Beyond Instruction Level Taint Propagation

Beng Heng Ng Earlence Fernandes Ajit Aluri Atul Prakash
Department of Computer Science and Engineering
University of Michigan
Ann Arbor, Michigan 48105, USA

Zijiang James Yang
Department of Computer Science
Western Michigan University
1903 West Michigan Avenue Kalamazoo, MI 49008, USA

ABSTRACT

Dynamic taint analysis (DTA) plays a fundamental role in computer security research. However, current implementations of DTA are often inefficient as taint information is propagated for each instruction. Previous work has suggested propagating taint information at higher abstractions such as functions. But, this has only been achieved by manually instrumenting taint rules for library functions. Research on automatically creating taint propagation rules for higher levels of abstraction is lacking. Towards addressing the research gap, we propose the notion of straight line code units (SLCUs) and describe a technique to reduce higher abstractions like functions to SLCUs. Since a basic block is equivalent to a SLCU, current basic block summarization techniques can be applied to SLCUs. We propose an algorithm for automatically summarizing taint propagations for SLCUs with no or single pointer indirections, which we describe to be unaffected by memory aliasing. Preliminary results indicate that at least 87% of the basic blocks (the most basic form of SLCU) from a set of common Linux libraries fulfill this criteria.

1. INTRODUCTION

Dynamic taint analysis (DTA) is one of the two most commonly used dynamic analysis techniques, which are fundamental to computer security research [16, 5]. DTA is often used to analyze software binaries for security problems such as buffer overflows, format string vulnerabilities [12] and leakage of sensitive information [10]. When performing DTA on a program, untrusted or sensitive input data from different sources such as user input, network sockets, or files, is tainted, which are then propagated and tracked as the program executes. Depending on the analysis objectives, the program counter or the output data is then checked for the presence of the taints. DTA complements static binary analysis by providing precise information that is unavailable in the latter.

Traditional taint propagation techniques propagate taint information from source to destination operands for each instruction. However, this is often slow [12, 4, 13, 19]. For example, TaintCheck [12], a DTA tool based on Valgrind, can have slowdowns ranging from 3x to 37x, depending on the type of workload.

An obvious approach to improve the efficiency of DTA is to perform taint propagations at a higher abstraction level, such as the basic block abstraction, instead of propagating taint information at each instruction. Besides timing improvements, savings in other resources such as memory can be expected. While some work has pointed out this optimization [19], few studies actually discuss the nuances or measure the potential gains of this approach.

Our contributions are as follows.

- We compare the advantages and disadvantages of summarizing taint propagations at the basic block, loop, and function abstractions. We discuss the effects of memory aliasing and the resulting dilemma on whether taint information should be propagated before or after a basic block.
- We generalize the notion of a basic block to Straight-Line Code Units (SLCUs) and describe how we can reduce higher abstractions like functions to SLCUs to which our basic block summarization algorithm can be applied.
- We propose an algorithm for automatically generating summarized rules for taint propagations in SLCUs with no or single pointer indirections.
- We conducted an empirical study on common Linux libraries which shows that at least 87% of these basic blocks satisfy our criteria of containing only single or no pointer indirections.

2. RELATED WORK

Taint tracking related research can be grouped into three broad categories: binary instrumentation, whole system emulation, and hardware extensions. We outline the most important results related to high performance taint tracking systems from each of these groups.

Chang et al. propose a system that performs static inter-procedural flow analysis to determine points in a program where an attack could take place [3]. It then performs taint tracking instrumentation of these locations. The authors report 13% slowdown on server programs. The method relies on the existence of source code, whereas our system does not have this requirement. Similarly, in [11], Lam et al. propose a compiler that performs taint instrumentation and requires source code. It is not automatic since it relies on the programmer to specify interception points and proxy functions. This is a barrier to its widespread adoption since the taint tracking logic needs to be manually optimized by the programmer. TaintTrace, suggested by Zhao et al., uses DynamoRIO instrumentation on binaries to minimize register spilling by making use of dead registers to store taint values [4]. They do not try to minimize propagation overhead, which is what we are mainly concerned with.

