Control Flow Integrity Enforcement with Dynamic Code Optimization

Yan Lin[1], Xiaoxiao Tang[1], Debin Gao[1], and Jianming Fu[2]

[1] School of Information Systems, Singapore Management University, Singapore
[2] Computer School, Wuhan University, China

Abstract. Control Flow Integrity (CFI) is an attractive security property with which most injected and code reuse attacks can be defeated, including advanced attacking techniques like Return-Oriented Programming (ROP). However, comprehensive enforcement of CFI is expensive due to additional supports needed (e.g., compiler support and presence of relocation or debug information) and performance overhead. Recent research has been trying to strike the balance among reasonable approximation of the CFI properties, minimal additional supports needed, and acceptable performance. We investigate existing dynamic code optimization techniques and find that they provide an architecture on which CFI can be enforced effectively and efficiently. In this paper, we propose and implement *DynCFI* that enforces security policies on a well established dynamic optimizer and show that it provides comparable CFI properties with existing CFI implementations while lowering the overall performance overhead from 28.6% to 14.8%. We further perform comprehensive evaluations and shed light on the exact amount of savings contributed by the various components of the dynamic optimizer including basic block cache, trace cache, branch prediction, and indirect branch lookup.

Keywords: Control flow integrity, return-oriented programming, dynamic code optimization

1 Introduction

Control Flow Integrity (CFI) introduced by Abadi et al. [2] provides attractive security features because of its effectiveness in defending against most injected and code reuse attacks, including the recent and advanced attacking techniques like Return-Oriented Programming (ROP) [22]. Its basic idea is to enforce a control-flow graph (usually built from static analysis) so that the program only makes control transfers to intended target locations.

However, having an accurate and practical enforcement of CFI is known to be hard [2, 13, 18, 25]. First, it is generally difficult to accurately identify the target locations for all control transfers. Existing solutions typically apply a coarse-grained policy (e.g., to allow indirect calls to any functions [24]) or require compiler support or presence of relocation or debug information [4,19,25], which may not be applicable to Commercial Off-The-Shelf (COTS) software.

Second, intercepting control transfers and doing the necessary checking typically result in large performance overhead [7,10]. Many have proposed ways of striking the balance among reasonable approximation of the CFI properties, minimizing additional supports needed, and acceptable performance [20, 26]. Therefore, any noticeable reduction in the performance overhead would likely lead to more practical implementation and potentially better security properties.

An interesting observation is that prior to the introduction of CFI in 2005, there have already been a lot of research on dynamic code optimization to improve performance of dynamic program interpreters, e.g., Wiggins/Redstone [11], Dynamo [3], Mojo [8], and DynamoRIO [5]. Dynamo and DynamoRIO are among the more popular and mature ones. Dynamo targets a PA-RISC machine and uses a speculative scheme MRET (Most Recently Executed Tail) to pick hot traces without doing any path or branch profiling. DynamoRIO uses the same scheme to pick hot traces, except that it targets the x86-64 system. Although most of these were not proposed by the security community, there is at least one noticeable work called *program shepherding* [15] which makes use of a general purpose dynamic optimizer RIO [5] to enforce security policies. DynamoRIO and program shepherding provide nice interfaces for enforcing security policies on control transfers, which makes us believe that they can be good candidate architectures for CFI enforcement. Since these well established and mature dynamic code optimizers are proven to introduce minimal overhead, we believe that they could result in a system that outperforms existing CFI implementations.

In this paper, we propose *DynCFI* that enforces a set of security policies on top of DynamoRIO for CFI properties. We show that *DynCFI* achieves similar security properties when compared to a number of existing CFI implementations while experiencing a much lower performance overhead of 14.8% as opposed to 28.6% of *BinCFI*. We stress that *DynCFI* is not necessarily a CFI enforcement implementation that has the lowest performance overhead. Instead, our contribution lies on the utilization of the dynamic code optimization system which is a matured system proposed and well studied before CFI was even introduced. To the best of our knowledge, *DynCFI* is the first implementation of CFI enforcement on top of a dynamic code optimizer.

In the second half of this paper, we further investigate the exact contribution to this performance improvement. We propose a three-dimensional design space and perform comprehensive experiments to evaluate the contribution of each axis in the design space in terms of performance overhead. Among many interesting findings, we show that traces in the dynamic optimizer, which consist of cached basic blocks stitched together, had contributed the most performance improvement. Results show that traces have decreased the performance overhead from 22.7% to 14.8%. We also evaluate how branch prediction and indirect branch lookup have changed the performance. To the best of our knowledge, this is the first comprehensive evaluation on the performance overhead contributed by various components of the system, and we believe that this detailed understanding would aid future research and development of efficient CFI enforcement systems.

The remainder of this paper is structured as follows. Section 2 summarizes related work and outlines our motivation of using a dynamic optimizer. Section 3 introduces the security policies of *DynCFI* we enforce on top of DynamoRIO and compares them with a number of existing CFI enforcement implementations. In Section 4, we propose a three-dimensional design space for *DynCFI* and present a set of experiments to evaluate the contributing factors of various components of the dynamic optimizer. We present our security evaluation and some discussion in Section 5. In the end, we conclude in Section 6.

2 Related Work and Motivation

In this section, we first cover some important related work on CFI and dynamic code optimization, and then motivate our idea of enforcing CFI on top of one of the most well-established dynamic optimizers.

2.1 Control flow integrity

Control-flow Integrity (CFI) was first introduced by Abadi et al. [2]. The basic idea of CFI is to mark the valid targets of indirect branches with unique identifiers and then insert ID-checks into the program before each indirect branch transfer. Since its introduction in 2005, there have been a large body of CFI variants introduced [4, 10, 12, 20, 24–26].

