Multi-Domain SFI

Nicholas Boucher
Harvard University

April 28, 2019

1 Introduction

Software Fault Isolation, or SFI, is a set of well-defined, assembly-language-specific application binary restrictions which provide certain security guarantees [1]. Specifically, SFI allows a single virtual address space to be partitioned into two domains such that each domain can only access memory within itself or via specific trampolines to the opposite domain. Multi-Domain SFI is a generalization of SFI that allows for up to `O(n)` unique isolation domains for any n-bit virtual address space.

1.1 Motivation

The motivation for this work is a desire to add arbitrary process-like code and memory isolation on platforms that do not support classical *nix-stlye processes. Specifically, the development of Unikernels – small binaries that contain both compiled application code and the minimum number of operating system components needed to run that application code [2] – has demonstrated the need for multi-domain SFI.

One common criticism of Unikernels is the difficulty in debugging live systems. When something causes a deployed Unikernel to crash, the lack of an underlying operating system means that it can be difficult to examine the state of the program at the time of its failure. While remote debugging exists for a number of major Unikernel systems (such as a remote GDB stub for the popular `IncludeOS` [3] Unikernel[1]), remote debugging capabilities are not guaranteed to be online if the application error in any way modified the in-memory machine instructions or runtime memory utilized by the remote debugging stub.

While it would seem that traditional SFI would offer the ability to isolate the remote debugging region of virtual memory from Unikernel application code, further investigation revealed that the two domain limitation of traditional SFI is inadequate for solving this problem. This is illustrated in the C++ standard library. In the case of `IncludeOS`, the standard library is referened *both* by the remote debugging stub and Unikernel application code. Sharing this segment of memory between the two different SFI isolation domains is non-ideal as it requires placing trust in both isolation domains not to corrupt the standard library code for the other domain. As such, a third isolation domain with appropriate

[1]https://www.includeos.org/blog/2018/debugging.html

trampolines into the required portions of the standard library would allow decreasing the amount of trust to the standard library not corrupting itself, which is significantly more ideal. This context proved to be the motivation for multi-domain SFI.

1.2 Document Layout

Following the introduction and motivation for multi-domain SFI, this paper is divided into three substantive sections.

The first section focuses on the theoretical model upon which multi-domain SFI is built. This section provides deep detail on the specific compiler and assembly-level tools which multi-domain SFI utilizes. The second section describes the methods that developers use to annotate code which is to be compiled with multi-domain SFI. This section focuses on a C++ specific code annotation paradigm. The third section presents a practical example walkthrough of multi-domain SFI using a simple toy program.

This paper concludes with a high-level summary of multi-domain SFI and notes related to ongoing work.

2 Theoretical Design

The purpose of multi-domain SFI is to partition the address space into $d \in D$ isolation domains such that each domain is restricted in performing potentially dangerous memory operations across domains. Specifically, any domain is allowed to write and jump to targets within itself. Write operations, however, are not permitted across domains. Jumps are permitted across domains only to targets to which the invoking domain has explicitly been given access permission, and the targets of these jumps must be direct (i.e. statically verifiable targets). Reads are not restricted, as they do not pose a threat in corrupting memory across domains. A domain shall be formally defined as a compile-time specified, contiguous region of memory in the virtual address space to which a set of developer specified, immutable memory operation permissions are assigned. These memory permissions are manifested as bitmasks and are described in section 2.2.

Multi-domain SFI uses two tools to constrain memory access: memory alignment and bitmasks preceding indirect jumps. These two tools directly affect the assembly code constructed at compile time, and conformance to the alignment and bitmasking specifications can be verified statically for any generated binary.

Each of the two tools employed by multi-domain SFI targets a different aspect of constraining memory access; required bitmasks are used to ensure that the target of any indirect jump is a region of memory which the invoker is allowed to access, while forced memory alignment ensures that the destinations of any jump are indeed valid starting memory addresses for a full machine instruction.

The remainder of this section will address the application of memory alignment and bitmasks in depth, and then outline the limitations of the multi-domain SFI system.

