ShadowReplica: Efficient Parallelization of Dynamic Data Flow Tracking

Kangkook Jee
Columbia University
jikk@cs.columbia.edu

Vasileios P. Kemerlis
Columbia University
vpk@cs.columbia.edu

Angelos D. Keromytis
Columbia University
angelos@cs.columbia.edu

Georgios Portokalidis
Stevens Institute of Technology
gportoka@stevens.edu

ABSTRACT

Dynamic data flow tracking (DFT) is a technique broadly used in a variety of security applications that, unfortunately, exhibits poor performance, preventing its adoption in production systems. We present ShadowReplica, a new and efficient approach for accelerating DFT and other shadow memory-based analyses, by *decoupling analysis from execution and utilizing spare CPU cores to run them in parallel*. Our approach enables us to run a heavyweight technique, like dynamic taint analysis (DTA), *twice as fast*, while concurrently consuming *fewer* CPU cycles than when applying it in-line. DFT is run in parallel by a second shadow thread that is spawned for each application thread, and the two communicate using a shared data structure. We avoid the problems suffered by previous approaches, by introducing an off-line application analysis phase that utilizes both static and dynamic analysis methodologies to generate optimized code for decoupling execution and implementing DFT, while it also minimizes the amount of information that needs to be communicated between the two threads. Furthermore, we use a lock-free ring buffer structure and an N-way buffering scheme to efficiently exchange data between threads and maintain high cache-hit rates on multi-core CPUs. Our evaluation shows that ShadowReplica is on average $\sim$2.3$\times$ faster than in-line DFT ($\sim$2.75$\times$ slowdown over native execution) when running the SPEC CPU2006 benchmark, while similar speed ups were observed with command-line utilities and popular server software. Astoundingly, ShadowReplica also reduces the CPU cycles used up to 30%.

Categories and Subject Descriptors

D.4.5 [**Operating Systems**]: Reliability; D.4.6 [**Information flow controls**]: Security and Protection; D.4.7 [**Parallelization**]: Organization and Design; D.4.8 [**Optimization**]: Performance

Keywords

Security, Information flow tracking, Optimization, Parallelization

CCS'13, November 4–8, 2013, Berlin, Germany.

http://dx.doi.org/10.1145/2508859.2516704.

1. INTRODUCTION

Dynamic data flow tracking (DFT) is being used extensively in security research for protecting software [24, 11], analyzing malware [32, 21], discovering bugs [23, 7], reverse engineering [29], information flow control [34], etc. However, dynamically applying DFT tends to significantly slow down the target application, specially when a virtualization framework [22, 5] is used to apply it on binary-only software. Overheads can range from a significant percentage over native execution to several orders of magnitude, depending on the framework used and particular traits of the implementation [19]. When performance is not an issue, the overhead can still be problematic: (a) if it changes the behavior of the application (e.g., when network connections timeout or the analysis is no longer transparent), or (b) when computational cycles are scarce or CPU energy consumption needs to be kept to a minimum, like in mobile devices.

We present ShadowReplica, a new and efficient approach for accelerating DFT and other shadow memory-based analyses [23], by *decoupling analysis from execution and utilizing spare CPU cores to run them in parallel*. Our approach enables us to run a heavyweight technique like dynamic taint analysis (DTA) *twice as fast*, while concurrently consuming *fewer* CPU cycles than when applying it in-line. The main motivation behind ShadowReplica has been to accelerate security techniques with well established benefits, such as DFT, but it can also host other types of analyses. We demonstrate this by also implementing a control-flow integrity (CFI) [1] tool over ShadowReplica, however, the greater benefits can be reaped by techniques based on memory shadowing [23, 4].

Decoupling analysis from execution to run it in parallel is by no means a novel concept [14, 8, 31, 25, 33, 6, 26]. Previous work can be classified into three categories. The first is based on recording execution and replaying it along with the analysis on a remote host, or simply a different CPU [8, 26, 6]. These are geared toward off-line analyses and can greatly reduce the overhead imposed on the application. However, the speed of the analysis itself is *not* improved, since execution needs to be replayed and augmented with the analysis code. These solutions essentially hide the overhead from the application, by sacrificing computational resources at the replica. Due to their design, they are not a good fit for applying preventive security measures, even though they can be used for post-fact identification of an intrusion.

The second category uses speculative execution to run application code including any in-lined analysis in multiple threads running in parallel [25, 31]. While strictly speaking the analysis is not decoupled, it is parallelized. These approaches sacrifice significant *processing power* to achieve speed up, as at least two additional

threads need to be used for any performance gain, and the results of some of the threads may be discarded. Furthermore, handling multi-threaded applications without hardware support remains a challenge.

The third category aims at offloading the analysis code alone to another execution thread [14, 33]. These instrument the application to collect all the information required to run the analysis independently, and communicate the information to a thread running the analysis logic alone. In principle, these approaches are more efficient, since the application code only runs once. However, in practice, they have not been able to deliver the expected performance gains, due to *inefficiently* collecting information from the application and the high overhead of communicating it to the analysis thread.

ShadowReplica belongs to the third category of systems. Our main contribution is an off-line application analysis phase that utilizes both static and dynamic analysis approaches to generate optimized code for collecting information from the application, greatly reducing the amount of data that we need to communicate. For running DFT independently from the application, such data include dynamically computed information like memory addresses used by the program, control flow decisions, and certain operating system (OS) events like system calls and signals. We focus on the first two that consist the bulk of information. For addresses, we exploit memory locality to only communicate a smaller set of them, and have the DFT code reconstruct the rest based on information extracted by the off-line analysis. For control flow decisions, we exploit the fact that most branches have a binary outcome and introduce an intelligent encoding of the information sent to DFT to skip the most frequent ones.

DFT is run in parallel by a second shadow thread that is spawned for each application thread, and the two communicate using a shared data structure. The design of this structure is crucial to avoid the poor cache performance issues suffered by previous work. We adopt a lock-free ring buffer structure, consisting of multiple buffers (N-way buffering scheme [33]). After experimentation, we identified the optimal size for the sub-buffers of the structure, so that when two threads are scheduled on different cores on the same CPU die, we achieve a high cache-hit rate for the shared L3 cache and a low-eviction rate on each core's L1 and L2 caches. The latter is caused when two cores are concurrently read/write in memory that occupies the same cache line in their L1/L2 caches.

The code implementing DFT is generated during off-line analysis as a C program, and includes a series of compiler-inspired optimizations that accelerate DFT by ignoring dependencies that have no effect or cancel out each other [18]. Besides the tag propagation logic, this code also includes per-basic block functionality to receive all data required (e.g., dynamic addresses and branch decisions). Note that even though the code is in C, it is generated based on the analysis of the binary without the need for application source code. The implementation is also generic, meaning that it can accommodate different tag sizes in shadow memory, and it can be easily extended to different architectures (e.g., x86-64). Such flexibility is partly allowed by decoupling DFT from execution.

We implemented ShadowReplica using Intel's Pin dynamic binary instrumentation (DBI) framework [22] to instrument the application and collect the data required to decouple DFT. Shadow threads running the DFT code run natively (without Pin), but in the same address space, also implementing dynamic taint analysis [24] protection from memory corruption exploits. Our evaluation shows that compared with an already optimized in-lined DFT framework [18], it is extremely effective in accelerating *both* the application and DFT, but also using less CPU cycles. In other words, we do not sacrifice the spare cores to accelerate DFT, but exploit parallelization to improve the efficiency of DFT in all fronts. ShadowReplica is on average $\sim$2.3$\times$ faster than in-lined DTA when running the SPEC2006 benchmark ($\sim$2.75$\times$ slowdown over native execution). We observed similar speed ups with command-line utilities, like `bzip2` and `tar`, and the Apache and MySQL servers. We also discovered that with ShadowReplica applying DFT requires less CPU cycles than the in-lined case, reaching a 30% reduction in the 401.bzip2 benchmark.

The contributions of this paper can be summarized as follows:

- We propose a novel approach to efficiently parallelize in-line analysis by implementing low-cost communication between the primary original process and the secondary analyzer process.
- Our approach does not require a hardware component at runtime, but instead it is based on static and dynamic program analysis performed in advance.
- ShadowReplica preserves the functionality of both the original program being monitored, and the analysis logic that would be otherwise applied in-line.
- We implement a DFT analysis prototype, but our approach can be also applied to other analyses based on memory shadowing.
- We evaluate our prototype with the SPEC2006 CPU benchmark suite and various real-world applications, and our results confirm both the efficiency and effectiveness of our approach. Furthermore, we show that through ShadowReplica DFT uses less CPU cycles and energy, than when applied in-line.

