the similar feature.

ARMlock allocates a disjoint block of memory for each sandbox and assigns to it a unique domain ID. ARMlock manages the DACR register so that only memory of the currently running domain is accessible. Any attempt to access other domains' memory will lead to a hardware domain fault and be trapped. Cross-domain communications such as function invocation and memory references need to be converted to remote procedure calls. Figure 1 shows the architecture of ARMlock. ARMlock has two collaborating components: a kernel module and a user library. The kernel module operates on the (privileged) hardware to create and manage domains and to facilitate inter-domain communication. It extends the kernel's system call interface to expose its services to applications. The user library wraps ARMlock's services in an easy-to-use API. The host application uses this library to set up sandboxes and perform inter-module calls. In ARMlock, untrusted modules are compiled as position-independent shared libraries so that they can be loaded at any suitable locations [1]. After loading the module into memory, the host requests the kernel to set up the sandbox for the module by specifying its base address, length, and domain ID. The kernel then goes through the host's page table and updates the domain ID of the entries associated with the sandbox. At run-time, the host and the sandbox use the library to execute inter-module calls with the help of the kernel.

2.3 Fault Domain

Each domain is allocated a block of memory with a unique domain ID. ARMlock manages the domain assignment and domain access control register (DACR) independently for each process: domain IDs are assigned in the page table. They are naturally updated when the page table is switched during context switch, and DACR is saved and restored in the thread control block during context

[1]They have to be loaded at addresses that are 1MB-aligned because only the first-level page table entries contain the domain ID field.

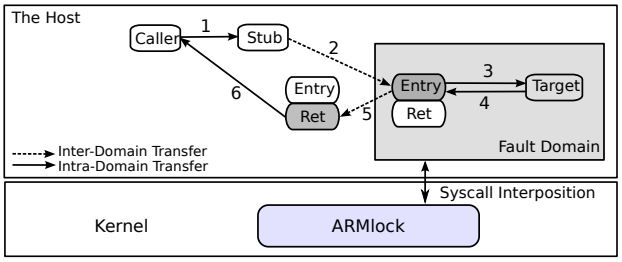

Figure 1: The architecture of ARMlock

switch. As such, each process is free to assign domains and control their access rights. No overhead will be incurred to processes that do not use ARMlock. At run-time, ARMlock updates DACR during domain switch so that only the currently running domain is accessible. Specifically, it sets the current domain (and the kernel [2]) to the `Client` access right and all other domains to `No Access`. Any attempt to access other domains will lead to a `domain fault` and be trapped by the hardware. The trap is then routed to the host application, which can respond by destroying the sandbox or terminating the whole process. Note that multithreading is naturally supported in ARMlock. First, each CPU core has its own DACR register. The domain access right is thus determined individually by each core. One core can run in the sandbox and another in the host application. Second, each thread has its own DACR setting that is saved in the thread control block. Switching thread will save the current thread's DACR and load the next thread's DACR. If a thread running in the sandbox preempts another one running in the host, the former cannot access the host memory as determined by its DACR.

Data can be legitimately exchanged across domains using the kernel-assisted memory copy or shared memory. In the former, the owner of the memory provides an unforgeable token [11] to the target domain, which then passes the token to the kernel along with a destination buffer address. The kernel verifies authenticity of the token and copies the memory on behalf of the target domain if verified. Specifically, the kernel temporarily makes both domains accessible in DACR and copies the memory into the destination buffer (there is no race condition in doing so because DACR is saved and restored during thread context switch.) After that, DACR is restored to the original value. In the latter, the memory shared by those domains bears a third domain ID. This domain is set to be accessible by both the host and the sandbox in DACR. In both cases, the host should treat data from a sandbox as untrusted and careful sanitize it. The application can choose which method to use for data exchange. Particularly, the shared memory method has better performance but may require more careful input sanitation.

Untrusted modules are compiled as the position-independent shared libraries. Function calls inside a domain proceed as usual without any performance penalty. Inter-module calls (line 2 and 5 in Figure 1) require help from the kernel because their target function lies in a different domain and is not directly accessible from the current domain. To this end, ARMlock installs in each domain an entry gate and a return gate. Those two gates are the only entry points into a domain. ARMlock also creates a stub for each imported external function. To make an inter-module call, the caller calls the stub (line 1 in Figure 1), which saves the current states and loads the target function address and the parameters into registers before calling the kernel for help (line 2). The kernel, instead of directly

[2]The kernel needs to be always available. It is protected from the user-space by the hardware (not memory domains).

jumping to the target function, dispatches it through the entry gate (line 3). When the remote function returns (line 4), the entry gate in the target domain asks the kernel to return back to the original domain (line 5), which then returns to the caller (line 6). In this way, the kernel only needs to know the entry and return gates of a domain but not any of the exported functions. Inter-module calls involve delicate operations on the program states. The complicity is handled for the programmers by the user library as described in Section 3.1.

ARMlock imposes no constraints on the instructions that can be executed by the untrusted module including system call instructions. Each sandbox has an associated in-kernel run-time environment managed by ARMlock to support system call and signal. System call is the interface for applications to request a wide range of kernel services. For example, recent versions of the Linux kernel have more than 330 system calls. Most of these system calls are not used by a particular application (or untrusted module) but remain accessible thus unnecessarily exposing a large attack surface. On the other hand, discreetly selected system calls can be safely allowed to simplify implementation as well as to improve performance. For example, an untrusted module can be allowed to receive and send data on existing sockets created by the host application, but not to create new ones. Each domain (including the host) has its own system call interposition settings. The ARMlock kernel module swaps this setting during domain switch.

The host and the sandboxed modules can specify their own signal handlers. ARMlock updates the signal handlers during domain switches. To deliver a signal, the kernel needs to manipulate the user-space memory (particularly the stack) and registers. A signal can only be delivered to the currently active domain because other domains are inaccessible. ARMlock allows only synchronous signals (i.e., signals caused directly by the execution of instructions such as `SIGSEGV`, the segment fault) to be delivered to a sandbox. Asynchronous signals currently are not allowed for sandboxes because the domain may have changed before the signal is delivered. For example, the `alarm` function in `libc` schedules the `SIGALRM` signal to be sent to the process in the future. The signal may be delivered to an unrelated domain (whichever domain is active at the time of signal delivery). In ARMlock, asynchronous signals can only be handled by the host. When running in a sandbox, those signals will be re-routed to the host for processing.

2.4 A Programming Model for Sandbox: Coroutine

To take full advantage of SFI systems (ARMlock included), programmers need to shift the programming paradigm to which modules with different privileges, origins, or security requirements are isolated from each other [45]. Isolating untrusted modules can significantly lower the risk of the host application being compromised. However, cross-domain access is rather unnatural and clumsy. Notice that in ARMlock, the stub issues a system call to perform an inter-module call (line 2 in Figure 1). This system call enters in one domain (the host) but emerges from another domain (a sandbox). This structure can naturally support a more intuitive programming model for sandboxing, namely coroutine, in which a set of routines can collaborate by voluntarily ceding CPU time to others. Coroutines are useful in solving many problems such as state machines, producer-consumer problems, and generators [6]. ARMlock provides built-in support for coroutines.