Some of these proposals focus on extracting accurate targets of indirect transfers. For example, CFL [4] requires recompilation of the target application to obtain such target information, and performs a "lock" operation before each indirect control flow transfer with a corresponding "unlock" operation at valid destinations only. ROPdefender [10] makes use of the dynamic binary instrumentation tool Pin [16] to implement a shadow stack where the return addresses are recorded and later compared with the return target address executed. It suffers from performance issues due to its checking for every return instruction executed. CFIMon [24] makes use of BTS [14] supported by hardware to collect in-flight branch transfers. Once the BTS buffer is full, a monitor process will start to detect whether these branch transfers are valid. However, BTS is a debugging mechanism that records all branches in a user-defined memory area, and there will be high overhead because of the large number of memory accesses. In BinCFI [26], potential candidates of indirect branch targets are recorded and all indirect branches are instrumented to be a jump to a CFI validation routine. BinCFI will cause high performance overhead as it has to translate all indirect branch targets executed, especially for programs which have a large percentage of indirect branches. *Lockdown* [17] is implemented in a dynamic binary translation platform called *libdetox*, which also uses shadow stack similar to ROPdefender to restrict the targets of return branches. However, its security policy for indirect jumps is relatively weak in allowing the target of a jump instruction to be any function entry points or any addresses inside the current function. This gives a lot of flexibility to attackers in using various gadgets.

Others focus on efficient ways of enforcing the CFI property for lower performance overhead. For example, in CCFIR [25], all control flow targets for indirect branches are allocated on a so-called springboard section, and indirect branches are only allowed to use control flow targets contained in the springboard section. The main restriction is that it requires relocation information to be included in the binaries. kBouncer [20] uses LBR [14] on Intel to record branch transfers. It checks whether the target of a return instruction is call-proceeded when a system call is invoked. It can be bypassed because the LBR mechanism only records limited number of branch transfers. ROPGuard [12] also performs CFI validation on Windows API calls. Like kBouncer, it requires that return addresses are call-preceded and the memory word before each return address is the start address of the API function.

In general, all existing proposals of CFI implementation enforce an approximation of the original and strict security policies due to the lack of accurate indirect transfer target information and performance considerations. Many have to trade security for better performance of the resulting system. Research has shown that some of these approximated CFI implementation are vulnerable to various attacks [6,13,21]. Therefore, any noticeable reduction in the performance overhead not only would lead to better user acceptance, but might translate into a better approximation of the CFI security policy.

2.2 Dynamic code optimization

We notice that another body of work called dynamic code optimization, mostly done by the software engineering community, could potentially be useful for improving the performance overhead. Most of them build hot traces for blocks frequently executed to boost execution. Dynamo [3], a dynamic optimizer for a PA-RISC machine, acts as a native interpreter which allows it to observe runtime behavior without instrumentation. Wiggins/Redstone [11] uses performance counters on the Alpha to build traces. Mojo [8] uses the same mechanism in Dynamo to pick hot traces and targets Windows NT running on IA-32. DynamoRIO [5] is an x86 system based on Dynamo. Some of these platforms provide nice interfaces of intercepting control flow transfers of the target program with very low overhead, e.g., DynamoRIO [5], to the extent that the overhead could be negative (performance improvement) for some situations. Such platforms could be perfect candidates on top of which CFI properties are enforced.

We are not the first to make use of such systems for security purposes. Program Shepherding [15] successfully makes use of DynamoRIO to restrict code origins and control transfers. DynamoRIO provides a suitable platform for security enforcement because the sandboxing checks added cannot be bypassed [15]. Due to this reason and the fact that it provides efficient interfaces of intercepting control flow transfers, we choose it for our CFI enforcement, too.

2.3 DynamoRIO

Figure 1 shows an overview of *DynamoRIO* [5], with darker shading indicating the application code to be monitored.

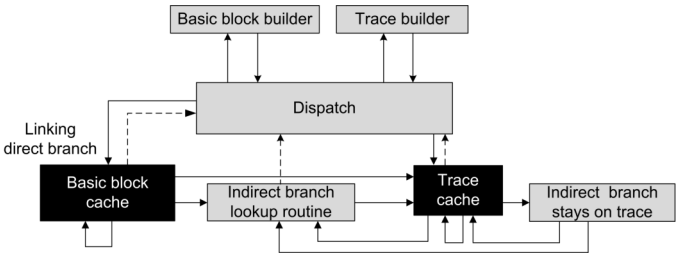

Fig. 1: Overview of *DynamoRIO*

DynamoRIO first copies basic blocks into the basic block cache. If a target basic block is present in the code cache and is targeted via a direct branch, *DynamoRIO* links the two blocks together with a direct jump. If the basic block is targeted via an indirect branch, *DynamoRIO* goes to the indirect branch lookup routine to translate its target address to the code cache address. Basic blocks that are frequently executed in a sequence are stitched together into the trace cache. When connecting beyond a basic block that ends in an indirect branch, a check is inserted to ensure that the actual target of the branch will stay on the trace. If the check fails, it will go to the indirect branch lookup routine to find the translated address.

To make itself a secure platform on which programs are executed, *DynamoRIO* splits the user-space address into two modes: the untrusted application mode and the trusted and protected RIO mode. This design protects *DynamoRIO* against memory corruption attacks. Meanwhile, the beauty of *DynamoRIO* (and the corresponding good performance) come mainly from the indirect branch lookup which is very efficient in determining control transfer targets with a hashtable. This hashtable maps the original target addresses with addresses in the basic block cache and trace cache so that most control transfers require minimal processing. We delay further details of *DynamoRIO* to Section 3 and Section 4 when we explain policies to be enforced on top of it and when we evaluate the improved performance achieved by individual components of *DynamoRIO*.

