A General Approach for Efficiently Accelerating Software-based Dynamic Data Flow Tracking on Commodity Hardware

Kangkook Jee[1], Georgios Portokalidis[1], Vasileios P. Kemerlis[1], Soumyadeep Ghosh[2], David I. August[2], and Angelos D. Keromytis[1]

[1]*Columbia University, New York, NY, USA*
[2]*Princeton University, Princeton, NJ, USA*

Abstract

Despite the demonstrated usefulness of dynamic data flow tracking (DDFT) techniques in a variety of security applications, the poor performance achieved by available prototypes prevents their widespread adoption and use in production systems. We present and evaluate a novel methodology for improving the performance overhead of DDFT frameworks, by combining static and dynamic analysis. Our intuition is to separate the program logic from the corresponding tracking logic, extracting the semantics of the latter and abstracting them using a Taint Flow Algebra*. We then apply optimization techniques to eliminate redundant tracking logic and minimize interference with the target program. Our optimizations are directly applicable to binary-only software and do not require any high level semantics. Furthermore, they do not require additional resources to improve performance, neither do they restrict or remove functionality. Most importantly, our approach is orthogonal to optimizations devised in the past, and can deliver additive performance benefits. We extensively evaluate the correctness and impact of our optimizations, by augmenting a freely available high-performance DDFT framework, and applying it to multiple applications, including command line utilities, server applications, language runtimes, and web browsers. Our results show a speedup of DDFT by as much as* $2.23\times$*, with an average of* $1.72\times$ *across all tested applications.*

1 Introduction

Fine-grained dynamic data flow tracking (DDFT) has been shown to be a useful tool for a broad variety of application domains, including software security [14, 22, 10], information flow control [39, 45, 25, 44], data lifetime analysis [8], software and configuration problem diagnosis [2, 12], and others. As there is no explicit support for DDFT in commodity hardware, the only practical means for applying these techniques are various software-based implementations. Unfortunately, these exhibit prohibitive performance overhead, ranging from $3\times$ to $100\times$ when applied to pure binaries, and $1.5\times$ to $3\times$ when inserted during compilation [41, 21]. Efforts to alleviate the high cost of DDFT include exploiting parallelization, applying the instrumentation selectively, and dynamically deactivating it when no data needs to be tracked [32, 28, 36, 7, 17, 29, 20]. Despite improvements, the overheads remain significant. Additionally, some optimizations require availability to source code, additional resources (*e.g.,* CPU cores), or only perform well under certain conditions (*e.g.,* when the tainted data are seldom accessed). Consequently, these shortcuts are generally seen as too impractical, or undesirable, except for specific scenarios.

Existing binary-only DDFT systems operate by individually instrumenting each program instruction that propagates data, adding one or more instructions that implement the data tracking logic. Higher-level semantics are not available and cannot be easily extracted so, for example, it is not possible to automatically determine that a function copies data from one buffer to another, and directly generate the tracking code for that. This introduces a lot of redundant instructions and greatly reduces the proportion of "productive" instructions in the program. The problem is exacerbated in register-starved architectures like Intel's *x86*, where the program and data tracking logic compete for general purpose registers.

We present a novel optimization approach to DDFT, based on combining static and dynamic analysis, which significantly improves its performance. Our methodology is based on separating program logic from taint tracking logic, extracting the semantics of the latter, and representing them using a *Taint Flow Algebra*. We apply multiple code optimization techniques to eliminate redundant tracking logic and minimize interference with the target program, in a manner similar to an optimizing compiler. We draw on the rich theory on basic block optimization and data flow analysis, done in the context of compilers, to argue the safety and correctness of our algorithm using a formal framework.

We evaluate the correctness and performance of our methodology by employing a freely available DDFT framework libdft [19] and show that the code generated by our analysis behaves correctly when performing dynamic taint analysis (DTA) [27]. We evaluate the performance gains achieved by our various optimizations using several Linux applications, including commonly used command-line utilities (`bzip`, `gzip`, `tar`, `scp`, *etc.*), the SPEC CPU 2000 benchmarks, the MySQL database server, the runtimes for the PHP and JavaScript languages, and web browsers. Our results indicate performance gains as high as $2.23\times$, and an average of $1.72\times$ across all tested applications.

The main contributions of this paper are:

- We demonstrate a methodology for segregating program logic from data tracking logic, and optimizing the latter separately using a combination of static and dynamic analysis techniques. Our approach is generic and can benefit most binary-only DDFT approaches.
- We define a Taint Flow Algebra that we use to represent the tracking logic of programs to assist the analysis process.
- We demonstrate the application of classical compiler optimization techniques to the defined Taint Flow Algebra. For instance, dead code elimination, copy propagation, and algebraic simplification are implicitly performed in the course of our static analysis.
- The resulting optimized taint tracking code is up to $2.23\times$ faster than unoptimized data tracking. Furthermore, our optimizations are orthogonal to other performance improvement approaches, and can provide additive benefits. Although the overall performance impact of DDFT remains significant, we believe that it has come much closer to becoming practical for (certain) production environments.

The rest of this paper is organized as follows. Section 2 presents background information (also placing related work in context) and introduces our approach. Section 3 describes the Taint Flow Algebra and our optimizations. We discuss the tools developed and interfacing with an existing DDFT framework in Section 4. We evaluate our proposal in Section 5. Discussion and future work are in Section 6 and Section 7. We conclude the paper in Section 8.

2 Background and Overview

We briefly describe some of the many uses of DDFT in the security field, as a core motivation for our work. We also discuss the limiting factors regarding performance, casting the problem in terms of code optimization and proposing a new methodology based on combining static and dynamic analysis to coalesce and minimize tracking code.

2.1 Dynamic Data Flow Tracking

DDFT is the process of accurately tracking selected data of interest, as they flow during program execution. Among other uses, DDFT has been employed to provide insight in the behavior of applications and systems [14, 8, 25, 12], and to assist in the identification of configuration errors [2]. Most prominently, it has been used in the security field to defend against various software exploits [27, 18, 31, 3, 41], and to enforce information flow by monitoring and restricting the use of sensitive data [39, 24]. For the former, the network is usually defined as the source of interesting or "tainted" data, while the use of tainted data is disallowed in certain program locations (*e.g.,* in instructions manipulating the control flow of programs, such as indirect branch instructions and function calls). For the latter, the developer or the user is responsible for specifying the data that needs to be tracked and the restrictions on their use.

The specifics of DDFT can vary significantly depending on ones goals, performance considerations, and deployment platform. One possible classification of existing mechanisms can be made based on the means by which the tracking logic is augmented on regular program execution. For example, DDFT can be performed by inserting data tracking logic *statically* during the compilation of software, or by performing source-to-source code transformation [33, 21, 5]. It can also be applied *dynamically* by augmenting instrumentation code on existing binaries using dynamic binary instrumentation (DBI) [9, 32] or a modified virtual machine (VM) [31, 17]. Finally, DDFT can be also performed in hardware [11, 38, 13, 40].

A different classification can be made depending on the granularity of the tracking, which can be *fine- or coarse-grained*. Assuming a single-bit tag that can be either "on" or "off", fine-grained mechanisms frequently

assign one tag per byte or four-byte word. Larger tags that enable the tracking of different types, or "colors", of data are also possible, but they complicate the process and significantly raise overhead. On the other hand, coarse-grained mechanisms may track data at the file level and tag entire processes [43]. Additionally, data tracking can be restricted to *explicit* data dependencies, or extend to *implicit* dependencies [9, 39]. For example, in `count += len`, the variable `count` depends explicitly on variable `len` as its value is the result of an arithmetic operation with `len`. However, in `if (len > 0) then count += 1`, variable `count` only depends implicitly on `len` because its value is not the result of a direct operation with `len`, even though it is only increased when `len` > 0. The tracking of implicit data flows was introduced in Dytan [9], but it results in an excessive amount of false positives (*i.e.,* over-tainting). DTA++ [18] attempts to address taint-explosion by allowing the user to only track certain implicit flows.

Compiler-based DDFT systems incur lower overheads than dynamic systems, but are not applicable to binary-only software. Considering the huge number of legacy software and third-party libraries, this greatly limits the applicability of such systems in a large class of software. In contrast, dynamic systems operate on binaries, but incur high runtime overheads. The most practical solutions (*i.e.,* fast and work with binaries) require hardware support, which is not available. *This paper aims to provide a general methodology for improving the performance of software (DBI- or VM-based) fine-grained dynamic data flow tracking for binaries*. Our approach can handle both explicit and implicit data dependencies. However, for simplicity and lack of space, the examples we use only show explicit dependencies. We do not consider coarse-grained tracking mechanisms, since they incur little overhead, and are significantly different in nature and purpose from their fine-grained counterparts.