ARMlock introduces two primitives, `yield` and `resume`, to support coroutines. `Yield` explicitly gives CPU to the collaborating routine, while `resume` resumes execution of the previous routine. Figure 2 contains a simple program to demonstrate how to use

```
 1: procedure CONSUMER(fd)
 2:     PRODUCER(fd)
 3:     while (i = YIELD()) ≠ -1 do
 4:         USE(i)
 5:     end while
 6: end procedure
 7:
 8: procedure PRODUCER(fd) /*untrusted, sandboxed*/
 9:     READ_AND_DECRYPT(fd, buf)
10:     RESUME()
11:     for p = buf; p < end_of_buf; p++ do
12:         ch = DECODE_NEXT(buf, &p)
13:         RESUME(ch)
14:     end for
15:     RETURN(-1)
16: end procedure
```

Figure 2: Coroutine example

coroutines in ARMlock. In this example, the host consumes data generated by the `producer`, which uses a "complex" algorithm to parse the untrusted input file. `Consumer` cedes the CPU time to `producer` in line 3, and `producer` parses the next input item and returns it to `consumer` in line 13. The execution continues at line 3 as if `yield` has returned with the data passed to `resume` in line 13. With coroutines, the sandboxed code can be expressed in its natural logic. For example, `producer` can loop through the input and simply return each item implicitly using `resume`. Also, closely related states can be kept together (e.g., `buf`, `p`, and `end_of_buf` are all in the sandbox). An advantage of ARMlock is that coroutines can be implemented solely in the user library by leveraging the ARMlock kernel interface. No additional kernel modification is required to support `yield` and `resume`. The implementation details are described in Section 3.2.

The use of coroutines in ARMlock-protected programs is optional. Programmers can use either coroutines or the traditional function calls for the sandbox, whichever suits the problem better. In section 4, we give some examples of using coroutines to simplify refactoring of some existing applications.

3. IMPLEMENTATION

In this section, we give details about our prototype of ARMlock for Linux. The prototype supports the ARMv6 and ARMv7 sub-architectures due to their popularity and the support of memory domain. These two sub-architectures cover a wide variant of ARM processors from the low-end ARM11 processors in Raspberry Pi [29] to the powerful Cortex A9 and Cortex A15 processors that are popular in high-end mobile devices and a potential competitor in data centers [4].

3.1 ARMlock Kernel Module

ARMlock leverages the hardware feature to isolated untrusted modules, and thus requires the kernel privilege to change important hardware/software states (e.g., the page table). Our prototype has a kernel module and a user library that collaborate to provide SFI for applications. The kernel module has a compact design to avoid significant increase to the kernel's TCB (design goal 3) and to enable fast domain switch (design goal 2). Our prototype introduces less than 500 lines of code in the kernel, and the main kernel remains intact except for two minor changes (in 5 lines) to address incompatibility with ARMlock: first, there is a small section of kernel memory wrongfully set to the user domain (domain 2). Two lines

are changed in the kernel memory configuration table (`mem_types` in `arch/arm/mm/mmu.c`) to correct it. Second, the kernel considers every domain fault as a kernel exception and panics upon one (a reasonable design since no other components used memory domain before.) Three lines are added in `arch/arm/mm/fsr-2level.c` to deliver them to the application if they happen in the user mode.

ARMlock's kernel module is responsible for creating and switching domains. The host application can specify the settings for a fault domain, including the memory region and its domain ID, the address of the entry gate and the return gate, the initial stack (each domain has its own stack), as well as allowed system calls. Given those parameters, the kernel creates the sandbox by manipulating the host's page table. Specifically, it locates the first-level page table entries that map the memory of the sandbox, and sets their memory domain to the provided one. However, the main kernel is not aware of ARMlock. It might overwrite the domain ID when updating its page table, for example, to swap out and swap in pages. To address this problem, ARMlock registers a callback to the MMU notifier (`change_pte`) so that ARMlock will be notified whenever those page table entries have been changed. ARMlock can then recover their domain IDs. The host application also needs to provide its own entry and return gate addresses to the kernel so the untrusted module can remotely call the host's functions. Once the sandbox has been set up, the host issues an `ENABLE_ARMlock` command to the kernel that will prevent any further changes to the sandbox. That is, the sandbox cannot call the kernel to change itself.

The second responsibility of ARMlock's kernel module is to facilitate inter-module calls. To this end, it provides two commands, `ARMlock_CALL` for inter-module calls, and `ARMlock_RET` for inter-module returns. In both cases, ARMlock first modifies DACR to make the current domain inaccessible and the next domain accessible. It then updates the signal handlers and system call interposition. Finally, it manipulates the saved application states so that execution will continue at the next domain when the system call returns. Specifically, the kernel saves the user registers to the kernel stack (`struct pt_regs`) when entering the kernel. These saved registers can be located by ARMlock. To switch domains, ARMlock overwrites the saved stack pointer (`r13`) and program counter (`r15`) with those of the next domain. When kernel returns from the system call, it restores the saved registers and returns to the user-space. The execution continues at the restored program counter with the restored stack, and transits to the next domain. From the application's point of view, those two system calls enter from one domain but emerge from the other domain (instead of the original domain as normal system calls do). This provides the necessary structure to support coroutines. Moreover, the other registers are not modified by ARMlock. They can be used to pass parameters for inter-module calls. It is important for the current domain to clear unused registers to prevent cross-domain information leakage.

ARMlock's kernel module only recognizes the entry and return gates of a domain. It does not need to know any of the exported functions. Particularly, the `ARMlock_CALL` command always transits to the entry gate of the next domain, which further dispatches the execution to the target function. `ARMlock_RET` always transits to the return gate of the caller domain, which subsequently returns to the original caller. As such, the kernel is oblivious of how function calls are dispatched or returned. The application needs to maintain adequate states for inter-module calls. For example, the caller could push the return address (in `r14`) to the stack and load the target function address in one of the registers. The details of inter-domain calls are handled by the user library and are mostly transparent to the programmers (Section 3.2). The entry and the return gates are the only two entry points to a domain. This provides an effective location to control the exported functions. For example, the host application can create a list of functions callable by a sandbox, and check that only these functions are called by the sandbox at the entry gate. On the other hand, there is no need to perform access control at the return gate because the return address is always saved on the caller domain's own stack and thus cannot be changed by other domains.