Another methodology seen in the literature [13, 8, 15] is to maintain two versions of a code unit, a taint tracked (slow) version and an optimized (fast) version. For example, in LIFT [13], a work by Qin et al., the Fast-Path optimization verifies whether the live-ins of a code unit (basic block or hot trace) are safe. If they are, the faster non-taint-tracked version is executed, otherwise the slow version is executed. Note that we provide this guarantee as well and at the same time provide a best effort taint tracking through "unsafe" code units via our basic block summaries. Therefore, their slow version is potentially executed faster on our system. Similarly, LIFT dynamically switches between heavily instrumented QEMU and faster XEN virtual machines. Saxena et al. use static analysis to determine what code is on the fast-path [15].

Bosman et al. re-implement an emulator with taint tracking in mind in their work on minemu [2]. They make use of SSE registers to minimize register pressure on traditional x86 arch. This is orthogonal to our approach and it can draw further benefits from summarized taint tracking. RIFLE [17] is a compiler to convert conventional ISA to information-flow security ISA. This is similar to previous mirror-approaches that insert a taint-instruction corresponding to a program instruction. The ISA defines additional registers to hold taint values.

Another approach to improve DTA performance is to parallelize taint tracing. In [14], Ruwase et al. present techniques on how to execute taint tracking code in parallel by relaxing the rules of taint propagation. While this does produce speedup, it is orthogonal to our work. We focus on redefining the taint data flow problem in terms of basic block summarization.

In a closely related work, Jee et al. **??** separate the taint tracking logic from the program logic and apply compiler optimization techniques on the former to minimize the tracking code. Their main goal is to remove redundant tracking instructions by defining a Taint Flow Algebra that helps in applying dead code elimination, copy propagation and other standard optimization techniques to the taint tracking logic. While our approach is similar in thought, we aim to achieve near elimination of taint tracking logic through a basic block by constructing a taint propagation summary for it. Our ideas, with minimal or few changes can be easily extended to higher levels of abstractions such as loops and functions.

Another related work, that we drew inspiration from is TaintEraser by Zhu et al. **??**. This work proposes turning off taint propagation at the function level, and use a patching function to propagate taints between the inputs and outputs. They achieved roughly 10.7x performance improvement. However, TaintEraser uses "human experts" to generate the summaries of highly utilized functions. We make a step towards the automated generation of summaries for basic blocks.

3. POSSIBLE APPROACHES

An important consideration when approaching the problem of optimizing taint propagations is the different abstractions, i.e., basic block, loop, or function level abstractions, since each abstraction will require a different strategy. Perhaps the primary advantage of the basic block abstraction is simplicity. A basic block, defined as a contiguous sequence of instructions with a single entry and a single exit point, allows classical static data-flow analyses such as reaching definitions to be applied. Moreover, straight-line code enables robust reasoning. These factors enhance the correctness of the proposed strategy.

Besides summarizing taint propagations at the basic block level, another approach is to summarize them at the loop level. For loops with known bounds, static analysis is straightforward using techniques such as loop unrolling. However, for loops whose bounds are unknown, this becomes a challenge. One possibility to resolve this may be to leverage symbolic execution to compute the upper bounds for these loops. The outcomes for the static analysis can be either *constant* or *varying* taint propagations. For constant taint propagations, the sources and sinks remain unchanged for every iteration. For such cases, the taint information need only be propagated once regardless of the number of times the loop iterates. In contrast, for varying taint propagations, the sinks and sources differ between different iterations for a single run of the loop, or differing sinks and sources between different runs of the loop. Such loops may not be amenable for taint propagation summarizations and may require summarization at lower abstractions.

For nested loops, the summarizations may be performed prior to and after the execution of an inner loop. This is because taint propagations for the inner loop may be dependent on taint propagations before entering the inner loop. The taint propagations for the inner loop may also affect the taints after exiting the loop. Further study is needed to determine if the overheads for such a mechanism exceed that of summarizing the taint propagations at a lower abstraction.

When using the function abstraction for summarizing taint propagations, there are several advantages. Firstly, there are likely more opportunities to identify redundant taint propagations at the function abstraction, and thus leading to better optimizations. Function abstractions may also be better suited for analysis by normal users and developers. For example, it may be more semantically intuitive to the user that a taint is propagated through a certain function, rather than a basic block.

However, complexities can arise due to the presence of branches and loops in functions. One potential pitfall is the path explosion problem. Function calls within a function can complicate analysis, particularly in the case of recursive functions. For a normal function call within a function, the approach can be similar to loops, i.e., summarizations are performed prior and after the function call. For recursive functions, if the sources and sinks are constant regardless of the recursion depths, then summarization is straightforward. On the other hand, if the sources and sinks vary, then the summarization may be performed at a lower abstraction.