2.2 Previous Optimization Approaches

DDFT approaches that operate on binary-only software typically incur high slowdowns ranging from $3\times$ to $100\times$ on CPU-intensive workloads [32, 8]. Much research focused on the overhead of DDFT for systems using binary instrumentation and virtualization, which suffer the largest overheads. Important paths explored by previous research include:

Additional resources Data tracking is decoupled from program execution and is offloaded to one or more CPUs [28, 35] or even to remote hosts [7, 30]. These techniques can also parallelize the tracking itself and scale to multiple cores.

Static analysis The application is statically analyzed before inserting the tracking code to avoid injecting redundant tracking instructions [21]. The speed-up can be considerable when narrowing down the data tracking to particular elements of a program [36], while the analysis can also be user assisted [45].

Intermittent tracking Performance is improved by dynamically deactivating data tracking under certain conditions (*e.g.,* when the program does not operate on tagged data [32, 17]). Data tracking can also be applied on-demand based on CPU usage, or due to manual activation [29]. Tracking only parts of an application also reduces overhead [20].

Many of these approaches improve performance, but are not always costless. For instance, approaches utilizing additional hardware resources do not always scale (*i.e.,* adding CPUs does not further improve performance). Also, static analysis methods frequently require access to source code, and oftentimes sacrifice functionality and/or accuracy. Intermittent tracking offers significant performance improvements on certain favorable scenarios, but the cost of activating and deactivating it can actually increase overhead on unfavorable ones.

The optimizations we describe in this paper are based on statically and dynamically analyzing program binaries. More importantly though, they can be combined with some of the approaches listed above to offer additive performance benefits. For example, our approach can optimize the tracking logic that runs on itself on spare core or remote host. It can also be combined with other static analysis approaches. For instance, Saxena *et al.* [36] use static analysis to recover the high-level semantics of a program (*e.g.,* local variables, stack conventions, *etc.*). Their approach is not applicable to all binaries, but it is orthogonal to ours. Finally, our methodology can also benefit intermittent tracking approaches to accelerate the tracking logic when it is activated.

Moreover, fast DDFT systems have been also created by approaching the problem from a systems perspective. libdft [19] carefully utilizes the Pin [23] DBI framework to provide customizable DDFT with low overhead. Minemu [3] introduces a new emulation framework for DDFT, that sacrifices virtual address space and utilizes the SSE extension on x86 and x86-64 CPUs to achieve very low overhead. These approaches can also benefit from our optimizations as both approaches focus on the cost for each metadata access while our optimization reduces the frequency of those accesses.

2.3 Still not Fast Enough

Despite the various improvements in speed, the cost of DDFT is still considerably high for use in production systems. To better understand the limiting factors,

we examine some of its basic characteristics. First, additional memory is required to mark the data being tracked (*i.e., data tags*). The minimum amount of memory required is one bit for every tagged piece of data, which can be a bit, byte, or larger. Second, tracking instructions to propagate tagged data are inserted in the program. This introduces overhead because additional code is executed, but also stresses the runtime. For instance, the same eight general purpose registers of the x86 architecture are now used to implement both the original program and data tracking logic, which leads to register spilling (*i.e.,* temporarily saving and restoring a register to/from memory) and adds pressure to the instruction and data cache.

Furthermore, instrumentation usually occurs on a per instruction level, ignoring higher level semantics. As a consequence, all instructions and values are being instrumented, even if they could be eventually ignored. Consider the following code snippet (left is C code and right the corresponding data propagation, where `T(A)` indicates the tag of `A`):

Listing 1: Assignment through temporary

```
1 tmp1 = array1[idx1]; T(tmp1) = T(mem[array1+idx1])
2          ...                      ...
3 array2[idx2] = tmp1; T(mem[array2+idx2]) = T(tmp1)
```

Copying a value from `array1` to `array2` uses the intermediate variable `tmp1`. This can occur frequently, either because `tmp1` is modified in some way before being copied (for example, its bits are rotated), or because the underlying architecture (*e.g.,* x86) does not provide a memory-to-memory instruction. If `T(n)` represents the tag of `n`, this example requires two tag propagations, instead of one `T(array2[idx2]) = T(array1[idx1])`. The wastefulness of naively instrumenting this code snippet is made more apparent, if we look at how it is translated to binary code:

Listing 2: *x86*-like assembly of Listing 1

```
1 mov reg0,mem[array1+idx1] mov reg1,T[array1+idx1]
2                           mov T[reg0],reg1
3             ...                 ...
4 mov mem[array2+idx2],reg0 mov reg1, T[reg0]
5                           mov T[array2+idx2],reg1
```

The compiler allocates a register to store `tmp1`, but its tag `T(tmp1)` is stored in memory. As memory-to-memory instructions are not available, this will result in one additional register and two instructions to perform each propagation.

Injecting the instrumentation code during or before compilation can take advantage of the compiler's optimizations to circumvent some of these problems. For example, the compiler could determine that `mov reg1,T[reg0]` of line 4 is redundant. Furthermore, taint operation `mov T[reg0],reg1` of line 2 could also be removed, if the taintedness of intermediate variable (`tmp1`) is not needed from subsequent executions. However, we cannot benefit from such optimizations when applying DDFT on binaries.

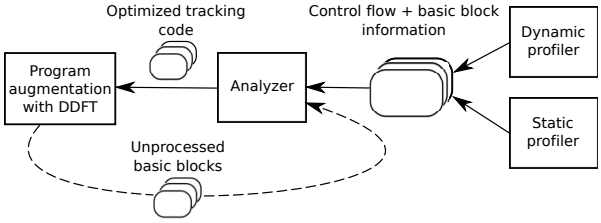


Figure 1: **Overview**. We extract basic blocks and control flow information, combining dynamic and static analysis to produce optimized data tracking code.

2.4 Optimization Through a Taint Flow Algebra (TFA)

Figure 1 shows a high-level overview of our approach for optimizing data tracking. We start by dynamically and statically profiling the target application to extract its basic blocks and control flow information. A basic block of code consists of a sequence of instructions that has only one entry point and, in our case, a single exit point. This means that no instruction within a basic block is the target of a jump or branch instruction, and the block is only exited after its last instruction executes. These properties are desirable for various types of analysis, like the ones performed by compilers. The control flow information describes how the basic blocks are linked. It is frequently impossible to obtain the complete control flow graph (CFG) for an entire program, but fortunately our analysis does not require a complete CFG. Nonetheless, the combination of dynamic and static profiling provides us with a significant part of the CFG, including the part that dominates in terms of execution time, and would benefit the most from optimization.

The analyzer receives the profiler information and extracts data dependencies from the code, separating program from data tracking logic. It then transforms the latter to an internal representation, based on the Taint Flow Algebra (TFA) described in Section 3.1, which is highly amenable to various optimizations. The optimizations performed by the analyzer are described in Section 3, and include classic compiler optimizations like *dead-code elimination* and *copy propagation*. Our goal is to remove redundant tracking operations, and reduce the number of locations where tracking code is inserted. Finally, the analyzer emits optimized tracking code, which is applied on the application. Note that the type of tracking code generated depends on the original tracking implementation to be optimized. In our imple-

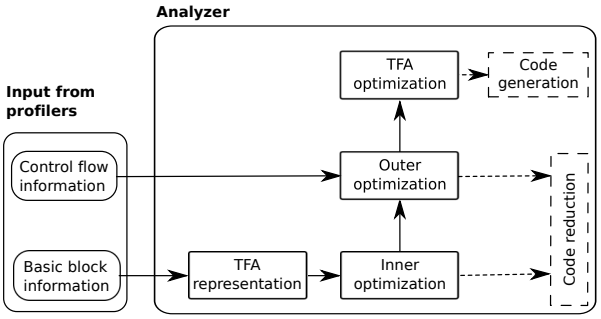


Figure 2: **TFA analysis**. After transforming the data dependencies to the taint flow algebra, three optimizations are performed: *inner*, *outer*, and *TFA*.

mentation, as it operates on binary programs the analyzer produces primitive C code, which can be compiled and inserted into the application using a DBI framework such as Pin [23].

It is possible that the profiling of the program is incomplete, so when running it through a DBI framework we may encounter basic blocks that have not been yet optimized/analyzed, which will be instrumented with unoptimized data tracking code. We can assist the profiler by exporting the unprocessed blocks, so that they be added to the analysis. This process establishes a feedback loop between the runtime and the analyzer, as shown in Figure 1.

There are a number of interesting previous work closely related to our analysis. Ruwase *et al.* [34] proposed a decoupled analysis similar to ours but it simply batch-processed the tracking logics and generated a synchronization issue between the original execution and tracking logic. A primitive form of TFA was implemented by Chen *et al.* [6] to complement their hardware solution. Ruwase *et al.* [35] again proposed a system that utilized data dependency extracted from a binary to maximize a parallelization of tracking logics.