The rest of this paper is organized as follows. Section 2 presents an overview of our approach. Section 3 describes the off-line analysis stage, which includes most of our optimizations. Section 4 explains the dynamic runtime. In Sec. 5, we provide implementation details, and the evaluation of our framework is presented in Sec. 6. After discussing related work in Sec. 7, we conclude the paper in Sec. 8.

2. OVERVIEW

2.1 In-line vs. Decoupled DFT

Dynamically applying DFT on binaries usually involves the use of a dynamic binary instrumentation (DBI) framework or a virtual machine monitor (VMM) that will transparently extend the program being analyzed. Such frameworks enable us to inject code implementing DFT in binaries, by interleaving framework and DFT code with application code, as shown in Fig.1 (*in-line*).

ShadowReplica proposes an efficient approach for accelerating dynamic DFT and similar analyses by decoupling them from execution and utilizing spare CPU cores to run the instrumented application and DFT code in parallel. We replace the in-line DFT logic in the application with a stub that *extracts* the minimal information required to independently perform the analysis in another thread, and *enqueues* the information in a shared data structure. The DFT code, which is running on a different CPU core, is prefixed with a consumer stub that *pulls out* the information and then *performs* the analysis.

Decoupling the analysis from execution enables us to run it completely independently and without involving the instrumentation framework, as illustrated in Fig. 1 (*decoupled*). Depending on the cost of the analysis (e.g., tracking implicit information flows is more costly than explicit flows), it can accelerate both application and analysis. In short, if I_i, A_i, and P_i are the instrumentation,

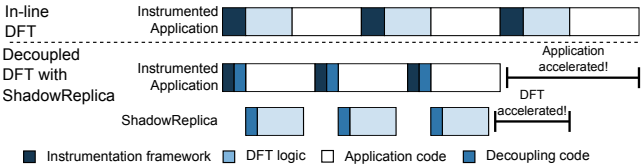


Figure 1: In-line vs. decoupled application of DFT with ShadowReplica and binary instrumentation.

analysis, and application code costs with in-line analysis, and I_d, A_d, P_d, E_d and D_d are the costs of instrumentation, analysis, application, enqueueing and dequeueing code (as defined in the above paragraph), then decoupling is efficient when:

$$I_i + A_i + P_i > max(I_d + P_d + E_d, A_d + D_d) \quad (1)$$

Essentially, decoupling is more efficient when the following two conditions are met: *(a)* if the cost of the in-line analysis is higher than the cost of extracting the information and enqueueing, and *(b)* if the cost of program execution combined with instrumentation interference is higher than dequeueing cost. Ha et al. [14] provide a more extensive model of the costs and benefits involved with decoupling analysis.

Analyses that are bulky code-wise can experience even larger benefits because replacing them with more compact code, as decoupling does, exerts less pressure to the instrumentation framework, due to the smaller number of instructions that need to be interleaved with application code. For instance, when implementing DFT using binary instrumentation, the developer needs to take extra care to avoid large chunks of analysis code and conditional statements to achieve good performance [19]. When decoupling DFT, we no longer have the same limitations, we could even use utility libraries and generally be more flexible.

We need to emphasize that ShadowReplica does *not* rely on complete execution replay [8, 26] or duplicating execution in other cores through speculative execution [25, 31]. So even though other cores may be utilized,[1] it does not waste processing cycles. Application code runs exactly once, and the same stands for the analysis code that runs in parallel. The performance and energy conservation benefits gained are solely due to exploiting the true parallelism offered by multi-cores, and being very efficient in collecting and communicating all the data required for the analysis to proceed independently.

2.2 Decoupling DFT

DFT involves accurately tracking selected data of interest as they propagate during program execution, and has many uses in the security domain [12, 9, 3, 27, 32, 24, 34]. In ShadowReplica, we implemented DFT along with its most popular incarnation, DTA [24], where input data (e.g., network data) are deemed "tainted" and their use is prohibited in certain program locations (e.g., in instructions manipulating the control flow of programs, such as indirect branch instructions and function calls).

We created a code analysis module for ShadowReplica that implements DFT by generating optimized C code for performing data tracking for all the BBLs discovered during profiling, and generic code for newly discovered code. We follow the methodology we introduced in prior work for optimizing in-lined DFT code [18].

[1]We do not demand the kernel to schedule the replica on a different core, but we do enforce each replica to be on the same package/die with the process to benefit from L3 cache sharing.

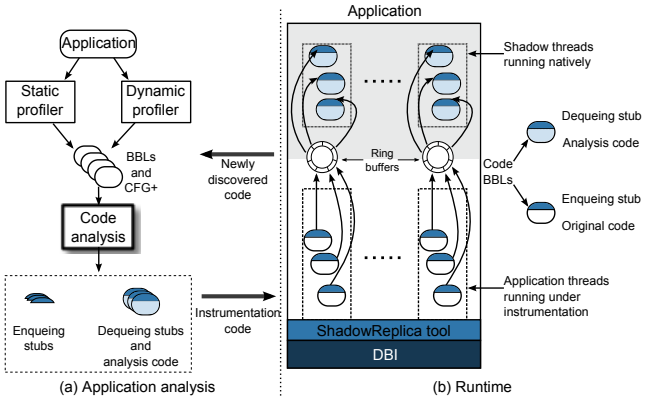


Figure 2: The architecture of ShadowReplica.

To correctly and independently track data and propagate tags, the analysis code has two sources of information: the code itself and the application executing in parallel. For example, an x86 instruction like `mov eax, ebx` indicates that a tag should be propagated between registers, so no runtime information is required. However, for instructions that reference memory using dynamically calculated addresses, e.g., `mov eax, (ecx, ebp,4)`, we need to obtain the effective address used by the instruction during execution. Remember, that the analysis code does *not* execute application code, nor does it have access to data contained in the registers because it is executing in parallel. The application needs to also supply information regarding control-flow decisions, as well as any other data required by DFT, like when to tag (taint) an address range that is only available at runtime. *This is all the information required to accurately perform DFT in parallel.*

Security checks are normally performed in parallel with the application. However, it is necessary that we have the ability to synchronize with the analysis code, by waiting for it to complete. This is important because in the case of DTA we want to ensure that program control flow has not been compromised, and in the case of information flow control that no sensitive data has been leaked. This is achieved by injecting code in the application that checks whether the analysis has completed. For instance, to protect from leaking sensitive information, our prototype implementation synchronizes before system calls used to transmit data, like `write()` and `send()`.

Changing characteristics of DFT, like the size of tags it supports that affect the number of different data classes or labels it can track, only affects the analysis code. The application is only delayed when synchronization is required and, as our evaluation shows, currently this is not an issue because DFT executes as fast (if not faster) as the application.

2.3 Architecture

Figure 2 depicts the architecture of ShadowReplica, which comprises of two stages. The first stage, shown in the left of the figure and discussed thoroughly in Sec. 3, involves profiling an application both statically and dynamically to extract code blocks, or basic blocks (BBLs), and control-flow information (CFG+). The latter includes a partial control-flow graph showing how the extracted BBLs are connected, and frequency data indicating which branches are taken more frequently than others.

This data is processed to generate optimized code to be injected in the application, and code for running the analysis in parallel. The first contains code stubs that enqueue the information required to

decouple DFT in a shared data structure. Note that ShadowReplica does not naively generate code for enqueueing everything, but ensures that only information that has potentially changed since the previous executed block are enqueued. This is one of our main contributions, and problems of previous work [25, 33] that failed to satisfy equation (1). The second includes code stubs that dequeue information along with the analysis code.

The generated code is passed to the runtime component of ShadowReplica, shown in Fig. 2(b) and discussed in Sec. 4. We utilize a DBI framework that allows us to inject the enqueueing stubs in the application in an efficient manner and with flexibility (i.e., on arbitrary instructions of a binary). Our motivation for using a DBI is that it allows us to apply ShadowReplica on unmodified binary applications, and it enables different analyses, security related or others, by offering the ability to "interfere" with the application at the lowest possible level.

Application threads are executing over the DBI and our tool, which inject the enqueueing stubs. We will refer to an application thread as the *primary*. For each primary, we spawn a shadow thread that will run the analysis code, which we will refer to as the *secondary*. While both threads are in the same address space, applications threads are running over the DBI's VMM, but shadow threads are executing *natively*, since the code generated in the first phase includes everything required to run the analysis. Our current design spawns secondary threads in the same process used by the DBI and the application. In the future, we are considering hosting the secondary threads in a different process for increased isolation.