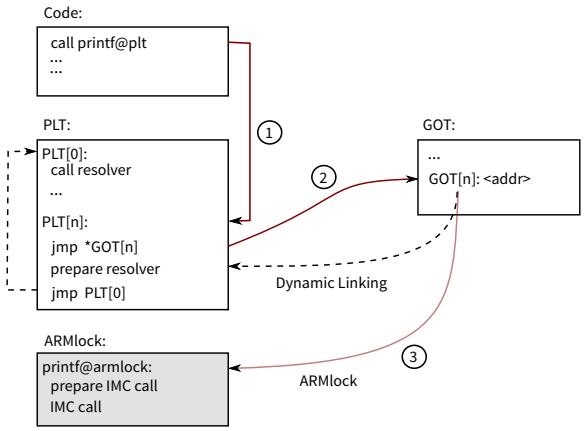

Figure 3: ARMlock external function dispatch

3.1.1 Signal/Exception Handling

Untrusted modules in ARMlock may cause exceptions such as the divided-by-zero fault and the illegal-instruction fault. Unlike many SFI systems, ARMlock allows exceptions (or synchronous signals) to be delivered to the sandbox. Each domain can assign its own set of signal handlers. When ARMlock changes domains, it also switches the signal handlers. To this end, ARMlock maintains a bitmap of interested signals for each domain. During domain switch, ARMlock restores their handlers to those of the next domain. Considering the fact that a signal is a relatively rare event for most applications, it could reduce the domain-switch latency if signal handlers are lazily updated, say, right before the signal is handled. However, this method is more intrusive and requires non-trivial modifications to the main kernel, an approach avoided by ARMlock in favor of easier deployment. In ARMlock, only synchronous signals can be delivered to the sandbox. Asynchronous signals are handled by the host itself. Nevertheless, an asynchronous signal might happen when the CPU is executing in the sandbox. To address that, ARMlock registers a signal handler for those events on behalf of the sandbox. The handler simply forwards the signal to the host by calling the host's handler.

The way a signal is handled in ARMlock is also worth mentioning. To deliver the signal to a thread, the kernel allocates a signal frame (`struct sigframe`) on the thread's user stack. Signal frame consists of the user registers (`struct pt_regs`, used to resume the thread upon signal return) and a piece of the executable `retcode`. `Retcode` is responsible for cleaning up the signal frame and resuming the interrupted thread through a `sigreturn` system call (`arch/arm/kernel/signal.c`). Because signal handling is highly kernel-specific, the thread does not know how to clean up the signal frame by itself and should instead rely on `retcode` for this purpose. `Retcode` is usually synthesized by the kernel, either on the user stack or in the memory shared by the kernel and the user-space such as VDSO [42]. In either case, we need to make sure the synthesized code is executable by the sandbox. In our prototype, the Linux kernel actually does not use `retcode` on the

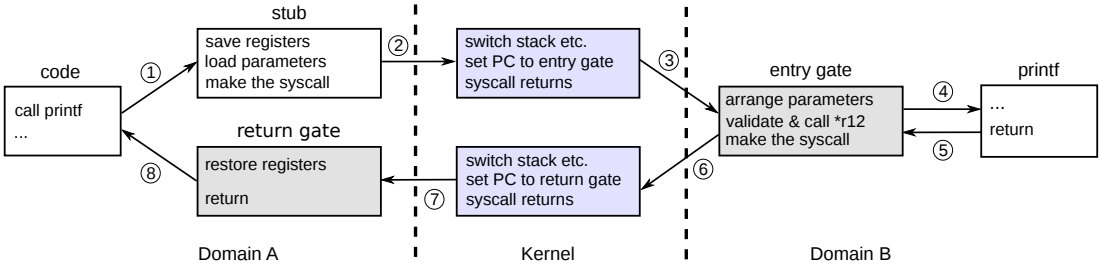

Figure 4: ARMlock inter-module call

stack because the stack is non-executable (but `retcode` is created nevertheless). Instead, it uses a copy of `retcode` in the kernel (`KERN_SIGRETURN_CODE`). The kernel's memory has a domain ID of 0 with the access right of `Client`, thus its access right is determined by the page table. ARMlock sets the page table to make `retcode` executed by both the host and the sandbox. Finally, the kernel dispatches the signal by manipulating the saved user registers so that execution will continue at the signal handler and "return" to the correct `retcode` when the handler returns, similar to the way ARMlock switches domains.

3.1.2 System Call Interposition

An untrusted module in ARMlock can make its own system calls for convenience and performance. However, the kernel's system call interface is dangerously wide, and less-exercised system calls are often a source of kernel exploits. It is thus necessary to intercept and regulate system calls by the untrusted module. Our prototype relies on the `Seccomp-BPF` framework [32] in the Linux kernel for system call interposition. `Seccomp-BPF` is an extension to `Seccomp` [31] that was designed to securely run third-party applications by disallowing all system calls except for read and write of already-open files. `Seccomp-BPF` generalizes `Seccomp` by accepting a `BPF` program [22] to filter system calls and their parameters. In ARMlock, the host can assign a `Seccomp` program for the sandbox. By default, it disallows all system calls except those for the ARMlock kernel module. However, the host can permit other system calls when necessary. ARMlock delegates the host to configure system call interposition for the sandbox. The host should exercise discretion in this task. Particularly, it should not permit any unnecessary system calls and pay close attention to the memory-related system calls such as `mprotect`, `mmap`, and `brk`. Currently, ARMlock does not support `fork`, `exec` and other related system calls inside a sandbox. A process can be forked by the host, followed by re-initialization of the sandbox. This limitation can be lifted should more intrusive changes to the base kernel were allowed.

At run-time, the kernel needs to change the `Seccomp` program during domain switch. In Linux, each task (`struct task_struct` in `sched.h`) has its own program container (`struct seccomp_filter`). The container is organized into a tree structure. Specifically, there is a `prev` pointer in `seccomp_filter` that points to the parent process's program. When a system call needs to be filtered, the kernel executes all the programs along this reverse linked list. In other words, a parent's program is inherited by the child processes. To interpose the sandbox's system calls, ARMlock creates a new `seccomp_filter` structure for the sandbox (it also inherits the parent's program). When switching domains, ARMlock only needs to overwrite the task's program container with the pointer to this new container (`current->seccomp.filter`). Switching domains will not affect the program of the host's children because they inherit the host's program, not the sandbox's.

3.2 ARMlock Fault Domain

ARMlock has a kernel module and a user library. The kernel module is responsible for creating domains and facilitating inter-module calls, while the user library provides an easy-to-use programming interface for applications. In this rest of this section, we present details of the fault domain, particularly the user library.