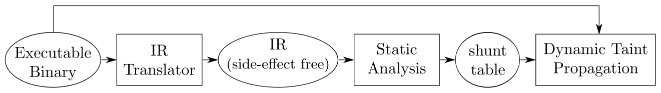


Figure 1: Process overview.

4. OVERVIEW

Figure 1 provides an overview of the process. For the ease of subsequent analysis, the executable binary to be analyzed is first statically translated into an Intermediate Representation (IR) that is free of side-effects. An example of such a side-effect is the modification of the `eflags` register by x86 arithmetic instructions.

Next, we statically analyze the IR to produce Straight Line Code Units (SLCUs). A SLCU is essentially a sequence of instructions along the same execution path. We define *inputs* of a SLCU as register reads and memory loads, and *outputs* as register writes and memory stores. The set of taint *sources* is a subset of the inputs. Similarly, the set of taint *sinks* is a subset of the outputs.

The SLCUs are then used to generate a *shunt table* for looking up taint propagation rules during dynamic analysis. A shunt table contains rules that specify the optimized taint propagations. Each rule, $(sig, sink, src_1, src_2, \ldots, src_n)$, which specifies that taint records from $src_1, src_2, \ldots, src_n$ need to be propagated to $sink$ for a SLCU having signature sig (which can be the address).

During dynamic taint propagation, our system looks up the shunt table to find taint propagation rules for the SLCU. If rules exist, the SLCU is executed without the instruction-level taint propagation logic. Taint propagations are carried out based on the rules. On the other hand, if no rule exists, then our system falls back to propagating taints at the instruction-level.

A disadvantage of the SLCU abstraction is the overhead needed to look up and evaluate the taint propagation rules for each SLCU. Besides, if the SLCU happens to be a basic block with few optimization opportunities, the overhead may exceed the savings over per-instruction taint propagation. However, we envisage that proper heuristics to guide the process will result in savings.

A heuristic we are developing involves an instruction cost model. While generating the SLCU summaries, we need to speculate about the potential savings and summarize units only when it is feasible for us. We measure this using estimated overheads of executing a summary of an SLCU and executing normal per-instruction taint propagation. Let Δ_s be the former, and $C_T(I)$ be the cost associated with executing the corresponding taint instructions for a machine instruction I. Then, the heuristic can be expressed as:

$$\Delta_s - \textstyle\sum_{\forall I \epsilon SLCU} C_T(I) < 0$$

The overhead to execute a set of taint instructions for every original instruction is the sum of the costs as determined by the cost model. This model gives us an estimate on the number of processor cycles it takes to complete the taint propagations for an SLCU. For example, a `mov` instruction's corresponding taint instructions can involve two loads: one arithmetic and one store operation. In this way, every machine instruction has an estimated cost associated with it. Hence, $\sum_{\forall I} C_T(I)$ represents the cost of executing SLCUs with per-instruction taint tracking. The overhead to execute the summary, Δ_s, involves the overhead to lookup rules for all live-outs of the unit, the overhead to generate the instrumentation and the number of loads, arithmetic and stores needed for the summary. If there is a distance between the two overheads, which is determined empirically, we conclude that it is feasible to summarize an SLCU.

To facilitate our discussions, we will use the syntax for the C language to describe the instructions. However, to keep our discussion focused on the optimization techniques, we will assume that the operations do not have side-effects. We also limit the variables to represent two types of registers: system registers (e.g., `eax`, `ebx`, `ecx`, etc.), and temporary registers (tn, where n is an integer). Temporary registers are used to store temporary values in-between computations. The use of temporary registers allows for the Static Single Assignment (SSA) property to be upheld. The SSA property greatly simplifies analysis since each temporary register is only defined once.