2.1 Memory Alignment

Like classic SFI [4]–[6], multi-domain SFI enforces 32-byte memory alignment for all basic blocks of machine code. Phrased differently, the virtual memory address of any possible jump target must be aligned to a 32-byte memory region. In practice, this means that the least significant 4 bits of the virtual memory address for the target of any jump must all be zeros.

Of course, most machine languages (such as x86) do not use fixed sized instructions, and instruction sizes tend to be far less than 32-bytes. This means that any basic block of code which is not 32-byte aligned by chance must be padded with `NOOP` instructions until the block is properly aligned. As with traditional SFI, which uses the same mechanism, this incurs some amount of binary size overhead as emitted by the compiler and memory overhead at runtime.

The reason for required memory alignment is subtle yet critical to proper SFI domain isolation. If we did not require the targets of any jumps to be memory aligned in some fashion, then any piece of code could jump to any virtual memory address *even if* the target address was not the intended start of a machine instruction. A clever adversary may be able to select a virtual memory address that represents the middle of a machine instruction which, misaligned, represents a different instruction [7]. By chaining together these instruction "gadgets," the adversary may be able to jump to any region of memory in the virtual address space *regardless* of any bitmasks prefacing intended jump instructions.

By forcing alignment for jump targets (for which we followed the convention of classic SFI in selecting 32-bytes), we can construct a compiler which enforces that no machine instruction crosses a 32-byte boundary. Through this, we can prevent the previously described fractional instruction attack with no runtime overhead.

2.2 Bitmasks

The more difficult challenge in isolating regions of memory is establishing a mechanism which properly restricts memory access within a given SFI isolation domain. In general, there are three classes of memory operations: reads, writes, and jumps. For all memory operations, we will define two values: *invoking domain* and *target domain*. These values are formally defined as:

- **Invoking Domain**: The SFI isolation domain for which a given machine instruction is located within the virtual address space of a muti-domain SFI compiled application.
- **Target Domain**: The SFI isolation domain for which any read, write, or jump machine operation targets. It may be the same or different than the invoking domain. The term target domain is only defined for machine instructions which reference exactly one virtual address within the instruction's arguments.

For any memory operation for which there is a well defined target domain, we further define two categories of operations: *direct memory operations* and *indirect memory operations*. Every memory operation for which there is a well defined target domain must be either a direct or indirect memory operation. We will explicitly define these terms as:

- **Direct Memory Operation**: Direct memory operations are machine instructions whose target memory argument is given as an explicit, hard-coded value. An example in x86 is `JMP 0x0804cfc0`.
- **Indirect Memory Operation**: Indirect memory operations are machine instructions whose target memory argument is given as a value that is not resolvable at compile time, such as a register. An example in x86 is `JMP EAX`.

Restricting memory access takes different forms for direct and indirect memory operations. The explicit tools used in both contexts are defined in the following sections.

2.2.1 Restricting Direct Memory Operations

Restricting memory access to direct memory operations is a trivially simple task when performed at compile time, and does not differ from the mechanism used by classic SFI. By definition, the target address is known at compile time since the target address is emitted directly by the compiler. Since the target domain is derived from the target address, the compiler can also easily identify the target domain for any direct memory operation.

As such, we shall define a compiler restraint for multi-domain SFI compilers that the compiler must check all potentially dangerous direct memory operations – writes and jumps – before emitting compiled binaries to ensure that every direct memory operation has a target address that is within a target domain to which the invoking domain has been granted access by the developer. If the compiler detects a memory operation that is not permitted and cannot be resolved to a permissible operation, the compiler shalt halt the compilation process and output a compiler error highlighting the offending source code.

Through this compiler design, we can guarantee that all direct memory operations successfully emitted from a correctly implemented multi-domain SFI compiler will indeed be valid memory operations. Note that if desired, this can be statically verified by examining any binary emitted by by a multi-domain SFI compiler [5], since all relevant information is contained statically within the generated binary.

2.2.2 Restricting Indirect Memory Operations

Restricting indirect memory operations requires a more significant effort than direct memory operations.

The key insight in multi-domain SFI is that for every isolation domain $d_i \in D$, we define a corresponding immutable bitmask m_i which completely defines the regions of memory upon which d_i is permitted to perform memory operations.

