Binary Code Continent: Finer-Grained Control Flow Integrity for Stripped Binaries

Minghua Wang[1,2,4], Heng Yin[2], Abhishek Vasisht Bhaskar[2], Purui Su[1,3], Dengguo Feng[1]
{wangminghua, supurui, feng}@tca.iscas.ac.cn, {heyin, abhaskar}@syr.edu
[1]Trusted Computing and Information Assurance Laboratory, Institute of Software, Chinese Academy of Sciences
[2]Syracuse University
[3]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
[4]University of Chinese Academy of Sciences

ABSTRACT

Control Flow Integrity (CFI) is an effective technique to mitigate threats such as code-injection and code-reuse attacks in programs by protecting indirect transfers. For stripped binaries, a CFI policy has to be made conservatively due to the lack of source code level semantics. Existing binary-only CFI solutions such as BinCFI and CCFIR demonstrate the ability to protect stripped binaries, but the policies they apply are too permissive, allowing sophisticated code-reuse attacks. In this paper, we propose a new binary-only CFI protection scheme called `BinCC`, which applies static binary rewriting to provide finer-grained protection for x86 stripped ELF binaries. Through code duplication and static analysis, we divide the binary code into several mutually exclusive code continents. We further classify each indirect transfer within a code continent as either an Intra-Continent transfer or an Inter-Continent transfer, and apply separate, strict CFI polices to constrain these transfers. To evaluate `BinCC`, we introduce new metrics to estimate the average amount of legitimate targets of each kind of indirect transfer as well as the difficulty to leverage call preceded gadgets to generate ROP exploits. Compared to the state of the art binary-only CFI, BinCFI, the experimental results show that `BinCC` significantly reduces the legitimate transfer targets by 81.34% and increases the difficulty for adversaries to bypass CFI restriction to launch sophisticated ROP attacks. Also, `BinCC` achieves a reasonable performance, around 14% of the space overhead decrease and only 4% runtime overhead increase as compared to BinCFI.

Categories and Subject Descriptors

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

General Terms

Security

Keywords

Control Flow Integrity

ACSAC '15, December 07-11, 2015, Los Angeles, CA, USA

http://dx.doi.org/10.1145/2818000.2818017.

1. INTRODUCTION

ASLR [22] and DEP [2] have mitigated traditional threats to computer programs. However, attackers are still able to launch attacks through code reuse [4, 21] even if ASLR and DEP are enabled. Return Oriented Programming [20] (ROP) is one such code reuse technique. Over time this technique has gained popularity and presents a challenge to program safety. Several works [3, 6–8, 13, 19, 24] have proposed solutions to tackle these kinds of attacks and they have all made improvement to some extent.

Control flow Integrity [1] plays an important role in combating control flow hijack attacks. It forces the control flow transfers in the program to follow the policy represented by the control flow graph. The policy can be strict based on the source code as the the control flow graph tend to be completed. However, for stripped binaries, because of the lack of source code or debug information, the CFI policy is coarse-grained. Although many advanced CFI implementations such as CCFIR and BinCFI can prevent the vast majority of control flow hijack threats, they may still be vulnerable to sophisticated ROP attacks as illustrated in the works [5, 10].

In this paper, we extend one state of the art CFI solution, BinCFI, and propose a new binary-only CFI protection scheme, `BinCC`, which provides finer-grained protection for stripped binaries. By duplicating a little amount of code and performing static analysis, we divide the binary code into several mutually exclusive code continents, and classify each indirect transfer as either an Intra-Continent transfer or an Inter-Continent transfer. We further apply strict CFI polices to constrain these two kinds of transfers. Under our policy, Intra-Continent transfers have determined targets within the continent, and Inter-Continent transfers are only permitted to reach specific types of targets across continents. As a result, we significantly refine the legitimate transfer targets.

To evaluate our policy, we introduce new metrics to estimate the average amount of legitimate targets of each kind of indirect transfer as well as the difficulty to leverage call preceded gadgets to generate exploit. As compared to BinCFI, the experimental results show that `BinCC` makes great improvement on both these aspects. `BinCC` reduces the legitimate transfer targets by 81.34% compared to BinCFI. Especially, `BinCC` provides much finer-grained protection for returns and degrades the average legitimate targets by 87%, thereby significantly increasing the difficulty to launch sophisticated ROP attacks by leveraging call preceded gadgets. Apart from that, `BinCC` has a reasonable performance, 4% higher runtime overhead and 14% less space overhead than BinCFI.

In summary, `BinCC` has the following contributions:

- `BinCC` proposes code duplication and code continents construction and thus classifying indirect transfers as either Intra-

Continent transfers or Inter-Continent transfers, allowing to enforce a finer-grained CFI policy.

- `BinCC` can considerably refine the legitimate targets for a binary's indirect transfers, especially for returns, as compared to binary based CFI implementations, BinCFI and CCFIR.
- `BinCC` can not only eliminate common control flow hijack threats, but also significantly increase the difficulty to launch sophisticated ROP exploits, for instance, leveraged by call preceded gadgets.
- `BinCC` has reasonable performance, around 14% less space overhead and 4% higher runtime overhead as compared to BinCFI.

We organize the remainder of paper as follows. We discuss background and related work at Section 2, and then present the concept of code continent and our policy in Section 3. We describe code continent construction in Section 4 and CFI enforcement in Section 5. Section 6 presents our evaluation. Discussion is in Section 7 and conclusion is in Section 8.

2. BACKGROUND AND RELATED WORK

CFI related implementations can be generally classified into two categories, namely, source code based and binary only based. Since in practice a large number of binaries we face are closed source, we lay more emphasis on binary only based solutions. Particularly, we discuss more on two state of the art implementations CCFIR as well as BinCFI and the possible attacks towards them.

2.1 Source Code Based CFI

Many CFI implementations [3,11,12,16,23] need source code to enforce CFI policy. The works [11,23] mainly focus on the protection of virtual function calls. They leverage class hierarchy analysis to identify legitimate targets and insert checking code to perform method and vtable checks. Both solutions can provide fine-grained protection to calls but little protection to returns. CFL [3] works by performing a lock operation before each indirect transfer and a corresponding unlock operation only at valid destinations. It shares the similar insight with ours in terms of constraining returns from relatively called functions, but it relies on source code, which is not always available in practice, and additionally, it lacks modular support. MCFI [15] is a CFI solution that supports separate compilation. It uses several tables to store legitimate targets of indirect transfers, and uses auxiliary type information to update their targets when modules are dynamically loaded. Instrumented code is inserted before indirect branches and runtime checks are needed.

CPI [12], RockJIT [16] also focus on control flow integrity. RockJIT [16], an extension of MCFI [15], is able to prevent control flow attacks caused by JITed code. It computes the program's precise CFG using the JIT compiler's source code and updates the CFI policy when dynamic code is generated at runtime. The work [12] introduces code pointer integrity and code pointer separation. It can guarantee the program safety by selectively protecting code pointer accesses, which are susceptible to control flow hijacking attacks.