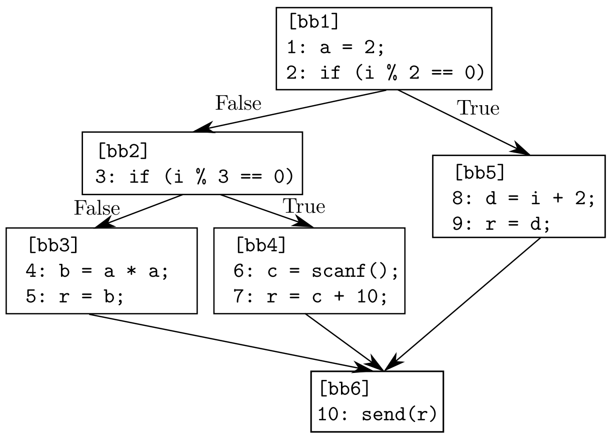
4.1 Taint Propagation Syntax

Taint records can be read/written from/to different locations. We now explain the syntax we will use to describe the propagation of the taint records for both the taint propagation syntax and the taint propagation rules. $\tau(a)$ returns the set of taint records for variable a, while $\tau(*a)$ returns the set of taint records at the location pointed to by a. $\varsigma(b, T)$ overwrites the existing taint records for b with those for T. The only operator allowed on two sets of taint records is the union operator, i.e., $\cup$.

For simplicity, we will use the taint policy described in

Instruction	Taint Policy
$t = *u$	$\varsigma(t, \tau(*u))$
$t = g$	$\varsigma(t, \tau(g))$
$*t = u$	$\varsigma(*t, \tau(u))$
$g = u$	$\varsigma(g, \tau(u))$
$t = u$	$\varsigma(t, \tau(u))$
$t = \langle unop \rangle u$	$\varsigma(t, \tau(u))$
$t = u \langle binop \rangle v$	$\varsigma(t, \tau(u) \cup \tau(v))$

Table 1: Taint policy. t, u, and v denote temporary registers. g denotes a system register.



```
[bb1]
1: a = 2;
2: if (i % 2 == 0)

[bb2]
3: if (i % 3 == 0)

[bb3]
4: b = a * a;
5: r = b;

[bb4]
6: c = scanf();
7: r = c + 10;

[bb5]
8: d = i + 2;
9: r = d;

[bb6]
10: send(r)
```

Figure 2: Example for summarizing taint propagations at function level.

Table 2. Constants are never tainted. The simplistic taint policy does not describe control-flow taint propagations for implicit flows, which we will defer to future work.

5. FROM FUNCTION TO SLCU

Summarizing taint propagations at the function level can provide more opportunities for performance gains. For example, the overheads incurred for looking up the propagation rules can be amortized over more basic blocks. The intuition is that the summarization techniques are essentially the same for different abstractions if the input is straight-line code, i.e., the analysis for a function without branches is the same as a basic block. Thus, towards summarizing taint propagations at the function level, we propose the following steps.

1. Identify all paths in a function.
2. Combine all basic blocks along each path to form SLCUs. The result is a straight-line code for each path.
3. Summarize taint propagations for each SLCU to generate shunting rules.

A challenge that may arise with identifying all the paths in a function is the path explosion problem. To mitigate this problem, we propose only summarizing taint propagations for functions with cyclomatic complexity of less than 15. We examined 582,959 functions from executables on various Linux distributions and found that 455,078 (78.06%) of the functions satisfy this criteria.

We will use the example in Figure 2 to illustrate our proposed steps. The paths identified will be as follows.

1. bb1 $\rightarrow$ bb2 $\rightarrow$ bb3 $\rightarrow$ bb6
2. bb1 $\rightarrow$ bb2 $\rightarrow$ bb4 $\rightarrow$ bb6
3. bb1 $\rightarrow$ bb5 $\rightarrow$ bb6

Next, we can join the basic blocks along each path to obtain SLCUs. For each SLCU, we can apply the summarization techniques to obtain the summarized taint propagation rules. We use the path conditions to describe the paths. The summary table is shown in Table 2, where we use P_n to indicate the predicate at line n.

Path Conditions	Summarized Taint
$\neg P_2 \wedge \neg P_3$	$\varsigma(r, \emptyset)$
$\neg P_2 \wedge P_3$	$\varsigma(r, \{\tau(c)\})$
P_2	$\varsigma(r, \{\tau(i)\})$

Table 2: Taint summaries for various paths in Figure 2. P_n refers to the predicate on line n.

```
1: t1 = *eax
2: *ebx = t2
3: ecx = *t1
```

(a) Example 1.

$\varsigma(*ebx,\ \tau(t2))$
$\varsigma(ecx,\ \tau(*(*eax)))$

(b) Taint rules for Example 1.

Figure 3: Example 1 for optimized taint rules.