Communication between primary and secondary threads is done through a ring-buffer structure optimized for multi-core architectures (Sec. 4.2). The ring buffer is also used for the primary thread to synchronize with the secondary, when it is required that the analysis is complete before proceeding with execution. For instance, ensuring that integrity has not been compromised before allowing a certain system call or performing a computed branch.

Multi-threaded applications accessing shared data within a critical area protected by locks (e.g., mutexes) are handled by using an additional ring buffer structure for every critical area. When a primary enters such an area by acquiring a lock, it first synchronizes with the secondary, and then they both switch to the new ring buffer, which now receives all the information generated by the enqueueing stubs. Before exiting the critical area, the primary synchronizes again with the secondary, and then both switch to the original buffer.

Finally, we export any new BBLs and CFG edges that are discovered at runtime, which can be passed back for code analysis. Extending the coverage of our analysis means that we can generate optimal code for a larger part of the application. Note that our analysis also generates generic code for handling application code not discovered during profiling. This "default" code performs all necessary functionality, albeit slower than the optimized code generated for known BBLs and control-flow edges.

2.4 Other Applicable Analyses

The main motivation behind ShadowReplica has been to accelerate security techniques with well established benefits, by providing a methodology and framework that can utilize the parallelism offered by multi-core CPUs. Besides DFT and DTA, we also implemented a control-flow integrity [1] tool to demonstrate the flexibility of our approach. We argue that many other analyses can benefit from it.

Control-flow Integrity. CFI [1], similarly to DTA, aims at preventing attackers from compromising applications by hijacking their control flow. Programs normally follow predetermined execution

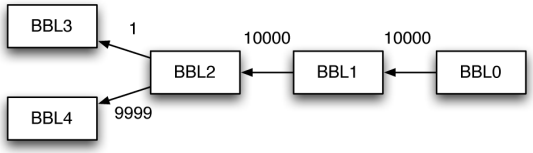


Figure 3: Example CFG. Nodes represent basic blocks and edges are control transfers. During dynamic profiling, we count how many times each edge is followed (edge labels).

paths. CFI enforces program flow to follow one of these predetermined paths. Determining a complete CFG of a program is a challenging task in itself, but assuming such a CFG is available, CFI operates as follows. Run-time checks are injected in the application to ensure that control flow remains within the CFG. These checks are placed before every indirect control flow transfer in the application and check that the destination is one of the intended ones. Basically, all possible destinations of a particular control flow instruction are assigned the same id, which is validated before transferring control. While this can be overly permissive, because multiple instructions may have the same targets assigning the same id to the super-set of destinations, this approach allows for fast checks. We implemented CFI with ShadowReplica by using the CFG information extracted during the application profiling, and by generating analysis code that checks whether a control-flow transfer is allowable, using the same control flow enqueueing stubs we used for DFT.

Other Analyses. Dynamic analyses that use DBI frameworks [22, 5, 23] can also readily make use of ShadowReplica (e.g., techniques that focus on memory integrity detection). Valgrind's Memcheck [23] uses shadow memory to keep track of uninitialized values in memory and identify their (dangerous) use; Memcheck is an ideal candidate for accelerating through ShadowReplica. Dr. Memory [4] discovers similar types of programming errors including memory leaks, heap overflows, etc. Software-based fault isolation [30] mechanisms can also be easily supported through our framework by using the existing code analysis for the primary and small modifications to the secondary. Approaches that do not depend on shadow memory can also be supported with moderate engineering effort. Examples include call graph profiling, method counting, and path profiling. Finally, ShadowReplica can be extended to work for analyses that refer to memory contents such as integer overflow detection. Note that analyses of this type are less common for binaries, as they require access to source code.

Limitations. There are problem areas where ShadowReplica is not a good fit, exactly because analysis code is decoupled from execution. For instance, cache simulation [16] requires that the ordering of memory accesses can be reconstructed accurately. This task is challenging, even when using in-lined analysis code. It is even more so, when the analysis code runs in parallel.

3. OFF-LINE APPLICATION ANALYSIS

We analyze the application off-line to generate optimized instrumentation code to be injected in primary threads, as well as a new program in C that implements the analysis in the secondary (based on the execution trace recorded by the primary). This section describes our methodology for doing so.

3.1 Application Profiling

The first step of application analysis involves profiling (Fig. 2(a)) to gather code and control-flow information. ShadowReplica uses

both static and dynamic profiling. Currently, we perform static profiling using the IDA Pro [15] disassembler, using its scripting API. To complement our data, we built a tool over the Pin [22] DBI framework that dynamically collects information about a binary by executing it with various inputs (e.g., test inputs and benchmarks). New methodologies that improve code and CFG identification are orthogonal to our design and could be included alongside our toolkit with little effort.

Every BBL identified during profiling is assigned a unique id (BBID). BBIDs are calculated by combining the block's offset from the beginning of the executable image it belongs to with a hash of the image's text section to produce a 32-bit value. An example control-flow graph collected during profiling is shown in Fig. 3. During dynamic profiling, we also keep a counter of how many times each control-flow transfer is observed. In the example in Fig. 3, when executing BBL2, BBL4 was followed 9999 times while BBL3 only once.

3.2 Primary Code Generation

During code analysis we generate optimized code for the primary thread to enqueue information necessary for performing DFT in the secondary. The most frequently enqueued data are effective addresses used in memory accesses and control-flow transfers. This section discusses our approach for minimizing the amount of data we need to send to the secondary.

3.2.1 Effective Address Recording

A naive approach would transfer all addresses involved in memory operations to the secondary, leading to excessive overhead. We developed a series of optimizations to reduce the number of addresses needed to be enqueued without compromising the soundness of the analysis. Fig. 4 will assist us in presenting our methods.

We begin by transforming the BBL in Fig. 4 (a) into the DFT-specific representation in Fig. 4 (b), which captures data dependencies and data tracking semantics, as defined in an intermediate representation called Taint Flow Algebra (TFA) [18]. Each register used in the BBL is treated like a variable. Whenever a register variable is updated, a new version of the variable is created (e.g., `eax1` in line 1 of Fig. 4(b)) that is valid within the BBL. For example, in lines 1 to 4 and 6, of Fig. 4(b), a tag is copied, while in line 5 two tags are combined, denoted by the OR ('|') operator.

Intra-block Optimization. We search for effective addresses within a BBL that correlate with each other to identify the minimum set required that would correctly restore all of them in the secondary. For instance, we only need to enqueue one of `[esp0]` or `[esp0 + 4]` from Fig. 4 (b), as one can be derived from the other by adding/subtracting a constant offset. This search is greatly facilitated by the DFT transformation and register variables versioning, but it is applicable to all shadow memory analyses, as it only tries to eliminate related addresses.

DFT Optimization. This optimization identifies instructions, and consequently memory operands, which are not going to be used by the analysis in the secondary, by applying compiler optimizations, such as dead code elimination and data-flow analysis, against our DFT-specific representation of the BBL [18]. For instance, in Fig. 4(b) we determine that the propagation in line 1 is redundant, as its destination operand (`eax1`) is overwritten later in line 3, before being referred by any other instruction. This allows us to ignore its memory operand `[esp0]`.

Inter-block Optimization. We extend the scope of the intra-block optimization to cover multiple blocks connected by control transfers. This implements backward data-flow analysis [2] with the partial CFG gathered during profiling. We begin by defining the input and output variables for each BBL. We then produce a list of input and output memory operands which are live before entering and when leaving a BBL. Using our representation, input memory operands are the ones with all of its constituent register variables in version 0, and output memory operands are the ones that have all of its constituent register variables at their maximum version. In our example in Fig. 4, the inputs list comprises of `[esp0]`, and the outputs list includes `[esp0]`, `[ebx1]`, `[eax2 + ebx1]`, and `[eax2 + 2 × ebx1 + 200]`. If all predecessors for the BBL contain `[esp0]` in their outputs list, the block can harmlessly exclude this from logging because the secondary still has a valid reference to the value. The optimization has greater effect as more inputs are found on the outputs lists of a block's predecessors.

