BinCFP: Efficient Multi-threaded Binary Code Control Flow Profiling

Jiang Ming and Dinghao Wu
College of Information Sciences & Technology
The Pennsylvania State University, University Park, PA 16802
{jum310, dwu}@ist.psu.edu

***Abstract*—In many tasks of reverse engineering and binary code analysis (e.g., hybrid disassembly, resolving indirect jump, and decoupled taint analysis), the knowledge of detailed dynamic control flow can be of great value. However, the high runtime overhead beset the complete collection of dynamic control flow. The previous efforts on efficient path profiling cannot be directly applied to the obfuscated binary code in which an accurate control flow graph is typically absent. To address these challenges, we present *BinCFP*, an efficient multi-threaded binary code control flow profiling tool by taking advantage of pervasive multi-core platforms. BinCFP relies on dynamic binary instrumentation to work with the unmodified binary code. The key of BinCFP is a multi-threaded fast buffering scheme that supports processing trace buffers asynchronously. To achieve better performance gains, we also apply a set of optimizations to reduce control flow profile size and instrumentation overhead. Our design enables the complete dynamic control flow collection for an obfuscated binary execution. We have implemented BinCFP on top of Pin. The comparative experiments on SPEC2006 and obfuscated common utility programs show BinCFP outperforms the previous work in several ways. In addition, BinCFP's control flow profile sizes are only about 49.2% that of the conventional design.**

***Index Terms*—Control Flow Profiling, Dynamic Binary Instrumentation, Multi-core**

I. Introduction

Dynamic control flow information is typically represented as a sequence of basic blocks executed during run time [1]. In many applications of reverse engineering and binary code analysis, the detailed dynamic control flow can be very handy. For example, control flow information can resolve indirect jump targets, which are a known challenge for binary code static analysis [2]. In this way, we can increase the accuracy of static disassembly [3] and reverse the effect of control flow obfuscation [4]. Moreover, efficiently recording control flow data can facilitate decoupled taint analysis [5]–[7] to improve the overall taint analysis performance. However, the significantly high runtime overhead is a major barrier to the complete collection of dynamic control flow. The performance penalty further slows down the related security analysis tasks. On the other hand, the topic of efficient trace profiling has been well studied in the code generation and optimization area [8]–[10]. Unfortunately, these previous efforts cannot be directly applied to the obfuscated binary code. They either require an accurate control flow graph for path profile encoding or cannot work when continuous control flow information is required. Therefore, an efficient and complete binary code control flow collection tool is necessary to reverse engineering.

In this paper, we present a novel technique, called *BinCFP*, to allow efficient and complete binary code control flow profiling on ubiquitous multi-core platforms. Based on the logged data, we can construct a straight-line code trace for the further offline analysis. BinCFP is built on dynamic binary instrumentation so that it works with unmodified binary code. The core of BinCFP is a multi-threaded fast buffering scheme, which allows the application to continue executing while trace buffers are processed asynchronously. We also adopt an enhanced control flow profile structure to produce compact profile size. In addition, we apply a set of instrumentation optimizations to achieve better performance gains. Furthermore, BinCFP supports multi-threaded applications as well.

We have developed BinCFP on top of Pin [11], a dynamic binary instrumentation platform. BinCFP relies on very lightweight logging of the execution control flow information to reconstruct a straight-line code later. We have evaluated BinCFP on a number of applications such as SPEC2006 and obfuscated common utility programs. The performance experiments show that BinCFP imposes a small impact on application runtime performance and a considerable speedup over the previous approaches. Besides, BinCFP's control flow profile sizes are only about 49.2% that of conventional profiles. With optimization and compression, the profile size can be further lowered. Such experimental evidence shows that BinCFP can be applied for various software security applications. In summary, we make the following contributions:

1) We present BinCFP, an efficient multi-threaded binary code control flow profiling tool. We take advantage of pervasive multi-core platforms to greatly reduce the program execution slowdown. The source code of our tool is available at https://github.com/s3team/bincfp.
2) BinCFP's control flow profile does not require fine-grained static analysis so that BinCFP is fit for analyzing obfuscated binary code.
3) Based on our control flow profiling data, it is straightforward to construct a straight-line code trace for the further offline analysis. BinCFP has been adopted by decoupled taint analysis work [6], [7] as a part of the logging infrastructure.