In ARMlock, the untrusted module needs to be compiled as a position-independent shared library (e.g., ELF dynamic shared object (DSO) in Linux [7]) so that it can be loaded at a proper location (1MB aligned). Each module maintains a list of imported and exported functions. This defines the interface between domains and only functions in this list can be called by other domains using inter-module calls. For each imported function, ARMlock synthesizes a stub to facilitate the inter-module call. ARMlock leverages the structure of position-independent shared libraries to seamlessly integrate those stubs into a domain. Specifically, the compiler creates a structure called `PLT/GOT` [26] for each external function (e.g., `printf`) whose address has to be resolved at run-time by the loader and linker. As shown in Figure 3, `PLT` is a short sequence of code that represents `printf` in the `DSO`. Calls to `printf` will be directed to this `PLT` entry, which contains an indirect jump to the instruction address in its associated `GOT` entry. Initially, the `GOT` entry points back to the `PLT` to prepare and call the dynamic resolver. The first call to `printf` therefore will resolve the address of `printf` and update the `GOT` entry (line 1, 2, and dotted lines in Figure 3). Subsequent calls will be directly dispatched to the actual function. ARMlock leverages `PLT/GOT` by eagerly calling the resolver to locate the external function, and replacing the `GOT` entry with the address of its stub (e.g., `printf@armlock`, line 3). Consequently, calls to external functions will be conveniently replaced by inter-module calls. In addition, if a call-back function is passed to a domain (e.g., a `compar` function to `qsort`, the quick sort function), a stub needs to be created in that domain to invoke the call-back function with inter-module call.

Figure 4 shows the sequence of an inter-module call. ARMlock creates an entry gate and a return gate for each domain, and synthesizes a stub for every imported function. Calls to an imported function will be dispatched to its stub (line 1 in Figure 4). The stub handles the low-level details of inter-module call. It saves the state of the current domain onto the stack and loads the parameters into the registers. Particularly, it stores the target function address into one of the registers ($r12$ in our prototype). It then makes a system call to the ARMlock kernel module (line 2). The kernel changes the active domain by updating the thread states such as DACR, saved registers, and signal handlers (Section 3.1). It then sets the saved instruction pointer ($r15$) to the entry gate of the target domain. Therefore, execution will resume in the target domain when the system call returns (line 3). The entry gate first validates that the target function (in $r12$) is exported to the calling domain and calls it if the check is passed (line 4). Memory-based parameters need

to be copied over with the help of the kernel, if not shared. When the function returns (line 5), the entry gate makes another system call to the kernel (line 6) which "returns" to the return gate of the calling domain (line 7). The return gate subsequently restores the saved program states and returns to the original call site (line 8). Therefore, each inter-module call requires two system calls. As shown in our experiment, these system calls are very light-weight and close to a null system call (e.g., `getpid`).

ARMlock's user library encapsulates and hides the complicity of inter-module calls. The programmer only needs to provide a list of imported functions, from which the library will automatically generate the corresponding stubs. Even though we rely on PLT/GOT to place those stubs, there is no need to customize the compiler because it already supports PLT/GOT, a required structure for dynamic shared libraries. In our current prototype, the programmer needs to manually call some library functions to marshal/unmarshal memory-based parameters. This can be easily automated with the tools for remote procedure call [28].

In Section 2.4, we re-introduced the coroutine programming paradigm to support sandboxes. Coroutines can be straight-forwardly implemented in ARMlock using return gates: `yield` and `resume` save the registers of the current domain on the stack, and then issue an `ARMlock_RET` system call to jump to the target domain's return gate. The return gate restores the registers of the target domain and resumes its execution. For example, in Figure 2, consumer first calls `producer` to initialize it (line 2). After initialization (line 9), producer calls `resume` to save its states to the stack and return to consumer (line 10). When consumer later `yields` to producer asking for more data (line 3), the return gate of producer will restore previously saved registers and resume execution at line 11.

Compiler optimization may also complicate the implementation (and debugging) of sandboxes. Particularly, modern compilers support intrinsic functions, also called `builtins`, that are implemented by the compiler and available for use (implicitly) by the user program. Note that some intrinsic functions are not inlined in the code, and could be located in different domains from the code that invokes them. For example, `gcc` provides built-in functions for division on the ARM platform if the processor does not have native division instructions. The compiler may also silently replace some instruction sequences with a built-in function during optimization (e.g., the `memset` intrinsic). However, only a single copy of built-in functions will be loaded and linked into a process. When an intrinsic function is called in a sandbox, it will lead to a domain fault because the function lies in the host domain. To address that, we need to load and link to a copy of built-in functions in each domain or provide our own equivalent implementation.

4. EVALUATION

In this section, we first systematically analyze the security guarantee provided by ARMlock to demonstrate its effectiveness, and then evaluate the performance overhead incurred by ARMlock.

4.1 Security Analysis

Similar to other SFI systems, we assume a threat model where the kernel, including the ARMlock kernel module, is trusted and the host application is benign but may contain exploitable vulnerabilities. The host needs to safely execute some untrusted modules such as benign code that handles untrusted inputs or code of a questionable origin. Therefore, the code in the sandbox could have been compromised and is potentially malicious. The goal of ARMlock is to securely and efficiently isolate the untrusted module from the host. Figure 5 shows the interaction between the components of a protected application. We will use this figure to illustrate the attacks against ARMlock and the defenses built into our system.

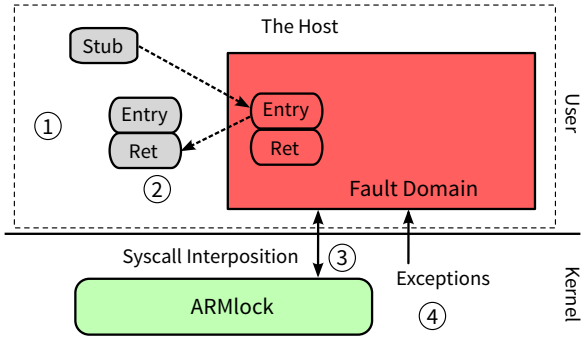

Figure 5: Threat model of ARMlock

Direct memory access: since the sandbox is a part of the host's address space, an attacker may try to directly access the host memory (#1 in Figure 5). This attempt will be intercepted by the CPU as a *domain fault* because ARMlock sets the host's memory inaccessible when running inside the sandbox. The kernel delivers the exception to the handler of the sandbox. ARMlock registers a handler for the sandbox that forwards domain fault and other selected signals to the host. This might be disrupted by the untrusted module since we assume it can compromise everything inside the sandbox (it is feasible to secure the handler using techniques similar to the Chrome sandbox for Linux [12]. We choose to have a simpler attack model.) Even so, the attempt to access the host memory will be foiled. Some kernel memory may also be accessible to the untrusted module such as VDSO [42] used by Linux to speed up some system calls. This will not pose a new threat to the kernel because the kernel always assumes applications are untrusted and protects itself from them.