2.2 Binary Only Based CFI

There are plenty of binary only based solutions [14,24,25] which enforce control flow integrity. O-CFI [14] applies a coarse-grained policy to constrain control flow transfers. The integrity checking is performed by consulting a bounds lookup table which stores the legitimate range for each indirect branch. It also uses code-randomization which helps the CFI enforcement and also enables the program to resist information disclosure attacks. SFI [24] is a sandboxing technique that helps enforce control flow integrity. The basic idea is to make untrusted modules execute in the same process' address space without allowing them to access each other's data and code. PittSFIled [13] and NaCl [25] are SFI-based implementation to secure native code and they restrict indirect transfers' targets to meet specific alignment requirement.

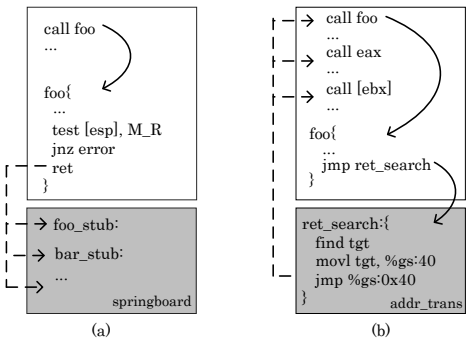

Figure 1: Possible Attacks to CCFIR and BinCFI

Lockdown [17] applies fine-grained policies that protect binaries. It uses shadow stacks to enforce integrity for returns. However, shadow stacks could introduce high runtime overhead as more memory read and write operations are needed to maintain call/return pairs. More importantly, shadow stacks need to be stored in safe memory regions, which needs support from segmentation provided by hardware or isolation techniques like SFI. The safe regions could be vulnerable to information leakage and thus controlled by attackers to perform exploits. One real attack is shown in [9].

CCFIR [26] is a binary rewriting based CFI solution that targets Windows x86 executables. It arranges the targets for indirect transfers into a newly introduced section called a "springboard". The targets are aligned in the springboard, and each indirect transfer is instrumented to check whether the runtime target meets the correct alignment. If so, the transfer will be performed through the corresponding stub in the springboard, otherwise the target is regarded as invalid. BinCFI [27] is another CFI work based on binary rewriting and targets ELF binaries. BinCFI disassembles the binary, instruments indirect transfers and puts the instrumented code into a newly introduced code section. For each instruction, it maintains a mapping between the original location and the new location. Indirect transfers are instrumented to jump to address translation routines, which look for the targets of these transfers. If found, the routine will execute the target at the new location. Also, BinCFI supports inter-module CFI policy by modifying the loader to behave as a hub to transfer control among modules.

The CFI policy towards stripped binaries is coarse-grained as the CFGs are imprecise. The permissive policy is still likely to be violated although it has the ability to mitigate the vast majority of common control flow hijacks. Figure 1 shows the possible attack models to CCFIR and BinCFI. The solid line indicates the execution flow at run time, while the dashed line indicates the possible targets that could be leveraged in an attack. For CCFIR, as shown in Figure 1(a), the target, residing in the springboard, is only required to align against the constant M_R, so any call-site's address would be regarded as legitimate. Similarly, for BinCFI, a controlled return would be able to reach any call sites in the binary, as shown in Figure 1(b). Those returns are left unprotected and give adversaries chances to leverage call preceded gadgets to launch attacks. One recent practical exploit has been shown [10].

3. BINARY CODE CONTINENT

We propose a finer-grained CFI policy that is able to significantly refine the legitimate targets of indirect transfers. In general, we achieve this by duplicating some necessary code and performing static analysis to separate the binary into several mutually exclusive code continents, and more importantly, assigning each indirect transfer to be either an Intra-Continent transfer or an Inter-Continent transfer. This enables us to enforce separate, strict policy to achieve finer-grained protection. In the following sections, we first describe the concept of code continent through a sample and then present our CFI policy.

3.1 Code Continent

Code continents are constructed from Super-CFGs of functions. A Super-CFG (Super Control Flow Graph) is constructed, for a function, from its CFG (Control Flow Graph) by connecting all direct call sites in the CFG to the entry point of the callee's Super-CFG and the end point of the callee's Super-CFG is connected back to the call site. This process is repeated recursively until all the direct calls in the function are handled. A code continent is a directed graph that is constructed from merging functions' Super-CFGs based on their common edges. Therefore, code continents are mutually exclusive.

We use Figure 2 to illustrate code continents that represent the sample code. Suppose a binary originally contains the functions, `main`, `foo`, `bar`, `qux` and `start`. `start` is the binary's entry point. The ordinals within graph nodes represent the corresponding instructions in code. In Figure, CC_1 represents the code continent generated from the Super-CFG of `main`. `foo`'s Super-CFG, which is constructed by `5`, `6` and `7`, is included because `foo` is directly called at `3`. CC_2 represents the code continent generated from `bar`'s Super-CFG, which has only four nodes. As no direct call site is present in `bar`, no callee's Super-CFG needs to be added in.

We divide the graph nodes contained in a code continent into three categories: *root nodes*, *border nodes* and *inner nodes*. *root nodes* represent entry points of indirect called functions, and they are represented in grey in the Figure. For instance, `1` and `5` are *root nodes* in CC_1. *border nodes* are the indirect transfer instructions whose targets cannot be identified while computing the Super-CFG, and they are represented striped in the Figure, for example, `2`, `4` and `6` are border nodes in CC_1. The nodes that are neither root nor border nodes are denoted as *inner nodes*, and they are represented in white in the Figure. The instructions represented by *inner nodes* are either non-control flow transfer instructions or control flow transfers that have determined targets present inside Super-CFGs, for example, `3` and `7`.

By dividing the nodes, we are able to classify indirect transfers into two categories, Intra-Continent transfers and Inter-Continent transfers. Intra-Continent transfers are the indirect transfers originating from inner nodes, while Inter-Continent transfers originating from border nodes. More importantly, we guarantee that each indirect transfer is either an Intra-Continent or an Inter-Continent transfer and thereby enforcing separate, strict policies. This is achieved by performing code duplication before code continent construction.

We know that in general the functions in a binary fall into two categories, *Indirectly Called Functions* (ICFs) and *Directly Called Functions* (DCFs). There might be some functions that are called in both ways, and by duplicating those functions, we can partition the functions into two mutually exclusive sets by considering the duplicated functions as ICFs, as shown in Figure 3. At an indirect call site, we perform runtime dispatch to execute the duplicated function when the original function is called. As such, a function will be called only in a certain way, either indirectly or directly, and because of this, a return will go back to a specific type of call site. We thus can divide all the returns into two mutually exclusive sets: *direct returns*, which only target direct call sites, as well as *indirect returns*, which only target indirect call sites.