II. Efficient Control Flow Profiling

Figure 1 illustrates the architecture of BinCFP, which consists of three parts: online logging, multi-threaded fast buffering, and straight-line code reconstruction. To work with

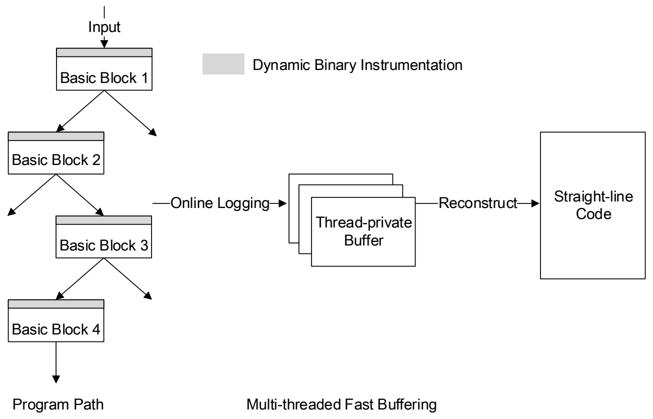

Figure 1. The architecture of BinCFP.

unmodified program binaries, BinCFP are built based on dynamic binary instrumentation (DBI). Application thread(s) are executing over BinCFPṪhe logging tool records dynamic control flow information by instrumenting basic blocks. Each basic block is represented as a unique tag. BinCFP first writes the basic block tags to a trace buffer and then store them into disk storage when the buffer becomes full. Three design goals guide us to achieving low execution overhead: 1) design an optimized trace buffer data structure to produce compact profile size; 2) process buffers asynchronously, allowing application code (i.e. producer) to continue executing while buffers are being consumed; 3) avoid unnecessary application instrumentation overhead. We meet the first requirement by adopting an optimized control flow profile structure. To handle the second challenge, we parallelize profile consumption through an n-way fast buffering scheme based on multi-cores. At last, we carefully design our logging tool to remove redundant instrumentation code. The details will be discussed thoroughly in the following subsections.

A. Control Flow Profile

A naive approach to recording dynamic control flow is to log each basic block executed, for example, recording each basic block's entry address. On 32-bit machines, a basic block can be represented as its 4-byte entry address. However, a 4-byte tag is an excessive use. Zhao et al. [1] proposed *Detailed Execution Profile* (DEP), an efficient method that only uses 2-byte tags for most basic blocks and represents special cases with extra escape bytes. DEP divides a 4-byte address into 2 high bytes for *H-tag* and the left 2 low bytes for *L-tag*. During online logging, if two continuing basic blocks share the same two high bytes (i.e., H-tag), only L-tag is logged into the profile buffer. If the H-tag varies, instead, a special escape tag 0x0000 followed by the new H-tag will be recorded. Furthermore, DEP's scheme does not require control flow graph or any fine-grained static analysis, making it suitable for binary code analysis. Our control flow profile is based on the DEP's scheme and improves it in several ways.

B. Optimizations

Certain x86 instructions are related to string operations, such as `MOVS`, `LODS`, `STOS`, `CMPS`, and `SCAS`. These instructions

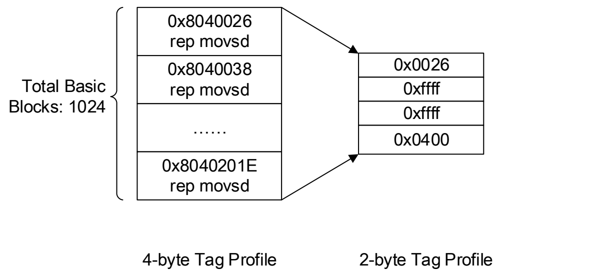

Figure 2. Optimization to the single basic blocks caused by REP-prefixed instructions.