Bitmasks are constructed as follows: each isolation domain d_i will be associated with a tag t_i. No two isolation domains can have the same tag. A tag is a unique virtual memory address which is a power of 2, thus having a single 1 in its binary representation at bit b. An isolation domain will begin at the memory address specified by its tag and contain the entire virtual address range whose address is larger than t_i and smaller the first address with bit $b+1$ equal to 1. We will further define the bitmask generator G to be the following value for 32-bit address spaces:

$$G \triangleq (\neg \prod_{i=0}^{n}(t_i)) \mid 0xFFFFFFF0 \tag{1}$$

where $\mid$ represents logical OR, $\prod$ represent the logical OR of all elements in the specified set, and $\neg$ represents the bitflip operation. That is, G is defined at the value where all the corresponding bits b_i for each t_i are set to zero, the lowest 4 bits are set to zero (for 32-byte alignment), and all other bits are set to 1. Finally, the bitmask m_i for region d_i is defined as:

$$m_i \triangleq (\prod_{j \in \texttt{access}(d_i)} t_j) \mid G \tag{2}$$

where $\texttt{access}(d_i)$ returns the set of indices j for the domains $d_j \in D$ that domain d_i should be allowed to access.

Bitmasks are used to ensure that indirect jumps have targets that are within an allowed isolation domain. This is enforced by requiring that the registers containing the targets of all indirect memory operations from invoking domain d_i are logically ANDed with bitmask$_{d_i}$ in the instruction immediately preceding the indirect memory operation.

For example, the following generic indirect jump assembly instruction:

```
JMP EAX
```

Would be emitted by the multi-domain SFI compiler as:

```
AND EAX, bitmask_d_i
JMP EAX
```

for where bitmask_d_i is the bitmask as defined in Eq. 2. for the invoking domain.

2.2.3 Restricting Return Addresses

Function returns present a subtle and challenging problem. Normally, return addresses are placed on the stack by a function caller. Subsequent return instructions then jump to the runtime-specified return address on the stack. However, we cannot apply a standard bitmask before the indirect jump taken within a return statement, as we cannot be sure at compile time which SFI domains are valid targets for any given application of the function (since we cannot know the caller before execution).

One simple solution is to have the function caller place a bitmask that, when applied via a logical AND, would restrict the target domain to a valid region of memory as with standard indirect jumps. This necessitates getting rid of all RET instructions and replacing them with indirect jumps. Since the called function could arbitrarily modify the return address placed on the stack, however, we cannot use this solution nor any implementation that places trust on the integrity of the addresses on the stack.

Instead, we will chose to implement returns as a direct jump to a hard-coded assembly function called VerifyReturn. This function verifies the return address and performs an indirect jump to the verified address.

Verifying the identity of the function caller and associating a correct bitmask within `VerifyReturn` becomes the crux of the problem. We propose that the bitmask for the function-calling domain d_i implicitly passed by the caller on the stack in addition to the return address via a modification to standard function calling conventions within the compiler implementation. However, just as a function could maliciously modify its return address, so too could it maliciously modify its bitmask. Thus, we propose that the bitmask be encrypted using a low-round Feistel cipher. Furthermore, the lowest 4 bits of the bitmask, which are normally 0, will be replaced with a canary value fixed at program initialization and known at runtime only to `VerifyReturn`. Therefore, even if an adversary could supply a return address that decrypted as a useful bitmask, that bitmask would only be accepted if the lowest four bits were the correct canary values.

The behavior of `VerifyReturn` is as follows: it first decrypts bitmask on the stack using the low-round Feistel cipher. It then verifies that the lowest four bits of the plaintext are equal to the canary value. If the values are equal, it clears the lowest four bits and applies and logical `AND`s the bitmask with the return address on the stack. If the resulting value is equal to the starting return address, we know that the return address is within a valid region of memory and `VerifyReturn` simply invokes a `RET` statement. If any of these tests fail, `VerifyReturn` suspends program execution and fails to return.