Inter-domain communication: the attacker may also target the inter-domain communication, including inter-module calls, cross-domain memory copy, and shared memory (#2 in Figure 5). First, the attacker may try to call a dangerous function in the host (e.g., `system()` in `libc`) or pass malicious parameters to an exported function. ARMlock maintains a list of the exported functions that is checked against by the entry gate. As such, only these functions may be called by the untrusted code. They should always treat parameters from the sandbox as malicious and carefully sanitize them. The host risks being compromised should some security checks be neglected. To mitigate this threat, the programmer could refine the exported functions for a narrower attack surface, and provide defense-in-depth by interposing the host's syscalls [32] or using a capability-based system [45]. The return gate in the host cannot be misused by the attacker to manipulate control flows because the return address is saved on the host's stack and thus cannot be changed by other domains. The return value, if used, should also be sanitized.

Cross-domain memory copy requires the assistance of the ARMlock kernel module. The host passes an unforgeable token [11] to the kernel and only then the sandbox can issue a memory copy command. Therefore, the source domain has full control over the cross-domain memory copy. During memory copy, there is a short window of time in which both the host and the sandbox are accessible. However, there is no race condition that would allow the attacker to access the host memory: DACR (domain access control register) is saved and restored during the context switch in the thread control blocks. If another thread is scheduled to interrupt

Item	Configuration
CPU	ARM1176JZF-S 700 MHz
Memory	512MB
OS	Raspbian (based on Debian)
Kernel	Linux 3.6.11
LMbench	version 2
nbench	version 2.2.3

Table 2: Configuration of the experiments

the memory copy, DACR will be restored to its value, rendering the host memory inaccessible. Moreover, each CPU core has its own independent DACR. A thread running simultaneously on another CPU core cannot access the host memory unless that core's DACR legitimately allows it.

The host and the sandbox may also use shared memory to communicate. This may be exploited by the attacker, say, through race conditions. For example, the attacker may modify the shared memory when it is being accessed by the host if the host is multi-threaded. To mitigate it, the host should copy data from the shared memory before processing it. If large blocks of memory are frequently exchanged, ARMlock can be extended to support swapping the ownership of the shared memory so that only one domain can access it at a time.

System call interposition and exceptions: ARMlock imposes no limitations on the instructions of the sandboxed code. It can issue system calls or cause exceptions (#3 and #4 in Figure 5). A wide open system call interface is detrimental for security because many system calls are less exercised and have a higher chance to contain vulnerabilities. In ARMlock, the host defines the allowed system calls and their parameters for an untrusted module. Similar to a firewall policy, the host should by default deny all system calls and only allow necessary ones. For example, our porting of tcpdump puts its protocol parsers in the sandbox and gives them access to the open socket. The parsers can read that socket but cannot create new ones. This minimalist policy helps to avoid many pitfalls in the system call interposition for whole applications [10]. Moreover, exceptions can be delivered to the sandbox. ARMlock leverages the existing signal delivery system in the kernel, one of the most mature features in the kernel. As mentioned earlier, the forwarding of selected signals to the host may be disrupted by the untrusted code, a denial-of-service threat. However, such behavior can be easily detected by ARMlock, for example, by maintaining a counter of domain faults in the kernel and the host and comparing them for differences.

Case studies: to demonstrate the effectiveness of ARMlock, we examine vulnerabilities in some popular programs and show how ARMlock can help to confine damages. The first program we examined is `gzip`. A recent version of gzip (1.2.4) contains a buffer overflow vulnerability that can be triggered when an input file name is longer than 1,020 bytes [13]. This bug may be exploited remotely if gzip is running by a server such as the Apache web server. To confine this vulnerability, we use ARMlock to isolate the functions that handle the untrusted command line input (in `getopt.c`). Even though the bug can still be exploited, the damage is confined to the sandbox and cannot spread to the host application or the whole system. Functions like `getopt` can be easily converted to coroutines because they closely follow the producer/consumer model. Moreover, gzip contains several algorithms to compress/decompress untrusted data. They should also be isolated.

The second program (library) we examined is `libpng`, the official `PNG` reference library [18]. Despite having been "extensively tested for over 18 years", new security vulnerabilities are still being discovered in the library with 27 CVE reports (the most recent one was reported in 2012). We can use ARMlock to sandbox the library together with the `zlib` it depends on. However, this will lead to many inter-module calls because `libpng` has a fairly fine-grained interface. To address that, we provide a simple wrapper function around `libpng` with a higher level interface (e.g., decode this `png` file into the buffer we provided), and sandbox `libpng` along with the wrapper. Moreover, `libpng` uses `setjmp/longjmp` to handle errors, a feature that can naturally be supported by ARMlock without any special handling.

The last program we examined is *tcpdump* [40]. The current version of tcpdump contains 124 protocol parsers. Vulnerabilities have been reported for multiple protocol parsers in tcpdump such as BGP, GRE and LDP. These parsers can be easily isolated in ARMlock and prevented from compromising the host.

4.2 Performance Evaluation

The performance of ARMlock can be characterized by the domain switch latency and the execution speed of code in the sandbox. Normally, the host and its untrusted modules are tightly coupled and have close interactions that require inter-domain communications. It is therefore critical for ARMlock to keep the domain switch latency as low as possible. Because each inter-module call requires two system calls to ARMlock's kernel module, this latency is bounded by twice the time of a *null* system call (e.g., `getpid`). Moreover, it is ideal for instructions running inside a sandbox to execute as fast as those outside the sandbox. There is a trade-off between the domain switch latency and the execution speed of the sandboxed code in the design of SFI systems. For example, SFI systems that rely on binary rewriting may have lower domain switch latency but constantly suffer from the performance overhead when running inside the sandbox. The way a program is organized can also affect its performance. Domain switch has a (mostly) constant latency, it is beneficial to structure the program so that the frequency of domain switches is minimized, for example, by defining the interface at a higher semantic level. Our performance evaluation accordingly consists of two sets of experiments: micro-benchmarks that measure the domain switch latency, and macro-benchmarks that test the performance of sandboxed programs. Our experiments are based on Raspberry Pi, a popular and affordable single board computer. It has a low-end ARMv6 processor and 512MB memory [29]. Table 2 lists the configuration of our experiment environment.

To measure the domain switch latency, we implemented a simple `inc` function in the sandbox which returns its parameter increased by one. We call this function from the host for 10 million times and measure the elapsed time with the `clock` libc function. The result shows that even on this relatively low performance ARM processor, ARMlock can perform 903,342 inter-module calls every second. That is, each inter-module call takes about $1.1\mu s$. Therefore, each ARMlock system call takes about $0.55\mu s$ because every inter-module call consists of two domain switches (thus two system calls). To compare it with common system calls, we also measured the latency of `getpid` [3] and several other system calls of the original Linux kernel with `LMbench` [19]. Figure 6 shows the latency of these system calls relative to `getpid` (in log scale). In particular, an inter-module call in ARMlock takes about 2.6 times of `getpid` or the `null` system call in LMbench.

Our second set of experiments measures the performance overhead for instructions running inside the sandbox. Specifically, we

[3] We use `syscall (__NR_getpid)` to make sure the kernel is entered to avoid the impact of VDSO [42].