As in the sample code, ICFs are composed by `main`, `foo`, `qux`, `bar`, while DCFs is composed by `foo`. Only `foo` is called in both ways. Suppose the execution starts from `start`, originally, the function `foo` is indirectly called at first time by `9` and directly called at second time by `3`, so the return `7` would go back to those two call sites respectively. In `BinCC`, `foo'` is a new function generated from duplicating `foo` and will be executed when `foo` is indirectly called at `9`. This makes `7` become an Intra-Continent transfer and `7'` become an Inter-Continent transfer, which means that `7`, as a direct return, would only return to `3`, while `7'`, as an indirect return, would return to `9`, as shown by the two dot arrows.

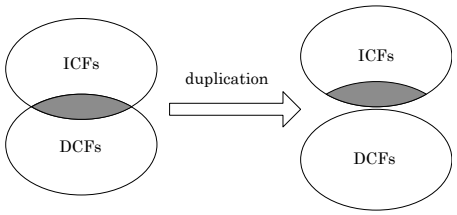

Figure 3: Indirectly Called Functions(ICFs) and Directly Called Functions(DCFs). Functions are partitioned into two mutually exclusive parts through duplicating the functions in the intersection. Duplicated functions are considered as ICFs.

3.2 CFI Policy

We propose Intra-Continent Policy and Inter-Continent Policy to constrain Intra-Continent and Inter-Continent indirect transfers respectively.

Intra-Continent Policy.

This policy is to constrain the inner nodes representing indirect transfers whose targets can be determined statically. Their targets are always present inside their own code continent and are always determined by the Super-CFGs that compose this continent. Within a continent, there are only two kinds of indirect control flow transfers we need to be concerned with, one being direct returns and the other being indirect jumps associated with switch-case jump tables. For each direct return, the legitimate targets are its corresponding target call sites within the current code continent. For each indirect jump, the legitimate targets are all the case branches in the corresponding jump table. The case branches in a jump table can be identified by static analysis, and we connect the indirect jump with all its case branches when building Super-CFGs, which makes its targets deterministic.

For the sample illustrated, there is one Intra-Continent indirect transfer in those continents. It is the return at `7` in CC_1. It is only allowed to return back to the call site `3` (or say the instruction `4`).

Inter-Continent Policy.

This policy is to constrain the border nodes, which are indirect transfers whose targets cannot be statically determined from Super-CFGs. Based on the transfer types, we apply the following policies.

i Indirect call nodes can only reach root nodes that represent the entry points of ICFs.

ii Indirect return nodes can only go back to the border nodes that represent indirect call sites.

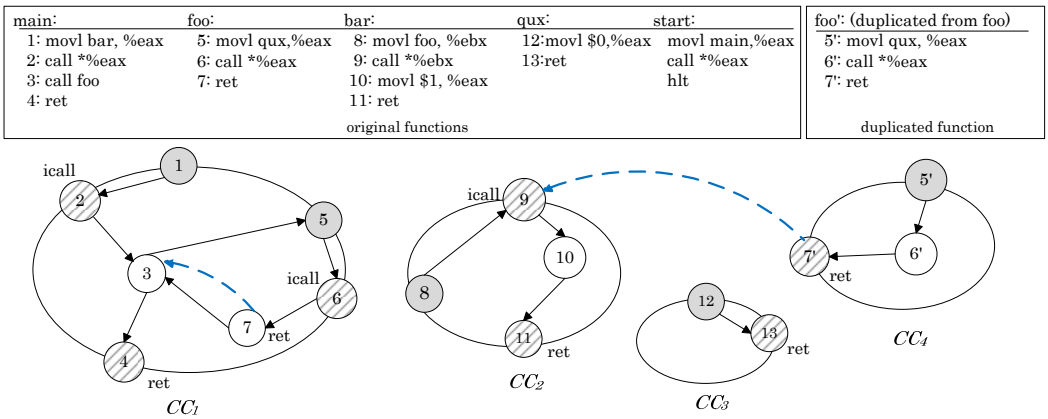

Figure 2: Code Continents Sample

iii Indirect jump nodes, whose targets cannot be resolved by static analysis, can go to either root nodes or border nodes that represent call sites.

The policies are enforced based on the following insights. After the functions are partitioned into two mutually exclusive sets, an ICF can only be called by indirect call instructions (i), and due to this, indirect returns from an ICF will only go back to indirect call sites (ii). Indirect jumps normally have associations with jump tables, where their targets can be determined by static analysis, but there may be some indirect jumps whose targets cannot be known statically. Their targets should be constants present in the binary, and they can either be the entries of ICFs or call sites (iii).

For the sample above, the root nodes are entry nodes of ICFs (`1`, `5`, `5'`, `8` and `12`), the border nodes contain indirect call nodes (`2`, `6` and `9`) and indirect return nodes (`4`, `11`, `13` and `7'`). In `BinCC`, the indirect call nodes (`2`, `6` and `9`) are permitted to call the ICFs (`1`, `5`, `5'`, `8` and `12`). The indirect returns (`4`, `11`, `13` and `7'`) are allowed to return back to indirect call sites (`2`, `6` and `9`).

4. CODE CONTINENT CONSTRUCTION

To construct code continents, we identify all the DCFs and ICFs in the binary, and then perform control flow analysis to compute the CFGs of those functions, which helps determine what functions we need to duplicate and also to compute Super-CFGs. After the code duplication, we finally construct the code continents based on Super-CFGs. We now discuss these phases.

4.1 DCF Identification

Each DCF is called through a relative call. The function entry's address could be computed through the relative call instruction's address and operand value. Through this, we identify all the direct call sites from the disassembly to obtain all the DCFs.

4.2 ICF Identification

ICF identification is based on the observation that the address of an ICF stems from the constants in the binary. We search for any constant that could be used to represent or to compute a function's address.

In binaries that contain relocation tables, the table entries, which are constants, cover the addresses of ICFs. We check whether the sum of the each constant with the code base address is present in the code section. If so, this sum is resolved as the address of an ICF. In binaries that contain no relocation tables, we scan the binary for the constants and handle it differently for non-PIC and PIC modules.

For a non-PIC module, we use a integer-sized window (e.g., 4-byte in 32-bit systems) to scan the constants within the data, .rodata, .init_array, exported symbol sections and any other sections possibly containing integers representing valid code addresses. We also collect constants used within the code section. By consulting the disassembly, if the constant or the sum of the constant and the code base is a valid code address and meets one valid instruction's boundary, we considered it as a constant function pointer candidate.

For a PIC module, functions can be reached by PC thunks. Therefore, in addition to performing the same approach as for non-PIC modules to collect constants, we identify all the PC thunks and see whether they are used to have access to functions. If so, we consider those functions as constant function pointer candidates.

