Multi-Domain SFI

Nicholas Boucher
Harvard University

April 18, 2019

1 Design

- 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

1.1 Notes

- This model does indeed propose moving away from the single trampoline model. The reasoning behind this was that assuming an arbitrary number of isolation domains where each domain is either defined as accessible or non-accessible to every other domain, having a single trampoline function (acting as something of a central dispatch to all inter-domain accesses) would require that function to both verify the identity of the caller and then check that it has permission to access the requested target. While both of these problems are quite solvable, adding some sort of caller verification + permission lookup (perhaps by hashtable or multi-dimensional array) seemed like a lot of overhead for what I believe will be a common operation. As such, the new model strives to remove the need for a central-dispatch-esque trampoline function and instead require in more traditional SFI style that every indirect jump is appropriately masked (as output by the compiler and optionally verifiable through static binary analysis). Here, permissions are being encoded using these masks instead of in another data structure verified by a trampoline function. This only leaves the question of verifying return addresses, which I believe is addressed in the next questions.

  The trampoline functions as referenced in "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" is a proposed way to keep some functions "hidden" within an isolation domain. For instance, if sfi_domian_0 contains functions foo and bar, but only wants to expose foo to sfi_domain_1 and keep everything else non-exposed, foo and bar are placed in an isolation zone (let's call it SFI_0) by the compiler and a new isolation zone (let's call it SFI_2) is generated by the compiler which contains only a trampoline function for foo (since that's the only function that was to be exposed). In this situation, SFI_1 (the code from sfi_domain_1) is able to access SFI_2 but not SFI_0 directly.

- Key verification — referencing the Feistel cipher keys — would be exclusively used for verifying RETs, not for function calls. Per the above explanation, function calls/jumps would be required to have bitmasks before each indirect call/jump. These bitmasks represent the access permissions of the calling isolation domain and could be verified statically. However, this still leaves open the question of how to prevent an isolation domain from modifying the stack to place a return address for which it does not have permission to access. As such, this proposal suggests a modification to standard calling conventions such that a key is silently added by the compiler as the first argument to every function call. These keys take the form of the encryption of the permission bitmask referenced above (more on that in the answer to the last question). Additionally, x86 RET statements are universally disallowed in the generated code except in one specific, compiler-output-hardcoded function VerifyReturn (another modification to standard calling conventions). Where return statements would normally be, this system would place a JMP instruction to VerifyReturn. VerifyReturn then decrypts the key placed as the silent first argument on the stack, checks to see that the return address is unmodified by the bitmask contained within the plaintext, and then invokes a RET instruction (or fails in some manner if the return address was not valid).

- Indeed, it sounds like capability token may be a more appropriate name than "key" for the proposed purpose of this data (I'll keep using the word "key" in this last answer for consistency though). While the key could be any randomly generated value, the goal of this proposal was to store access rights within the key itself so as to prevent the need for any key-¿permission data structure queries. However, simply using the bitmask (which in this model represents the access rights of an isolation domain) as the key would not suffice, because an adversary who has control within an isolation domain could trivially set the "key" to all 1's in order to gain access to the whole address space. The goal of the Feistel cipher isn't to add cryptographic security (the proposal is to use a very small number of rounds, e.g. n=2 or n=4), but rather to make it so that (1) looking at a single key does not give information about the permissions it contains, and (2) looking at a single key does not give enough information to arbitrarily generate another valid key. In order to accomplish goal (2), there is a randomly generated, short set of bits that is placed in the lowest bits of the plaintext of the "key" in order to verify the integrity of the key (these bits would correspond to the alignment specific by SFI and therefore would be unused by the bitmask). Perhaps a more sound argument could be made for having randomly generated keys for each section, but in my mind this system seemed potentially simpler/faster in implementation.

- 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.

2 Multi-Domain SFI Example

This section gives an example of a toy program written using multi-domain SFI. It is broken down into sections on C++, virtual memory, and assembly.