Since we only have a partial CFG of the application, it is possible that at runtime we identify new execution paths that invalidate the backwards data-flow analysis. We tackle this issue by introducing two heuristics that make inter-block optimization more conservative. First, if we find any indirect jumps to a known BBL, we assume that others may also exist and exclude these blocks from optimization. Second, we assume that function entry point BBLs, may be reachable by other, unknown indirect calls, and we also exclude them. We consider these measures to be enough to cover most legitimate executions, but they are not formally complete, and as such may not cover applications that are buggy or malicious. Note that with the latter, we are not referring to vulnerable applications that may be compromised, an event that can be prevented by DTA, but malicious software that one may wish to analyze. In the case of malware, this optimization can be disabled with negligible impact on performance (see Sec. 6).

Linear Lower Bound. To enhance the intra- and inter-block optimizations, we introduce the concept of *Linear Lower bound*, which interprets memory operands as a series of linear equations:

$$b_{reg} + s \times i_{reg} + d \tag{2}$$

Memory addressing in x86 architectures can be expressed as a linear equation (2). It contains two variables for base and index registers (b_{reg} and i_{reg}), a coefficient s for scale, and a constant d for displacement.[2] Thus, we can say each BBL is associated with a group of linear equations, where each equation represents a distinct memory operand (Fig. 4(c)). For every BBL, we solve their group of linear equations, which results in reducing the number of memory operands that need to be enqueued and decreasing the size of a BBL's inputs and outputs lists for the inter-block optimization. For example, in line 5 of Fig. 4(c), `[ebx1]` is no longer required, as it can be calculated from `[eax2 + ebx1]` and `[eax2 + 2 × ebx1 + 200]` in lines 4 and 6. It also helps inter-block analysis by adding `[eax2]` to the BBL's outputs list.

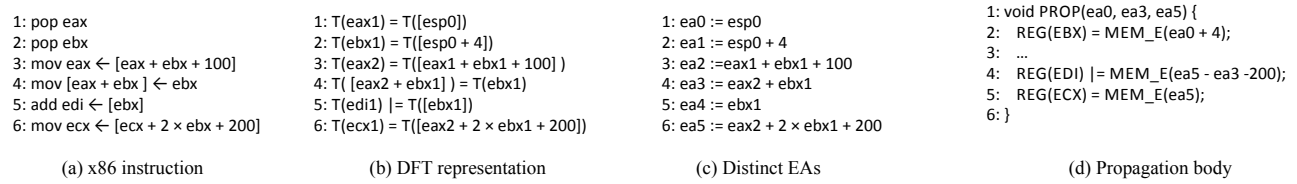
Having applied all, except the inter-block optimization, the number of effective addresses that need to be transferred for our example is reduced from six to three (from Fig. 4(c) to (d)). The effects of individual optimizations are discussed in Sec. 6.1.

3.2.2 Control Flow Recording

As in the previous section, a naive approach to ensure control flow replication would involve the primary enqueueing the BBID of every BBL being executed. However, simply doing this for all BBLs is *too* costly.

After examining how control-flow transitions are performed in x86 architectures, we identify three different types of instructions:

[2]Segmentation registers see limited use today, mostly for referring to thread local storage (TLS). We ignore segmentation-based addressing for the Linear Lower Bound optimization.



(a) x86 instruction (b) DFT representation (c) Distinct EAs (d) Propagation body

Figure 4: Example of how a BBL is transformed during code analysis.

(a) direct jumps, (b) direct branches, and (c) indirect jumps. For direct jumps, BBIDs for successor BBLs can be excluded from logging, since there is only a single, fixed exit once execution enters into BBL0. For example, the transitions from BBL0 to BBL1, and then to BBL2 in Fig. 3 can be excluded. Direct branches can have two outcomes. They are either taken, or fall through where execution continues at the next instruction. We exploit this property to only enqueue a BBID, when the least frequent outcome, according to our dynamic profiling, occurs. For instance, when BBL3 follows BBL2 in Fig. 3. We use the *absence* of BBL3's id to signify that BBL4 followed as expected. Note that if a BBL has two predecessors and it is the most frequent path for only one of them, we log its BBID. Last, for indirect jumps we always record the BBID following them, since they are hard to predict. Fortunately, the number of such jumps are less compared to direct transfers.

Applying our approach on the example CFG from Fig. 3, we would only need to enqueue the id of BBL3 once. Obviously, this approach offers greater benefits when the profiling covers the segments of the application that are to run more frequently in practice. It does not require that it covers all code, but unknown code paths will not perform as well. We evaluate the effects of this optimization for various applications in Sec. 6.

3.2.3 Ring Buffer Fast Checking

For every enqueueing operation from the primary, we should check whether there is available space in the ring buffer to avoid an overflow. However, performing this check within the DBI is very costly, as it requires backing up the `eflags` register. We attempt to reduce the frequency of this operation by performing it selectively. For instance, every 100 BBLs. However, to correctly placing the checks in the presence of CFG loops, so that they are actually performed every 100 executed blocks, is challenging. We mitigated this problem by introducing an algorithm, which finds basic blocks that program execution is ensured to visit at most every k block executions. The problem formally stated is as follows.

We are given a CFG defined as $C = (V, E)$. C is a weighted directed graph where V represents a set of basic blocks and E represents a set of edges that correspond to control transfers among blocks. We also have a weight function $w(v)$ that returns execution counts for $v \in V$. Given that we want to find a subset of vertices S such that:

- For a given parameter k, we can assure that the program execution will visit a node from S at most every k block executions.
- $\sum_{v \in S} w(v)$ is close to the minimum.

The above problem is identical to *the feedback vertex set problem* [13], which is NP-complete when any non-cyclic paths from C is smaller than k. Thus, we can easily reduce our problem to this one and use one of its approximation approaches. Additionally, to take into consideration new execution paths not discovered during profiling, the secondary monitors the ring buffer and signals the primary, when it exceeds a safety threshold. Finally, we also allocate a write-protected memory page at the end of the available space in the ring buffer that will generate a page fault, which can be intercepted and handled, in the case that all other checks fails.

3.3 Secondary Code Generation

During the off-line analysis we generate C code that implements a program to dequeue information from the shared ring-buffer and implement DFT. Listing 1 contains a secondary code block generated for the example in Fig. 4.

3.3.1 Control Flow Restoration

Each code segment begins with a goto label (line 2), which is used with the `goto` statement to transfer control to the segment. Control transfers are made by code appended to the segments. In this example, lines 21 $\sim$ 31 implement a direct branch. First, we check whether the ring buffer contains a valid effective address. The presence of an address instead of BBID (i.e., the *absence* of a BBID) indicates that the primary followed the more frequent path (BBID 0xef13a598), as we determined during code analysis (Sec. 3.2.2). Otherwise, the code most likely did not take the branch and continued to the BBL we previously identified (in this case 0xef13a5ba). We do not blindly assume this, but rather check that this is indeed the case (line 25). We do this to accommodate unexpected control-flow transfers, such as the ones caused by signal delivery (Linux) or an exception (Windows). If an unexpected BBID is found, we perform a look up in a hash table that contains all BBLs identified during the analysis, and the result of the search becomes the target of the transfer in line 29. Note that unknown BBIDs point to a block handling the slow path. While a look up is costly, this path is visited rarely. We also use macros `likely()` and `unlikely()` (lines 5, 21, and 25) to hint the compiler to favor the likely part of the condition.

3.3.2 Optimized DFT

Each code segment generated performs tag propagation for the BBL it is associated with. Effective addresses are referenced directly within the ring buffer, and shadow memory is updated as required by code semantics. For each BBL, optimized tag propagation logic is generated based on the methodology introduced in our previous work [18]. Briefly, this involves extracting tag propagation semantics and representing them in a DFT-specific form, which is susceptible to multiple compiler-inspired optimizations that aim at removing propagation instructions that have no practical effect or cancel out each other, as well as reducing the number of instructions required for propagation by grouping them together.

The propagation code in lines 6 $\sim$ 12 is generated based on the example in Fig. 4. `rbuf()` is a macro that returns a value in the ring buffer relative to the current reading index. Ring buffer accesses correspond to `ea0`, `ea3` and `ea5` in Fig. 4(d). The `MEM_E()` macro in lines 8 $\sim$ 11 translates memory addresses from the real execution context to shadow memory locations, while `REG()` does the same for registers.

```
1  /* BBL label */
2  BB_0xef13a586:
3
4  /* LIFT's FastPath(FP) optimization */
5  if (unlikely( REG(EBX) || MEM_E(rbuf(0) + 4) || REG(EDI) || . . . ))
6  {
7      /* propagation body */
8      REG(EBX) = MEM_E(rbuf(0) + 4);
9      . . .
10     REG(EDI) |= MEM_E(rbuf(2) − rbuf(1) −200);
11     REG(ECX) = MEM_E(rbuf(2));
12 }
13
14 /* update the global address array */
15 garg(0) = rbuf(1);
16
17 /* increase index */
18 INC_IDX(3) ;
19
20 /* control transfer */
21 if (likely(IS_VALID_EA(rbuf(0) )) {
22     /* direct jump */
23     goto BB_0xef13a598;
24 } else {
25     if (likely(rbuf(0) == 0xef13a5ba))
26         goto BB_0xef13a5ba;
27     } else {
28         /* hash lookup for computed goto */
29         goto locateBbIdLabel();
30     }
31 }
```

Listing 1: Example of secondary code for a BBL.