Note that not all obtained candidates are function entry points. Some constants are actually the addresses of case branches in a switch-case jump table. Case branches are within a function and cannot be function entries. Some candidates are actually the addresses of some return call sites but not function entry points. So we remove these two kinds of constants from the candidates, and the remaining are the final ICFs.

4.3 Control Flow Analysis

We perform static control flow analysis to compute the CFGs of all the functions, which are composed of identified ICFs and DCFs. Our analysis is launched in a conservative way. We try to identify all possible targets for each indirect branch. The main difficulty is resolving the possible targets of indirect jumps. In most common cases, an indirect jump is used to dispatch execution from a jump table, and all legitimate targets of this jump are the corresponding case branches, which can be identified by the previously discovered constants. Therefore, for an indirect jump, we check whether or not it is associated with a jump table and connect it to the corresponding case branches if so. As BinCFI is able to perform this, we reuse that component to get the desired result to help construct CFGs.

4.4 Code Duplication

As discussed earlier, to enforce our policy, we duplicate ICFs that fall in the intersection of the ICFs and DCFs. However, in some cases, due to compiler's optimizations, some ICFs are likely to have common returns with DCFs, and each of those returns cannot be given a clear transfer type (either Intra or Inter), which violates our CFI policy. To resolve this issue, if an ICF has common returns with a DCF, as determined by referencing these two functions' CFGs, we also duplicate this ICF. The duplicated function will become a new ICF in the binary.

After identifying the functions to be duplicated, we duplicate all

the instructions in their CFGs. All the duplicated code and the original code are put into a new code section after being instrumented with CFI policy (discussed at Section 5). The original code section is marked non-executable and all existing data sections are kept unchanged. The constants with duplicated instructions are unchanged as well, so the values in the data sections are still accessed correctly.

Additionally, in the duplicated functions, we need to fix up the operand values of relative calls and jumps. For relative calls, the operands are adjusted to simulate the call instructions of the original version of the function. Similarly, for relative jump instructions, the operands are adjusted to jump to the corresponding target branches in the duplicated function.

4.5 Code Continent Construction

Code continents are constructed from the Super-CFGs of functions. Algorithm 1 shows how we compute a function's Super-CFG. We search the function's CFG for direct calls, and at a direct call site we add the callee's Super-CFG into the graph through $AddSuperCFG$. This function introduces two new edges, one being from the the direct call site(i.e., node i) to the callee's entry, and the other being from the callee's return to the direct call site.

Algorithm 1 $SuperCFG(CFG_{func})$: Compute $func$'s Super-CFG based on its CFG

Input: CFG_{func} : the CFG of the function $func$
Output: sg: the super-graph of $func$
1: $sg \Leftarrow CFG_{func}$
2: **for** each $i \in Nodes(CFG_{func})$ **do**
3: **if** i is a direct call to f **then**
4: $sg = AddSuperCFG(sg, i, SuperCFG(f))$
5: **end if**
6: **end for**
7: **return** sg

Algorithm 2 shows how we construct code continents and classify nodes in code continents. The algorithm takes all ICFs as input. It computes Super-CFGs of each ICF and then merges these Super-CFGs based on their common edges (e.g., the common callees), which is achieved by $MergeGraph$. The algorithm finally produces the set of mutually exclusive code continents, with root nodes, border nodes and inner nodes classified in each code continent. From the algorithm we see that the nodes are classified based on the types of nodes of Super-CFGs that compose a code continent. The root nodes comes from the entry points of ICFs in the current continent. The border nodes comes from the nodes representing indirect calls($icall$), indirect returns($iret$) and indirect jumps whose targets are not statically determined($ijmp_u$). The inner nodes are composed by the nodes representing non-control flow instructions and control flow transfers whose targets are determined (e.g.,direct calls, direct returns) in the current continent.

There is one corner case that we need to consider. There might be orphaned code pieces, which are not ever reached by static control flow analysis, for instance, unresolved target branches for indirect jumps or dead code. We need to restrict the indirect transfers in such code pieces as they may be invoked at runtime. To achieve this, we also generate a code continent, denoted as an orphaned code continent, for each of them. An orphaned code continent is composed by the Super-CFG of a orphaned code piece, and the Super-CFG is constructed by applying Algorithm 1 by considering the instructions of the code piece as its "CFG"(which is CFG_{func}). Also, $ibrnch$ of the Super-CFG(i.e.,$sg.ibrnch$), composed by $icall$, $iret$ and $ijmp_u$, are *border nodes*, and inn of the Super-CFG(i.e.,$sg.inn$) are *inner nodes*. Considering that an orphaned continent is not discovered as a function and thus cannot be invoked by a call, we assign the entry of an orphaned continent as a *border node*, instead of a *root node*.

Algorithm 2 $ConstructCC(ICFs)$: Construct Code Continents by taking ICFs as input

Input: all the ICFs: $ICFs$;
Output: all the code continents: CC
1: $CC \Leftarrow \phi; SG \Leftarrow \phi; SG_{done} \Leftarrow \phi$
2: **for** each $icf \in ICFs$ **do**
3: $sg = SuperCFG(CFG_{icf}); sg.ent = icf.entry$
4: **for** each $i \in Nodes(sg) - sg.ent$ **do**
5: **if** i is a $icall$ or a $ijmp_u$ or a $iret$ **then**
6: $sg.ibrnch = sg.ibrnch \cup \{i\}$
7: **else**
8: $sg.inn = sg.inn \cup \{i\}$
9: **end if**
10: **end for**
11: $SG = SG \cup \{sg\}$
12: **end for**
13: **for** $sg \in SG - SG_{done}$ **do**
14: $cc_{cur} = sg; SG_{done} = SG_{done} \cup \{sg\}$
15: $cc_{cur}.border = cc_{cur}.border \cup \{sg.ibrnch\}$
16: $cc_{cur}.inner = cc_{cur}.inner \cup \{sg.inn\}$
17: $cc_{cur}.root = cc_{cur}.root \cup \{sg.ent\}$
18: **for** $sg' \in SG - SG_{done}$ **do**
19: **if** $HasCommonEdges(cc_{cur}, sg')$ **then**
20: $cc_{cur} = MergeGraph(cc_{cur}, sg')$
21: $cc_{cur}.border = cc_{cur}.border \cup \{sg'.ibrnch\}$
22: $cc_{cur}.inner = cc_{cur}.inner \cup \{sg'.inn\}$
23: $cc_{cur}.root = cc_{cur}.root \cup \{sg'.ent\}$
24: $SG_{done} = SG_{done} \cup \{sg'\}$
25: **end if**
26: **end for**
27: $CC = CC \cup \{cc_{cur}\}$
28: **end for**
29: **return** CC

5. CFI ENFORCEMENT

After code continent construction, we perform instrumentation on different nodes in the continents to enforce our CFI policy. This is implemented on top of BinCFI, so we briefly describe the basic instrumentation structure BinCFI provides and then give details on our enforcement.