3 Design, Implementation, and Security Comparison

As discussed in Section 1, our motivation is to use *DynamoRIO* to enforce CFI properties in anticipation for improved performance. Our objective is to design a practical and efficient CFI enforcement without the extra requirement of recompilation or dependency on debug information. In this section, we first present the design of *DynCFI* that can be effectively enforced on *DynamoRIO* and the

implementation of it, and then compare the security property it achieves with some existing CFI (and related defense) approaches.

3.1 Returns

The most frequently executed indirect control transfer instructions are returns. *DynCFI* maintains a shadow call stack for each thread to remember caller information and the corresponding return address. The whole process is shown in Figure 2. For a call instruction, we store the return address on our shadow stack. For a return instruction, we check whether the address on the shadow stack equals to the address stored at the stack memory specified by %esp. Such a shadow stack enables *DynCFI* to apply a strict policy that only returning to the caller is allowed, although a relaxed version could also be applied to reduce overhead (see Section 3.4 for more discussion). *DynCFI* also takes care of the following exceptions in special cases.

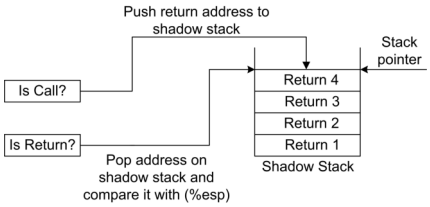

Fig. 2: Shadow stack operations

- **Signals** A signal comes with a return address but not a call instruction. Fortunately, *DynamoRIO* records all necessary signal information for us to maintain a correct shadow stack.
- **Lazy binding** The procedure `dl_runtime_resolve()` in lazy binding uses `ret` (without a corresponding `call`) to perform a `jmp` operation. The pattern of the code is fairly easy to identify though.
- **setjmp and longjmp** `setjmp` and `longjmp` allow bypassing of multiple stack frames. We `pop` out return addresses continuously until a match is found or when the shadow stack is empty.
- **C++ exception handling** We use the second argument of `Unwind_SetIP` as the return address for proper enforcement of our policy.

3.2 Indirect jumps and indirect calls

We further classify indirect jumps into normal indirect jumps and PLT jumps, such as `jmp offset (base_register)`, which are used to call functions in other modules, target of which can only be exported from other modules. To obtain target information for every indirect branch, we use the static analysis engine provided by another well-known CFI enforcement *BinCFI* [26], which combines

linear and recursive disassembling techniques and uses static analysis results to ensure correct disassembling. Targets of indirect calls are function entry points and targets of indirect jumps are function entry points and targets of returns. Meanwhile, targets of PLT jumps are exported symbol addresses. These valid jump and call targets are organized into three different hashtables to improve performance — one for indirect jumps, one for indirect calls, and one for PLT jumps. Most importantly, the shadow stack and hashtables readable only in the user mode; so attackers cannot modify them.

3.3 Implementation

As discussed in Section 2, *DynamoRIO* maintains a hashtable that maps original control transfer target addresses with addresses of code caches. The hashtable has to be built when the control transfer occurs the first time though. This process, together with the dispatcher which is invoked when matches are not found in the hashtable (see Figure 1), become the natural place of our CFI enforcement, since CFI mainly concerns control transfer targets.

We obtained the source code of *DynamoRIO* version 5.0.0 from the developer's website [1], and added more than 700 lines of code (in C) to implement *DynCFI*. Most of the additional code is added to the dispatcher where checks of control flow transfers are performed. Some code is also added to basic block cache building to implement our shadow call stack and to initialize *DynamoRIO* to load the valid jump/call target addresses into our own hashtables.

DynCFI does not implement the full sets of CFI properties originally proposed by Abadi et al. [2]. We only perform checks on indirect control transfers at the first time when the target of an indirect branch occurs. However, it does not really impact security, and it is exactly the reason why *DynamoRIO* is widely accepted as an efficient dynamic optimizer — original code is cached in short sequences and security policies, if any, need only be checked the first time the code cache is executed [5]. Subsequent executions of the same code cache will be allowed (without checking) as long as the control transfer targets remain unchanged. Any violations to our policy will miss the (very efficient) indirect branch hashtable lookup and go back to the dynamic interpreter which will consider the control transfer a first timer and perform all the checks (inefficient).

3.4 Security comparison

Table 1 shows the security policy of *DynCFI* when compared with some existing CFI implementations and ROP defense solutions. A caveat here is that we make use of the shadow call stack information only when a new target is added to the hashtable. This will make the policy effectively call-proceeded only. Since call-proceeded policy is widely considered as adequate by many other approaches, we apply this performance improvement in our subsequent evaluation. This relaxed policy also enables a fair comparison between *DynCFI* and other CFI enforcement schemes since many others also use a call-proceeded policy.