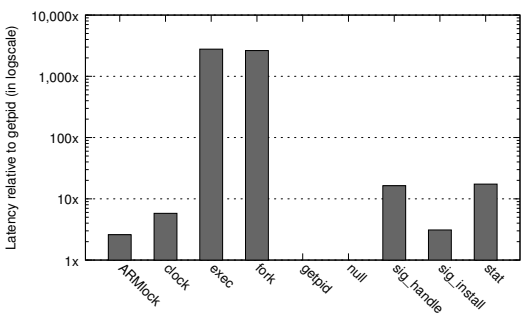

Figure 6: ARMlock domain switch latency relative to that of `getpid`

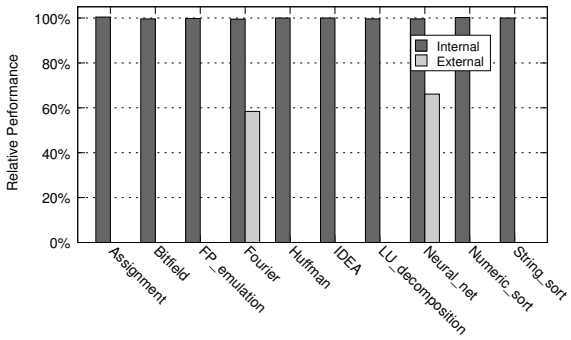

Figure 7: Relative performance of nbench in ARMlock

run `nbench` [24] – a computation-intensive benchmark of CPU, FPU, and memory system – both inside the sandbox and natively, and then compared their relative performance. The result is shown in Figure 7 (marked as internal). It is clear that in ARMlock instructions running inside the sandbox is as fast as outside the sandbox. Notice that this experiment represents the ideal scenario for ARMlock because the sandboxed code is almost self-contained and thus cross-domain communication is not frequent. To further measure the performance under frequent inter-module calls, we modify two benchmarks in `nbench`, `Fourier` and `Neural Net`, to use floating-point functions in the host. Both benchmarks rely heavily on floating-point functions such as `pow`, `sin`, and `cos`. This setting represents almost a worst-case scenario for ARMlock because of the frequent inter-module calls. In both cases, the performance drops to about 62% of the native `nbench`. Moving those floating point functions back into the sandbox makes the performance almost the same as the native `nbench`. In general, it will benefit performance (sometimes significantly) to move closely related functions into the sandbox together. This observation also applies to the three programs in our case studies (Section 4.1). None of them exhibited more than 1% performance overhead. For example, our modified `tcpdump` reads packets from an open socket and parses them in the sandbox. It has virtually no overhead.

5. DISCUSSION

In this section, we discuss issues related to ARMlock and software fault isolation in general. We also suggest possible future work for ARMlock.

Architecture support for fault isolation: ARMlock is a hardware-based fault isolation for the ARM architecture. It leverages the memory domain feature in ARM to confine memory access. The memory domain feature has clear advantage in implementing SFI over previous hardware features, namely segmentation [9, 46] and paging [36, 39]. With segmentation, an application can create multiple segments that each specifies the valid base and length of a memory region. It can further use an instruction prefix to specify which data or code segment the instruction refers to. This creates a number of issues. Particularly, the untrusted code needs to be instrumented to prevent it from changing segments and/or jumping over the segment override prefix to refer to the unintended segment. Self-modifying code and dynamically-loaded code are also challenging to support: the system needs to have the capability to instrument code at run-time. Meanwhile, page-table based sandboxes suffer from high domain switch overhead because TLB and cache may be affected. Likely, it could not support applications that require frequent inter-module calls well. In contrast, memory domain in ARM fits better than segmentation and paging. Specifically, it assigns to each top-level page table a domain ID, an index into the domain access control register (DACR). This can be efficiently integrated into the processor implementation: each TLB entry can be extended to include a domain ID field. When there is a memory access, the processor uses the domain ID to retrieve the access right from DACR. This can be handled in parallel to the memory access. Although memory domain is not available on x86, we speculate that recent x86 CPUs could be extended to support it with some minor modifications. Intel introduced a feature called page attribute table (PAT) in Pentium III to allow the kernel to specify cacheability of virtual memory [15]. PAT has a similar structure as memory domain in ARM: a special register (`IA32_PAT MSR`) contains eight page attribute fields to specify a memory type such as `uncacheable`, `write-combined`, `write-through`, `write-back`, `reserved`, etc. A page table entry has three bits (the `PAT`, `PCD`, and `PWT` bits) that index into this register to determine the cacheability of its mapped memory. However, attempting to write the `reserved` memory type into the `IA32_PAT` register will trigger a general-protection fault [15]. To enable the memory domain like support, we need to allow `reserved` to be written into `IA32_PAT` and delay the fault until the memory of this type is accessed. After this change, we could use a similar design to implement ARMlock on the x86 architecture.

Software support for fault isolation: Hardware assisted fault isolation often requires software support at both the kernel and the user levels. ARMlock's kernel module is more or less an afterthought "patch" to the existing kernel in favor of compatibility. A cleaner design could introduce domains into the kernel's process management. A domain is a group of resources (similar to the name space in Linux but at a lower level). Each domain should have its own memory, system call table, open file table, signal delivery etc. A master and slave relation can also be established among domains. A master domain can control a slave domain's resource allocation and override its system call table. The kernel may also provide a hook for SFI to customize domain switch. Without doubt, this cleaner design requires more intrusive and extensive changes to the kernel and may not be practical.

A SFI system should facilitate the adoption of its programming paradigm. ARMlock leverages the `PLT/GOT` structure to shield the programmer from the details of inter-module calls. Inevitably, the programmer still needs to refactor the application into domains, or compartments [45]. This task can be aided with software development tools to suggest source code partitioning and wrap remote calls in RPC. Moreover, the system should provide common libraries ported to the sandbox. Domain switch is a waste of CPU time because it does not perform useful computation. Porting these libraries to the sandbox may significantly reduce the overhead (Section 4). Projects such as NaCl have ported many libraries to their

sandbox and may be reused by other SFI systems. In addition, the dynamic loader and linker could be made SFI-aware so that the binary of these libraries can be shared by all the sandboxes in the system to reduce the memory footprint.

Many sub-architectures of ARM: unlike the stable x86 architecture, the ARM architecture has a large number of slightly incompatible sub-architectures each with different features, power consumption, and performance. Some of them might not support memory domain required by ARMlock. Moreover, the ARM architecture has two profiles. Profile A targets full-fledged applications with virtual memory support; profile M is designed for low-performance and lower-power embedded devices which do not need paging. In addition, recent extensions in high-end ARM processors add a new page table format, i.e., the long format. This format removes the memory domain support despite the fact that there are enough unused bits to accommodate a domain ID. However, these processors are backward compatible with ARMlock because memory domain is still supported by the short page-table format. In the case that the short format is also deprecated, ARMlock can switch to page table based isolation. Even though ARMlock is not completely future proof for high-end ARM processors, it will remain useful and functional for low-to-medium-end processors, for example, to run untrusted plugins in smart devices (e.g., smart watch) or Internet of Things.