5.1 Basic Infrastructure

BinCFI instruments the disassembly and inserts the instrumented code into a new code section with making the original code section non-executable. It uses address pairs of the form <orig_addr, new_addr> to associate the new locations in the instrumented code with their corresponding original locations. Specifically, BinCFI generates address pairs for all indirect transfer targets and maintains them in two different address translation hash tables, one for returns and the other for indirect jumps and calls. All address translation tables are read-only.

BinCFI instruments indirect calls/jumps and returns. For the indirect jumps associated with jump tables, their operands are replaced by expressions of the form *(CE_1+Ind)+CE_2, where CE_1 and CE_2 are constants, and CE_1 indicates the jump table associated, and *(CE_1+Ind) indicates all possible case branches. Also, BinCFI introduces a new jump table based on every CE_1, with transformed case branches' addresses inside. For the remaining in-

direct transfers, they are instrumented as shown in Figure 4. We illustrate it by taking an indirect call as an example, the other indirect transfers are handled in the same way. Firstly the runtime target(i.e.,%eax) is saved to a thread local variable(i.e.,%gs:0x40), and then the control is transferred to a routine, `addr_trans`, which performs checking and address translation.

```
call *%eax          movl %eax, %gs:0x40
                    jmp addr_trans
```

Figure 4: BinCFI's instrumentation for indirect transfers, shown by taking an indirect call as an example. Left is the original instruction and right shows the instrumentation.

Figure 5 shows how `addr_trans` works. In the routine, it checks whether the transfer is against the CFI policy. If not, it performs the address translation. As %gs:0x40 stores the address that falls in the original code section, namely, orig_addr, BinCFI consults the relevant address translation table for the corresponding translated address, namely, new_addr. If found, it jumps to new_addr. Otherwise, it calls the global lookup routine, which helps address translation across different modules.

```
proc addr_trans:
        check_cfi_policy(orig_addr)
        if invald: trigger_alert()
        new_addr = find_trans_tgt(addr_trans_table, orig_addr)
        if found: goto new_addr
        else:  goto global_lookup_routine
```

Figure 5: Address Translation Routine in BinCFI.

The global lookup routine works by consulting the GTT (Global Translation Table). For every loaded module, the GTT records the relationship between the base address of the module and the address of the module's `addr_trans`. For the above example, the global routine checks which module the address %gs:40 belongs to, if no module is found in the GTT, an alert is triggered, and if found, the control is transferred to that particular module's `addr_trans` routine which takes care of the address checking and translation as mentioned above. The global lookup routine and the GTT are added in the loader, and they will loaded at different memory address every time. Also, the loader will update the GTT when a new module is loaded during the runtime.

5.2 Our Instrumentation

We apply BinCFI's infrastructure and make improvement over BinCFI to enforce our CFI policy.

Extensions to Address Translation Tables.

To achieve our enforcement, we make two extensions to address translation tables. Firstly, we introduce new table entries for the newly introduced indirect transfer targets due to the code duplication, which are duplicated functions and the return call sites within them. Secondly, we modify the relevant table entries of the functions that have been duplicated, so that if the original function is called at runtime the corresponding new function will actually be executed.

We take `foo` in the code of Figure 2 as an example to show such an extension. Figure 6(a) shows how BinCFI records the address pair for `foo`, while Figure 6(b) shows the extension made by `BinCC`. Two new targets for indirect transfers, the function entry <5', 5'>, the call site <6', 6'>, are added into the table. 5_new is changed to 5', so `foo'` will be actually executed when `foo` is indirectly called at runtime.

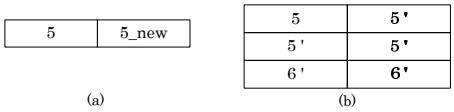

Figure 6: Address translation table extension for `foo`. Indirect transfer targets are represented by ordinals, of which 5_new is the corresponding translated address of `foo` in BinCFI.

Although new indirect transfer targets are introduced through the code duplication, it has not brought challenges to our protection in practice. The number of newly added targets is small as only a little proportion of functions need to be duplicated (shown at Section 6.1). Apart from that, the new transfers are constrained in the same way and the experimental results show that they have no impact on our protection.

Intra-Continent Policy Enforcement.

This policy is to constrain the *inner nodes* that represent indirect transfers within a code continent: direct returns and indirect jumps related to jump tables. Their targets are determined and can be obtained from the Super-CFGs that compose the code continent.

To constrain direct returns, we prepare each of them a separate address translation hash table to store its legitimate targets. For each direct return, the targets are the call sites which this direct return is connected with in the Super-CFGs. We get those call sites and put them into the corresponding address translation table. Apart from that, we instrument each direct return as shown in Figure 7. The `start` and `size`, embedded in two `prefetchnta` instructions, indicate where to find the corresponding address translation table. We use another thread local variable %gs:0x50 to store the first `prefethnta` instruction's address, `_addr`. The translation routine `addr_trans_dret` uses %gs:0x50 at runtime to access `start` and `size` for locating the right table, and then perform the same operations as `addr_trans` as shown Figure 5.

```
_addr:  ret          _addr:  prefetchnta start
                             prefetchnta size
                             movl _addr, %gs:0x50
                             movl %(esp), %gs:0x40
                             jmp addr_trans_dret
```

Figure 7: `BinCC`'s instrumentation for direct returns. Left is the original instruction and right shows the instrumentation.

For the indirect jumps related to jump-tables, we can use a similar structure. However, as mentioned above, BinCFI has instrumented this kind of indirect jumps by restricting them to only go to their corresponding case branches, so we retain its instrumentation.

Inter-Continent Policy Enforcement.

Intra-Continent policy constrains indirect transfers whose targets could be statically determined. In this part, we constrain the rest of the indirect transfers in the binary, which correspond to the *border nodes* of continents. We enforce the Inter-Continent policy as described in Section 3.2 on these nodes. The same instrumentation as described in Figure 4 is performed, except that each kind of transfer is given its own address translation table and address translation routine to conduct address checking and translation.

We make an extra effort to handle one kind of *border nodes*, the indirect jumps that reside in PLT entries. They are used for dynamic symbol resolution. For such an indirect jump, only two legitimate targets should be allowed, one being the next instruction's address, the initialization value before the symbol being resolved,

which is statically determined, and the other being the symbol's resolved address, which is determined at runtime. We can further restrain such a runtime target, since the loader can intercept the symbol resolution process and retrieve the resolved address. To this end, we arrange all the targets of these indirect jumps into one address translation table and put it into a newly introduced read-only data section. Also, we modify the loader to be able to change the property of the data section to writable, update the corresponding table entry with the resolved address after symbol resolution, and change the property back. The change is restricted within this new data section, so there is no impact on other read-only address translation tables.