are typically combined with `REP`-prefix to execute repeatedly. Dynamic binary instrumentation tools [11], [12] usually treat `REP`-prefixed instructions as implicit loops. If a `REP`-prefixed instruction iterates more than once, each new iteration will cause DBI to generate a new basic block, which only contains a single instruction. As a result, much more basic blocks than we expect are produced. In SPEC CPU2006, we find several cases that `REP`-prefixed string instructions are heavily used. For example, as high as 13.4% of total executed instructions of the `h264ref` benchmark are related to `REP` iterations. BinCFP extends DEP's profile to handle the implicit loops introduced by the `REP`-prefixed instructions, which could otherwise become a performance overkill. Particularly, we first detect the first loop of `REP`-prefixed instructions. And then two consecutive escape value 0xffff, followed by a repeat count (stored in the `ecx` register) are entered into the trace buffer to represent the entire implicit loops of `REP`-prefixed instructions. Note that the maximum `REP`-prefixed loop count in our evaluation comes from `gcc` benchmark. The loop count is $1,770$, which is far less than two-byte number limit. Figure 2 presents an example of our enhanced profile encoding. The left part shows a total $1,024$ single instruction basic blocks due to the implicit loop unrolling. BinCFP only encodes the first repetition and appends two consecutive 0xffff to indicate such `REP` loops. After that, the number of repetitions (0x0400 in our example) is put into the profile buffer. As a result, BinCFP only consumes 8 bytes to represent the total 1024 basic blocks. In contrast, the raw 4-byte profile takes up 1024×4 bytes space, and DEP needs 1024×2 bytes.

Besides, we also configure Pin to disable unrolling `REP`-prefixed instructions, because Pin otherwise inserts analysis code into each implicit iteration, introducing additional overhead. In our evaluation, our optimization can reduce DEP profile space at most 55.2% and with up to 63.5% runtime overhead reduction. Note that it is also possible to use a single bit to log a basic block by recording the binary decision of conditional jump [13], which leads to a much denser log data. However, 1-bit encoding also brings several drawbacks. First, Pin's notion of basic blocks is not strictly the same as compiler-level basic blocks. Pin also breaks basic blocks on some specific instructions, such as `CPUID`, `POPF`, and `REP`-prefixed instructions. 1-bit encoding will miss these special cases. Second, encoding 1-bit does not favor Pin code inline, which introduces more instrumentation overhead. Third, unlike BinCFP, recovering the whole execution trace from 1-bit profile has to walk through the control flow graph, which is quite expensive as well.

Although our example is 2-byte tag profile on IA-32, it is straightforward to extend to 4-byte tag profile for 64-bit machines. Current BinCFP accommodates both 32-bit and 64-bit binaries. It is worth mentioning that frequently checking of H-tags during online logging can introduce additional instrumentation overhead. We will further discuss this issue in Section II-D.

C. Multi-threaded Fast Buffering Scheme

In this section, we discuss our concurrently buffering data scheme design, which supports multi-threads on a multi-core platform. The goal is to exploit underutilized computing resources to mitigate the disk I/O bottleneck. The key to our design is an n-way buffering thread pool, which enables application threads (i.e., *producer threads*) to continue executing and writing data to free buffers, while multiple Pin-tool internal spawned threads (i.e., *worker threads*) process full buffers asynchronously. Figure 3 presents how the multi-threaded fast buffering scheme works. Let us assume the application under examination contains two threads. The processing steps are as follows.

1) When a program starts running, each application thread allocates a number of thread-private buffers (5 buffers in our example), which means each thread can only write data into its private buffers. Available free buffers are formed as a free-buffer queue. Also, each free buffer contains a thread ID, indicating the associated application thread of the free buffer.
2) At the same time, multiple Pin-tool internal threads (i.e., *worker threads*) are spawned as well. The number of worker threads in our example are 10. Note that different from application threads, worker threads are not JITed and therefore execute natively. A worker thread takes a buffer from the full buffer queue and then writes the data into disk storage. After that, the worker thread put the buffer into the free buffer queue of the associated application thread. To access a full buffer exclusively, multiple worker threads acquire and then release the full buffer's lock.
3) Worker threads communicate with application threads via counting semaphores. After being created, worker threads are waiting for the arrival of full buffers. When a free buffer becomes full, a callback function, *BufferFull* will be called to carry out three tasks: 1) enqueue the full buffer to a global full-buffer queue; 2) schedule one worker thread to process the full buffer; 3) return the thread-private free buffers to the application. If no free buffer, the application thread has to be blocked until a new free buffer becomes available.