2.1 C++

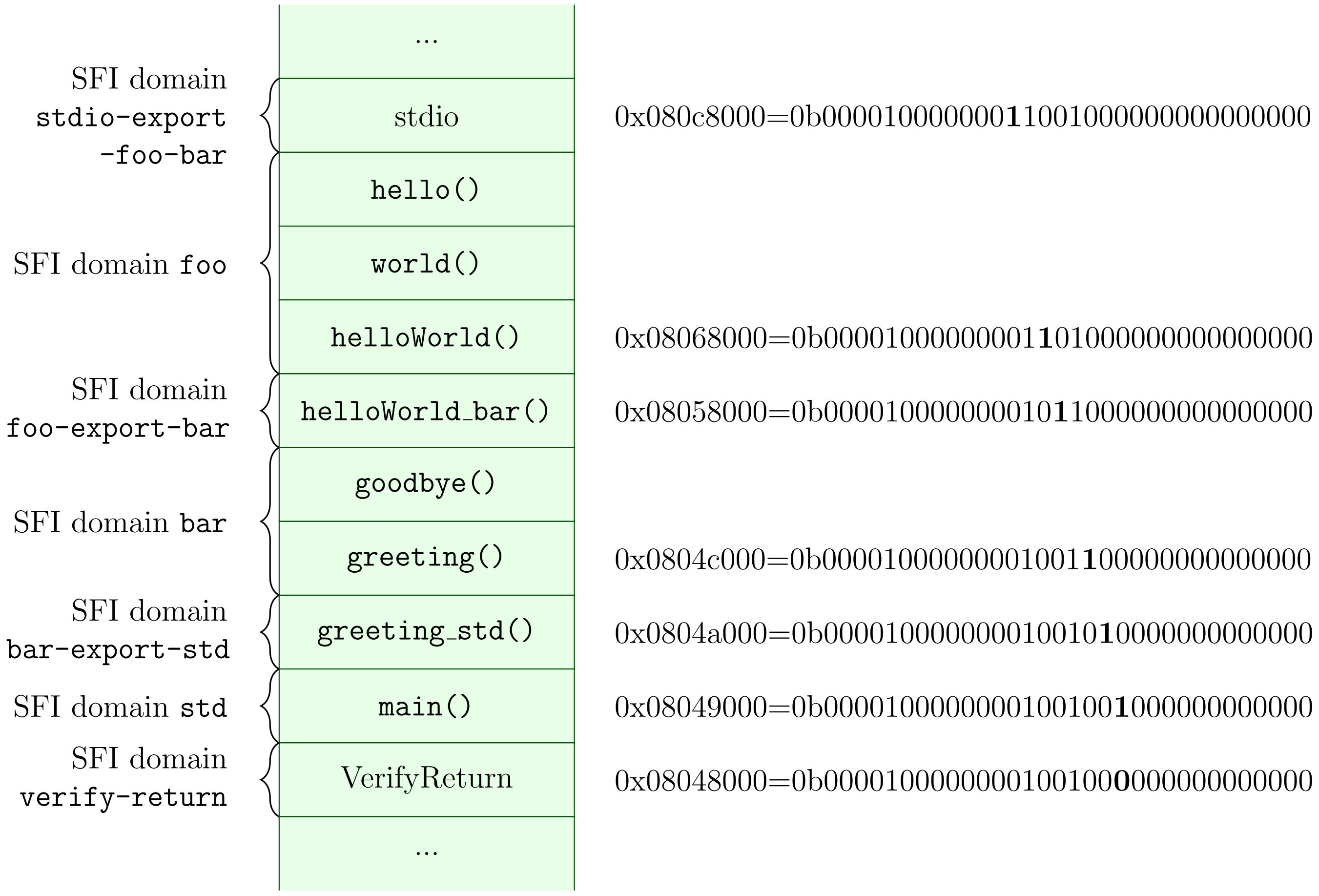
Let us consider the following C++ code, which is a simple program that outputs:

```
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;
}
```

2.2 Memory Layout

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.