In addition, we lay consideration on the indirect returns in the orphaned continents, denoted as orphaned returns. As we cannot identify the invoker of orphaned continents by static analysis, we allow their targets to be any call sites.

C++ Exceptions.

Another problem we need to pay attention to is C++ exceptions. In C++ programs, the necessary information for exception handling is stored in the .eh_frame section. When the exception triggered, the system would use current execution context to perform stack unwinding to identify the corresponding catch branch. We introduce new code through code duplication and there is no exception metadata about this code in the .eh_frame, so if a duplicated function contains C++ exceptions logic and actually triggers the exception, no exception handler could be found through stack unwind and the program would thus run incorrectly. To avoid this problem, we do not duplicate the functions containing C++ exceptions logic, and treat the returns from such functions similar to orphaned returns, allowing their targets to be any call site.

6. EVALUATION

We evaluated `BinCC` by testing SPEC CPU 2006 benchmark programs, which are compiled with GCC version 4.6.1 and -O2 optimization level. The evaluation was performed on a Ubuntu-11.10 32-bit virtual machine with one processor, 1.0GB Memory and a 20G Hard Disk.

6.1 Code Duplication Evaluation

One key operation to our CFI policy enforcement is code duplication. We evaluate the quantity of code we need to duplicate.

program	|ICFs|	|DCFs|	$\#total$	$\#dupl$	per %
lbm	5	41	45	1	2.22
gcc	6803	3287	9846	431	4.38
perlbench	2263	950	3185	83	2.61
libquantum	6	104	109	1	0.92
omnetpp	1362	727	1976	306	15.49
sjeng	142	147	287	2	0.70
gobmk	2126	731	2851	209	7.33
bzip2	55	76	130	1	0.77
milc	45	238	282	1	0.35
hmmer	249	349	597	1	0.17
povray	2011	1069	3037	136	4.48
sphinx	16	298	313	1	0.32
h264ref	133	473	598	16	2.68
astar	9	103	109	3	2.75
mcf	5	44	48	1	2.08
namd	54	76	129	1	0.78
soplex	636	487	1062	113	10.64
average	3539	1692	5637	183	3.42

Table 1: Duplicated Functions Statistics

Table 1 shows the number of ICFs and DCFs, the total number of all the functions and duplicated functions for each tested benchmark program. As some ICFs share returns with DCFs, we duplicate these ICFs. This is the reason why $\#dupl$ is different from |ICFs| + |DCFs| - $\#total$ in some samples. The result shows only a little amount of functions, nearly 3.4% of all the binary's functions, need to be duplicated so as to achieve a fine grained protection.

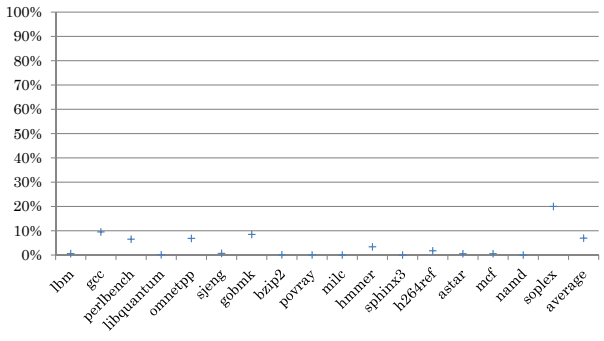

Figure 8: Percentage of Duplicated Instructions

From the table, we see C++ programs such as omnetpp and soplex generally need more functions to duplicate than C programs. In these C++ programs, the vast majority of the duplicated functions are C++ virtual functions. The function pointers are in vtables and identified as ICFs. They could be also directly called, for instance, by the functions from the same class hierarchy, so they need to be duplicated according to our solution.

Figure 8 shows that the percentage of instructions we need to duplicate for each sample. Overall, we only need to duplicate less than 7.0% of the binary's instructions on average.

6.2 Indirect Transfer Targets Metrics

BinCFI introduces the AIR(Average Indirect target Reduction) to evaluate the quality of protection. The definition is as follows.

$$AIR = \frac{1}{N}\sum_{j=1}^{N}(1 - \frac{|T_j|}{S})$$

In this definition, T_j stands for the legitimate target set for the indirect transfer i_j. S stands for the binary's code size. For `BinCC`, the code size S, the number of indirect transfers N, and some legitimate target sets $|T_j|$ can be increased due to the code duplication. We take these into consideration to calculate the AIR and it turns out that `BinCC` achieves a higher AIR, 99.54% as compared to BinCFI, 98.86%.

This metric is not balanced because in a binary S is far higher than $|T_j|$, even coarse-grained CFI solutions can also achieve high AIR, which does not necessarily mean a high quality of protection. We propose a new metric, RAIR (Relative AIR), defined as follows, and use it to demonstrate the extent to which `BinCC` refines the legitimate targets compared to BinCFI.

$$RAIR = \frac{1}{N}\sum_{j=1}^{N}(1 - \frac{|T_j|}{|T'_j|})$$

In the definition, T'_j represents the legitimate targets of indirect transfer i_j by BinCFI, and T_j represents the i_j's legitimate targets by `BinCC`. Figure 9 shows the statistics about this metric. On average, `BinCC` reduced indirect transfers targets by 81.34% from BinCFI.

As compared to BinCFI, `BinCC` refines legitimate targets for each kind of indirect transfer. To evaluate such an improvement,

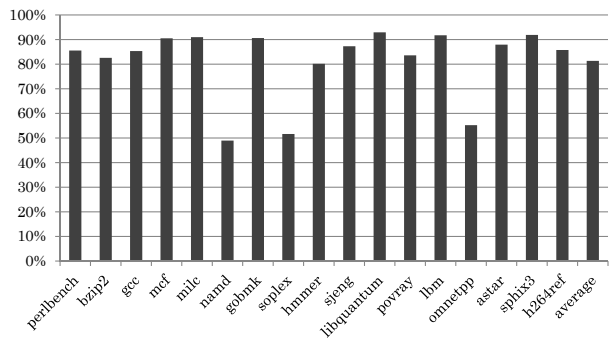

Figure 9: RAIR for Tested Samples

we use the formula defined bellow to estimate the average number of legitimate targets for each kind of indirect transfer.

$$AVG = \frac{1}{N}\sum_{j=0}^{N}|T_j|$$

Suppose i_j is an indirect transfer of a specific kind and T_j is the legitimate target set for i_j in a CFI enforced binary, and N represents how many indirect transfers of this kind exists in the binary. We calculate this metric for three kinds of indirect transfers and illustrate the percentage reduction of legitimate targets, given by $\frac{AVG_{BinCFI}-AVG_{\texttt{BinCC}}}{AVG_{BinCFI}}$, for all the three types of indirect transfers in Figure 10, 11 and 12.