It is apparent that the less time application threads spend on waiting for free buffers, the better performance it would achieve. Therefore, we bias our fast buffering scheme design to favor better application runtime performance. In particular, we create plenty of worker threads to process full buffers timely. Also, we dynamically adjust the number of buffers and worker threads to optimize load balancing. In Figure 3, the number of worker threads is 10, equal to the total number of buffers allocated by application threads. It is evident that the number worker threads and profile buffer size are critical for the better runtime performance. In Section IV-A, we will perform a set of experiments to tune these two factors.

D. Instrumentation Optimization

BinCFP's online logging tool is built on Pin. The way a Pin tool is implemented can have great impact on the performance of the instrumented application. Besides the optimization to the `REP`-prefixed instructions, this section introduces other methods we applied to reduce BinCFP's instrumentation overhead.

Conceptually, Pin's instrumentation contains two major components, namely instrumentation code and analysis code. Instrumentation code inspects program to decide where the analysis code should be injected. In our tool, the instrumentation granularity is on the basic block level. The instrumentation code is executed only once for every sequence of basic blocks, and the translated code, including original code and analysis code, is saved in Pin's code cache for efficiency. Analysis code will be invoked at beginning of each basic block. In our case, the analysis routines record dynamic control flow and manage fast buffering scheme. As analysis routines are executed very frequently, carefully tuning the analysis code is paramount for better performance. Pin tends to inline the compact, branch-less analysis routines into translated code cache; while the analysis code with conditional branches will force Pin to emit a function call to the respective analysis routines instead. In our case, we have to frequently check whether the H-tag has changed upon the execution of each basic block and write the tag to the trace buffer accordingly. To favor Pin's inlining, we shift the check of H-tag to the instrumentation phrase, which is performed only once when each basic block is translated. In this way, the branch instructions in the analysis routines are removed, and profile tags are updated as well. As a result, the overhead introduced by frequently checking H-tags is reduced. In Section III, we will introduce other Pin-specific optimizations we adopted.

III. Implementation

We have implemented BinCFP based on Pin DBI framework [11] (version 2.12). BinCFP works with multi-threaded applications running on both x86-32 and x86-64 machines. We create thread local storage (TLS) slot to store and retrieve per-thread buffer structure. One challenge here is that Pin-tools cannot work with either pthreads library or Win32 threading API. We have to implement a counting semaphore using Pin's own binary semaphore to spawn worker threads. To fully utilize Pin's code cache effect, we also set the maximum number of basic blocks per Pin trace from 3 to 8. In addition, we use GCC's built-in macro "`__builtin_expect()`" to facilitate compiler's branch prediction.

To manage the per-thread free-buffer queue and the global full-buffer queue, we leverage Pin fast buffering APIs to perform low-overhead buffering of data. More specifically, `INS_InsertFillBuffer()` inlines a call to write trace profile tags directly to the given buffer. The callback defined in `PIN_DefineTraceBuffer()` is used to process the buffer

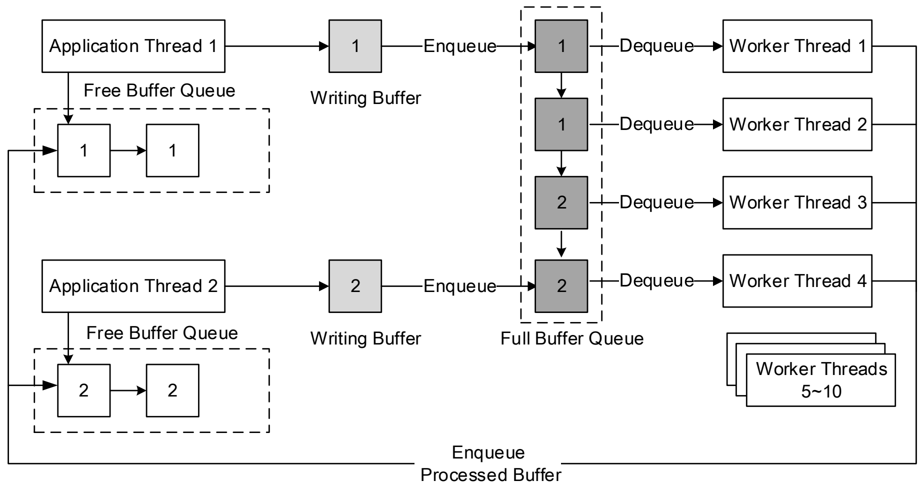

Figure 3. Multi-threaded fast buffering scheme.