3 Static Analysis

This section presents a formal definition of the optimizations performed. Figure 2 shows the various analyses we perform to generate optimized data flow tracking instrumentation code. Throughout this section, we consider the basic block (BB), a set of instructions with a single entry and exit, as the primitive unit for profiling, analyzing, and executing a binary. [1]

[1]To be persistent across separate analysis stages, a basic block identifier (BBID) is computed by hashing the full path name of the binary image it belongs to and adding its offset from the image's base address. This facilitates TFA to identify blocks that begin from different entry points but exit at the same point as separate ones, and makes each block have its own optimization result.

reg : *variable-name unsigned*
mem : *'[' mem-expr ']'*
var : *reg | mem | const*

binary-opr : *'|' | '&'*
unary-opr : *'~'*
assign-opr : *':='*
rng-map : *'r (' var | expr | statement ')'*

mem-expr : *(reg |const) |{ '×' | '+' | '−' (reg|cost)};*
expr : *var | expr {var binary-opr} | '(' expr ')'*
| unary-opr expr
statement : *(reg | mem) assign-opr expr*

Figure 4: The abstract syntax tree used by our Taint Flow Algebra.

Furthermore, we assume that single-bit tags are in use, and the smallest amount of information that can be tracked is a byte. To assist the reader better comprehend the various steps of this process, as we present them in detail below, we employ the code sample depicted in Figure 3. It demonstrates how we process a given *x86* block of binary code as it goes through the different analyses. As a result of our optimization, the five taint propagation operations required originally (Figure 3(a)) are reduced to three, inserted in two locations (Figure 3(d)).

3.1 Definition of a Taint Flow Algebra

TFA offers a machine independent representation of the data flow tracking logic that exposes optimization opportunities. It expresses a system with memory and an infinite number of registers, so it can accommodate a variety of instruction-set architectures (ISAs). Figure 4 shows its abstract syntax tree (AST) in BNF form.

It supports three different types of variables: constants, registers, and memory. The latter two correspond to the tags that hold the taint markings for the underlying architecture, while constants are primarily used to assert or clear tags. For example, when using a single-bit mark for each byte of data, the constant `0xf` (`1111` in binary) represents four tainted bytes. Other tag sizes are also supported by simply modifying the constants and variable size. Register variables are versioned (*i.e.,* a new register version is generated on update) to help us identify live ranges and redundant operations, while memory follows a flat model with array-like index addressing. Also, all variables in a single statement are of the same size, and we use the constants combined with the AND (bitwise &) and NOR (unary ∼) operators to extend or downsize a variable due to casting. We express data dependencies among variables using the OR operator (bitwise |), and determine operator precedence using parentheses ("(", ")").

(a) Original x86 instructions	(b) TFA representation	(c) TFA optimization	(d) Live range realignment
1: mov ecx, esi	1: ecx1 := esi0	1:	[0,1]: tmp0 := 0x1 & eax0 | esi0
2: movzxb eax, al	2: eax1 := 0x1 & eax0	2:	
3: shl ecx, 0x5	3:	3:	
4: add edx,0x1	4:	4:	
5: lea esi, ptr [ecx+esi]	5: esi1 := ecx1 | esi0	5:	
6: lea esi, ptr [eax+esi]	6: esi2 := eax1 | esi1	6: esi2 := 0x1 & eax0 |esi0	[6,9]: esi2 := tmp0
7: movzxb eax, ptr [edx+esi]	7: eax2 := 0x1 & [edx0+esi2]	7: eax2 := 0x1 & [edx0+esi2]	[7,9]: eax2 := 0x1& [edx0+esi2]
8: testb al, al	8:	8:	
9: jnzb 0xb7890200	9:	9:	

Figure 3: A block of x86 instructions as it is transformed by our analysis. First, it is abstracted using the TFA to (b). After performing our optimizations it is reduced to (c), and finally the locations of the instrumentation operations are corrected in (d). Note that instructions 3, 4, 8, and 9 do not involve data dependencies.

Unlike other general representations like LLVM-IR or gcc's GIMPLE, our TFA does not require primitives to express control transfers made by branch instructions or function calls. Instead, we employ a separate optimization stage to handle inter-block dependencies. As a result, branch choices solely depend on the original execution of the program. A *range operator* (`rng-map`) also differentiates the TFA from other representations, allowing us to specify live ranges for its various elements (*i.e.,* statements, expressions, and variables), so we can move certain statements in different positions and still correctly preserve code semantics. Constant variables are valid for entire blocks, while register variables are live from their previous till their next definition. In the case of memory variables, we also need to consider the registers and constants used to calculate their address. Hence, we determine their liveness range by looking for ranges, where both the variables used in addressing and the memory variable itself are valid concurrently (we discuss issues emerging from memory aliasing in Section 3.4.2). The liveness range for expressions and statements can be obtained by combining the ranges of their contained elements. Statements in Figure 3(d) are prefixed with a liveness range gained by applying the range operator($r(\cdot)$) to the statements in Figure 3(c). For a formal definition of the operational semantics of the range operator, we refer interested readers to Appendix A.1.

TFA aims to be a general representation that can accommodate common aspects of tracking logic extracted from different ISAs. Thus, it ignores details specific to a certain deployment platform. For instance, it does not represent operations like the x86's `SET` and `CMOV` instructions, whose exact behavior can only be determined at runtime, and refrains from optimizing them. Also, while there are similarities with previous work [37], our TFA represents a more applied than theoretical language that can easily express the logic incorporated by existing DTA tools [19, 3].

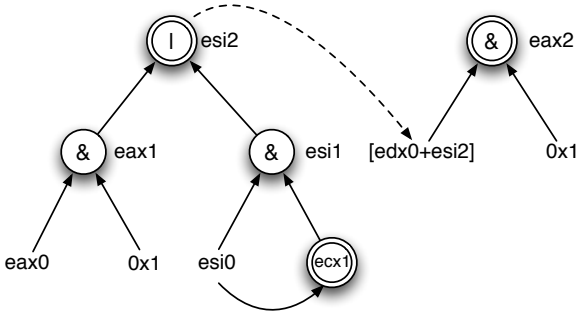


Figure 5: The taint-map for Figure 3(b), viewed as a direct acyclic graph (DAG).

3.2 Data Dependencies Extraction and TFA Representation

Our analysis begins by parsing a basic block of code and representing it in our TFA. This stage is specific to the type of code. For example, the x86 binary code in Figure 3(a) is represented as shown in Figure 3(b). To "translate" an instruction to TFA, we first determine the data dependencies it defines. In other words, we extract the data tracking logic or taint tracking semantics from the block's instructions. For instance, the `MOV` instruction propagates taint from source to destination, and the *ALU* instruction family (`ADD`, `SUB`, `XOR`, `DIV`, *etc.*) tags the destination operand, if one of source operands is also tagged. Even though it may seem like a straightforward process, *in practice different taint-tracking tools may adopt slightly divergent semantics*. For instance, sign extending a tainted 16-bit number to 32 bits may taint the entire result [19], or just the lower 16 bits [31]. Currently, we have adopted the interpretation used by libdft [19] (also used for our prototype implementation), but it can be effortlessly modified to cater to other implementations or problem domains.

While extracting dependencies, we can already discard some instructions, as most DDFT systems also do.

For instance, arithmetic operations with constants (lines 2 and 3 in Figure 3(a)) do not propagate taint. We also handle language idioms like `xor eax,eax` and `sub eax,eax`, which are used to clear registers, by clearing the corresponding tags (*e.g.,* `t(eax)`$\leftarrow$`0`). At this stage, we also cast operands of different sizes to the same width using masking and unmasking operations (*i.e.,* using the logical operators and constants). For instance, instruction 7 in Figure 3(a) transfers one byte from `[edx+esi]` to `eax`. We use the constant `0x1` to mask the taint propagation, ensuring that the higher bytes of the destination are correctly marked as clean.

The results of data dependency analysis (Figure 3(b)) are stored in a hash-table data structure, which we call the *taint-map*. The taint-map holds one entry for every TFA statement, using the destination operand as *key* and the *right hand-side* (*rhs*) expression as *value*. Combining all its entries eventually presents us with a directed acyclic graph (DAG), where inputs are the children of outputs as shown in Figure 5. The solid lines in the figure represent data dependencies within a statement, while dashed lines represent dependencies between variables. The leaf nodes of the DAG are input variables, and double-lined nodes are output variables. Finally, during this phase, we can introduce different dependency interpretations based on tracking policies such as implicit data flow tracking, and schemes like pointer tainting [42] or heap pointer tracking [40].