In Figure 10 we see that on average `BinCC` reduced the legitimate targets for an indirect call by around 40% compared to BinCFI. According to BinCFI implementation, all the possible constant code pointers are potential targets of indirect calls. However, some of those constants are actually addresses of call sites and jump-table's case entries, which are not function entries. In our implementation, we remove them from the target set.

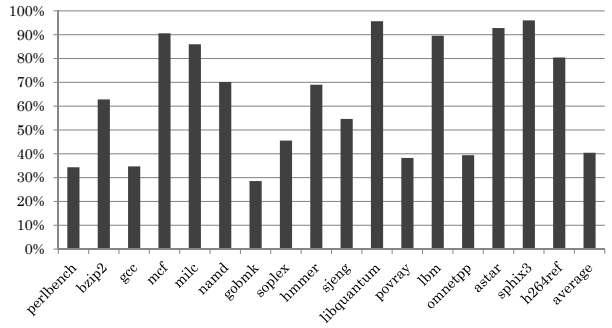

Figure 10: Percentage Reduction of legitimate targets for an indirect call

For indirect jumps, there could be many indirect jumps that reside in the PLT section, and these jumps have certain targets within the current module. `BinCC` takes them into account and thus reduces the legitimate targets by 35% on average compared to BinCFI, as shown in Figure 11.

Figure 12 shows that `BinCC` reduces legitimate targets for a return by 87% on average compared to BinCFI, this is because `BinCC` significantly refines the targets for direct returns by enforcing a much more strict policy on them as compared to BinCFI. We see gcc achieved the largest improvement. There are more direct returns than indirect returns in this binary, and each direct return

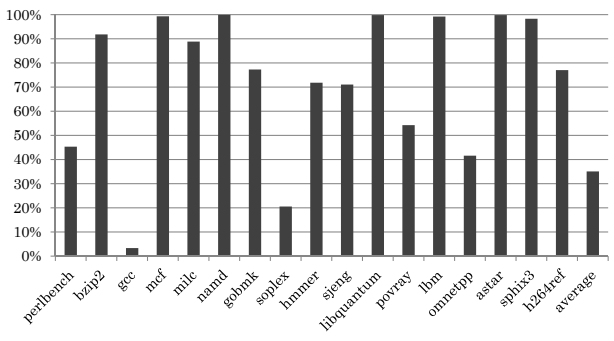

Figure 11: Percentage Reduction of legitimate targets for an indirect jump

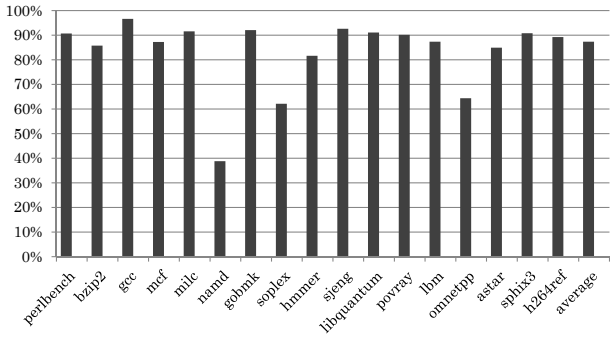

Figure 12: Percentage Reduction of legitimate targets for a return

only has far less legitimate targets on average than each indirect return. All these factors contribute to the improvement.

6.3 ROP Attacks Evaluation

Because of new security mechanisms such as ASLR and DEP, ROP has gained much more popularity among attackers as a technique for launching exploits. CFI implementations such as CCFIR and BinCFI significantly mitigate ROP attacks, since the vast majority of ROP gadgets are instruction-misaligned and no longer feasible for generating exploit. However, the work [5, 10] recently showed that in some cases it is still possible for attackers to leverage call preceded ROP gadgets to build ROP chains and thus bypass the protection from BinCFI and CCFIR.

To evaluate the ability of `BinCC` in preventing ROP attacks, we introduce GS (Gadget Survivability) to represent the difficulty for attackers to leverage call preceded gadgets while establishing ROP chains for exploits under CFI protection. We define this metric as follows.

$$GS = \frac{1}{|R|}\sum_{i=0}^{|R|}\frac{|C_i|}{|C|}$$

Suppose in a CFI enforced binary a return instruction was fully controlled by attackers, how likely this return could be used to reach a call preceded gadget in this binary.

In the definition, C_i represents the legitimate target set for a return instruction r_i under CFI enforcement, R represents the set composed by all the returns, and C represents the set composed by all the call preceded gadgets. For the return r_i, it could reach the $|C_i|$ number of call preceded gadgets. So the probability that it was controlled and could return to a call preceded gadget is $\frac{1}{|R|} * \frac{|C_i|}{|C|}$.

After taking all the returns into consideration, the average probability is $\sum_{i=0}^{|R|} \frac{1}{|R|} * \frac{|C_i|}{|C|}$, or, $\frac{1}{|R|} \sum_{i=0}^{|R|} \frac{|C_i|}{|C|}$.

For BinCFI, from the formula, for each r_i, $|C_i|$ equals $|C|$, so this metric value is 100%. This is consistent with the fact that the hijacked return could reach any call preceded gadget within the binary they protect.

However, for BinCC, this probability is much smaller, as both direct returns and indirect returns are restricted to have far less legitimate targets. Specifically, if the return was an indirect return, it would be allowed to reach any indirect call sites. If the return was a direct return, it would only be allowed to reach gadgets starting at specific direct call sites. We calculate the probability under BinCC as well as BinCFI for all tested samples, and present the statistics in Table 2. From the table, we see that BinCC considerably degraded the probability to leverage call preceded ROP gadgets for ROP attacks, only around 0.70%, with comparison to 100% for BinCFI.

program	BinCC	BinCFI
lbm	2.554%	100%
gcc	0.321%	100%
perlbench	0.530%	100%
libquantum	0.210%	100%
omnetpp	1.145%	100%
sjeng	0.299%	100%
gobmk	1.138%	100%
bzip2	0.474%	100%
povray	0.573%	100%
milc	0.283%	100%
hmmer	0.569%	100%
sphinx3	0.444%	100%
h264ref	0.420%	100%
astar	0.387%	100%
mcf	0.355%	100%
namd	1.237%	100%
soplex	1.017%	100%
average	0.701%	100%

Table 2: Gadget Survivability for BinCC and BinCFI

6.4 Performance

We evaluated both space and runtime overhead. For space overhead, we compared the increase of address translation tables between BinCC and BinCFI, and also evaluated code size increase and total file size increase compared to the original file. For runtime overhead, we ran binaries enforced by BinCC as well as BinCFI and finally made comparisons.