Table 1: Security comparison with other CFI and ROP defenses

Approach	Policy			
	Return	Indirect jump	Indirect call	PLT jump
BinCFI [26]	Call-preceded	Function entry,return address	Function entry	Exported symbol address
CCFIR [25]	Corresponding springboard section			Nil
CFIMon [24]	Call-preceded	Any address in the training set	Any function entry	Nil
ROPdefender [10]	Caller	Nil	Nil	Nil
kBouncer [20]	Call-preceded	Nil	Nil	Nil
LockDown [17]	Caller	Function entry, instruction in the current function	Function entry	Nil
DynCFI	First execution: Caller, Others: Call-preceded	Function entry,return address	Function entry	Exported symbol address

DynCFI achieves similar security when compared with these existing approaches. In particular, *DynCFI* is mostly comparable to *BinCFI* in that both maintain a list of valid target addresses to be checked at runtime, with one noticeable difference in the enforcement mechanism: *BinCFI* enforces the policies with static instrumentation to translate indirect target address while *DynCFI* uses *DynamoRIO* as the interpreter platform. This makes *BinCFI* the perfect candidate for performance overhead comparison with *DynCFI*, which is the topic of our next Section.

4 Detailed Performance Profiling

In this section, we conduct a comprehensive set of experiments on the performance overhead of *DynCFI*. Besides the overall performance overhead, we run some detailed performance profiling to find out the contribution to such overhead by various components of the dynamic optimizer. We wish that such a detailed profiling could shed light on the part that contributes most to the performance overhead, and give guidance to future research in further improvement.

To better understand our evaluation strategy, we present our first attempt in the profiling, show the results, and explain the limitation of this attempt. We then choose an existing CFI implementation for the detailed comparison with *DynCFI*. We analyze the design space of CFI enforcement implementation and organize it along three axes on which the two systems under comparison could be clearly identified. Lastly, we perform a sequence of experiments by modifying individual components of *DynCFI* so that the contribution of each to performance overhead can be evaluated.

4.1 Target applications

To evaluate the performance overhead, we need to subject *DynCFI* (and another CFI implementation for comparison purposes) to some applications. To enable

fair comparison with existing work, we used twelve pure C/C++ programs we can find in SPEC CPU2006, which are also used in the evaluation of the original work of *BinCFI* [26], as our benchmarking suite.

Experiments were executed on a desktop computer with an i7 4510u CPU and 8GB of memory running x86 version of Ubuntu 12.04. Each individual experiment was conducted 10 times, average of which is reported in this paper.

4.2 First attempt in performance profiling

As an initial attempt to understanding the performance overhead contributed by various components of *DynCFI*, we use program counter sampling to record the amount of time spent in various components of *DynCFI*. We use the `ITIMER_VIRTUAL` timer which counts down only when the process is executing and delivers a signal when it expires. The handler used for this signal records the program counter of the process at the time the signal is delivered. We sample the program counter every ten milliseconds.

Table 2: Percentage of time spent on various components

Application	Application code	IBL inlined	IBL not inlined	Basic block building	Trace building	Dispatch	Others
`bzip2`	97.99	0.60	0.00	0.20	1.20	0.00	0.00
`gcc`	86.78	7.46	0.26	0.91	3.42	1.10	0.07
`mcf`	97.48	0.42	1.26	0.14	0.07	0.14	0.49
`gobmk`	80.00	1.08	0.00	2.70	11.35	4.86	0.00
`sjeng`	94.10	5.67	0.11	0.02	0.09	0.02	0.00
`libquantum`	99.51	0.49	0.00	0.00	0.00	0.00	0.00
`omnetpp`	84.88	14.50	0.38	0.06	0.15	0.03	0.01
`astar`	94.36	4.79	0.78	0.00	0.01	0.04	0.01
`namd`	99.89	0.69	0.00	0.00	0.02	0.00	0.00
`soplex`	74.21	25.42	0.03	0.10	0.10	0.10	0.02
`povray`	89.71	6.88	0.82	0.76	1.01	0.76	0.06
`lbm`	99.99	0.00	0.00	0.00	0.01	0.00	0.00
Average	91.57	5.62	0.30	0.41	1.45	0.59	0.06

Table 2 shows the percentage of time each application spends in various steps in *DynCFI*. It suggests that more than 90% of the time is spent on the application's code on average. Other non-negligible processes include Indirect Branch Lookup (IBL) inlined with the application's code and that not inlined, basic block and trace cache building, as well as the dispatcher.

In an attempt to explain why some applications, e.g., `gcc`, `omnetpp`, `soplex`, and `povray`, incur larger overhead, we count the number of different control transfers in each application (runtime) and present statistics in Table 3. The correlation between the two tables suggests that larger number of control transfers could lead to the higher overhead.

Although it sounds like we have obtained detailed understanding of the performance overhead, there is one important factor that we have overlooked so far — the overhead contribution of the dynamic optimizer on executing the application's code (second column of Table 2). In other words, Table 2 does not tell us if the dynamic optimizer had sped up or slowed down the execution of

Table 3: Statistics of different types of control transfers

Application	%Indirect call	%Indirect jump	%Return	%Direct branch	Total
bzip2	0.002	0.002	0.774	99.222	2813437750
gcc	0.434	1.958	7.767	89.841	40789466606
mcf	0.001	0.029	5.402	94.568	5000155956
gobmk	0.001	0.027	4.811	95.161	687830197
sjeng	1.072	2.289	4.718	91.921	122978889385
libquantum	0.000	0.000	0.242	99.758	706839248554
omnetpp	1.609	1.763	33.998	62.630	87535408451
astar	1.698	0.049	19.738	78.515	30621019276
namd	0.000	0.008	3.292	96.700	115933566091
soplex	0.002	0.018	23.239	76.741	73160950993
povray	2.776	0.154	26.279	70.791	8195937460
lbm	0.000	0.017	0.035	99.948	15270883768

the application's code, and what had contributed to that speedup or slowdown. Our further comparison verifies this suspicion, see Table 4, as there is noticeable difference in the amount of time spent.