We exploit the fact that we can now run conditional statement fast, since we are running the analysis outside the DBI, to implement a simplified version of the FastPath (FP) optimization proposed by LIFT [28]. This is done in line 8, where we first check whether any of the input and output variables involved in this block are tagged, before propagating empty tags.

Last, in line 15 we save the EA that the inter-block optimization determined it is also used by successor BBLs in the global arguments array `garg`, and progress the increasing index of the ring buffer in line 18.

4. RUNTIME

The off-line application analysis produces the enqueueing stubs for the primary and the analysis body for the secondary. These are compiled into a single dynamic shared object (DSO) and loaded by ShadowReplica at start up. In this section, we describe various aspects of the runtime environment.

4.1 DFT

4.1.1 Shadow Memory Management

The shadow memory structure plays a crucial role in the performance of the secondary. Avoiding conditional statements in in-line DFT systems, frequently restricts them to flat memory structures like bitmaps, where reading and updating can be done directly. An approach that does not scale on 64-bit architectures, because the bitmap becomes excessively large. Due to our design, conditional statements do not have such negative performance effects.

The approach we adopted is borrowed from libdft [19] that implements byte-per-byte allocation, with a low-cost shadow address translation, and dynamic allocation to achieve a small memory footprint. Shadow memory is maintained in sync with the memory used by the application by intercepting system calls that manage memory (e.g., `mmap()`, `munmap()`, `brk()`) and enqueueing special control commands that allocate and deallocate shadow memory areas. The information sent includes the command code and an address range tuple *(offset, length)*.

To access shadow memory, we first consult a page table like data structure, which points to a per-page flat structure. The mechanism does not require a check for page validity because intercepting the memory management calls ensures it. Accesses to unallocated space are trapped to detect invalid memory addresses.

4.1.2 DFT Sources and Sinks for Taint Analysis

Tagging memory areas, when data are read from a file or the network, is performed by generating code that enqueues another control command in the ring buffer. The data sent to the secondary include the command code, an address range tuple *(offset, length)*, and the tag value (or label). The value with which to tag the address range in shadow memory depends on the particular DFT-logic. For instance, DTA uses binary tags, data can tainted or clean. In this case, we can simply omit the tag value.

Checking for tagged data can be either done by enqueueing a control command, or it can be done entirely by the secondary, as it is in the case of DTA. For instance, whenever a BBL ends with a `ret` instruction, the secondary checks the location of the return address in the stack to ensure it is clean. If we wanted to check for sensitive information leaks, we would have to issue a command from the primary whenever a `write()` or `send()` system call is made. Note that all checks are applied on the secondary, which can take action depending on their outcome. For DTA, if a check fails, we immediately terminate the application, as it signifies that the application's control flow has been compromised. Other actions could involve logging memory contents to analyze the attack [32], or allowing the attack to continue for forensics purposes [27].

Moreover, the primary synchronizes with the secondary before potentially sensitive system calls (e.g., `write()` and `send()`) that may leak sensitive information from the system. This ensures that the secondary does not fall behind, allowing an attack to go undetected for a long time, or at least not before leaking information. This granularity of synchronization does leave a small but, nevertheless, existing time window that the attacker can leverage to suppress the secondary and our checks. We have complemented our implementation to also require synchronization whenever a new BBL (i.e., a new execution path) is identified at run time to guard against such an event.

4.1.3 Generic Block Handler

BBLs that were not identified during profiling are processed by the Generic Block Handler (GBH), our slow-path implementation of DFT. In this case, the primary on-the-fly transforms instructions and their operands into their DFT-specific representation of TFA. Then, it implements a stack-based interpreter that enqueues the DFT operations (i.e., `AND`, `OR`, `ASSIGN`), which need to be performed by the secondary, in the ring buffer.

4.2 Ring Buffer

We implemented the ring buffer, as a lock-free, ring buffer structure for a single producer (primary), single consumer (secondary) model. Lamport et al. [20] initially proposed a lock-free design for the model, however, the design suffers from severe performance overheads on multi-core CPUs due to poor cache performance [14]. The most serious problem we observed was *forced* L1 eviction that occurs as the secondary attempts to read the page that was just modified by the primary, while it is still in the primary's L1 cache and before it is committed to RAM. In certain cases, the overhead due to ring-buffer communication increased to the point that our framework became unusable. To mitigate the problem, we switched to

a N-way buffering scheme [33], where the ring-buffer is divided into N separate sub-buffers, and primary and secondary threads work on different sub-buffers at any point of time, maintaining a distance of least as much as the size of a single sub-buffer.

Synchronizing between primary and secondary is also performed through the shared data structure. We extended the typical N-way buffering implementation, by adding a control entry for each sub-buffer. When the primary finishes updating a sub-buffer, it records the number of entries it enqueued in the control entry and moves the next sub-buffer. If the next sub-buffer's control entry is zero, the secondary already finished its work, so the primary can proceed to fill that up as well. Otherwise, the primary waits, periodically polling the control entry. Likewise, the secondary polls the control entry to check whether there are data for it to process. When primary and secondary need to be synchronized (e.g., before the execution of sensitive system calls), the primary retires early from its current sub-buffer and waits until the secondary clears up all entries and updates the control entry with zero, thus synchronizing.

Data Encoding. Data enqueued in the ring buffer belong to one of the following three groups: (a) effective addresses (EAs), (b) BBIDs, and (c) control commands. We exploit the fact that valid EAs point to user-space ranges (0G $\sim$ 3G range on x86 Linux) to quickly differentiate them from other entries without using additional bits. We map BBIDs and control commands to address ranges that correspond to kernel space (3G $\sim$ 4G range on x86 Linux), which immediately differentiates them from EAs that are always in lower address ranges. This design is easily applicable on 64-bit architectures and other operating systems.

4.3 Support for Multi-threaded Applications

Multi-threaded applications are composed by *critical regions*, where they access shared data in a safe manner by acquiring a lock. To ensure that secondary threads operate on shadow memory in the same order that the primary threads operated on real data, we need to identify when a thread enters such a critical region. We accomplish this by intercepting mutex related calls in the POSIX threads library, namely `pthread_mutex_lock()` and `pthread_mutex_unlock()`. Our approach can be easily extended to also support other locking schemes, but currently does not guarantee shadow memory consistency for shared-data accesses where no locking is performed.

Lock-protected critical regions are handled as follows. When one of the primary threads acquires a lock, it waits until its secondary clears all entries from the ring buffer, essentially synchronizing with the secondary. Every lock in the primary is associated with a separate ring buffer structure and another secondary thread attached to it, when it is created. The primary proceeds to switch its ring-buffer pointer to the one that belongs to the particular lock and resumes execution. Similarly, when releasing the lock, the primary synchronizes with the current secondary and resumes using its original ring buffer. This process ensures that the order of shadow memory operations is consistent with the order of operations in the application. In the future, we plan to explore additional optimizations targeting multi-threaded applications.

5. IMPLEMENTATION

We implemented the dynamic profiler as tool for the Pin DBI framework in $\sim$ 3,800 lines of C++. The runtime environment, responsible for injecting the enqueueing stubs in the application and hosting the secondary threads, was also build as a tool for Pin in $\sim$4,000 lines of C++ code. The code analysis and generation represents the majority of our effort. It consists of $\sim$12,000 lines of Python code.

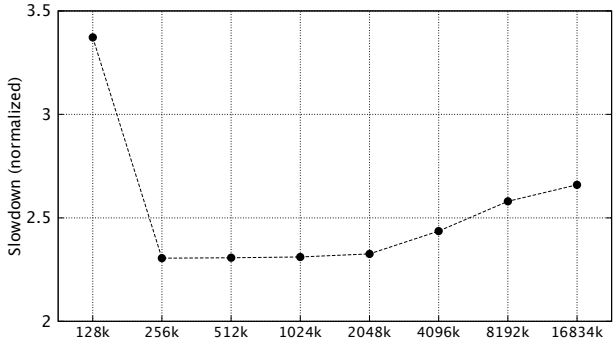


Figure 5: Choosing right ring-buffer size.