Note that this same functionality could be implemented with each SFI domain having a secret key which it passes on every function call with the return address instead of the Feistel ciphertext. However, we believe that a low-round Feistel network decrypter and plaintext verifier can be implemented more efficiently than querying a data structure containing secret key values for target verification. Indeed, benchmarking on a 2018 MacBook Pro indicated that a 2-round Feistel network was approximately 20% faster than C++'s `std::map::find` implementation.

2.3 Restricting Inter-domain Jumps

2.4 Standard Isolation Domains

There are certain isolation domains that must be present in every program: at least one code segment, a stack/heap, and `VerifyReturn`.

Code segments will have developer-specified permissions, which are described in a subsequent section of this paper. `VerifyReturn`, by construction, must be accessible from all domains. Furthermore, this implementation of muli-domain SFI uses a single stack and heap space for all code isolation domains, so the domain containing these regions must also be accessible from all other domains.

To implement this efficiently, we will always place `VerifyReturn` and the stack/heap in the lowest domain of the virtual address space (where it's t_i=`0x0`) such that all domains automatically have access to these areas and we do need to consume a tag bit for this purpose.

2.5 Limitations

Because each isolation domain requires a tag with a uniquely positive binary bit, the number of isolation domains is limited by the size of the virtual address space. The maximum number of domains is the number of addressable bits minus the amount of overhead. Overhead is defined as the number of bits required for alignment and to fit the contents of the largest domain within one contiguous memory region.

3 Language Implementation

To gain the benefits of multi-domain SFI, developers must annotate their source code in such a way as to indicate how the address space is to be partitioned into isolation domains. In this section, we present a method of annotation for C++ which is consumed by multi-domain SFI C++ compilers to generate multi-domain SFI complaint code.

In C++, SFI domains are defined using namespaces. Formally, isolation domains are constructed as C++ namespaces named `sfi_X` where X is the name used to refer to the isolation domain throughout the annotated code. For example, an sfi domain named `foo` would be created by placing the isolation-desired code in a namespace titled `sfi_foo`.

The global namespace, such as where the `main` function must be placed, will be labeled as the isolation domain `sfi_std`. Thus, to make a function within an isolation domain available to be accessed by `main`, the function should be annotated with `export(std)`.

Import macros must also be exported to specific domains. Exporting an import makes all functions within that import available to the target domain. If imports are not annotated and are in the global namespace, they will be made available only to `sfi_std`. Otherwise, imports should be annotated just as regular functions, such as the following example which makes C++'s `stdio` available to isolation domain `foo`:

```
#export(foo)
#import <stdio.h>
```

3.1 Silent Domain Generation & Trampolines

Functions which are to be made available to other isolation domains are decorated with a C++ Macro that contains the name of the zone(s) which should be able to access the decorated function. This macro is called `export`. Specifically, if a function `myFunc` in isolation domain `foo` was to be made available to isolation domains `bar` and `baz`, `myFunc` would be immediately preceded by the macro `export(bar, baz)`. Naturally, all isolation domains are allowed to access themselves without any additional annotation.

Under the hood, the compiler will silently generate a new isolation domain for every utilized permutation of domain permissions created through `export` annotations. For example, when domain `foo` exported function `myFunc` to isolation domains `bar` and `baz` in the above example, a new isolation domain `foo-export-bar-baz` would be silently generated. This isolation domain would contain only a *trampoline* function to `myFunc`. A trampoline function is defined as a standard function emitted by the compiler using the same function calling conventions as the rest of the program in which the function does nothing other than calling the target function with the same arguments that it was passed.

The compiler will output as few isolation domains as necessary to achieve the security goals annotated by the programmer. In that spirit, silently generated isolation domains will be combined when possible. For example, if `foo` exported two different functions both to `bar` and `baz`, then only one `foo-export-bar-baz` domain is created and the trampoline functions for both are placed in this single silently generated domain.

4 Practical Example

This section gives an example of a toy program written using multi-domain SFI. It is broken down into a view of the toy program in C++ source code and a view of the compiled application's virtual address space during runtime.