Table 4: Time spent in application code

Application	in *DynCFI* (sec)	Natively (sec)	Overhead (%)
bzip2	4.88	4.86	0.41
gcc	60.73	56.25	7.96
mcf	13.91	14.19	-1.97
gobmk	1.48	1.35	9.62
sjeng	158.93	150.01	5.95
libquantum	813.12	821.63	-1.04
omnetpp	138.54	122.23	13.34
astar	76.16	75.44	0.95
namd	735.51	733.73	0.24
soplex	64.81	61.15	5.98
povray	14.21	14.12	0.64
lbm	375.45	388.14	-3.27

Therefore, we want to further investigate the contribution of various components of the dynamic optimizer in speeding up or slowing down the application's code. We present our second attempt in the rest of this section. With the objective of finding out contributions to the performance overhead by individual components of the dynamic optimizer, our strategy is to

1. Find an existing CFI implementation X for comparison.
2. Continuously disable or modify individual components of *DynCFI* so that the modified system eventually becomes similar to the implementation of X.
3. In every step of disabling or modifying the components, perform experiments to find the corresponding (difference in) performance overhead.

4.3 Picking BinCFI for detailed comparison

With this strategy, it is important that we choose an X that

– Is an independent, state-of-the-art implementation of CFI enforcement;

– Shares the same high-level idea with *DynCFI* while validating control transfers with a different approach (e.g., by binary instrumentation) from that of the dynamic optimizer/interpreter as in *DynCFI*.

so that our evaluation could attribute the difference in performance overhead to the dynamic optimizer.

As discussed at the end of Section 3, *BinCFI* and *DynCFI* are similar in that both maintain a set of valid control transfer targets and use a centralized validation routine for CFI enforcement. In both cases, the validation routine maintains a hashtable for the valid control transfer targets. Figure 3 shows the work-flow of *BinCFI*.

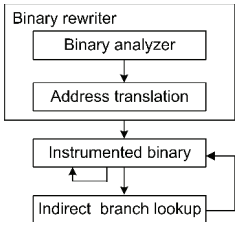

Fig. 3: Overview of *BinCFI*

The difference between *BinCFI* and *DynCFI* is that *BinCFI* obtains the valid target addresses of indirect branches statically and records their corresponding instrumented target addresses into the hashtable, and then replaces the indirect instructions with a direct jump to the CFI validation routine. *BinCFI* satisfies our requirements for the performance comparison, and is therefore chosen for our subsequent detailed evaluation.

4.4 Overall comparison and the design space

The overall performance overhead of executing the benchmarking applications under (original, unmodified) *DynamoRIO*, *DynCFI*, and (original, unmodified) *BinCFI* is shown in Figure 4. Results are shown in terms of percentage overhead beyond natively executing the applications on an unmodified Linux Ubuntu system. We obtained the source code implementation of *BinCFI* [26] from its authors.

An interesting observation is that the original *DynamoRIO* and *DynCFI* do not differ much in terms of overhead (a relatively small 1.3% difference). This shows that the interfaces provided by *DynamoRIO* are convenient and effective for CFI enforcement, which confirms our intuition since *DynamoRIO* intercepts all control transfers and no additional intercepting is needed in our modification to *DynamoRIO*.

DynCFI experiences a significantly smaller overhead of 14.8% compared to *BinCFI* at 28.6%. This suggests that the dynamic optimizer provides a more efficient platform for CFI enforcement compared to existing approaches like binary instrumentation as in *BinCFI*. That said, the two systems differ in other

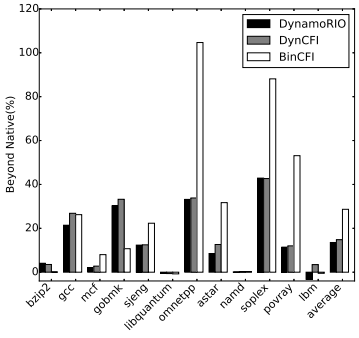

Fig. 4: Overall performance overhead

aspects and therefore this overall evaluation result is insufficient in attributing the majority of the performance gain to mechanisms of the dynamic optimizer.

As discussed in Section 4.3, our strategy to this difficulty is to continuously disable or modify individual components of *DynCFI* so that eventually it becomes similar to *BinCFI*, in terms of their operating mechanism as well as the performance overhead. By doing so, we would likely observe degradation of performance (increase in overhead) of the modified system which is definitely due to the corresponding feature disabled or modified. The question is – which individual component or feature to disable or modified?

To answer this question, we analyze the internal validation mechanisms of the two approaches and identify three main factors that could significantly contribute to the different performance overhead.

1. **Trace** Trace is the most important mechanism in *DynamoRIO* to speed up indirect transfers. Traces are formed by stitching together basic blocks that are frequently executed in a sequence. Benefits include avoiding indirect branch lookups by inlining a popular target of an indirect branch into a trace (with a check to ensure that the target stays on the trace and otherwise fall back to the full security check), eliminating inter-block branches, and helping branch prediction. Trace is unique in *DynamoRIO* and is not in *BinCFI*.
2. **Branch prediction** Modern processors maintain buffers for branch prediction, e.g., Branch Target Buffer (BTB) and Return Stack Buffer (RSB). The effectiveness of these predictors could get seriously affected due to the modifications to the control transfers. For example, turning a return instruction into a indirect jump would make RSB useless in the branch prediction, potentially leading to an increase in the performance overhead.
3. **Indirect branch lookup routine** Besides implementation details that are not necessarily due to the architectural design (to be discussed more in Section 4.5), a dynamic optimizer could use a single lookup routine for the entire application including the dynamically loaded libraries, while systems that apply static analysis and binary instrumentation would likely have to use a dedicated lookup routine for each module because some dynamically loaded libraries might not have been statically analyzed or instrumented. This could contribute to noticeable differences in performance overhead.