Inner Optimization After obtaining the TFA representation of a block, we process it to identify the operands used as inputs and outputs. The first version of every operand on the *rhs* of an assignment is marked as input. Similarly, the greatest version of an operand on the *left-hand side* (*lhs*) of an assignment is marked as output. From Figure 3(b), {`esi0`, `eax0`, `[edx0+esi2]`} are inputs, whereas {`ecx1`, `eax2`, `esi2`} are outputs. This process enables us to perform rudimentary dead code elimination [1], to discard taint propagation operations that do not affect output variables from the basic block. From our toy example, the taint operation in line 5 of Figure 3(b) can be removed as the taintedness of `esi1` solely depends on prior version of the same register variable (*i.e.,*`esi0`.)

3.3 Incorporating Control Flow Information - Outer Optimization

We make use of the control flow information collected during the code profiling stage to extend *the inputs/outputs identification* to span multiple chained basic blocks. For example, if the CFG points out that BB `Block0` is directly followed by either `Block1` or `Block2`, and no other BBs, we can use the inputs/outputs lists of those BBs to eliminate some of `Block0`'s output variables, if they are overwritten without being used as inputs in either `Block1` or `Block2`. This implements a type of data flow analysis (or live variable analysis) common in compiler optimizations, and allows us to purge more redundant tracking operations, based on the new, smaller outputs lists. *Our analysis differs from commonly used algorithms in that the available CFG used may be incomplete, and the BBs we use, can span across the boundary of functions and even libraries.*

In a case where we cannot identify all successor blocks due to indirect calls or jumps, we handle it conservatively by considering all output variables of a BB as live ones. This modification should not have much impact on performance, since a profiling result for SPEC CPU2000 benchmark suite shows that about 95.7% of BBs are ended with deterministic control transfers such as direct and conditional jumps which only allow one and two successor nodes respectively. Theorem 1 formally addresses the soundness of outer analysis whose correctness can be proved using a semi-lattice framework [1], modified to cover cases where the CFG is incomplete. Due to space constraints, we accommodate the proof in Appendix A.2.1.

Theorem 1. ***Soundness of outer analysis:*** *Outer analysis (live variable analysis) with incomplete CFG converges and is safe.*

In our toy example (Figure 3(b)), the output variable `ecx1` taints `esi1` in line 5, but our analysis determines that it is not used in any of subsequent blocks. Therefore, we remove `ecx1` and the corresponding propagation operation. Note that taint propagations in line 5 and line 6 preserve original taint semantics without `ecx1` as their taintedness solely depends on `esi0`.

3.4 TFA Optimization

The most powerful optimization enabled by our analysis is the *TFA optimization*, which also utilizes results from previous stages (*i.e.,* the taint-map and the truncated BB outputs lists). This optimization is more aggressive than the previous ones, as it does not only identify redundant code, but it also generates optimized tracking code.

3.4.1 Pruning Redundant Expressions and Merging Statements Using Copy Propagation

Copy propagation is the process of replacing the occurrences of targets of direct assignments (*e.g.,* `eax1` in Figure 3(b)) with the expression assigned to them (*e.g.,*

`0x1` & `eax0`), where they appear in other statements (*e.g.,* line 6 in Figure 3(b)). We perform this process by recursively substituting variables in the *rhs* expressions of assignments, until they solely contain variables from the inputs and outputs lists. We use the taint-map to quickly look for targets of assignments. The result is an updated taint-map that contains entries that use a variable from BB outputs as key, while its value is the taint expression assigned to the key. This can also be seen as a process of directly connecting output and input nodes from DAG representation of a block (*i.e.,* Figure 5) by eliminating intermediate nodes. The number of tracking operations needed is greatly reduced, as shown in Figure 3(c). Even though, some of the generated assignments are more complex, they will not generate longer expressions than the original (Theorem 2 in Appendix A.2.2,) while they also reduce the number of locations where propagation code needs to be inserted. Due to its recursive nature, the running time of our substitution algorithm increases exponentially with the size of BB. We alleviate this problem by memoizing previously visited variables.

Along the way, we also perform *algebraic simplification* on arithmetic operations such as (`0xffff` & (`0x00ff` & `eax0`) ⇒ (`0x00ff` & `eax0`). This optimization may be also performed in later stages from the deployment platform, like the DBI framework. However, we make no assumptions about the underlying framework, so we proactively exploit optimization opportunities like this, whenever they are feasible.

Resolving Range Violations in Merged Assignments Substituting variables can raise issues, when variables in a statement cannot be live (*i.e.,* valid) concurrently. For example, `esi2` and `eax0` in instruction six of Figure 3(c) have ranges of [5,9] and [0,1] respectively, which we calculate and express using the range operator (`r(·)`), and are not valid concurrently. When a statement cannot yield a common range for their contained elements, we introduce temporary variables to resolve the conflict. For example, in Figure 3(d) we use `tmp0` to resolve the range conflict. Note that the range correction should be made in a way that the number of instrumentation locations is kept to the least possible, and should not result in more tracking operations than the original representation. Theorem 2 directly states this, its proof is provided in Appendix A.2.2.

Theorem 2. ***Efficiency of the TFA optimization****: The TFA optimization always produces less, or an equal number of, tracking statements than the original representation, for any basic block.*

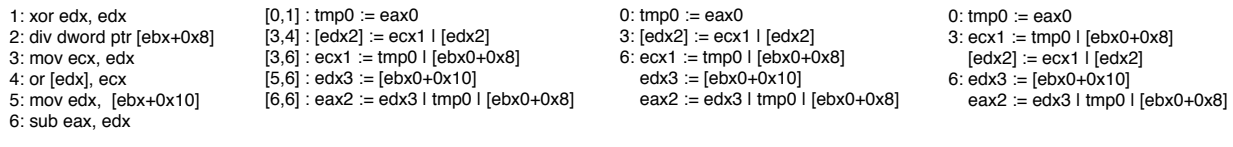
One can argue that showing to have less statements than the original may not be sufficient, since longer statements containing many variables could, in theory, be translated into more machine instructions. Even though precisely estimating the outcome of our optimization in the instruction level is not an easy task, we can still provide a *corollary* to the theorem that states that the optimization's results will always have less or the same number of variables than the original representation as a whole. This can be proven in similar way to Theorem 2. However, the theorem does not cover some corner-cases, where the TFA representation has more statements than the original instrumentation. For instance, in the presence of ISA specific instructions, such as x86's `xchg` or `cmpxchg` instructions which requires two or more statements to be expressed in TFA whereas a baseline system can have a specialized and rather efficient interpretation. We detect such cases and retain the original propagation logic.

3.4.2 Minimizing Instrumentation Interference with Aggregation

By default the TFA optimization produces *scattered* statements. This leaves the merged statements near the location where their target is actually being updated. As the range operator can yield multiple valid locations for each statement, we can adopt different policies to group statements, in order to minimize the instrumentation overhead. *Aggregation* aims to group statements into larger instrumentation units, so as to further reduce the number of locations where code is injected into the original program. Aggregation can be particularly effective when used with DDFT frameworks where the fixed overhead associated with every instrumentation is high.[2] In the example shown in Figure 3(d), aggregation will combine the second and third statements into a single block with an aggregated liveness range of [7,9].

Range Violations among Statements In section 3.4.1, we saw how we address invalid expressions produced because the elements of an expression are valid in different code ranges. Less obvious range violation errors may also occur with *aggregation*. For example, consider the BB in Figure 6(a), which after our analysis produces the statements in Figure 6(b) (prefixed with their calculated live ranges). Both Figures 6(c) and 6(d) are valid groupings consistent to each statement's liveness range, but the first will incorrectly propagate taint. That is because in Figure 6(c), the

[2]For most DBI/VM based frameworks each instrumentation requires additional instructions for context switching. For instance, Pin [23] introduces three additional instructions per instrumentation location for analysis routines that can be inlined and 12 otherwise.



(a) Input basic block (b) TFA representation (range realigned) (c) Incorrect aggregation (d) Correct aggregation

Figure 6: Aggregation example with range violations.

second statement uses `ecx1` before it is defined in the third statement. This *define/use violation* occurs because the aggregation algorithm greedily attempts to maximize the size of instrumentation unit. Also, ordering of the second and third statement causes a *memory alias violation*. In the original order shown in Figure 6(c), the memory variable `[ebx0+0x8]` is used before memory variable `[edx2]` is defined. Since we cannot determine if these two variables point to the same location or not, all memory variables are considered to be dependent on each other.

To address these issues, we impose dependencies between statements using a variable and the statement defining it. We implement these dependencies by topologically sorting the taint-map data structure, and relocating instructions based on their order. As it is shown by the corrected aggregation in Figure 6(d), we increase neither the number of instrumentation units nor the code size as a whole, but only redistribute the statements.