Currently, all of the secondary's functionality is implemented as a single function with labels for each BBL segment. This design choice was made to exploit the compiler's intra-procedure optimizations that can better organize the code and improve locality. An alternate design adopting a function abstraction per BBL cannot enjoy these optimizations, and would suffer higher function call/return overheads. We discovered that a large number of labels caused the GCC compiler to fail due to resource exhaustion when running CFG related optimizations.[3] This currently prevents us from building a single-function program for applications with more than $\sim$ 50K BBLs. As a workaround, we divide the analysis program into multiple sub-functions, where each represents a clustered subgraph from the CFG.

Fig. 5 shows how ShadowReplica's communication cost varies, as we set the size of the ring-buffer with different values. In this experiment, we ran `bzip2` compression against Linux kernel source (v3.7.10; $\sim$ 476MB) using a ring buffer comprising of 8 sub-buffers, and having the secondary only consume data. When the size of the whole structure is too small (128k), performance suffers due to L1 cache eviction, as each sub-buffer of 16KB is smaller than the size of L1 cache (32KB). When it becomes larger performance improves, until it becomes larger than 4096KB, where performance starts to suffer due to increased L3 cache misses. In our prototype implementation, we set the ring buffer size to 512KB and configured it with 8 sub-buffers. The size of each sub-buffer is 64KB, twice as large as our test system's L1 cache size.

6. EVALUATION

We evaluated ShadowReplica using common command-line utilities, the SPEC CPU2006 benchmark suite, and a popular web and database (DB) server. Our aim is to quantify the performance gains of our system, and analyze the effectiveness of the various optimizations and design decisions we took. We close this section with a security evaluation of the DTA and CFI tools we built over ShadowReplica.

The results presented throughout this section are mean values, calculated after running 5 repetitions of each experiment. Our testbed consisted of an 8-core host, equipped with two 2.66 GHz quad-core Intel Xeon X5500 CPUs and 48GB of RAM, running Linux (Ubuntu 12.04LTS with kernel v3.2). Note that ShadowReplica can run on commodity multi-core CPUs, such as the ones found on portable PCs. We used a relatively strong server for the evaluation because it was available at the time. The only crucial resource is the L3 cache, 32MB in our case, as the communication overhead

[3]Note that we have filed a bug for this issue [17], but it has not been completely resolved yet.

Category	Application	# BBLs	# Ins.	Unopt	Optimization				Rbuf	# DFT operands	Time (sec.)
					Intra	DFT	Inter	Exec			
Utilities	bzip2	6175	8.59	4.97	2.93	2.62	2.21	1.40	39.70 %	2.69	156.66
	tar	8615	5.20	2.31	1.82	1.70	1.53	0.69	33.49 %	4.52	141.79
SPEC2006	473.astar	5069	7.38	4.09	2.48	2.15	2.05	1.40	22.11 %	3.30	89.78
	403.gcc	78009	4.37	2.54	1.93	1.84	1.42	0.55	56.87 %	5.57	8341.53
	401.bzip2	4588	7.85	4.12	2.75	2.28	2.10	1.27	32.96 %	2.77	109.60
	445.gobmk	25939	5.97	3.04	2.11	1.90	1.72	0.95	27.09 %	3.56	674.74
	464.h264	11303	5.76	4.88	3.12	3.04	1.72	0.84	74.78 %	3.57	242.67
	456.hmmer	7212	19.14	12.62	6.99	6.66	6.34	5.95	59.62 %	6.64	115.81
	462.libquantum	3522	9.16	4.66	2.61	2.27	2.04	1.09	56.69 %	3.88	58.09
	429.mcf	3752	5.96	3.21	2.12	1.83	1.76	1.06	26.38 %	3.19	57.62
	471.omnetpp	15209	5.36	2.88	2.11	1.95	1.80	1.05	6.59 %	3.31	234.35
	400.perl	27391	5.71	2.86	1.89	1.82	1.64	0.84	7.79 %	4.47	421.10
	458.sjeng	5535	5.29	2.49	1.74	1.65	1.54	0.82	21.51 %	4.07	81.51
	483.xalanc	34881	4.79	2.02	1.54	1.44	1.34	0.50	21.40 %	5.97	1189.45
Server application	Apache	23079	8.39	3.17	1.91	1.84	1.6	1.05	28.08 %	5.57	706.63
	MySQL	40393	6.87	4.21	2.19	2.08	1.80	1.10	11.00 %	5.42	1486.60
Average		18792	7.24	4.17	2.52	2.32	2.04	1.29	32.88 %	4.28	881.74

Table 1: Results from static analysis. *Unopt* indicates the number of enqueue operations that a naive implementation requires. The *Intra*, *DFT*, *Inter*, and *Exec* columns correspond to the number of enqueue operations after progressively applying our intra-block, DFT, inter-block, and control flow optimizations. *Rbuf* shows the percentage of BBLs that need to be instrumented with analysis code that checks for ring buffer overflows. *# DFT operands* shows the operands from the secondary's analysis body. The last column, *Time*, shows the time for offline analysis.

heavily depends on its size. The version of Pin used during the evaluation was 2.11 (build 49306). While conducting our experiments hyper-threading was disabled and the host was idle.

6.1 Effectiveness of Optimizations

Table 1 summarizes the effects our optimizations had on reducing the amount of information that needs to be communicated to the secondary, and the effectiveness of the ring buffer fast checking optimization. The *# BBLs* column indicates the number of distinct BBLs discovered for each application, using the profiling process outlined in Sec. 3.1. *# Ins.* gives the average number of instructions per BBL. The *Unopt* column shows the average number of enqueue operations to the ring buffer that a naive implementation would require per BBL. The *Intra*, *DFT*, *Inter*, and *Exec* columns correspond to the number of enqueue operations after progressively applying the intra-block, DFT, and inter-block optimizations presented in Sec. 3.2.1, as well as the optimizations related to control flow recording (Sec. 3.2.2). *Rbuf* shows the percentage of BBLs that need to be instrumented with checks for testing if the ring buffer is full, according to the fast checking algorithm (Sec. 3.2.3). *# DFT operands*, shows *per BBL* average number of register and memory operands appeared from the secondary's analysis body. The last column, *Time* shows the time for offline analysis to generate codes for the primary and the secondary.

On average, a naive implementation would require 4.17 enqueue operations per BBL, for communicating EAs and control flow information to the secondary, and instrument 100% of the BBLs with code that checks for ring buffer overflows. Our optimizations reduce the number of enqueueing operations to a mere 1.29 per BBL, while only 32.88% of BBLs are instrumented to check for ring buffer overflows. A reduction of 67.12%.

While offline analysis requires a small amount of time (on average 881 sec.) for most programs, 403.gcc takes exceptionally long (8341 sec.) to complete. This is due to the program structure, which has a particularly dense CFG, thus making the *Rbuf* optimization wasting a long time ($\sim$ 6500 sec.) only to enumerate all available

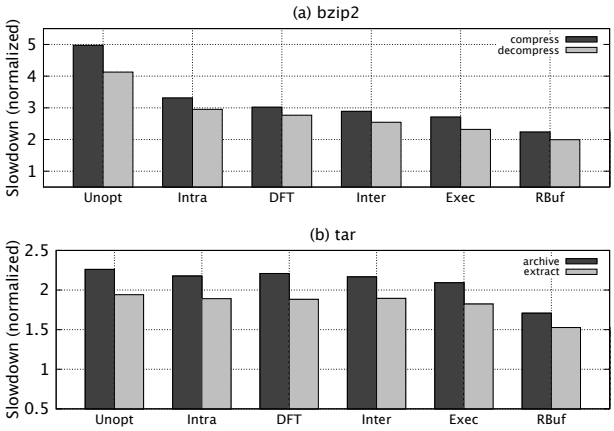


Figure 6: The slowdown of the primary process imposed by ShadowReplica, and the effects of our optimizations, when running `bzip2` and `tar`.

cycles to perform the analysis described in Sec. 3.2.3. We believe that this can be alleviated by parallelizing the algorithm.

To evaluate the impact of our various optimizations on the primary's performance, we used two commonly used Unix utilities, `tar` and `bzip2`. We selected these two because they represent different workloads. `tar` performs mostly I/O, while `bzip2` is CPU-bound. We run the GNU versions of the utilities natively and over ShadowReplica, progressively enabling our optimizations against a Linux kernel source "tarball" (v3.7.10; $\sim$476MB).