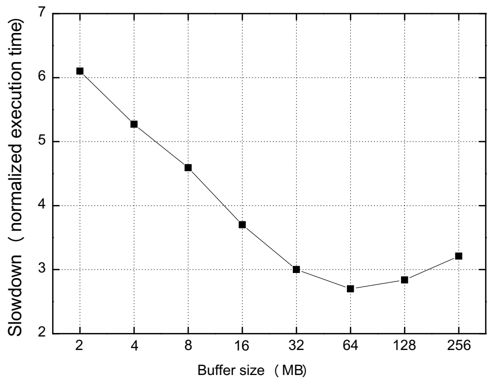

Figure 4. Normalized slowdown when profile buffer size varies.

when the buffer becomes full or when the thread exits. Moreover, we force Pin to use fastcall calling convention to pass arguments via registers, which avoids emitting stack access instructions required for the call (i.e., push and pop). We also use the GCC compile option “ `-fomit-frame-pointer`” to eliminate the instructions to save, set up and restore the frame pointer. Another benefit of fastcall linkages optimization is that it alleviates register spilling, which will otherwise become a performance bottleneck in many cases [14], [15].

IV. Evaluation

Our testbed contains a server machine, which is equipped with two Intel Xeon E5-2690 processors (16-core with 2.9GHz) and 128GB of RAM, running Ubuntu12.04. The data presented throughout this section are all mean values. We calculate them by running 5 repetitions of each experiment case.

A. Configuration of Buffer Size and Worker Threads

In this experiment, we tune the two factors that may affect BinCFP's performance: 1) the trace buffer size; 2) the number of worker threads. We first study the impact of buffer size by varying the buffer size from 2MB to 256MB. We also set the number of worker threads as 16 to achieve enough parallelism. In this way, the total buffer size is 16 $\times$ single buffer size. The training set is SPEC CINT2006 with *test* workload. Figure 4 shows a downward trend that the slowdown is reduced as the buffer size is increased. The reason is both the number of free/full buffer switches and the synchronization cost are reduced. However, when the buffer size is beyond 64MB, we see an increase on slowdown again. We attribute the additional overhead to the overlarge total buffer size (e.g., 16 $\times$ 128MB), which may affect application's working set. After that, we set the buffer size to 64MB and increase the number of worker threads progressively from one to sixteen. In summary, because of the tuned buffer size and the maximum parallelism, the combination of 16 worker threads and 64MB buffer size achieves an optimal result. Therefore, we set this configuration as default in the following experiments.

B. Performance

To evaluate the application performance slowdown imposed exclusively by BinCFP, we have developed a simple tool to measure Pin's environment runtime overhead, which runs a program under Pin without any form of analysis (“nullpin” bar). Besides, we also evaluate other three profile formats based on our multi-threaded buffering scheme. “BB_pc” bar refers the conventional 4-byte basic block profile, which records the entry address of each basic block. “CF_bit” indicate using a single bit to record the binary decision of a conditional jump. “DEP” refers the control flow profile scheme proposed by Zhao et al. [1].

a) SPEC CPU2006: Figure 5 shows the normalized execution time of running SPEC CINT2006 with *reference* workload. We expect these results can estimate the worst case. On average, BinCFP imposes a 2.97X slowdown to native execution, which is the lowest among the four profile formats. Due to the large profile size produced, BB_pc's application is often blocked for available free buffers. BinCFP outperforms DEP greatly in the test cases that use REP-prefixed instructions intensively, such as `h264ref` and `gcc`. CF_bit introduces as much as the 4.0X slowdown on average. The reason is CF_bit's instrumentation contains a number of conditional branches, which are hard to be inlined.

b) Utilities: Then we evaluate BinCFP on four obfuscated common Linux utilities. These utilities represent three kinds of workload. `tar` is I/O bounded, whereas `bzip2`