3.5 Code Generation

To apply the results of our analysis on a DDFT framework, we need to transform the statements from TFA to a language that the framework understands. We do so using both the statements produced pre- and post-aggregation, namely *TFA scatter* and *TFA aggregation*. The reason for doing so is that the number of instrumentation locations and the size of the routines that need to be injected affect varying frameworks in different ways (*i.e.,* some may prefer one instead of the other). Consequently, to implement the code generation back-end, we need to know certain features of the deployment platform, like *(i)* architectural properties such as the ISA type (*e.g., x86* or *x86_64*), the underlying OS, and the instrumentation framework employed (*e.g.,* PIN, DynamioRIO, QEMU, *etc.*), and *ii)* DDFT characteristics such as tag size, tracking granularity, and any particular rules used for determining data dependencies. Most of all, the produced code needs to use the DDFT framework's primitives for accessing and updating tags for different variable types. For our prototype implementation, we utilized the libdft [19] DDFT framework, which we discuss in the following section.

4 Implementation

To implement the static analysis described in Section 3, we developed a set of tools written in Python, consisting of approximately 7K lines of code. Since processing applications with large codebases, such as web-browsers, can take a considerable amount time, all of our tools utilize multiple threads to exploit available multiple CPU cores.

The majority of our analysis tools, with the exception of the ones performing the translation from binary to TFA and code generation are system agnostic, since they operate on TFA. Regarding the system dependent tools, we implemented them along with the profiler shown in Figure 1 for x86 binaries. To test and evaluate the generated instrumentation code we used libdft [19], a freely available framework that transparently applies DDFT on *x86* Linux binaries, developed over Intel's Pin DBI framework [23]. Our tools can be easily extended to operate on other operating systems, while moderate effort may be required to support other ISAs, like the *x86_64*. In the remainder of this section, we discuss the profiler we implemented for Linux, the libdft framework, and the components that depend on it.

4.1 A Pin-based Dynamic Profiler

The profiler is responsible for extracting basic blocks and a partial CFG from the application. While already available code analysis tools like IDA Pro [16] could be used for this purpose, we chose to implement a dynamic profiler using the Pin DBI framework. Briefly, Pin comprises of a VM library and an injector that attaches the VM in already running processes, or new processes that it launches. Using Pin, we created a tool (*i.e.,* a Pintool) to profile the target applications. This made our testbed more uniform and enabled us to better interface with libdft, as it is also based on Pin. Additionally, our dynamic profiler can help us identify frequently executed or "hot" code segments allowing us to better estimate performance during the static analysis phase. However, it will only reveal execution paths traversed during the profiling.

4.2 Integration with the libdft Framework

libdft is a highly optimized DDFT framework that shows comparable to or faster performance than most previous work. libdft takes the form of a shared library enabling developers to create of DDFT-enabled Pintools for binaries, using its extensive API. For example, libdft already includes a DTA tool, which can be used to protect applications from remote buffer overflow exploits.

We applied our optimizations by compiling the code generated by our analysis as a shared library object (`libtfa.so`), accompanied by an instruction file that specifies where to inject each instrumentation routine, and lists the arguments for each. We generated a unique function for each injection point, allowing us to generate very compact code, using less function arguments. However, there are limits to the number of functions that can be defined a within a program. To overcome this limitation, we implemented a simple code caching scheme that reuses functions that correspond to the same expression. This simple mechanism greatly reduced the number of generated functions (*e.g.,* from 621,601 to 64,271 for the Chrome web-browser). The size of shared library objects for different evaluated applications are presented from Table 1. Finally, we also modified libdft's initialization routine to load our files, and its instrumentation routine to use them.

Moreover, we applied the result of our code reduction analysis (*i.e.,* the inner and outer optimizations) to evaluate its distinct contribution (see Section 5). Since these optimizations only remove instructions, we simply supplied libdft with a list of instructions that should not be instrumented.

Also note that code that was not profiled does not get optimized, and is handled by the default taint tracking logic of libdft. Specific instructions such as *x86*'s `xchg` and `cmpxchg` are also not optimized because they are handled more efficiently by the default logic (we would require two or more statements to express this taint operation in the TFA). Another interesting observation made was that seemingly equivalent assignments like `eax1=eax0|ebx0` and `eax1|=ebx0` do no result in the same instrumentation code in libdft, nor do they perform the same. Fortunately, modifying our analysis to prefer the one form instead of the other is trivial.

5 Evaluation

In this section, we measure the impact of our optimizations using libdft v3.141 and Pin v2.8 (build 39028). In all cases, we evaluate the optimization schemes described in Section 3.5, namely *TFA-scatter* and *TFA-aggregation*, as well as our reduction analyses, *inner optimization* and *outer optimization*, the baseline implementation of *libdft* , and native execution. Our testbed consisted of two identical hosts, armed with 2×2.66GHz quad core Intel Xeon X5500 CPUs and 24GB of RAM, running Debian Linux v6 ("squeeze"; kernel version 2.6.32). While running the benchmarks the hosts were idle, and no other processes were running except from the corresponding evaluation suite.

5.1 Static Analysis Results

Table 1 contains various statistics regarding TFA analysis. Note that the percentage of instructions instrumented does not refer to the number of taint operations generated, but the number of locations being instrumented. Each instrumentation may actually define multiple taint propagations, specially in the case of TFA aggregation. For TFA scatter and TFA aggregation, we also list the average distance *(d)* of the propagation routines from their original locations in number of instructions. An instrumentation (function) is considered to be at distance 0 from its target, when the taint propagation occurs right before the instrumented instruction. As expected, the value of *(d)* is small when using TFA scatter, while it increases when using TFA aggregation.

5.2 Performance

Performance results were obtained by running 10 repetitions of each experiment. In all cases, we draw the average slowdown observed when running the various benchmarks using libdft and our optimized versions of it, compared with native execution. Additionally, we calculate the average speedup achieved by our optimizations. We obtain the average slowdown (S_{slow}) and speedup (S_{speed}) using the following formulas:

$$S_{slow} = \frac{Time_{avg}\ under\ libdft\ w/opt.}{Time_{avg}\ under\ native},$$

$$S_{speed} = \frac{Time_{avg}\ under\ libdft}{Time_{avg}\ under\ libdft\ w/opt.}$$

We also calculate and draw the relative standard deviation (RSD) for each experiment by dividing the standard deviation with the average.

Figure 7 shows the slowdown obtained for four common Unix command-line utilities that represents different type of workloads. We use `tar` to archive and extract a vanilla Linux kernel "tarball" (v2.6.35.6; $\sim$ 400MB) whereas `gzip` and `bzip2` were used for compressing and decompressing it respectively. For `scp`, we transferred 1GB of randomly generated data, first over an 100Mbps link and then over an 1Gbps link. As expected, we see that all CPU intensive applications running under DDFT are significantly accelerated by our

Application	Number of blocks	Percentage of instructions instrumented						Shared library size	Time to complete
		Inner	Outer	TFA scatter	(*d*)	TFA aggr.	(*d*)		
gzip	3,484	89.7%	79.4%	59.5%	(0.87)	30.4%	(2.46)	847 kb	125.55s
bzip2	5,253	86.5%	76.8%	54.8%	(1.21)	28.2%	(2.83)	1,352 kb	312.45s
tar	7,632	86.8%	76.8%	60.2%	(0.75)	30.0%	(2.30)	1,721 kb	230.07s
scp	19,617	85.1%	77.9%	57.5%	(1.75)	27.9%	(3.94)	4,919 kb	2,086.77s
SPEC CPU2000	170,128	84.84%	71.42%	59.14%	(0.99)	26.79%	(3.10)	11,388 kb	10317.37s
PHP	28,791	86.1%	78.9%	60.0%	(1.35)	25.2%	(3.76)	3,698 kb	2,417.45s
MySQL	45,262	86.8%	78.0%	58.6%	(0.83)	24.5%	(3.52)	8,191 kb	3,736.47s
Firefox	676,027	83.5%	72.95%	63.5%	(0.72)	25.9%	(3.41)	13,909 kb	42,691.54s
Chrome	496,720	83.6%	73.0%	59.56%	(0.76)	28.4%	(2.73)	17,438 kb	56,485.61s

Table 1: **Static analysis results**. The table lists various statistics regarding the static analysis of various applications. From left to right, it lists the number of basic blocks profiled/processed, the percentage of the instructions that are still instrumented after each optimization, the size of shared library objects (`libtfa.so`) which contain optimized propagation logics, and the time to complete the entire analysis (includes all optimizations). Note that each instrumentation may actually define multiple taint propagations, specially in the case of TFA aggregation. For TFA scatter and TFA aggregation *(d)* denotes the average distance of the propagation routines from their original locations in number of instructions.

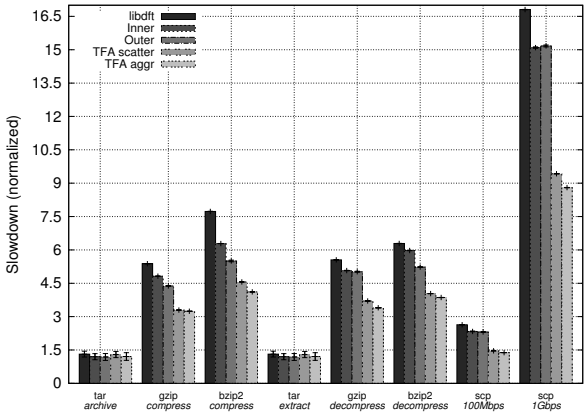


Figure 7: Slowdown for common Unix command-line utilities.