...
SFI domain stdio-export-foo-bar
stdio
0x080c8000=0b00001000000011001000000000000000
hello()
SFI domain foo
world()
helloWorld()
0x08068000=0b00001000000001101000000000000000
SFI domain foo-export-bar
helloWorld_bar()
0x08058000=0b00001000000001011000000000000000
goodbye()
SFI domain bar
greeting()
0x0804c000=0b00001000000001001100000000000000
SFI domain bar-export-std
greeting_std()
0x0804a000=0b00001000000001001010000000000000
SFI domain std
main()
0x08049000=0b00001000000001001001000000000000
SFI domain verify-return
VerifyReturn
0x08048000=0b00001000000001001000000000000000
...

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.

2.2.1 Bitmasks

It is important to note that each SFI domain (as depicted in the above figure), must be able to be isolated by bitmasks for jumps at the assembly level.

For example, a bitmask which would permit only access to SFI domain `verify-return` (while maintaining 32-byte alignment) would be `0x08048fc0=0b00001000000001001000111111000000`. Such a bitmask is applied by `AND`ing the bitmask with an arbitrary virtual memory address prior to `JMP` instructions to guarantee that indirect memory jumps will be to an allowed region of memory.

Bitmasks can also grant access to multiple SFI-domains simultaneously using this method. To extend the previous example, a bitmask which permits access to both the SFI domains `verify-return` and `bar` would be `0x0804cfc0=0b00001000000001001100111111000000`.

Note from the previous example that the carefully chosen addresses for the section of memory in which each SFI domain is placed allows bitmasks to give joint access to regions which are not adjacent. To complete the example, the full bitmask for the `bar` SFI domain should permit access to `verify-return`, `bar`, and `foo-export-bar`. This bitmask would be `0x0805cfc0=0b00001000000001011100111111000000`.

2.3 Assembly

There are two aspects of the generated assembly code that are different than standard compiler output.

1. All indirect jumps are masked
2. Function returns are implemented differently

2.3.1 Indirect Jumps

The main goal of SFI is to isolate different regions of code from each other. As such, jumps between SFI domains are restricted. Direct jumps – that is, jumps with hard-coded destinations – can be statically verified to be within valid domain targets.

However, indirect jumps cannot be verified before running the program by the compiler nor a static analyzer. As such, bitmasks are applied before every indirect jump which limit jumps to valid destination regions. Each SFI domain has it's own bitmask. The creation of the bitmask for each SFI domain is detailed in the prior section "Bitmasks". The bitmask of the containing SFI domain is applied with a logical `AND` at the assembly level immediately preceding every indirect jump.

2.3.2 Function Returns

Function returns present a subtle and challenging problem. Normally, return addresses are placed on the stack by a function caller. Return instructions jump to the runtime-specified return address. 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 destination to the valid region of memory. This necessitates getting rid of all `RET` instructions and replacing them with indirect jumps. For the purpose of having an assembly-level function which can verify return targets before performing jumps, we will chose to implete returns as a direct jump to a hard-coded assembly function called `VerifyReturn` which simply verifies the return address and performs an indirect jump to that address.However, this does not solve the problem. Since the function callee has control of the stack, the callee could maliciously change the return address and trivially set the bitmask passed by the caller to all ones.

As a result, we propose that the bitmask implicitly passed by the caller (via a modification to standard function calling conventions within the compiler) be encrypted using a low-round Feistel cipher. Within the lowest 6-bits of the plaintext (since all jumps must be

32-byte aligned per SFI specification resulting in these bits being unused by the mask) of the encrypted value would be a secret key known only by `VerifyReturn` in order to prevent adversary-controlled SFI domains from generating random bit for a malicious mask and having the mask be accepted by the return verifier. `VerifyReturn` masks the return address on the stack with the decrypted bitmask and performs and indirect jump of the decryption verification succeeds, otherwise it halts execution if the verification fails.

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, the Feistel network can be implemented in a small number of bitwise logical operations and a single conditional jump.