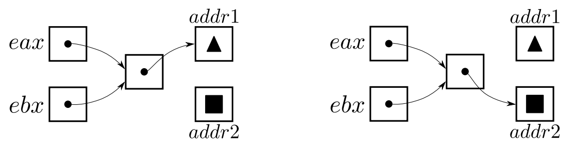
Once a summary table has been generated using the SLCU summarization algorithm, we need to form indices into this table at runtime. As our analysis is done statically, we need a mechanism to construct a path condition for an actual code path that was executed at runtime. We also need to insert the summarized taint code just before various sink points in the function since the table expresses the taint values of registers/memory at a sink point. Therefore, we propose an instrumentation that signals what branch is taken at each condition. When we reach a sink point (can be a return), we insert code to execute a summary. This summary is pulled in from a table indexed by path condition as stated earlier. Hence, a function is summarized. The key insight here is that we can reduce the function abstraction to a straight-line code abstraction and apply our SLCU summarization algorithm. Hence, our techniques are independent of the code abstraction.

5.1 Taint Propagation Dilemma

Summarized taint information can be propagated either before or after the SLCU. However, *memory aliasing* can lead to a dilemma as to when the taint information should be propagated. Memory aliasing, which can be detected using various techniques [6, 1, 7], occurs when different registers or memory locations can reference the same memory location. The dilemma arises if the data stored at the aliased memory location is the address of another memory location.

Suppose we have a SLCU as shown in Figure 3a, where eax and ebx are aliasing, i.e., they contain the address of the same memory location. The corresponding taint propagation rules are shown in Figure 3b. Figure 4 illustrates the pointer status before and after executing the SLCU. If the taint rules are effected *before* the SLCU is executed, the resulting taint records for ecx will be ▲ as shown in Figure 4. This is the expected result since $addr1$ is saved in $t1$ and used after Line 2. However, if the rules are effected *after* executing the SLCU, the resulting taint records for ecx will erroneously become ■ as in Figure 4. At Line 2, $*ebx$ is updated with the value of $t2$, and since eax aliases with ebx, this implies $*eax$ is also updated.

On the other hand, suppose we have a SLCU as shown in Figure 5a with taint rules shown in Figure 5b. Similar to Example 1, Figure 4 shows the pointer status before and after executing the SLCU. If the taint rules are effected *after* the SLCU is executed, the taint records for ecx will be ■, which is expected. However, if the rules are effected *before* executing the SLCU, the resulting taint records for ecx will



(a) Before SLCU. (b) After SLCU.

Figure 4: Pointer status for both Example 1 and 2. ▲ and ■ represent the taint records at $addr1$ and $addr2$ respectively.

```
1: *ebx = t2
2: t1 = *eax
3: ecx = *t1
```

(a) Example 2.

$\varsigma(*ebx,\ \tau(t2))$
$\varsigma(ecx,\ \tau(*(*eax)))$

(b) Taint rules for Example 2.

Figure 5: Example 2 for optimized taint rules.

```
1: t3 = esp
2: t1 = t3 + 0x18
3: t4 = *t1
```

(a) Single indirection.

```
1: t3 = esp
2: t1 = t3 + 0x18
3: t4 = *t1
4: t5 = t4 + 0x4
5: t6 = *t5
```

(b) Double indirection.

Figure 6: Examples of pointer indirections.

be ▲, which is incorrect, since we expect $*eax$ to be updated to $addr2$ before the taint records are propagated.

This results in a dilemma between propagating the taint records before or after the SLCU. In some cases, executing taint summaries before the block is correct and in some cases, executing the summary after the block is correct. To remediate the situation, we propose optimizing only SLCUs with at most single pointer indirections. In other words, we ignore SLCUs that have memory loads with more than one level of indirection.

An additional source of nondeterminism is that if multiple indirections were summarized, it is not sufficient to make a decision between executing the summary before or after the block. Consider Figure 6a which shows an example for a single indirection, while Figure 6b shows an example for a double indirection. The optimized taint rule for the instructions in Figure 6a will be $\varsigma(t4, \tau(*(esp + \texttt{0x18})))$. In comparison, the taint rule for the instructions in Figure 6b will be $\varsigma(t6, \tau(*(*(esp + \texttt{0x18}) + \texttt{0x4})))$. If we look at the doubly indirected rule, we cannot know in advance what is the final pointer value until the statement that computes the first indirection has completed. Hence, the summary code needs to be sprayed into the block as well, and this will result in additional overhead.