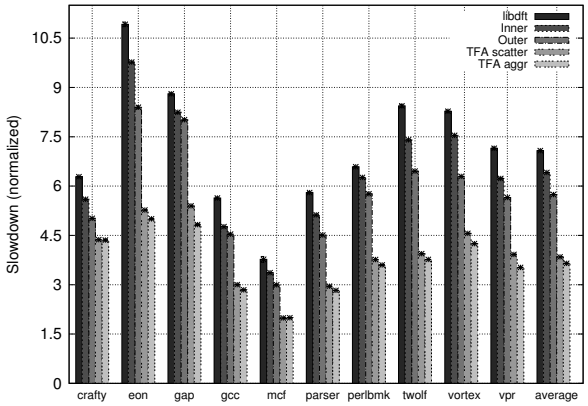


Figure 8: Slowdown for SPEC CPU2000 benchmarks.

optimization. For instance, compressing natively with `bzip2` takes 65.5s to complete, whereas libdft takes 506.1s and TFA aggregation only 269.12s. In contrast, `tar`, benefited the least from our optimizations as file I/O dominates. For `scp`, our optimization showed similar speedups of $\sim 1.90\times$.

We also evaluate the effects of our optimizations using the SPEC CPU2000 benchmark suite. Figure 8 illustrates selected benchmarks, while the rightmost bar represents the average overhead, computed over the whole benchmark (12 tests). Our optimizations reduced the average slowdown of libdft from $7.08\times$ to $3.65\times$ representing a speedup of $1.93\times$, while the largest speedup ($2.23\times$) was gained with the *twolf* test.

Next, we applied our optimizations on larger software, such as the the MySQL DB server (v5.0.51) and the PHP runtime (v5.3.3.7). Figure 9(a) shows the slowdown imposed by libdft and our optimizations, when running MySQL's own benchmark suite (`sql-bench`). The suite consists of four different tests, which assess the completion time of various DB operations, like table creation and modification, data selection and insertion, *etc.* CPU intensive benchmarks such as `test-alter` and `test-insert` benefit the most from our optimizations, achieving speedups of $1.47\times$ and $1.27\times$ respectively. `test-create` and `test-ATIS`, on the other hand, are less intensive and suffer less overhead when run over libdft, benefiting less from our analysis.

We also tested the runtime of the PHP scripting language runtime using the micro-benchmark suite PHP-Bench (`http://www.pureftpd.org/project/phpbench`), and report a representative subset of results in Figure 9(b) (the `casting` test experienced the least overhead, and the `md5` and `sha1` tests the most). The rightmost bar draws the average overhead, computed over the whole benchmark (556 tests). libdft incurs significantly high overheads when running the PHP runtime (an average slowdown of $16.58\times$), and as such

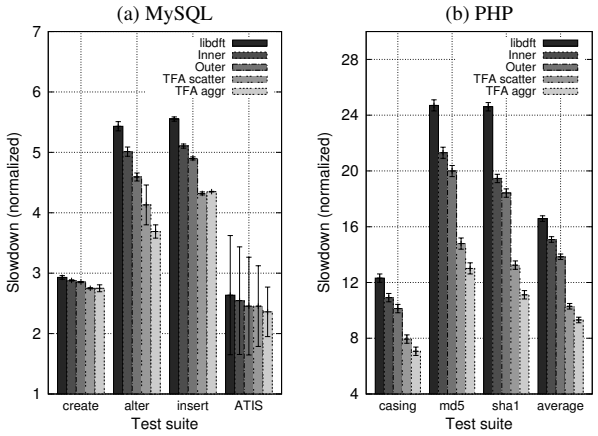


Figure 9: Slowdown for common MySQL benchmarks in (a) and PHPBench in (b).

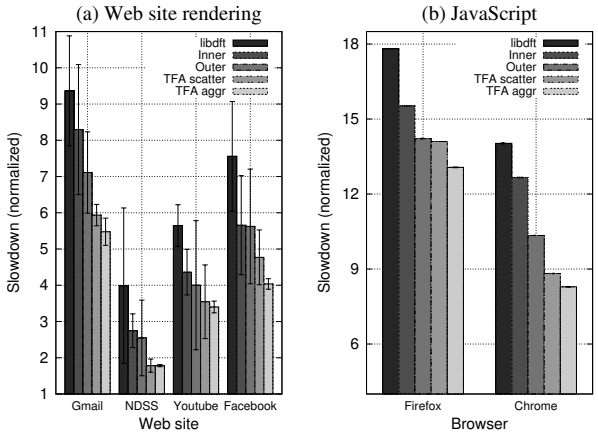


Figure 10: Web site rendering latencies in (a) and Dromaero JS benchmark slowdown in (b).

Application	Description
ATPhttpd v0.4 (BID 5215)	Stack-based buffer overflow
wu-ftpd v2.6.0 (BID 1387)	Remote format string stack overwrite
WsMp3 v0.0.2 (BID 6240)	Remote heap corruption
Echo server (handcrafted)	Heap-based function pointer overwrite

Table 2: List of detected control flow diversion vulnerabilities.

even after applying our optimizations, gaining a speedup of $1.78\times$, the average overhead remains high ($9.30\times$).

We conclude our performance evaluation by measuring the speedups obtained by our analysis with the Firefox(v3.5.16) and Chrome (v12.0.742.124) web browsers. First, we used Firefox to access the three most popular web sites according to Alexa's Top 500 (`http://www.alexa.com/topsites`), as well as the home page of the NDSS conference, and measured the time needed for the browser to render the accessed pages. We show individual results in Figure 10(a), while the average speedup gained by our optimizations was $1.90\times$. Additionally, we used the Dromaeo test suite (`http://dromaeo.com/?dromaeo`) to evaluate the JavaScript (JS) engines of both Firefox and Chrome (see Figure 10(b)). libdft also incurs large slowdowns with the Dromaeo benchmark, similar to the PHP benchmark. In this case, our optimizations managed to accelerate the data tracking by $1.36\times$ for Firefox and $1.69\times$ for Chrome.

5.3 Effectiveness

We close the evaluation by testing the correctness of our analysis, in terms of taint tracking. We do so by testing whether libdft's DTA tool can still correctly track network data, and prevent exploits, after applying our optimizations. We used the group of exploits listed in Table 2, and in all cases the modified DTA tool successfully detected and prevented the exploitation of all listed applications.

6 Discussion

Inconsistencies in shadow memory status The effectiveness of our approach was evaluated in Section 5.3, however TFA can produce two types of inconsistencies, which could result in different shadow memory status compared with unoptimized DDFT. The first occurs because TFA eliminates redundant tracking operations, which can lead to certain shadow registers not being updated. As these registers are not used later on by the program, this inconsistency does not affect the accuracy of tracking. The second is due to TFA relocating tracking operations. This introduces an ephemeral disparity in the state of shadow memory (*i.e.,* if we compare the shadow memory states of TFA and unoptimized DDFT after every instruction, they we will not always be equivalent). As the tracking operations execute by the end of each basic block and before control-flow is enforced, this inconsistency is resolved, and does not introduce any false positives or false negatives.

Effectiveness of aggregation Our evaluation results made clear that aggregating taint operations improves performance, but gains were smaller than what we originally expected. We attribute this behavior to Pin and libdft. Specifically, Pin achieves good performance by inlining instrumentation code between the application's instructions. However, this is not possible when the instrumentation routine grows in size, or when it contains branch instructions. libdft adheres to these limitations to achieve good performance. When we employ TFA aggregation, the instrumentation routines grow in size and are no longer inlined, increasing the cost of instrumen-

tation. We believe that the benefit of aggregation will be more apparent in DDFT systems with consistent instrumentation overheads.

TFA for multi-threaded applications Aggregation relocates the taint operations within a basic block, making sure that taint propagation semantics are preserved correctly. This execution model works well for sequential programs, however like all software DDFT approaches, it can have a correctness problem when applied to multi-threaded applications. Software DDFT approaches may suffer from races when multiple threads compete for the same tag location, since the original instruction and the tracking operation are not executed atomically. Our optimizations may increase the chance of such races, as the distance between original instruction and propagation increases (see Section 5.1). For this issue, we draw on the correctness of the original program protecting unsafe shared memory accesses. If the program's accesses to shared memory regions are properly guarded with synchronization primitives, the counterpart tracking operations will also be correct as both reside in the same basic block boundary.