We measured their execution time with the Unix `time` utility and draw the obtained results in Fig. 6. *Unopt* corresponds to the runtime performance of an unoptimized, naive, implementation, whereas *Intra*, *DFT*, *Inter*, *Exec*, and *RBuf* demonstrate the benefits of each optimization scheme, when applied cumulatively. With

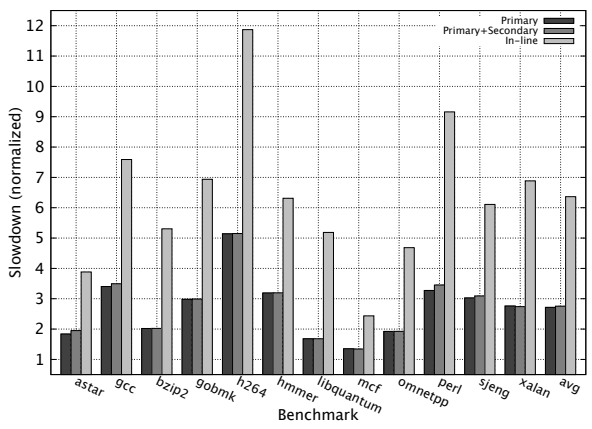


Figure 7: Running the SPEC CPU2006 benchmark suite with all optimizations enabled. *Primary* denotes the slowdown of the primary thread alone. *Primary+Secondary* is the overhead of ShadowReplica when it performs full-fledged DFT. *In-line* corresponds to the slowdown imposed when the DFT process executes in-line with the application.

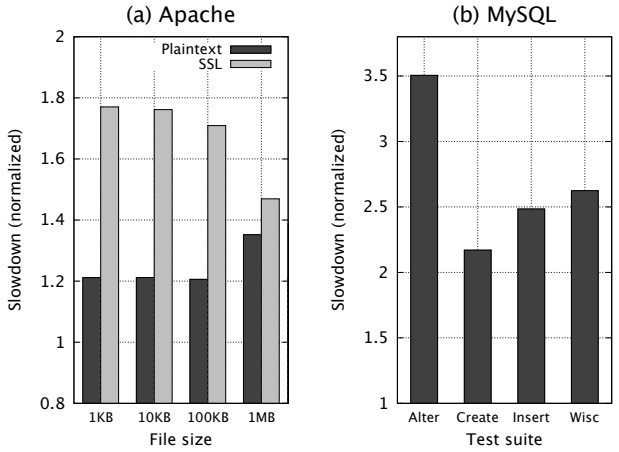


Figure 8: The slowdown incurred by ShadowReplica on the Apache web server and MySQL DB server.

all optimizations enabled, the slowdown imposed to `bzip2` drops from 5×/4.13× down to 2.23×/1.99× for compress/decompress (55%/51.82% reduction). Similarly, `tar` goes from 2.26×/1.94× down to 1.71×/1.53× for archive/extract (24.33%/ 21.13% reduction). It comes as no surprise that I/O bound workloads, which generally also suffer smaller overheads when running with in-lined DFT, benefit less from ShadowReplica. We also notice that *Intra* and *RBuf* optimizations have larger impact on performance.

6.2 Performance

SPEC CPU2006. Fig. 7 shows the overhead of running the SPEC CPU2006 benchmark suite under ShadowReplica, when *all* optimizations are enabled. *Primary* corresponds to the slowdown imposed by the primary thread alone. *Primary+Secondary* is the overhead of ShadowReplica when both the primary and secondary threads and running, and the secondary performs full-fledged DFT. Finally, *In-line* denotes the slowdown imposed when the DFT process executes in-line with the application, under our accelerated DFT implementation [18]. On average, *Primary* imposes a 2.72× slowdown on the suite, while the overhead of the full scheme (*Primary+Secondary*) is 2.75×. *In-line* exhibits a 6.37× slowdown, indicating the benefits from the decoupled and parallelized execution of the DFT code (56.82% reduction on the performance penalty).

During our evaluation, we noticed that for some benchmarks (astar, perlbmk, gcc, sjeng) the slowdown was not bound to the primary, but to the secondary. These programs generally required a high number of DFT operands to be sent to the secondary (*# DFT operands* from Table 1) compared to the number of enqueued entries.

The significant slowdown with the *h264ref* benchmark was due to the way Pin handles `rep`-prefixed string instructions (i.e., `rep movs`, `rep cmps`), heavily used in this benchmark. Pin assumes that the tool developer wants to instrument each one of these repetitions and transforms these instructions into explicit loops, introducing additional overhead. However, our DFT-specific representation [18], which captures source, destination, and offset from these instructions, does not require such a transformation. We used one of Pin's command-line options to disable this feature, which reduced our slowdown in this benchmark from 5× to 3.05×.

Apache and MySQL. We proceed to evaluate ShadowReplica with larger and more complex software. Specifically, we investigate how ShadowReplica behaves when instrumenting the commonly used Apache web server. We used Apache v2.2.24 and let all options to their default setting. We measured Apache's performance using its own utility `ab` and static HTML files of different size. In particular, we chose files with sizes of 1KB, 10KB, 100KB, and 1MB, and run the server natively and with ShadowReplica with all optimizations enabled. We also tested Apache with and without SSL/TLS encryption (i.e., *SSL* and *Plaintext*, respectively). Fig. 8 (a) illustrates our results. Overall, ShadowReplica has a 24% performance impact, on average, when Apache is serving files in plaintext. Unsurprisingly, the slowdown is larger when running on top of SSL (62%). The reason behind this behavior is that the intensive cryptographic operations performed by SSL make the server CPU-bound. In Fig. 8 (b), we present similar results from evaluating with the MySQL DB server, a multi-threaded application which spawned 10 $\sim$ 20 threads during its evaluation. We used MySQL v5.0.51b and its own benchmark suite (`sql-bench`), which consists of four different tests that assess the completion time of various DB operations like table creation and modification, data selection and insertion, etc. The average slowdown measured was 3.02× (2.17× – 3.5×).

6.3 Computational Efficiency

One of the goals of ShadowReplica was to also make DFT more efficient computation-wise. In other words, we not only desired to accelerate DFT, but also do it using less CPU resources. To evaluate this aspect of our approach, we chose two benchmarks from the SPEC CPU2006 suite; 401.bzip2 and 400.perl. Our choice was was not arbitrary. During our performance evaluation, we observed that in the first benchmark, DFT was running faster than the application. Performance is, hence, bound by the primary thread. On the other hand, in the second benchmark the secondary thread, performing DFT, was slower, hence its performance is bound by the secondary thread.

We run ShadowReplica and our accelerated in-line DFT implementation with these two benchmarks, and measured their CPU usage using the `perf` tool. Fig. 9 presents the results of our experiment. We run ShadowReplica with both primary and secondary

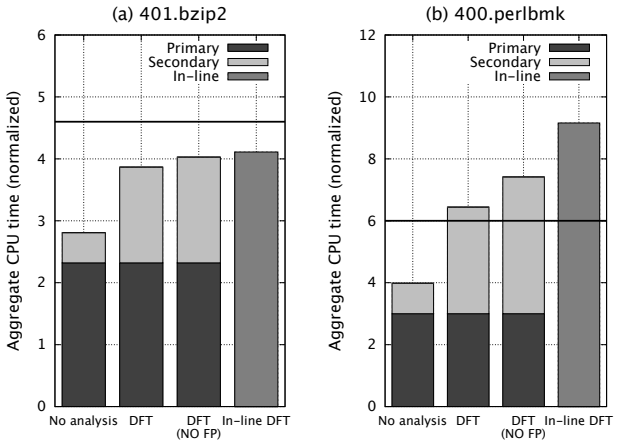


Figure 9: Aggregated CPU time consumed by two SPEC CPU2006 benchmarks when run under different configurations of ShadowReplica, and under in-line DFT. For ShadowReplica, we draw the CPU time taken by the primary and secondary threads separately. A darker horizontal line shows the threshold above which the secondary dominates CPU usage.