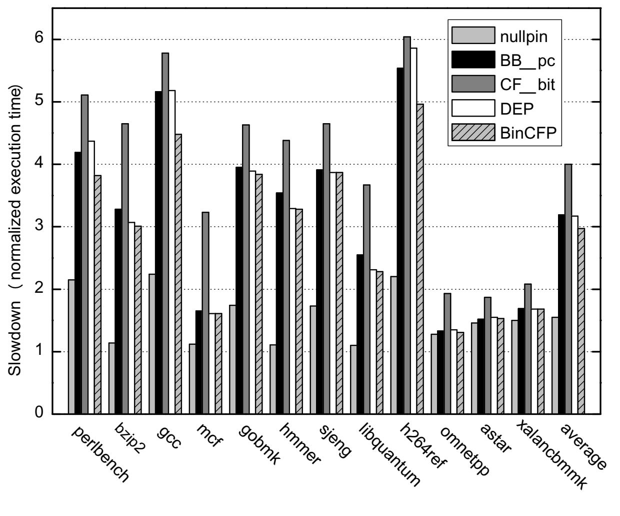

Figure 5. Slowdown on SPEC CPU2006.

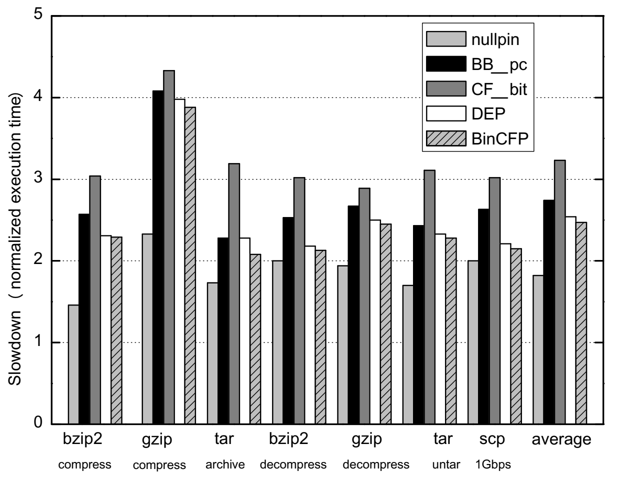

Figure 6. Slowdown on obfuscated common utilities.

and `gzip` are CPU intensive programs. Between these two cases is *scp*. We first run `tar` to archive and extract GNU Core utilities 8.13 package (∼50MB). And then the archive file of Core utilities is compressed and decompressed by `bzip2` and `gzip`, respectively. We use *scp* to copy the archive file of Core utilities over 1Gbps link. Furthermore, we obfuscate these four utilities by applying two obfuscators: Loco [16] and Obfuscator-LLVM [17]. Loco [16] is a link-time obfuscation tool, which is built on Diablo [18], a link-time optimizer. The Loco's common obfuscation options include control flow flattening [19] and opaque predicates [20]. Obfuscator-LLVM [17] is an obfuscation extension of the LLVM compilation suite [21]. Each obfuscation option is implemented as an LLVM pass. Table I shows the detailed obfuscation options we applied. Note that several options can greatly obfuscate control flow graph, e.g., "-funroll-loops" unrolls loops, "-mllvm -bcf" inserts opaque predicates, and "-mllvm -fla" does control flow flattening. Therefore, the previous efforts on efficient path profiling [8]–[10] cannot be directly applied to these obfuscated binary code. As shown in Figure 6, the experimental results are very similar to SPEC CPU2006. BinCFP imposes an average 2.47X slowdown to

Table I
Different optimization or obfuscation options.

Obfuscator	Options
Loco	-freorder-blocks (reorder basic blocks) -funroll-loops (unroll loops) -finline-small-functions (inline functions)
Obf-LLVM	-mllvm -sub (instructions substitution) -mllvm -bcf (opaque predicate) -mllvm -fla (control flow flattening)

Table II
Uncompressed trace profile size percentage (%).