TFA for other DDFT approaches Other DDFT systems build on binary instrumentation could also directly benefit from our code reduction schemes (*i.e.,* inner and outer optimizations). This would require some engineering effort to update TFA scatter and TFA aggregation to generate code suitable for the DDFT system being used. For instance, we can easily apply code reductions (inner, outer) to Minemu [3], while even the more complicated TFA scatter and TFA aggregation could be implemented, if we modified its translation mechanism. TFA can benefit Minemu because our optimizations focus on reducing the number of instrumented instructions, while Minemu achieves its high performance due to its low-cost access to shadow memory and the utilization of the SSE instruction set.

TFA and compiler-based DDFT Can we apply TFA to compiler-based DDFT and expect to have similar performance gains? Through access to high-level semantics, such as type information and loop invariants, compiler-based DDFT systems are generally already more optimized. Because tracking logic is inserted directly in the source code (or the compiler's intermediate representation), compiler-based approaches already benefit from compiler optimizations, which are applied to program and tracking logic as a whole. Most of TFA's optimizations overlap with the ones performed by compilers, while there are some that require access to source code (*e.g.,* Code Motion). Nevertheless, such approaches can still gain from our range-aware aggregation of tracking operations, albeit much smaller performance gains should be expected.

7 Future Work

In addition to the already applied TFA optimizations, we can also explore the following optimizations which are not yet applied:

Alias analysis Currently, we have limited rules on the equality of memory operands. We regard two memory operands as equal, only when all of the variables that determine their address are the same. If the variables are the same but differ by a constant, we regard them as unequal. In all other cases, the equality of memory operands cannot be decided. By leveraging a rather aggressive alias analysis, we expect to be able to determine memory operand equality more effectively, allowing us to further optimize DDFT.

Global common subexpression This can be implemented by extending the scope of the local common subexpression optimization beyond the boundary of basic blocks (note that TFA already performs the later during copy propagation). This also requires that we perform an *available expression* data flow analysis.

TFA also exposes opportunities for applying more advanced optimization schemes. For instance, we can implement compositional analysis [15] using TFA to produce summaries for code segments larger than a basic block. This would allow us to use *traces*, as defined by the Pin DBI framework [23], or *functions* as analysis units. Our approach can be also used to enhance the performance of other analyses based on memory shadowing and binary instrumentation, such as memory safety lifeguards [4, 26]. Even though, these systems define data dependencies differently, our methodology is still applicable, and can reduce the number of tracking operations required.

Another direction that we plan to explore is parellelization. Decoupling execution from DDFT has been explored in the past, however we believe that our approach reduces the number of instrumentation instructions sufficiently enough to make parellelization more attractive. Implicit data flow tracking can also benefit from our work, since TFA already extracts the control dependencies between basic blocks.

8 Conclusion

We presented a novel methodology that combines dynamic and static analysis to improve the performance overhead of DDFT. Our approach separates data tracking logic from program logic, and represents it using a Taint Flow Algebra. Inspired by optimizing compilers, we apply various code optimization techniques, like

dead code elimination and liveness analysis, to eliminate redundant tracking operations and minimize interference with the instrumented program. We incorporated our optimizations in a freely available DDFT framework and evaluated its performance with various applications. Our results show improved performance by $1.72\times$ on average across the tested applications, while in certain scenarios we observe speedups up to $2.23\times$. Although the overall performance impact of DDFT remains significant, our optimizations bring it closer to becoming practical for certain environments.

Acknowledgments

This work was supported by the US Air Force, DARPA and the National Science Foundation through Contracts AFRL-FA8650-10-C-7024, FA8750-10-2-0253 and Grant CNS-09-14312, respectively, with additional support by Google and 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, DARPA, the NSF, Google or Intel.

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison Wesley, 2006.

[2] M. Attariyan and J. Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *Proc. of the* 9^{th} *OSDI*, pages 237–250, 2010.

[3] E. Bosman, A. Slowinska, and H. Bos. Minemu: The World's Fastest Taint Tracker. In *Proc. of the* 14^{th} *RAID*, pages 1–20, 2011.

[4] D. Bruening and Q. Zhao. Practical Memory Checking with Dr. Memory. In *Proc. of the* 9^{th} *CGO*, pages 213–223, 2011.

[5] W. Chang, B. Streiff, and C. Lin. Efficient and Extensible Security Enforcement Using Dynamic Data Flow Analysis. In *Proc. of the* 15^{th} *CCS*, pages 39–50, 2008.

[6] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. Gibbons, T. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos. Flexible Hardware Acceleration for Instruction-Grain Program Monitoring. In *Proc. of the* 35^{th} *ICSA*, pages 377–388, 2008.

[7] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *Proc. of the 2008 USENIX ATC*, pages 1–14.

[8] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding Data Lifetime via Whole System Simulation. In *Proc. of the* 13^{th} *USENIX Security*, pages 321–336, 2004.

[9] J. Clause, W. Li, and A. Orso. Dytan: A Generic Dynamic Taint Analysis Framework. In *Proc. of the 2007 ISSTA*, pages 196–206.

[10] 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 the* 20^{th} *SOSP*, pages 133–147, 2005.

[11] J. R. Crandall and F. T. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *Proc. of the* 37^{th} *MICRO*, pages 221–232, 2004.

[12] J. Dai, J. Huang, S. Huang, B. Huang, and Y. Liu. HiTune: Dataflow-Based Performance Analysis for Big Data Clouds. In *Proc. of the 2011 USENIX ATC*, pages 87–100.

[13] M. Dalton, H. Kannan, and C. Kozyrakis. Real-world Buffer Overflow Protection for Userspace & Kernelspace. In *Proc. of the* 17^{th} *USENIX Security*, pages 395–410, 2008.

[14] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, and A. N. S. Patrick McDaniel. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proc. of the* 9^{th} *OSDI*, pages 393–407, 2010.

[15] P. Godefroid. Compositional Dynamic Test Generation. In *Proc. of* 34^{th} *POPL*, pages 47–54, 2007.

[16] Hex-Rays. IDA: Interactive DisAssembler. `http://www.hex-rays.com`.

[17] A. Ho, M. Fetterman, A. W. Christopher Clark, and S. Hand. Practical Taint-based Protection using Demand Emulation. In *Proc. of the 2006 EuroSys*, pages 29–41.

[18] M. G. Kang, S. McCamant, P. Poosankam, and D. Song. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In *Proc. of the* 18^{th} *NDSS*, 2011.

[19] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis. libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. Technical Report CUCS-044-11, Department of Computer Science, Columbia University, Oct 2011.

[20] H. C. Kim and A. D. Keromytis. On the Deployment of Dynamic Taint Analysis for Application Communities. *EICE Transactions on Information and Systems*, E92.D:548–551, 2009.

[21] L. C. Lam and T. Chiueh. A General Dynamic Information Flow Tracking Framework for Security Applications. In *Proc. of the* 22^{nd} *ACSAC*, pages 463–472, 2006.

[22] M. T. Louw and V. N. Venkatakrishnan. Blueprint: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers. In *Proc. of the* 12^{th} *IEEE S&P*, pages 331–346, 2009.

[23] 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 *Proc. of the 2005 PLDI*, pages 190–200.

[24] A. C. Myers and B. Liskov. Protecting Privacy using the Decentralized Label Model. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(4):410–442, 2000.

[25] S. Mysore, B. Mazloom, B. Agrawal, and T. Sherwood. Understanding and Visualizing Full Systems with Data Flow Tomography. In *Proc. of the* 13^{th} *ASPLOS*, pages 211–221, 2008.

[26] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proc. of the 2007 PLDI*, pages 89–100.

[27] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proc. of the* 12^{th} *NDSS*, 2005.

[28] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. Parallelizing Security Checks on Commodity Hardware. In *Proc. of the* 13^{th} *ASPLOS*, pages 308–318, 2008.

[29] G. Portokalidis and H. Bos. Eudaemon: Involuntary and On-Demand Emulation Against Zero- Day Exploits. In *Proc. of the 2008 EuroSys*, pages 287–299.

[30] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. Paranoid Android: Versatile Protection For Smartphones. In *Proc. of the* 26^{th} *ACSAC*, pages 347–356, 2010.

[31] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an Emulator for Fingerprinting Zero-Day Attacks. In *Proc. of the 2006 EuroSys*, pages 15–27.

[32] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *Proc. of the* 39^{th} *MICRO*, pages 135–148, 2006.

[33] I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel. Laminar: Practical Fine-Grained Decentralized Information Flow Control. In *Proc. of the 2009 PLDI*, pages 63–74.

[34] O. Ruwase, S. Chen, P. Gibbons, and T. Mowry. Decoupled Lifeguards: Enabling Path Optimizations for Dynamic Correctness Checking Tools. In *Proc. of the 2010 PLDI*, pages 25–35.