6.4.1 Space Overhead

As we use a similar infrastructure to BinCFI to manipulate binaries, the reason for file size increase is as the same as that for BinCFI, due to the new code section for instrumentation and new read-only data sections for address translation hash tables. The new code size increase 1.4 times the original code size, while the figure for BinCFI is around 1.2. We introduce four kinds of tables for storing targets of indirect calls, indirect jumps, indirect returns and direct returns, while BinCFI uses two kinds for storing indirect calls/jmps and returns. The total introduced tables size decrease around 20% of the figure for BinCFI. Overall, the space overhead is around 125% of the original binary size, 14% lower than the figure for BinCFI. The reduction of tables size contributes to the improvement of space overhead as compared to BinCFI. According to BinCFI's implementation, the hash table size is a power of 2 and is relevant to the number of legitimate targets, so the table size decreases greatly as we refine the legitimate transfer targets size.

6.4.2 Runtime Overhead

We compared the runtime overhead of the programs enforced by BinCC and BinCFI in Figure 13. In our test environment, the results showed that BinCFI increased around 18% percent, while our solution increased around 22% percent, 4% higher than BinCFI. The reason why our overhead is a bit higher than BinCFI is because we add extra instructions to help instrumentation for direct returns, and perform separate address checking and translating. We believe the overhead is reasonable and acceptable while comparing to the finer-grained protection provided to stripped binaries.

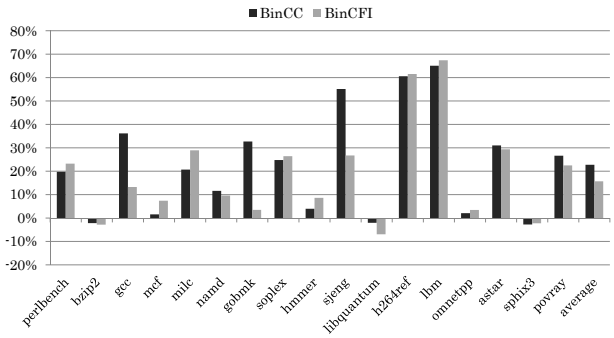

Figure 13: Runtime Overhead for Tested Samples

7. DISCUSSION

Although our solution has made considerable improvements for restricting indirect control flow transfers, the policy for indirect calls still need improvement compared to returns. In our solution, the protected binaries are still vulnerable to attacks altering indirect call targets. Indirect call targets are permitted to reach any ICF, and it would be considered as valid that if the target of an indirect call is modified to another ICF in the binary. However, other binary only CFI solutions face the same problem. Existing solutions [11, 18] make improvement in protecting specific indirect call targets such as C++ virtual functions. It is not hard for BinCC to combine them to achieve more strict protection. We leave this as our future work.

Dynamic code such as JIT code could not be identified simply through static analysis without source code, so our solution can not handle this code. This is also a common open issue with other current binary only CFI solutions. We leave it as our future work.

8. CONCLUSION

Existing binary-only CFI solutions apply relaxed CFI policies due to the lack of source code or debug symbols. Although they can mitigate common control flow hijack threats, it is still possible for adversaries to launch sophisticated attacks, for instance, ROP attacks launched by leveraging call preceded gadgets.

In this paper, we propose a new binary only CFI protection scheme BinCC, which provides finer-grained protection for x86 stripped ELF binaries. Through code duplication and static analysis, we divide the code into mutually exclusive code continents. We further classify indirect transfers in each code continent to be either Intra-Continent transfers or Inter-Continent transfers, and apply strict CFI polices to constrain these transfers. To evaluate BinCC, we introduce new metrics to estimate the average amount of legitimate targets of each kind of indirect transfer as well as the difficulty to leverage call preceded gadgets to generate exploits, and make comparisons with BinCFI. The experiments show that BinCC makes great improvement on both aspects with a reasonable overhead.

9. ACKNOWLEDGMENTS

We would like to thank anonymous reviewers for their feedback in finalizing this paper. This work was achieved when Minghua Wang was at Syracuse University as a visiting student. This research was supported in part by National Science Foundation Grant #1054605, Air Force Research Lab Grant #FA8750-15-2-0106, the Major State Basic Research Development Program of China Grant #2012CB315804, the National Natural Science Foundation of China Grant #91418206, and China Scholarship Council (CSC). Any opinions, findings, and conclusions made in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

10. REFERENCES

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):4, 2009.

[2] S. Andersen and V. Abella. Data execution prevention. changes to functionality in microsoft windows xp service pack 2, part 3: Memory protection technologies, 2004.

[3] T. Bletsch, X. Jiang, and V. Freeh. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 353–362. ACM, 2011.

[4] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40. ACM, 2011.

[5] N. Carlini and D. Wagner. Rop is still dangerous: Breaking modern defenses. In *USENIX Security Symposium*, 2014.

[6] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng. Ropecker: A generic and practical approach for defending against rop attacks. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[7] L. Davi, A.-R. Sadeghi, and M. Winandy. Ropdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 40–51. ACM, 2011.

[8] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. Xfi: Software guards for system address spaces. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 75–88. USENIX Association, 2006.

[9] I. Evans, S. Fingeret, J. González, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. Missing the point (er): On the effectiveness of code pointer integrity1. In *Security and Privacy (SP)*, 2015.

[10] E. Goktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 575–589. IEEE, 2014.

[11] D. Jang, Z. Tatlock, and S. Lerner. Safedispatch: Securing c++ virtual calls from memory corruption attacks. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[12] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[13] S. McCamant and G. Morrisett. Evaluating sfi for a cisc architecture. In *Usenix Security*, page 15, 2006.

[14] V. Mohan, P. Larsen, S. Brunthaler, K. Hamlen, and M. Franz. Opaque control-flow integrity. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[15] B. Niu and G. Tan. Modular control-flow integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 58. ACM, 2014.

[16] B. Niu and G. Tan. Rockjit: Securing just-in-time compilation using modular control-flow integrity. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1317–1328. ACM, 2014.

[17] M. Payer, A. Barresi, and T. R. Gross. Fine-grained control-flow integrity through binary hardening. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2015.

[18] A. Prakash, X. Hu, and H. Yin. vfguard: Strict protection for virtual function calls in cots c++ binaries. In *Network and Distributed System Security Symposium, NDSS*, volume 15, 2015.

[19] A. Prakash, H. Yin, and Z. Liang. Enforcing system-wide control flow integrity for exploit detection and diagnosis. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 311–322. ACM, 2013.

[20] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):2, 2012.

[21] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561. ACM, 2007.

[22] P. Team. Pax address space layout randomization, 2003.

[23] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in gcc & llvm. In *USENIX Security Symposium*, 2014.

[24] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *ACM SIGOPS Operating Systems Review*, volume 27, pages 203–216. ACM, 1994.

[25] 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 *Security and Privacy, 2009 30th IEEE Symposium on*, pages 79–93. IEEE, 2009.

[26] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 559–573. IEEE, 2013.

[27] M. Zhang and R. Sekar. Control flow integrity for cots binaries. In *Usenix Security*, pages 337–352, 2013.