Application	BB_pc	CF_bit	DEP	BinCFP
Utilities	100.0	21.9	56.6	51.6
SPEC CPU2006	100.0	22.6	53.0	47.8
Average	100.0	22.3	54.3	49.2

native execution, with a 1.30X speed up to CF_bit. Compared to nullpin, BinCFP only incurs a 1.36X slowdown. This evaluation shows that BinCFP supports the complete and efficient dynamic control flow collection of obfuscated binary code.

c) Profile Size: We also evaluate the profile size (uncompressed) introduced by the four different control flow profile formats: BB_pc, CF_bit, DEP, and BinCFP. We take the naive 4-byte profile (BB_pc) size as the baseline. Table II shows the average proportion related to BB_pc's profile size. The lower value in Table II means the size is more compact. On average, the relative size of BinCFP is only 49.2%, less than DEP's size by 5 percentages. It is worth pointing out that we see a significant profile size reduction for the *h264ref* benchmark, from 54.9% (DEP) to 24.6% (BinCFP). That is due to the intensive usage of `REP`-prefixed instructions. BinCFP optimizes this case (discussed in Section II-B) and produces more compact profile size. CF_bit results in a much denser profile size with the 22.3% percentage. However, CF_bit instrumentation is hard to be optimized, which leads to the highest overhead in our evaluation. In summary, BinCFP represents a practical solution that balances the runtime performance and the profile size. Note that trace compression methods (e.g., VPC3 [22], Sequitur [23]) are also be applied on BinCFP. We can benefit from these new trace compression algorithms to further reduce the profile size.

C. Instrumentation Optimization

In this experiment, we evaluate the set of optimizations we adopted for the online instrumentation. Figure 7 shows the impact of instrumentation optimizations when applied cumulatively on SPEC CPU2006 with the *reference* workload. We measure the logging overhead by not buffering the profile data to disk. The "unopt" bar represents an un-optimized version, which does not employ any optimization methods that we discussed in Section II and Section III. The bar denoted as O1, captures the effect of our effort on inlining analysis code, bringing a notable overhead step-down from average 5.69X to 3.19X. Optimization O2, which adds fast buffering APIs and fastcall linkages, reduces running time further to 2.73X. Optimization to `REP`-prefixed instructions (O3) offers an enhanced performance improvement by average 27.0%, with a peak value 63.5% to the `h264ref`. We attribute this to the fact that as high as 13.4% of total executed instructions of the `h264ref` are `REP` instruction repetitions.

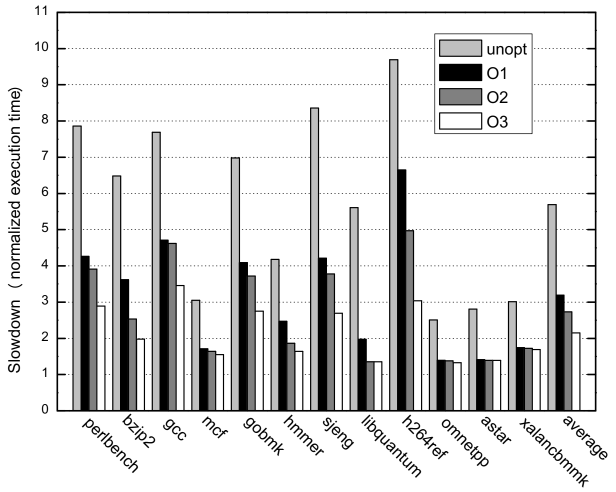

Figure 7. The impact of instrumentation optimization on SPEC CPU2006: O1 (inline analysis code), O2 (O1 + fast buffering APIs and fastcall linkages), O3 (O2 + optimize rep-prefixed instructions).

V. Discussion and Conclusion

Currently, BinCFP's control flow profile encodes basic block entry addresses. However, the entry address may not uniquely identify self-modifying code blocks. To address this issue, we can configure BinCFP to record the real executed instructions. BinCFP lacks isolation because all worker threads work in the same process running both Pin and the application. One direction is to switch the worker threads to different processes. Another interesting future work is to study efficient memory references profiling for the obfuscated binary code.

We have presented BinCFP, an efficient multi-threaded binary code control flow profiling tool by taking advantage of ubiquitous multi-core platforms. Unlike previous approaches, BinCFP does not rely on fine-grained static analysis, which enables broad applications in speeding up binary code analysis such as hybrid disassembly and decoupled taint analysis. The experimental results on SPEC2006 and obfuscated common utility programs are encouraging, which shows a considerable speedup over the previous work. BinCFP has been adopted by decoupled taint analysis work [6], [7] as a part of the logging infrastructure. PoL-DFA [24] adopts a simple version of our method, with less optimization on performance, and is built on LLVM [21].