An additional advantage of limiting the optimizations to SLCUs with no loads or first-order loads is simplicity when parsing the taint rules. One can consider parsing the rules to incur overheads, thus having to parse and evaluate rules for high-order loads may incur overheads that exceed the savings from the optimizations.

5.2 Automated Taint Summarization

Towards automatically computing the taint sources and sinks at the SLCU abstraction, we propose the algorithm shown in Figure 7. The `ComputeTaintProps` function takes in a sequence of instructions, live-ins, and live-outs for a SLCU and returns an ordered list where each element specifies a taint sink and a set of taint sources. The intuition behind the algorithm is that since branches are removed from a SLCU, the steps to propagate taint information during dynamic analysis will be the same as those during static analysis. Thus, assuming no side-effects, one can simulate the taint propagation steps trivially and extract the resulting taint sources and sinks.

The algorithm begins by initializing the initial pseudo-taint to a unique value, which is also added to the last pseudo-taint set. The pseudo-taints are used to emulate taint information, thus allowing us to keep track of the originating taints. Every instruction in the SLCU is then examined. The last pseudo-taint sets for all right-hand side operands are union-ed in a set T, which is then updated to the last pseudo-taint set of the written operand.

Finally, the algorithm extracts all sinks that are found in the live-out variables and gets all the originating sources us-

```
function ComputeTaintProps(SLCU $slcu$, LiveIns $I$, LiveOuts $O$)
    $pseudo\_taint = 0$
    Map $src$                    ▷ Tracks initial taints for sources
    Map $isrc$                   ▷ Tracks initial taints for indirected sources
    Map $last$                   ▷ Tracks last taints for locations
    Map $ilast$                  ▷ Tracks last taints for indirected locations
    for each $i$ in $I$ do
        $src[i] = pseudo\_taint$
        $isrc[i] = pseudo\_taint$
        $last[i] = \{pseudo\_taint\}$
        $ilast[i] = \{pseudo\_taint\}$
        $pseudo\_taint = pseudo\_taint + 1$
    end for
    for each instruction $i$ in $slcu$ do
        $T = \{\}$
        $rop$ = Get right operands for $i$
        for each right operand $ro$ in $rop$ do
            if UsedAsPointer($i$, $ro$) then
                $T = T \cup ilast[ro]$
            else
                $T = T \cup last[ro]$
            end if
        end for

        $lop$ = Get left operand for $i$
        if UsedAsPointer($i$, $lop$) then
            $ilast[lop] = T$
        else
            $last[lop] = T$
        end if
    end for

    $r$ = new Map()              ▷ Maps sinks to resolved taint sources
    ResolveIndirection($src$, $isrc$, $last$, $O$, $r$)
    ResolveIndirection($src$, $isrc$, $ilast$, $O$, $r$)
    return $r$
end function

function ResolveIndirection(Map $src$, Map $isrc$, Map $last$, LiveOuts $O$, Map $output$)
    for each $l$ in $last$ do
        if $l$ found in $O$ then
            $V = \{\}$
            for each $s$ in $last[l]$ do
                if $s$ found in $src$ then
                    $src$ = Get source from $src$ and $s$
                else if $s$ found in $isrc$ then
                    $src$ = Get source from $isrc$ and $s$
                    Mark $src$ for indirection
                end if
                $V = V \cup \{src\}$
            end for
            Mark $l$ for indirection if $l$ is from $ilast$.
            $output[l] = V$
        end if
    end for
end function
```

Figure 7: Algorithm for simulating taint propagations.

ing the pseudo-taints. If a pseudo-taint is found in `src`, then no indirection is required, i.e., taint information is propagated from the location specified. Otherwise, if it is found in `isrc` instead, then a single indirection is required, i.e., taint information is propagated from the memory pointed to by the location.

5.2.1 Algorithm Explanation

Consider the example in Figure 8. We will use it to demonstrate several important facets of our technique. The algorithm starts out by maintaining sets for taint values for all variables concerned with a sink. In Table 3, the following steps are represented.

Step 0 Initialization. All taint sets are initialized to $\emptyset$. `src(.)` represents the initial taint value of a register or memory location.

Step 1 The assignment of `ebx` to `t3` is simulated. This results in the sets being transferred.

Step 2 Similarly, another assignment is processed.

Step 3 Dereference. At this stage, the algorithm notices a dereference operation. Hence the set of `t4` is updated to indicate this fact. Observe that the taint set for `t4` is now `{src(*ebx)}`.