4.1 C++ View

Let us consider the following C++ code, which is a simple program that outputs:

```
Hello, World.
Goodbye.
```

```cpp
#export(foo, bar)
#include <stdio.h>

namespace sfi_foo {
        void hello() {
                printf("Hello ");
        }

        void world() {
                printf("World.\n");
        }

        #export(bar)
        void helloWorld() {
                hello();
                world();
        }
}

namespace sfi_bar {
        void goodbye() {
                printf("Goodbye.\n");
        }

        #export(std)
        void greeting() {
                sfi_foo::helloWorld();
                goodbye();
        }
}

int main() {
        sfi_bar::greeting();
        return 0;
}
```

4.2 Memory Layout View

The following memory layout is the in-memory representation of the `TEXT` portion of the virtual address space for the compiled binary of the above C++ code. Memory addresses indicate the lower bound of the adress range for the adjacent memory section.

Note that `NOP`s will likely have to be placed after the code within each section in order to maintain the alignment required for the multi-domain SFI memory layout.

4.2.1 Bitmasks View

As defined in Eq. 2, bitmasks are created by logically `OR`ing the tags of regions accessible by domain d_i together with the value G. Applying the definition of G from Eq. 1 yields a value of

$$\texttt{0x03fffff0} = \texttt{0b00000011111111111111111111110000}$$

With the value of G, we can easily calculate the bitmasks for each region from the information in the annotated code. Figure 2 shows all isolation domains with their corresponding bitmasks rather than their tags.

4.3 Stack View

Since multi-domain SFI modifies standard calling conventions, here we present a sample view of the stack during execution of `helloWorld()`.

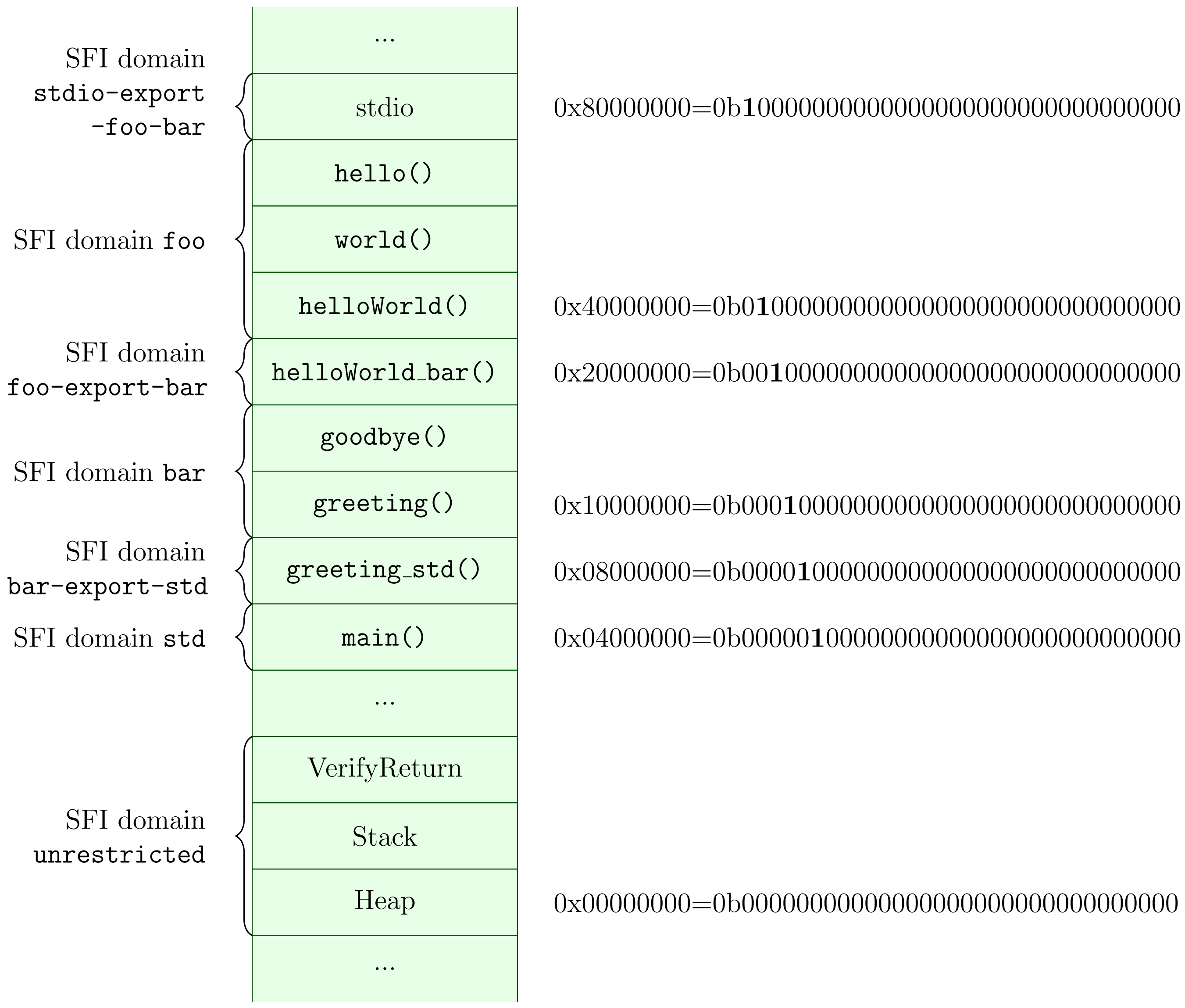
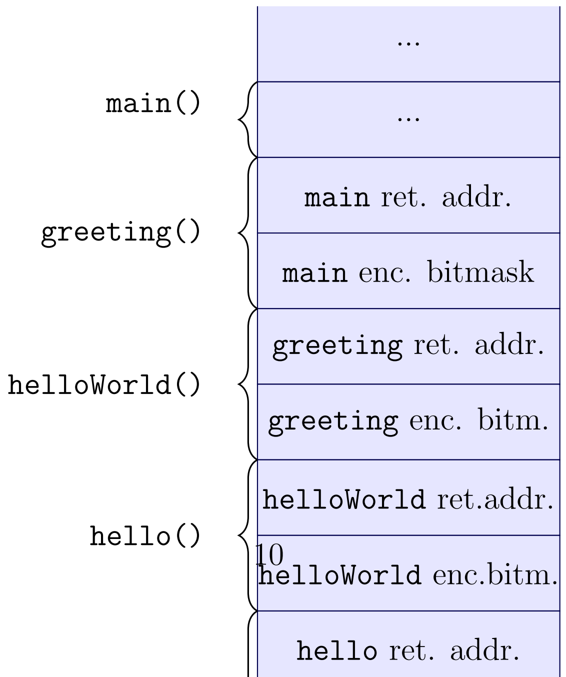