We want to explore details into these three axes to see how each of them affects the performance overhead. Other factors that might contribute to the overhead in *DynCFI* which we do not further investigate include

- Building basic block caches;
- Building trace caches;
- Inserting new entries into hashtables;
- Context switches between DynamoRIO and code caches.

4.5 Profiling along the three axes

With identification of the three axes, we make our second attempt in detailed understanding of the performance overhead of the two systems. Since executing on *DynCFI* and executing on the original unmodified *DynamoRIO* experience about the same overhead (see Figure 4), our subsequent experiments will only focus on comparing *DynCFI* and *BinCFI*. Also recall that our strategy is to disable or modify one component of *DynCFI* at a time and observe the corresponding change in performance overhead.

4.5.1 Traces Traces are unique in dynamic optimizers like *DynamoRIO* and *DynCFI*. There are potentially two ways in which traces impact the performance overhead. First, the stitching of basic blocks together eliminates some inter-block branches. Second, each trace has inlined code to check if the control transfer target is still on the trace (we call this InT). If the target is still on the trace, execution will just carry on without further checking; otherwise, a second inlined code (we call this InH) is executed to perform hashtable lookup without collisions. If collision happens, execution will go to the full indirect branch lookup routine (denoted as R). We examine contribution of InT and InH by disabling them individually. We also examine the effect of traces overall and present the results in Figure 5.

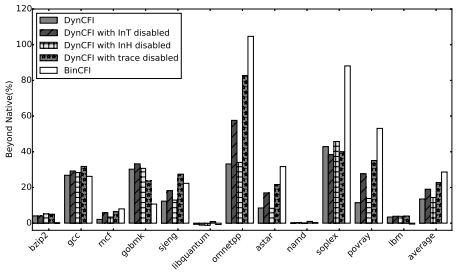

Fig. 5: Impact of trace on overhead

Figure 5 shows that the contribution due to InT is big, averaging to 5.5%. Exceptions go to `bzip2` and `soplex` which do not gain much with InT mainly because the fall-back of InH is very effective on them (which can be verified from the next-to-zero time spent in IBL not inlined in Table 2).

Although performance overhead increases when disabling InT (see Figure 5), *DynCFI* is still better than *BinCFI*. When disabling traces altogether, the overhead of *DynCFI* increases from 14.8% to 22.7% on average, with some going over the overhead in *BinCFI*. This shows that traces are contributing significantly in the low overhead of *DynCFI*. For applications with a large percentage of indirect branches (see Table 3), *DynCFI* with traces disabled still outperforms *BinCFI*. This suggests that there are other contributing factors in *DynCFI* which we have not evaluated.

4.5.2 Branch Prediction The way in which *DynCFI* and *BinCFI* intercept and deliver control flow transfers has an implicit effect on branch prediction. Branch prediction is typically achieved by remembering a history of control transfer targets by the same instruction. Both *DynCFI* and *BinCFI* could weaken branch prediction due to R using the same instruction (an indirect jump) to execute control transfers originally executed by different instructions in the application [5, 26]. Table 5 summaries how indirect control transfers in an application are executed in *DynCFI* and *BinCFI*.

Table 5: Execution of indirect control transfers

Original transfer		Return	Indirect call/jump
DynCFI	Basic block cache	Jump to R, indirect jump to target	
	Trace cache	InT or InH or jump to R, indirect jump to target	
BinCFI		Return	jump to R, indirect jump to target

In summary, *DynCFI* leads *BinCFI* in retaining branch prediction for indirect calls and jumps when trace caches are used due to InT and InH; however, *BinCFI* would perform better than *DynCFI* for returns. That said, note that there are typically far more return instructions than indirect calls and jumps executed for all the applications in our benchmarking suite, see Table 3.

To better understand the effect of various components of *DynCFI* and *BinCFI* on branch prediction, we count the number of mispredictions when executing the benchmarking applications on a number of different settings – *DynCFI*, *DynCFI* with InT disabled, *DynCFI* with InH disabled, *DynCFI* with traces disabled, *BinCFI*, *BinCFI* with returns being replaced by jumps to R, and present the results in Figure 6.

We observe that disabling InH has a larger impact on branch prediction than disabling InT in general. This shows that the inlined hashtable lookup has its fair share of its contribution on lower overhead. It also indirectly shows that the hashtable implementation in *DynCFI* is good in that collisions do not happen often (since R not inlined is not executed often as shown in Table 2). Another interesting finding is that replacing returns with indirect jumps on *BinCFI* adds a large number of mispredictions for some programs. In terms of overhead, this translates to about 2% more in the overhead as shown in Figure 7.