[35] O. Ruwase, P. B. Gibbons, T. C. Mowry, V. Ramachandran, S. Chen, M. Kozuch, and M. Ryan. Parallelizing Dynamic Information Flow Tracking. In *Proc. of the* 20^{th} *SPAA*, pages 35–45, 2008.

[36] P. Saxena, R. Sekar, and V. Puranik. Efficient Fine-Grained Binary Instrumentation with Applications to Taint-Tracking. In *Proc. of the* 6^{th} *CGO*, 2008.

[37] E. J. Schwartz, T. Avgerinos, and D. Brumley. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proc. of the* 13^{th} *IEEE S&P*, pages 317–331, 2010.

[38] G. E. Suh, J. Lee, and S. Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *Proc. of the* 11^{th} *ASPLOS*, pages 85–96, 2004.

[39] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. RIFLE: An Architectural Framework for User-Centric Information-Flow Security. In *Proc. of the* 37^{th} *MICRO*, pages 243–254, 2004.

[40] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. Flexitaint: A Programmable Accelerator for Dynamic Taint Propagation. In *Proc. of the* 14^{th} *HPCA*, pages 243–254, 2008.

[41] W. Xu, S. Bhatkar, and R. Sekar. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *Proc. of the* 15^{th} *USENIX Security*, pages 121–136, 2006.

[42] 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 the* 14^{th} *CCS*, pages 116–127, 2007.

[43] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making Information Flow Explicit in HiStar. In *Proc. of the* 7^{th} *OSDI*, pages 263–278, 2006.

[44] Q. Zhang, J. McCullough, J. Ma, N. Schear, M. Vrable, A. Vahdat, A. Snoeren, G. Voelker, and S. Savage. Neon: System Support for Derived Data Management. In *Proc. of the* 6^{th} *VEE*, 2010.

[45] D. Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. TaintEraser: Protecting Sensitive Data Leaks Using Application-Level Taint Tracking. *SIGOPS Oper. Syst. Rev.*, 45:142–154, 2011.

Appendices

A Formal Definitions and Semantics

A.1 The Range Operator

Figure 11 presents the operational semantics of the range operator (*rng-map*) defined from Figure 4. The operator calculates the valid insert locations of the element given as its arguments. It shows how the range operator is interpreted when it is applied to the different elements (variables, expression, and statement) of TFA. We do not present any other operational semantics, since they are not fundamentally different from previous work [37].

The rules in Figure 11 are read bottom to top, left to right. Given a range statement, we pattern-match its argument to find the applicable rule. *e.g.,* given the range statement `r(eax1 := 0x1 & eax0)`, we first match to the *Statement* rule. We then apply the computation given in the top of the rule, and if successful, a transition made from the original context (Σ) to the updated context (Σ'.) Σ is the only context we care for the operator, and it represents the instrumentation locations in the block being analyzed. For a BB with n instructions ($size(\Sigma) = n$), it allows $n + 1$ instrumentation locations ($[0, size(\Sigma)]$).

For *constant variables*, the range operator returns locations available in the entire block ($[0, size(\Sigma)]$). For *register variables*, it returns the range from the previous to the next definition ($[def(reg_{n-1}), def(reg_{n+1})]$.) In the case of *memory variables*, it looks for ranges where both the variables used in addressing and the memory variable itself are valid concurrently. The valid range for *expressions* and *statements* can be obtained by combining the ranges of their contained elements. Unlike other rules, the *Statement* rule updates the context (Σ) by inserting its input statement ($var ::= expr$) to a computed

$$\frac{\Sigma \vdash l = [0, size(\Sigma)]}{\Sigma \vdash r(constant) \Downarrow l}\ \textbf{Constant} \qquad \frac{\Sigma \vdash l = [def(reg_{N-1}), def(reg_{N+1})]}{\Sigma \vdash r(reg_N) \Downarrow l}\ \textbf{Register}$$

$$\frac{\Sigma \vdash l = \bigcap_{\forall reg \in mem_N} r(reg) \cap [def(mem_{N-1}), def(mem_{N+1})]}{\Sigma \vdash r(mem_N) \Downarrow l}\ \textbf{Memory} \qquad \frac{\Sigma \vdash l = \bigcap_{\forall var \in expr} r(var)}{\Sigma \vdash r(expr) \Downarrow l}\ \textbf{Expression}$$

$$\frac{\Sigma \vdash l = r(var) \cap r(expr) \quad \Sigma' = \Sigma[l \leftarrow var ::= expr]}{\Sigma \vdash r(var ::= expr) \rightsquigarrow \Sigma'}\ \textbf{Statement}$$

Figure 11: Operational semantics of the range operator (*rng-map*)

valid location (l). The algorithm to choose the insertion location is discussed in Section 3.4.2.

A.2 Theorems and proofs

A.2.1 Soundness Theorem

In this section, we provide the proof for Theorem 1 presented in Section 3.3. The efficiency and correctness of a typical live variable analysis is proven using a semi-lattice framework [1]. We apply the same methodology to guarantee the correctness of outer analysis. In our context, correctness means that our optimized code has the same effect as the original data tracking logic. While our analysis is almost identical to a typical live variable analysis, we only assume that an incomplete CFG is available.

Live variable analysis fits in a semi-lattice framework, which is employed to verify the safety and efficiency of an entire family of data flow problems. The data flow equations for the analysis can be stated directly in terms of $IN[B]$ and $OUT[B]$, which represent the set of variables live at the points immediately before and after block B. Also, we define

1. def_B as the set of variables *defined* (*i.e.,* definitely assigned values) in B prior to any use of that variable in B

2. use_B as the set of variables, whose values may be used in B prior to any definition of the variable.

The transfer functions relating def and use to the unknown IN and OUT are defined as:

$$IN[EXIT] = \phi \tag{1}$$

and for all basic blocks B other than $EXIT$,

$$IN[B] = use_B \cup (OUT[B] - def_B) \tag{2}$$

$$OUT[B] = \bigcup_{\text{S successor of B}} IN[S] \tag{3}$$

Equation 1 is a boundary condition, which defines that no variables are live when the program exits. Equation 2 specifies that a variable coming into a block is live, either if it is defined before usage in that block, or if it is live when coming out of the block. The last equation states that a variable coming out of a block is live, only if it is live when coming into any of its successors. These definitions comprise the most significant element of the semi-lattice framework. The monotone property of the above transfer functions is an important factor that confirms the safety of the analysis.

Since we only assume an incomplete CFG is available, we show that the same algorithm can be applied to our analysis. We modified the live variable analysis by adding a conditional statement to Equation 3 which defines all variables as live, if an unknown successor block exists (*e.g.,* due to an indirect branch). This in turn replaces Equation 3 with Equation 4.

$$OUT[B] = \bigcup_{\text{S successor of B}} IN[S] \tag{4}$$

(For any unknown S, $IN[S] = U$)

By having this equation, we claim the following theorem:

Theorem 1. ***Soundness of outer analysis:*** *Live variable analysis with incomplete CFG converges and is safe.*

Proof. The algorithm for the analysis will terminate, as we always have a *finite* number of blocks. Also, the additional statement in Equation 4 does not have any effect on the *monotone* property of the transfer functions, as it imposes conditions only on input arguments, and not on the function itself. Thus, it follows from the definition that the modified analysis still fits in the semi-lattice framework. □

A.2.2 Efficiency theorem

In this section, we provide the proof for Theorem 2 presented in Section 3.4.1. The theorem states:

Theorem 2. ***Efficiency of the TFA optimization****: The TFA optimization always produces less, or an equal number of, tracking statements than the original representation, for any basic block.*

Proof. Note that *n()* returns the number of statements in taint-map and T, T', and T'' represent the taint-map

data structures generated from the different stages of the TFA optimization.

T : the original taint-map directly translated from a given basic block B

T' : the copy propagation (substitution algorithm) applied to T

T'' : the range correction is applied to T'

then we want to show $n(T) \geq n(T'') \geq n(T')$.

An example taint-map data structure and its DAG representation is shown in Figures 3(b) and 5 respectively. Showing $n(T) \geq n(T')$ and $n(T'') \geq n(T')$ are trivial, as the substitution algorithm combines two or more statements into a single statement, and range correction splits a statement into two or more statements.

What we need to prove is $n(T) \geq n(T'')$. To show this, we use proof by contradiction. Let's assume that T'' is an optimal taint-map that does not have any range violations, and $n(T) < n(T'')$. WLOG, we can say that a taint-map can have statements such that $\exists s_0, s_1 \in T$ merged into $s'_0 \in T'$ by substitution algorithm, and s'_0 again split into $s''_0, s''_1, s''_2 \in T''$ by range correction. Then, we can have a new taint-map T''' such that $T''' = (T'' - s''_0, s''_1, s''_2) \cup s_0, s_1$. The newly created T''' does not have range violation and $n(T''') < n(T'')$. This contradicts our assumption that T'' is optimal taint-map with the minimal number of valid statements. $\square$