Kernel-level Sandbox: ARMlock is a user-level sandbox. It relies on memory domain for isolation. Memory domain is also available in the kernel space. It requires the following changes for ARMlock to become a kernel-level sandbox. *First*, the code running in the sandbox, even though confined, still has the kernel-level privilege. It is thus imperative to control the instructions of untrusted modules. For example, they should not be allowed to change DACR or even disable paging. This consequently will limit the support of self-modifying code or JIT (those features probably are not as useful for a kernel-level sandbox anyway). We can design a verifier to prove that a binary is safe to run in the kernel sandbox [46]. Second, the domain switch mechanism needs to be changed as well. ARMlock depends on the kernel to switch domains. The kernel is always accessible in ARMlock. This is not the case for a kernel-level sandbox since only part of the kernel is accessible at any time. We need to design a software-based domain switch mechanism, potentially with the help of some form of control-flow integrity [44, 46]. Third, the interrupt handling has to be adapted so that interrupt handlers are accessible in both the main kernel and the sandbox. In addition, the verifier should be applied to those handlers to prevent them from being misused. Interrupts happened in the sandbox are rerouted to the main kernel for processing.

Close-sourced OS support: our prototype targets Linux, arguably the most popular OS for the ARM processors. However, the design of ARMlock is neutral to the underlying OS. Particularly, it might be possible to adapt ARMlock to the Windows RT operating system. Close-sourced OSes pose extra challenges: although ARMlock can be implemented as a loadable kernel module or device driver, there might exist incompatibility in the main kernel such as the exception handling that we adjusted for Linux. To address this, the kernel could be dynamically patched to resolve the incompatibility. Different releases of the kernel may require different patches, leading to higher maintenance efforts.

6. RELATED WORK

SFI is designed to isolate untrusted modules in dedicated sandboxes to prevent them from accessing other domains' memory and escaping from the confinement. To this end, SFI systems often statically or dynamically rewrite the untrusted code to insert inline reference monitors to control the memory accessible. However, because memory reads are usually far more frequent than writes, some of these systems check only memory writes, rendering them unsuitable for applications such as untrusted native browser plugins (e.g., the attacker could search the browser memory for bank accounts.) The initial SFI system [43] and PittSFIeld [21] reserve dedicated register(s) for the generated inline monitors. This approach has its limitation on the 32-bit `x86` architecture due to its very limited number of general registers. In contrast, NaCl for `x86` [46] and VX32 [9] leverage the hardware segmentation of `x86` to confine memory accesses (segmentation has mostly been removed from the 64-bit `x86` architecture.) ARM does not have the similar segmentation support. ARMlock instead leverages the memory domain feature to sandbox both memory read and write. Binary rewriting based SFI inserts inline reference monitors to confine memory accesses. It is important to prevent these monitors from being bypassed by the untrusted code. This requires some level of control flow integrity [2] so that the untrusted code cannot jump over the monitor to execute unconfined memory access instructions. To this end, PittSFIeld and NaCl [33, 46] implement a chunk-based (coarse-grained) control flow integrity. A chunk contains the monitor and its related memory access instructions. Control transfer instructions can only target the beginning of a code chunk. As such, those monitors cannot be bypassed. VX32 is based on the dynamic binary translation and has a different strategy: it intercepts the real jump target at run-time and forbid any attempt to bypass the monitors. ARMlock does not need to ensure control flow integrity of the sandboxed code because the processor guarantees that it cannot access memory outside its domain. In fact, ARMlock imposes no constraints on the untrusted module. For example, the module can use self-modifying code or just-in-time compiling if necessary.

ARMor [49] and NaCl for ARM [33] are two closely-related systems. ARMor is a recent system to provide SFI for ARM applications. It leverages Diablo [27], a link time binary rewriting framework, to insert inline monitors into the untrusted binary. A monitor needs to be inserted for each memory access and control transfer instruction. It thus has a high performance overhead (almost 2x slow-down for computation-intensive tasks). Even though its performance could be optimized [48], ARMlock has much smaller performance overhead, particularly for the sandboxed code. NaCl for ARM [33] uses a customized compiler to mask out high bits of memory addresses and jump targets so that the accessible memory is limited to the lower 1GB. This limits the application of NaCl for ARM to a single sandbox with a fixed address space. Our system does not have such limitation and can also support advanced features such as JIT. Another closely related system is TLR [30], which leverages ARM TrustZone to build a trusted language runtime for mobile applications. TLR can significantly reduce the TCB of an open-source `.NET` implementation. Compared to TLR, ARMlock is a generic SFI system based on the light-weight memory domain support in ARM. Moreover, TrustZone provides a secure world in addition to a normal world. It's a better fit to isolate different applications (as demonstrated in the TLR system [30]) or OSes than to isolate modules in an application. For example, it might be difficult for TrustZone to support signals.

Since its introduction [43], SFI has been adopted by many systems for different purposes. Robusta [37] uses SFI to isolate native code of the Java virtual machine so that vulnerabilities in the native code cannot compromise the Java VM or the system. Program Shepherding [17] relies on dynamic binary translation to monitor control flow transfers at run-time and enforce security policies.

There is also a long stream of research to use SFI or similar technologies to secure device drivers. For example, VINO [34], SFI-Minix [14], BGI [5], XFI [8] and LXFI [20] leverage SFI to isolate kernel extensions (or device drivers) from the main kernel. ARMlock is a fault isolation system for user-space applications. Currently, it cannot be used to isolate kernel mode code yet. We leave it as a future work to extend our system for this purpose.

At last, ARMlock leverages a feature in the page table to split the address space into several domains. Page table has often been used for security purposes. For example, Nooks [39] maintains a copy of kernel page table for device drivers which only grants read access to the kernel memory so that misbehaving device drivers cannot directly change kernel data. SIM [36] protects the active in-VM monitor from the untrusted kernel in a separated address space enforced by the hypervisor. HyperLock [44] is a system that isolates the KVM hypervisor from the host kernel. It allocates a separate page table for KVM that can only be changed by KVM via explicit requests. A compromised KVM thus cannot modify the host OS memory or corrupt the whole system. ARMlock does not use separate page tables for untrusted modules. It instead relies on a more efficient hardware feature.

7. SUMMARY

In this paper, we have presented the design and implementation of ARMlock, a hardware-based fault isolation system for the ARM architecture. ARMlock uniquely leverages the memory domain feature in the commodity ARM processors to create multiple sandboxes for untrusted modules. We have implemented a prototype of ARMlock on Linux for the ARMv6 and ARMv7 sub-architectures. Our evaluation shows that the isolation provided by ARMlock is strong and efficient. For example, ARMlock can effectively isolate both memory write and memory read, and code running inside a sandbox executes as fast as that outside it. Moreover, ARMlock supports advanced features that many other SFI systems cannot or have difficult to support, such as self-modifying code, just-in-time compiling, and exception delivery.