Step 4 Union. Since `t4` and `edx` have taint sets, the taint set for `ecx` will be the union. This is in accordance with our taint policy.

Step 5 Summary. `t4` is assigned a constant and this is assigned to `eax`. Hence the taint set for `eax` is $\emptyset$.

```
live-ins = { ebx, edx }
live-outs = { ecx, eax }

1: t3 = ebx
2: t1 = t3 + 0x18
3: t4 = *t1
4: ecx = t4 + edx
5: t4 = 5
6: eax = t4
```

Figure 8: Example SLCU.

Library	%Amenable
libgcrypt.so.11.5.2	93.41
ld-2.11.1.so	87.12
libcrypt-2.11.1.so	88.84
libc.so.6	87.91
libdbus-1.so.3.4.0	94.29
libdl-2.11.1.so	93.12
libm.so.6	98.46
libpthread-2.11.1.so	94.51
libz.so.1.2.3.3	87.48

Table 4: Percentage of basic blocks with no or single pointer indirections.

6. EXPERIMENTS

Prevalence of Pointer Indirections.

To estimate the number of basic blocks (the most basic form of SLCU) amenable to the generation of shunting tables, we built an IDA plugin that analyzed whether basic blocks contained only single pointer indirections or multiple pointer indirections as well. The plugin converted x86 disassembly (from IDA Pro) to VEX IR. We then analyzed every load operation for each basic block and computed a backward slice of its operands. This allowed us to check if a load was dependent on another prior load. We ran our analysis on a sampling of commonly used Linux libraries and the results are summarized in Table 4. The second column shows the percentages of total basic blocks that do not have multiple indirections. We observe that at least 87% of the basic blocks in the libraries are suitable for summarization by our proposed algorithm.

Overheads.

Next, we examined tainting overheads on TEMU [18], an extension of QEMU. We chose TEMU for its ability to propagate taints system-wide. The following experiments were conducted on a machine that has an Intel Core I7-3770 processor and 32 GB RAM. The slowdowns ranged from about 8.8x to 1,775.5x for inputs (prime numbers) 3, 2000003, 4000037, 6000011, 8000009, 10000019, and 12000017. When the input is 12,000,017, the test program was not able to run to completion on TEMU due to insufficient memory. The results clearly show that work remains to improve the performance of tainting frameworks. However, our work is orthogonal to improving the performance of individual frameworks. Instead, we are focused on general taint summarization techniques that can be applied towards different frameworks.

We evaluated the potential gains that can be achieved by manually summarizing taint propagations at the basic block and loop abstractions for a test program shown in Figure 9. For the basic block abstraction, the taint propagation overheads can be reduced by between 72.77% and 77.84%, while for the loop abstraction, the overheads can be reduced between 87.02% and 98.03%.

Step	t1	t3	t4	ebx	edx	ecx	eax
0	$\emptyset$	$\emptyset$	$\emptyset$	{src(ebx)}	{src(edx)}	$\emptyset$	$\emptyset$
1	$\emptyset$	{src(ebx)}	$\emptyset$	{src(ebx)}	{src(edx)}	$\emptyset$	$\emptyset$
2	{src(ebx)}	{src(ebx)}	$\emptyset$	{src(ebx)}	{src(edx)}	$\emptyset$	$\emptyset$
3	{src(ebx)}	{src(ebx)}	{src(∗ebx)}	{src(ebx)}	{src(edx)}	$\emptyset$	$\emptyset$
4	{src(ebx)}	{src(ebx)}	{src(∗ebx)}	{src(ebx)}	{src(edx)}	{src(∗ebx), src(edx)}	$\emptyset$
5	{src(ebx)}	{src(ebx)}	$\emptyset$	{src(ebx)}	{src(edx)}	{src(∗ebx), src(edx)}	$\emptyset$

Table 3: Example for computing summary.

```
int bruteForcePrimalityTest(unsigned int n) {
  for (unsigned int i = 2; i < n; i++) {
    if (n % i == 0) return 0;
  }
  return 1;
}
```

Figure 9: Test function that computes the primality of input.