Application	Vulnerability	CVE-ID
exim-4.69	Format string	CVE-2010-4344
proftpd-1.3.3a	Stack overflow	CVE-2010-4221
nginx-0.6.32	Buffer underflow	CVE-2009-2629
memcached 1.1.12	Integer overflow	CVE-2009-2415
htget-0.93	Stack overflow	CVE-2004-0852
WsMp3-0.0.8	Heap overflow	CVE-2003-0338
athttpd-0.4b	Buffer overflow	CVE-2002-1816

Table 2: ShadowReplica-{DTA, CFI} successfully prevented the listed attacks.

threads running, having the secondary perform no analysis (*No analysis*), implementing DFT using all optimizations (*DFT*), and without the FastPath optimization from LIFT [28] (*DFT (NO FP)*). The last column (*in-line DFT*) shows the result for a DFT implementation that in-lines the analysis to the application process [18]. CPU usage is partitioned to show the amount of CPU cycles taken from the primary and secondary threads separately. The darker horizontal line visualizes the tipping point where the secondary thread starts dominating performance (i.e., it is slower than the primary), when we are running ShadowReplica with DFT and all optimizations enabled.

A take-out from these results is that the aggregated CPU usage of ShadowReplica is less or equal than that of in-line DFT analysis. In other words, we manage to satisfy equation 1 from Sec. 2.1. Astoundingly, in the case of 401.bzip2, we are so much more efficient that we require $\sim$30% less CPU cycles to apply DFT.

6.4 Security

The purpose of developing the DTA and CFI tools over ShadowReplica was not to provide solid solutions, but to demonstrate that our system can indeed facilitate otherwise complex security tools. Nevertheless, we tested their effectiveness using the set of exploits listed in Table 2. In all cases, we were able to successfully prevent the exploitation of the corresponding application.

During the evaluation, DTA did not generate any false positives, achieving the same level of correctness guarantees to our previous DFT implementation [19, 18]. However, CFI suffered from some false positives due to the inability to obtain an accurate CFG by static and dynamic profiling. We leave the task of improving the soundness of CFG as a future work.

Having DTA and CFI implemented mostly from the secondary, performance overhead was negligible by having $\sim$ 5% slowdown.

7. RELATED WORK

The idea of decoupling dynamic program analyses from execution, to run them in parallel, has been studied in past in various contexts [31, 8, 25, 33, 14, 26, 6]. Aftersight [8], ReEmu [6], and Paranoid Android [26] leverage record and replay for recording execution and replaying it, along with the analysis, on a remote host or a different CPU (replica). They are mostly geared toward off-line analyses and can greatly reduce the overhead imposed on the application. However, the speed of the analysis itself is not improved, since execution needs to be replayed and augmented with the analysis code on the replica.

SuperPin [31] and Speck [25] use speculative execution to run application and (in-lined) analysis code in multiple threads that execute in parallel. These systems sacrifice significant processing power to achieve speed up. Furthermore, handling multi-threaded applications without hardware support remains a challenging issue for this approach.

CAB [14] and PiPA [33] aim at offloading the analysis code alone to another execution thread, and they are the closest to ShadowReplica. However, neither of the two has been able to deliver the expected performance gains, due to (a) naively collecting information from the application, and (b) the high overhead of communicating it to the analysis thread(s). This paper demonstrated how to tackle these problems.

DFT has been broadly used in the security domain but also elsewhere. TaintCheck [24] utilizes DTA for protecting binary-only software against buffer overflows and other types of memory corruption attacks. It applies DTA, using Valgrind [23], to detect illegitimate uses of network data that could enable attackers to seize control of vulnerable programs. Panorama [32] makes use of DTA for analyzing malware samples similarly to Argos [27]. TaintBochs [9] utilizes a whole system emulator for studying the lifetime of sensitive data, whereas TaintEraser [34] relies on DTA for preventing sensitive information leaks. ConfAid [3] leverages DFT for discovering software misconfigurations. Dytan [10] is a flexible DFT tool, allowing users to customize its data sources and sinks, as well as the propagation policy, while it can also track data based on control-flow dependencies.

8. CONCLUSION

We presented ShadowReplica, a new and efficient approach for accelerating DFT and other shadow memory-based analyses, by decoupling analysis from execution and utilizing spare CPU cores to run them in parallel. We perform a combined off-line dynamic and static analysis of the application to minimize the data that need to be communicated for decoupling the analysis, and optimize the code used to perform it. Furthermore, we design and tune a shared ring buffer data structure for efficiently sharing data between threads on multi-core CPUs. Our evaluation shows that performing DFT using ShadowReplica is more than 2$\times$ faster and uses up to 30% less CPU cycles than in-line DFT. Although the overall performance impact of DFT remains significant, we hope that our optimizations will bring it closer to becoming practical for certain environments.

9. ACKNOWLEDGEMENT

We want to express our thanks to the anonymous reviewers for their valuable comments and to Rob Johnson, our shepherd, for his guidance. This work was supported by the US Air Force, the Office of Naval Research, DARPA, and the National Science Foundation through Contracts AFRL-FA8650-10-C-7024, N00014-12-1-0166, FA8750-10-2-0253, and Grant CNS-12-22748, respectively, with additional support by Intel Corp. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government, the Air Force, ONR, DARPA, NSF, or Intel.

10. REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proc. of CCS*, 2005.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2006.

[3] M. Attariyan and J. Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *Proc. of OSDI*, 2010.

[4] D. Bruening and Q. Zhao. Practical memory checking with dr. memory. In *Proc. of CGO*, 2011.

[5] D. Bruening, Q. Zhao, and S. Amarasinghe. Transparent dynamic instrumentation. In *Proc. of VEE*, 2012.

[6] Y. Chen and H. Chen. Scalable deterministic replay in a parallel full-system emulator. In *Proc. of PPoPP*, 2013.

[7] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: a platform for in-vivo multi-path analysis of software systems. In *Proc. of ASPLOS*, 2011.

[8] J. Chow, T. Garfinkel, and P. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *Proc. of USENIX ATC*, 2008.

[9] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *Proc. of USENIX Security*, 2004.

[10] J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *Proc. of ISSTA*, 2007.

[11] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. In *Proc. of SOSP*, 2005.

[12] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. of OSDI*, 2010.

[13] P. Festa, P. M. Pardalos, and M. G. Resende. Feedback set problems. *Handbook of Combinatorial Optimization*, 4:209–258, 1999.

[14] J. Ha, M. Arnold, S. M. Blackburn, and K. S. McKinley. A concurrent dynamic analysis framework for multicore hardware. In *Proc. of OOPSLA*, 2009.

[15] Hex-Rays. The IDA Pro Disassembler and Debugger, cited Aug. 2013. http://www.hex-rays.com/products/ida/.

[16] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob. Cmp$im: A pin-based on-the-fly multi-core cache simulator. In *Proc. of MoBS*, 2008.

[17] K. Jee. Bug 56113. GCC Bugzilla, cited Aug. 2013. http://gcc.gnu.org/bugzilla/show_bug.cgi?id=56113.

[18] K. Jee, G. Portokalidis, V. P. Kemerlis, S. Ghosh, D. I. August, and A. D. Keromytis. A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware. In *Proc. of NDSS*, 2012.

[19] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis. libdft: practical dynamic data flow tracking for commodity systems. In *Proc. of VEE*, 2012.

[20] L. Lamport. Specifying Concurrent Program Modules. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1983.

[21] K. H. Lee, X. Zhang, and D. Xu. High accuracy attack provenance via binary-based execution partition. In *Proc. of NDSS*, 2013.

[22] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proc. of PLDI*, 2005.

[23] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proc. of PLDI*, 2007.

[24] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. of NDSS*, 2005.

[25] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. Parallelizing security checks on commodity hardware. In *Proc. of ASPLOS*, 2008.

[26] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. Paranoid Android: Versatile protection for smartphones. In *Proc. of ACSAC*, 2010.

[27] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an Emulator for Fingerprinting Zero-Day Attacks. In *Proc. of EuroSys*, 2006.

[28] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In *Proc. of MICRO*, 2006.

[29] A. Slowinska, T. Stancescu, and H. Bos. Howard: a dynamic excavator for reverse engineering data structures. In *Proc. of NDSS*, 2011.

[30] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proc. of SOSP*, 1993.

[31] S. Wallace and K. Hazelwood. Superpin: Parallelizing dynamic instrumentation for real-time performance. In *Proc. of CGO*, 2007.

[32] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proc. of CCS*, 2007.

[33] Q. Zhao, I. Cutcutache, and W. Wong. PiPA: pipelined profiling and analysis on multi-core systems. In *Proc. of CGO*, 2008.

[34] D. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. TaintEraser: Protecting sensitive data leaks using application-level taint tracking. In *SIGOPS Oper. Syst. Rev.*, 2011.