VI. Acknowledgments

This research was supported in part by the National Science Foundation (NSF) grants CNS-1223710 and CCF-1320605, and the Office of Naval Research (ONR) grants N00014-13-1-0175, N00014-16-1-2265, and N00014-16-1-2912.

References

[1] Q. Zhao, J. E. Sim, L. Rudolph, and W.-F. Wong, "DEP: Detailed execution profile," in *Proc. of the 15th International Conf. on Parallel Architectures and Compilation Techniques (PACT'06)*, 2006.

[2] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS'03)*, 2003.

[3] J. Caballero, N. M. Johnson, S. McCamant, and D. Song, "Binary code extraction and interface identification for security applications," in *Proceedings of 17th Annual Network and Distributed System Security Symposium (NDSS'10)*, 2010.

[4] S. K. Udupa, S. K. Debray, and M. Madou, "Deobfuscation: Reverse engineering obfuscated code," in *Proceedings of the 12th Working Conference on Reverse Engineering (WCRE'05)*, 2005.

[5] K. jee, V. P. Kemerlis, A. D. Keromytis, and G. Portokalidis, "ShadowReplica: Efficient parallelization of dynamic data flow tracking," in *Proceedings of the ACM SIGSAC conference on Computer & communications security (CCS'13)*, 2013.

[6] J. Ming, D. Wu, G. Xiao, J. Wang, and P. Liu, "TaintPipe: Pipelined symbolic taint analysis," in *Proceedings of the 24th USENIX Security Symposium (USENIX Security'15)*, 2015.

[7] J. Ming, D. Wu, J. Wang, G. Xiao, and P. Liu, "StraightTaint: Decoupled offline symbolic taint analysis," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE'16)*, 2016.

[8] T. Ball and J. R. Larus, "Efficient path profiling," in *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO'29)*, 1996.

[9] M. D. Bond and K. S. McKinley, "Practical path profiling for dynamic optimizers," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO'05)*, 2005.

[10] R. Joshi, M. D. Bond, and C. Zilles, "Targeted path profiling: Lower overhead path profiling for staged dynamic optimization systems," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO'04)*, 2004.

[11] 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 *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI'05)*, 2005.

[12] D. Bruening, T. Garnett, and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," in *Proceedings of the international symposium on code generation and optimization (CGO'03)*, 2003.

[13] M. Renieris, S. Ramaprasad, and S. P. Reiss, "Arithmetic program paths," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*, 2005.

[14] E. Bosman, A. Slowinska, and H. Bos, "Minemu: The world's fastest taint tracker," in *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection (RAID'11)*, 2011.

[15] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, "libdft: Practical dynamic data flow tracking for commodity systems," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'12)*, 2012.

[16] M. Madou, L. Van Put, and K. De Bosschere, "Loco: An interactive code (de)obfuscation tool," in *Proceedings of ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation (PEPM'06)*, 2006.

[17] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-LLVM - software protection for the masses," in *Proceedings of the 1st International Workshop on Software Protection (SPRO'15)*, 2015.

[18] "Diablo Is A Better Link-time Optimizer," [online]. Available: http://diablo.elis.ugent.be/.

[19] C. Wang, J. Hill, J. C. Knight, and J. W. Davidson, "Protection of software-based survivability mechanisms," in *Proceedings of the 2001 International Conference on Dependable Systems and Networks (DSN'01)*, 2001.

[20] J. Ming, D. Xu, L. Wang, and D. Wu, "LOOP: Logic-oriented opaque predicate detection in obfuscated binary code," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*, 2015.

[21] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO'04)*, 2004.

[22] M. Burtscher, "VPC3: a fast and effective trace-compression algorithm," in *Proceedings of the joint International Conference on Measurement and Modeling of Computer Systems*, 2004.

[23] C. G. Nevill-Manning and I. H. Witten, "Identifying hierarchical structure in sequences: a linear-time algorithm," *Journal of Artificial Intelligence Research*, vol. 7, no. 1, pp. 67–82, 1997.

[24] G. Xiao, J. Wang, P. Liu, J. Ming, and D. Wu, "Program-object level data flow analysis with applications to data leakage and contamination forensics," in *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy (CODASPY'16)*, 2016.