4.5.3 Indirect branch lookup routine R The indirect branch lookup routine in *DynCFI* and *BinCFI* very much shares the same strategy. Both use an

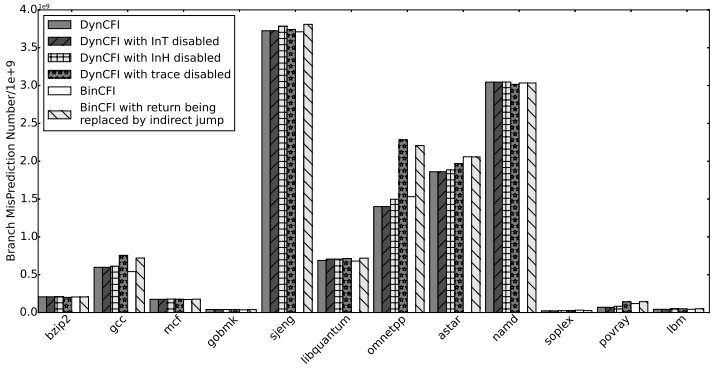

Fig. 6: Impact of traces on the number of branch mispredictions

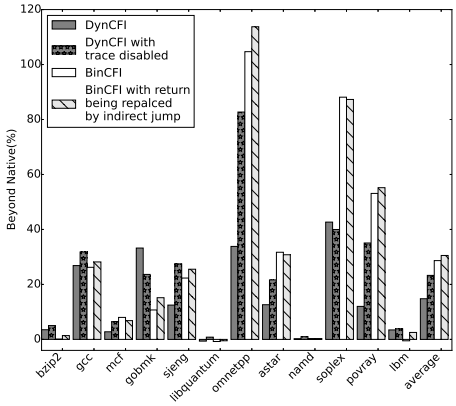

Fig. 7: Impact of branch prediction on overhead

efficient implementation of a hashtable to record valid control transfer targets. One noticeable difference, though, is that *BinCFI* requires an extra step to check if the target resides within the same software module before directing control to the corresponding R. Each software module has to implement its own copy of R because some dynamically loaded libraries might not have been statically analyzed or instrumented and *BinCFI* cannot use a centralized R for all modules.

On the other hand, *DynCFI* executes the application on top of a dynamic interpreter without static analysis or binary instrumentation, and therefore has three centralized R (one for returns, one for indirect jumps, and one for indirect calls) for all software modules. This architectural difference contributes to some additional performance overhead to *BinCFI*.

Besides the difference due to the architectural design, there are also lower level differences in implementing R between *DynCFI* (inheriting the same R from *DynamoRIO*) and *BinCFI*. In particular, they differ in the indirect jump instructions used (*DynCFI* uses a register to specify the target while *BinCFI* uses a memory), the number of registers used throughout the algorithm (and as

a result the number of registers to be saved and restored), and efficiency of the hashtable lookup algorithm.

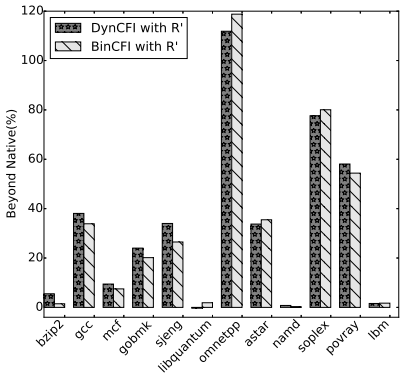

Fig. 8: Performance Overhead with unified R'

To evaluate the contribution of R in the overall performance overhead, we replace R in both *DynCFI* (with traces disabled) and *BinCFI* (with returned replaced with indirect jumps) with R', our (supposedly more efficient) implementation of the algorithm, and show the resulting performance overhead in Figure 8.

Comparing these results with those shown in Figure 7, we find that such low-level details in the implementation of R translates to significant differences in the overhead. In particular, the difference between *DynCFI* and *BinCFI* shrinks with R' replacing R, indicating that the original R used in *DynCFI* is more efficient than that in *BinCFI*.

4.5.4 Summary Recall that our strategy in the second attempt of detailed profiling of *DynCFI* is to continuously disable or modify various components to find the contribution of them in terms of performance overhead. Figure 8 shows the comparison between *DynCFI* with traces disabled (bringing both systems to the same configuration on the first axis) and *BinCFI* with returns replaced by indirect jumps (bringing both systems to the same configuration on the second axis) while they use the same R' (bringing both systems to the same configuration on the third axis). They are fairly close to each other in their performance, confirming that we manage to attribute their originally large difference being successfully attributed to the three axes.

We therefore believe that this second attempt provides a successful and accurate detailed profiling for *DynCFI* and *BinCFI*. With the detailed understanding of the contribution of each components on the three axes, we hope that future research could improve the performance further by, e.g., designing an indirect branch lookup routine that results in better branch prediction.

5 Security evaluation and discussions

5.1 Real world exploits

We use a publicly available intrusion prevention evaluator RIPE [23] to verify that *DynCFI* offers comparable security properties with existing CFI proposals (as analysis presented in Section 3.4). In particular, we check if *DynCFI* can detect exploits that employ the advanced Return-Oriented Programming (ROP) techniques.

RIPE contains 140 return-to-libc exploits out of which 60 exploit return instructions and 80 exploit indirect call instructions. For the 60 exploits on return instructions, our experiments confirm that *DynCFI* manages to detect all of them because they violate the call-preceded policy we enforced on return instructions. RIPE also contains 10 ROP attacks using return instructions, which are all successfully detected by *DynCFI* as the targets of these gadgets are not call-preceded.

DynCFI and *BinCFI* share the weakness in detecting exploits that change the value of a function pointer to a valid entry point of a function. Such attacks cannot be detected by most other CFI implementations either [24].

5.2 Average indirect target reduction

Zhang and Sekar [26] propose a metric for measuring the strength of CFI called Average Indirect target Reduction (AIR). As *DynCFI* uses different policy on return branches, we apply the same metric to test *DynCFI* when applied to the SPEC benchmarking suite. Table 6 compares the AIR metrics for *DynCFI* and *BinCFI*. We can find that average AIR for *DynCFI* is 98.80% which is comparable to 98.86% for the case of *BinCFI*.