7. CONCLUSIONS

DTA is slow and thus limiting its usability in security research. While the usefulness of DTA has been demonstrated in discovering security issues [16, 5] such as buffer overflows, format string vulnerabilities [12] , and sensitive information leakage [10], it is still not widely adopted, particularly in resource-constrained scenarios such as on mobile devices and browsers. The additional overheads associated with DTA lead to increased latency or, for mobile devices, decreased battery life. This limits the usefulness of DTA to offline analysis, which allows a window of opportunity for attacks to occur. Towards achieving real-time DTA, research is required to improve its performance.

Summarizing taint propagations is an intuitive approach to improve DTA's performance and has been suggested [19], but there has been no actual implementation beyond manually optimizing the taint propagations. Towards closing this research gap, we explored the nuances of summarizing taint propagations and estimated the potential gains of this approach.

We discuss the advantages and disadvantages of summarizing taint propagations at basic block, loop, and function abstractions. Based on these observations, we introduced the concept of the SLCU as a unit of summarization for taint propagations. The SLCU is attractive because higher code abstractions can be reduced to it, and they can be automatically summarized.

We also examine the effects of memory aliasing on choosing where to execute the summarized taint propagations, and propose the SLCUs be used as the basic unit for summarization. Our results show that approximately at least 87% of the basic blocks (the most basic form of an SLCU) in our test set of popular linux libraries fulfill this criteria, thus suggesting potential performance gains from optimizing such SLCUs should be significant. Additionally, we propose an algorithm for automatically summarizing taint propagations for SLCUs.

Using our analysis and experimental results as foundation and motivation, our future work will be towards fully automating taint propagation summarizations, thus making DTA more practical.

8. REFERENCES

[1] G. Balakrishnan and T. Reps. Analyzing Memory Accesses in x86 Executables. In *In CC*, pages 5–23. Springer-Verlag, 2004.

[2] E. Bosman, A. Slowinska, and H. Bos. Minemu: The World's Fastest Taint Tracker. In *Proceedings of the 14th international conference on Recent Advances in Intrusion Detection*, RAID'11, pages 1–20, Berlin, Heidelberg, 2011. Springer-Verlag.

[3] W. Chang, B. Streiff, and C. Lin. Efficient and Extensible Security Enforcement using Dynamic Data Flow Analysis. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 39–50, New York, NY, USA, 2008. ACM.

[4] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige. TaintTrace: Efficient Flow Tracing with Dynamic Binary Rewriting. *Computers and Communications, IEEE Symposium on*, 0:749–754, 2006.

[5] J. Clause, W. Li, and A. Orso. Dytan: A Generic Dynamic Taint Analysis Framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pages 196–206, New York, NY, USA, 2007. ACM.

[6] S. Debray and R. Muth. Alias Analysis of Executable Code. In *In POPL*, pages 12–24, 1998.

[7] M. Fernández and R. Espasa. Speculative Alias Analysis for Executable Code.

[8] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical Taint-Based Protection using Demand Emulation. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 29–41, New York, NY, USA, 2006. ACM.

[9] 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 *In Proc. of the 19 th NDSS*, 2012.

[10] H. C. Kim, A. D. Keromytis, M. Covington, and R. Sahita. Capturing Information Flow with Concatenated Dynamic Taint Analysis. In *ARES*, pages 355–362. IEEE Computer Society, 2009.

[11] L. C. Lam and T.-c. Chiueh. A General Dynamic Information Flow Tracking Framework for Security Applications. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, ACSAC '06, pages 463–472, Washington, DC, USA, 2006. IEEE Computer Society.

[12] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. 2005.

[13] 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 *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 135–148, Washington, DC, USA, 2006. IEEE Computer Society.

[14] O. Ruwase, P. B. Gibbons, T. C. Mowry, V. Ramachandran, S. Chen, M. Kozuch, and M. Ryan. Parallelizing Dynamic Information Flow Tracking. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA '08, pages 35–45, New York, NY, USA, 2008. ACM.

[15] P. Saxena, R. Sekar, and V. Puranik. Efficient Fine-grained Binary Instrumentation with Applications to Taint-Tracking. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, pages 74–83, New York, NY, USA, 2008. ACM.

[16] 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 *Proceedings of the IEEE Symposium on Security and Privacy*, 2010.

[17] 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 *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 243–254, Washington, DC, USA, 2004. IEEE Computer Society.

[18] H. Yin and D. Song. TEMU: Binary Code Analysis via Whole-System Layered Annotative Execution. Technical Report UCB/EECS-2010-3, University of California at Berkeley, January 2010.

[19] 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(1):142–154, Feb. 2011.