...
SFI domain stdio-export -foo-bar
stdio
0x80000000=0b10000000000000000000000000000000
SFI domain foo
hello()
world()
helloWorld()
0x40000000=0b01000000000000000000000000000000
SFI domain foo-export-bar
helloWorld_bar()
0x20000000=0b00100000000000000000000000000000
SFI domain bar
goodbye()
greeting()
0x10000000=0b00010000000000000000000000000000
SFI domain bar-export-std
greeting_std()
0x08000000=0b00001000000000000000000000000000
SFI domain std
main()
0x04000000=0b00000100000000000000000000000000
...
SFI domain unrestricted
VerifyReturn
Stack
Heap
0x00000000=0b00000000000000000000000000000000
...

Figure 1: Tag values t_i, equivalent to starting memory addresses, for each isolation domain



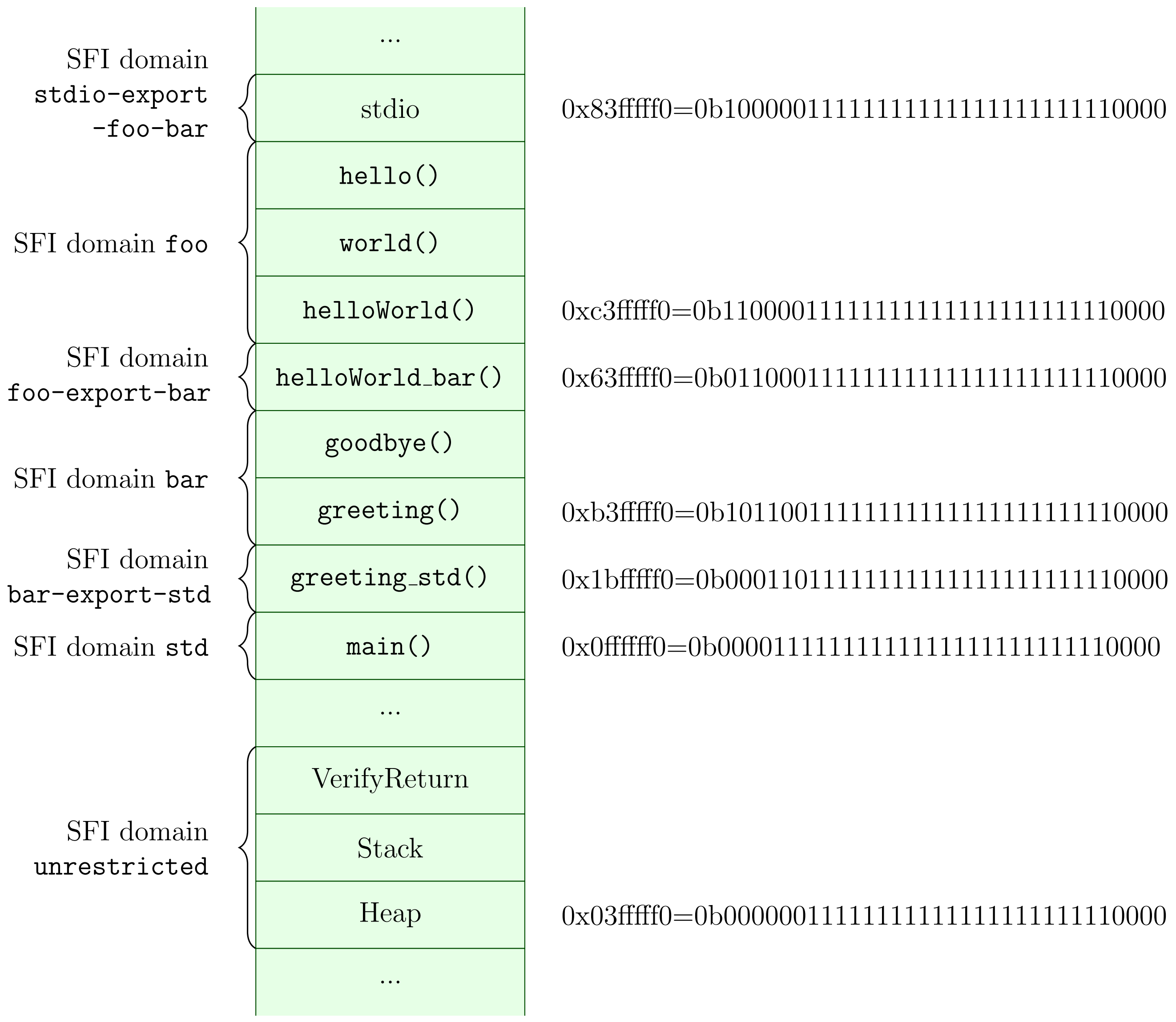

...
SFI domain stdio-export-foo-bar
stdio
0x83fffff0=0b10000011111111111111111111110000
SFI domain foo
hello()
world()
helloWorld()
0xc3fffff0=0b11000011111111111111111111110000
SFI domain foo-export-bar
helloWorld_bar()
0x63fffff0=0b01100011111111111111111111110000
SFI domain bar
goodbye()
greeting()
0xb3fffff0=0b10110011111111111111111111110000
SFI domain bar-export-std
greeting_std()
0x1bfffff0=0b00011011111111111111111111110000
SFI domain std
main()
0x0ffffff0=0b00001111111111111111111111110000
...
SFI domain unrestricted
VerifyReturn
Stack
Heap
0x03fffff0=0b00000011111111111111111111110000
...

Figure 2: Bitmask values for each isolation domain

5 Summary

The following section represents a high-level summary of multi-domain SFI as a brief summary of some of the key points of this paper.