Acknowledgements We would like to thank the anonymous reviewers for their comments that greatly helped to improve the presentation of this paper. This work was supported in part by the First Year Assistant Professor award of Florida State University. The first author of this paper was partially supported by the National Science Foundation of China under Grant No.61340031. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the FSU and NSFC.

8. REFERENCES

[1] Domain Access Control Register. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0434b/CIHBCBFE.html`.

[2] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity: Principles, Implementations, and Applications. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, November 2005.

[3] Update from the CEO. `http://googleblog.blogspot.co.uk/2013/03/update-from-ceo.html`.

[4] Calxeda. `http://www.calxeda.com/`.

[5] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast Byte-Granularity Software Fault Isolation. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2009.

[6] M. E. Conway. Design of a Separable Transition-Diagram Compiler. In *Communications of the ACM*, 1963.

[7] Linux Foundation Referenced Specification. `http://refspecs.linuxbase.org/`.

[8] U. Erlingsson, S. Valley, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software Guards for System Address Spaces. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, November 2006.

[9] B. Ford and R. Cox. Vx32: Lightweight User-level Sandboxing on the x86. In *Proceedings of 2008 USENIX Annual Technical Conference*, June 2008.

[10] T. Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *Proceedings of the 20th Annual Network and Distributed Systems Security Symposium*, February 2003.

[11] W. K. Giloi and P. Behr. An IPC Protocol and its Hardware Realization for a High-speed Distributed Multicomputer System. In *Proceedings of the 8th annual symposium on Computer Architecture*, 1981.

[12] Linux and Chrome OS Sandboxing. `https://code.google.com/p/chromium/wiki/LinuxSandboxing`.

[13] gzip. The gzip home page. `http://www.gzip.org/`.

[14] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Fault Isolation for Device Drivers. In *Proceedings of the 39th IEEE/IFIP International Conference on Dependable Systems and Networks*, 2009.

[15] Intel. *Intel 64 and IA-32 Architectures Software Developerś Manual Volume 3: System Programming Guide, Part 1 and Part 2*, 2010.

[16] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering Code-Injection Attacks with Instruction-Set Randomization. In *Proceedings of the 10th ACM conference on Computer and communications security*, CCS '03, October 2003.

[17] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution Via Program Shepherding. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.

[18] libpng. libpng home page. `http://libpng.org/pub/png/libpng.html`.

[19] LMbench - Tools for Performance Analysis. `http://www.bitmover.com/lmbench/lmbench.html`.

[20] Y. Mao, H. Chen, D. Zhou, , X. Wang, N. Zeldovich, and M. F. Kaashoek. Software fault isolation with API integrity and multi-principal modules. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, October 2011.

[21] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *Proceedings of the 15th conference on USENIX Security Symposium*, July 2006.

[22] S. McCanne and V. Jacobson. The BSD Packet Filter:A New Architecture for User-level Packet Capture. In *Proceedings of the 1993 USENIX conference*, 1993.

[23] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan. RockSalt: Better, Faster, Stronger SFI for the x86. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, june 2012.

[24] Linux/Unix nbench. `http://www.tux.org/~mayer/linux/bmark.html`.

[25] National Vulnerability Databasel. `http://nvd.nist.gov`.

[26] PLT and GOT - the Key to Code Sharing and Dynamic Libraries. `https://www.technovelty.org/linux/plt-`

and-got-the-key-to-code-sharing-and-dynamic-libraries.html.

[27] L. V. Put, D. Chanet, B. D. Bus, B. D. Sutter, and K. D. Bosschere. DIABLO: a Reliable, Retargetable and Extensible Link-time Rewriting Framework. In *Proceedings of the 2005 IEEE International Symposium On Signal Processing And Information Technolog*, 2005.

[28] Remote Procedure Call. http://en.wikipedia.org/wiki/Remote_procedure_call.

[29] Raspberry Pi, an ARM/GNU Linux Box for $25. http://www.raspberrypi.org/.

[30] N. Santos, H. Raj, S. Saroiu, and A. Wolman. Using ARM Trustzone to Build a Trusted Language Runtime for Mobile Applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, March 2014.

[31] seccomp. http://lwn.net/Articles/332974/.

[32] Yet another new approach to seccomp. http://lwn.net/Articles/475043/.

[33] D. Sehr, R. M. Karl, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *Proceedings of the 19th USENIX Security Symposium*, August 2010.

[34] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the USENIX 2nd Symposium on OS Design and Implementation*, 1996.

[35] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the Effectiveness of Address-Space Randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, CCS '04, October 2004.

[36] M. Sharif, W. Lee, W. Cui, and A. Lanzi. Secure In-VM Monitoring Using Hardware Virtualization. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, November 2009.

[37] J. Siefers, G. Tan, and G. Morrisett. Robusta: Taming the Native Beast of the JVM. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, 2010.

[38] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering Device Drivers. December 2004.

[39] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proceedings of the 19th ACM symposium on Operating Systems Principles*, October 2003.

[40] Tcpdump/Libpcap. http://www.tcpdump.org.

[41] Translation Lookaside Buffer. http://en.wikipedia.org/wiki/Translation_lookaside_buffer.

[42] On vsyscalls and the vDSO. http://lwn.net/Articles/446528/.

[43] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-based Fault Isolation. In *Proceedings of the 14th ACM Symposium On Operating System Principles*, December 1993.

[44] Z. Wang, C. Wu, M. Grace, and X. Jiang. Isolating Commodity Hosted Hypervisors with HyperLock. In *Proceedings of the 7th ACM SIGOPS EuroSys Conference*, 2012.

[45] R. N. Watson and J. Anderson. Capsicum: Practical Capabilities for UNIX. In *Proceedings of the 2010 USENIX Annual Technical Conference*, June 2010.

[46] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Orm, S. Okasaka, N. Narula, N. Fullagar, and G. Inc. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, May 2009.

[47] B. Zeng, G. Tan, and G. Morrisett. Combining Control-flow Integrity and Static Analysis for Efficient and Validated Data Sandboxing. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, October 2011.

[48] B. Zeng, G. Tan, and G. Morrisett. Combining Control-Flow Integrity and Static Analysis for Efficient and Validated Data Sandboxing. In *Proceedings of the 18th ACM Conference on Computer and Communication Security*, 2011.

[49] L. Zhao, G. Li, B. De Sutter, and J. Regehr. ARMor: Fully Verified Software Fault Isolation. In *Proceedings of the ninth ACM international conference on Embedded software*, EMSOFT '11, October 2011.