Table 6: AIR metrics for SPEC CPU 2006

Name	DynCFI(%)	BinCFI(%)
`bzip2`	99.95	99.37
`gcc`	97.60	98.34
`mcf`	98.58	99.25
`gobmk`	98.18	99.20
`sjeng`	99.60	99.10
`libquantum`	98.10	98.89
`omnetpp`	99.61	97.68
`astar`	96.70	98.95
`namd`	99.99	99.59
`soplex`	99.49	98.86
`povray`	99.19	98.67
`lbm`	98.56	99.46
Average	98.80	98.86

5.3 Shadow stack in full enforcement

As described in Section 3, in order to improve the performance, we do not check the shadow call stack if the target address is found in our hashtable (in which

all addresses have already been fully checked when they were first added to the hashtable).

We understand that a full enforcement of the shadow call stack is more secure as it ensures that every return jumps to its caller; however, its high performance overhead is also well documented in previous research [9,10]. To verify such high performance overhead, we modify *DynCFI* to check the shadow call stack for every return instruction, and show the results in Figure 9.

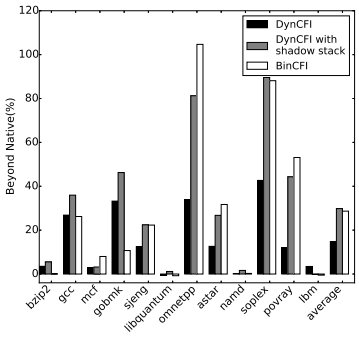

Fig. 9: Performance overhead with shadow stack

Figure 9 shows that *DynCFI* with full enforcement of the shadow stack runs with an average performance overhead of 29.8%, a big jump from our optimized implementation at 14.8%. Although such a full enforcement of the shadow stack takes away the performance advantage of *DynCFI* compared to *BinCFI*, *DynCFI* now offers much better security. We check the AIR metric and find that AIR for *DynCFI* with full enforcement of the shadow stack increases from 98.80% to 99.66% for SPEC CPU2006, which is better than that of *BinCFI* at 98.86%. Our experiments also show that *DynCFI* can now detect some more advanced ROP attacks, e.g., the ROP attack constructed by Goktas et al. [13] using call-preceded gadgets. A call-proceeded-only policy, e.g., that used in *BinCFI*, would miss such advanced attacks.

6 Conclusion

In this paper, we propose *DynCFI*, a new implementation of CFI properties on top of a well-studied dynamic code optimization platform. We show that *DynCFI* achieves comparable CFI security properties with many existing CFI proposals while enjoying much lower performance overhead of 14.8% on average compared to that of a state-of-the-art CFI implementation *BinCFI* at 28.6%. Our detailed profiling of *DynCFI* shows that traces, a mechanism in the dynamic code optimization platform, contribute the most to such performance improvement.

Acknowledgment This work was supported by Research on attack containment using randomization for Cloud Computing No. 61373168 and Research on software behavior model based on user intention No. 20120141110002.

References

1. DynamoRIO. `http://www.dynamorio.org/`.
2. M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.
3. V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. In *ACM SIGPLAN Notices*, volume 35, pages 1–12. ACM, 2000.
4. 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.
5. D. Bruening. *Efficient,Transparent,and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, 2004.
6. N. Carlini and D. Wagner. Rop is Still Dangerous: Breaking Modern Defenses. In *USENIX Security Symposium*, 2014.
7. P. Chen, X. Xing, H. Han, B. Mao, and L. Xie. Efficient Detection of The Return-Oriented Programming Malicious Code. In *Information Systems Security*, pages 140–155. Springer, 2010.
8. W.-K. Chen, S. Lerner, R. Chaiken, and D. M. Gillies. Mojo: A Dynamic Optimization System. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, pages 81–90, 2000.
9. T. H. Dang, P. Maniatis, and D. Wagner. The performance cost of shadow stacks and stack canaries. In *ACM Symposium on Information, Computer and Communications Security, ASIACCS*, volume 15, 2015.
10. 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.
11. D. Deaver, R. Gorton, and N. Rubin. Wiggins/Redstone: An On-line Program Specializer. In *Proceedings of the IEEE Hot Chips XI Conference*, 1999.
12. I. Fratric. Runtime Prevention of Return-Oriented Programming Attacks. *University of Zagreb*, 2012.
13. 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.
14. Intel Corporation. *Intell®64 and IA-32 Architectures Software Developer's Manual*, 2015.
15. V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure Execution via Program Shepherding. In *USENIX Security Symposium*, volume 92, 2002.
16. C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Acm Sigplan Notices*, volume 40, pages 190–200. ACM, 2005.

17. P. Mathias, B. Antonio, and R. Thomas. Fine-grained control-flow integrity through binary hardening, 2015.
18. 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.
19. 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.
20. V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP Exploit Mitigation using Indirect Branch Tracing. In *USENIX Security*, pages 447–462, 2013.
21. F. Schuster, T. Tendyck, J. Pewny, A. Maaß, M. Steegmanns, M. Contag, and T. Holz. Evaluating the Effectiveness of Current Anti-ROP Defenses. In *Research in Attacks, Intrusions and Defenses*, pages 88–108. Springer, 2014.
22. 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.
23. J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen. RIPE: Runtime Intrusion Prevention Evaluator. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 41–50. ACM, 2011.
24. Y. Xia, Y. Liu, H. Chen, and B. Zang. CFIMon: Detecting Violation of Control Flow Integrity using Performance Counters. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2012.
25. 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 *2013 IEEE Symposium on Security and Privacy (SP)*, pages 559–573, 2013.
26. M. Zhang and L. Sekar. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22th USENIX Security Symposium*, pages 337–352, 2013.