- Every namespace that begins with `sfi` is an SFI isolation domain
  - More formally, an SFI zone named `foo` will be denoted `sfi_foo`
- Functions which are to be made available to other isolation domains are decorated with a C++ Macro that contains the name of the zone(s) which should be able to access the decorated function

    - More formally, `#export(foo,bar)` makes the decorated function available to both of the isolation domains foo and bar
    - At compile time, trampoline functions to these decorated functions will be placed into separate isolation domain(s) that are able to access the isolation domain containing the decorated function

- At compile time, direct jumps (e.g. `JMP addr`, `CALL addr`) within an isolation domain are allowed

- At compile time, indirect jumps (e.g. `JMP reg`) must be masked by a bitmask which represents the specific domains the current domain has access to

- The function calling convention is universally modified such that a 1-word key is silently passed as the first argument to every function call
    - Keys are created by taking the desired bitmask for the current isolation zone, replacing the lowest 4 bits with a secret canary, and encrypting using a Feistel cipher with a low number of rounds (e.g. n=2 or n=4) and a secret key randomly generated during program initialization.
    - Functions are informed of their keys using the same mechanism that stack canaries are distributed

- `RET` commands are universally disallowed in the generated x86 output except for a small, special assembly function that handles returns
    - Where the compiler would normally output a return statement, it instead outputs a JMP statement to the special return assembly function
    - This special function then decrypts the first function argument on the stack using the Feistel cipher, verifies the plaintext secret canary, and then masks the return address on the stack with the plaintext (having zeroed out the verified canary).
    - If the canary is not correct, halt execution or throw some kind of error

- All compiler-generated assembly will follow SFI-defined alignment specifications

6 Notes

- In both the case of the Feistel Cipher and randomly generated keys, it remains how ASLR would affect the compiler's ability to generate relevant bitmasks.

- It is likely that binaries will need to be statically linked in order for this implementation to work in the wild. This is because the SFI implementation modifies standard assembly calling conventions. This is easily implemented within standalone, statically-linked binaries, but is more difficult for dynamically loaded shared libraries. If dynamically loaded shared libraries were to be used, they would need to be versions of the standard library compiled to support the modified calling conventions.

References

[1] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," *SIGOPS Oper. Syst. Rev.*, vol. 27, no. 5, pp. 203–216, Dec. 1993, ISSN: 0163-5980. DOI: `10.1145/173668.168635`. [Online]. Available: `http://doi.acm.org/10.1145/173668.168635`.

[2] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library operating systems for the cloud," *SIGPLAN Not.*, vol. 48, no. 4, pp. 461–472, Mar. 2013, ISSN: 0362-1340. DOI: `10.1145/2499368.2451167`. [Online]. Available: `http://doi.acm.org.ezp-prod1.hul.harvard.edu/10.1145/2499368.2451167`.

[3] A. Bratterud, A. Walla, H. Haugerud, P. E. Engelstad, and K. Begnum, "IncludeOS: A minimal, resource efficient unikernel for cloud services," in *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, Dec. 30, 2015, pp. 250–257. DOI: `10.1109/CloudCom.2015.89`.

[4] S. McCamant and G. Morrisett, "Evaluating SFI for a CISC architecture," in *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, ser. USENIX-SS'06, event-place: Vancouver, B.C., Canada, Berkeley, CA, USA: USENIX Association, 2006. [Online]. Available: `http://dl.acm.org/citation.cfm?id=1267336.1267351`.

[5] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," in *2009 30th IEEE Symposium on Security and Privacy*, May 2009, pp. 79–93. DOI: `10.1109/SP.2009.25`.

[6] D. Sehr, R. Muth, 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 Conference on Security*, ser. USENIX Security'10, event-place: Washington, DC, Berkeley, CA, USA: USENIX Association, 2010, pp. 1–1, ISBN: 888-7-6666-5555-4. [Online]. Available: `http://dl.acm.org/citation.cfm?id=1929820.1929822`.

[7] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 15, no. 1, 2:1–2:34, Mar. 2012, ISSN: 1094-9224. DOI: `10.1145/2133375.2133377`. [Online]. Available: `http://doi.acm.org/10.1145/2133375.2133377`.