StraightTaint: Decoupled Offline Symbolic Taint Analysis

Jiang Ming, Dinghao Wu, Jun Wang, Gaoyao Xiao, and Peng Liu
College of Information Sciences and Technology
The Pennsylvania State University
University Park, PA 16802, USA
{jum310,dwu,jow5222,gzx102,pliu}@ist.psu.edu

ABSTRACT

The multifaceted benefits of taint analysis have led to its wide adoption in *ex post facto* security applications, such as attack provenance investigation, computer forensic analysis, and protocol reverse engineering. Unfortunately, the high runtime overhead imposed by dynamic taint analysis makes it impractical in many scenarios. The key obstacle is the strict coupling of program execution and taint tracking logic code. To alleviate this performance bottleneck, recent work seeks to offload taint analysis from program execution and run it on a spare core or a different CPU. However, since the taint analysis has heavy data and control dependencies on the program execution, the massive data in recording and transformation overshadow the benefit of decoupling. In this paper, we propose a novel technique to allow very lightweight logging, resulting in much lower execution slowdown, while still permitting us to perform full-featured offline taint analysis, including bit-level and multi-tag taint analysis. We develop StraightTaint, a hybrid taint analysis tool that completely decouples the program execution and taint analysis. StraightTaint relies on very lightweight logging of the execution information to reconstruct a straight-line code, enabling an offline symbolic taint analysis without frequent data communication with the application. While StraightTaint does not log complete runtime or input values, it is able to precisely identify the causal relationships between sources and sinks, for example. Compared with traditional dynamic taint analysis tools, StraightTaint has much lower application runtime overhead.

CCS Concepts

•**Security and privacy** → **Software security engineering;** *Information flow control; Software reverse engineering;*

1. INTRODUCTION

Taint analysis, as a special form of data-flow analysis [21, 38], has a variety of compelling applications in security tasks. In addition to the runtime enforcement security policies [30, 35], taint analysis on the binary code is also broadly used in *ex post facto* security applications, such as attack provenance investigation [24, 46], computer forensic analysis [23], malware analysis [7, 51], and reverse engineering [8, 28, 49]. Static taint analysis (STA) [1, 36, 44] aims to reason the causal data flow relationships between sources and sinks prior to execution. However, static taint analysis is not precise enough when the source code is unavailable, especially for the obfuscated binary code. On the other hand, dynamic taint analysis (DTA) [14, 30, 35], propagating taint tags along the program execution path, is accurate in many scenarios wherein static taint analysis cannot achieve the needed precision. However, dynamic taint analysis typically suffers from a high performance penalty. In general, the state-of-the-art dynamic taint analysis tools such as libdft [20] typically impose more than a 6X slowdown. In the worst cases, the slowdown can easily go up to 20–30X [14, 30]. The high runtime overhead imposed by dynamic taint analysis has severely limited its application scope.

The key obstacle to further improving the performance of dynamic taint analysis is the tight coupling of program execution and taint tracking logic code [39]. Taint analysis has to maintain a shadow memory to map instruction operands to their corresponding taint tags. To propagate one taint tag between different residences, it typically takes 6–8 additional instructions [12]. In addition, since the taint tracking code is interleaved with the program execution, the frequent "context switches" between the application and taint analysis code impose further pressure on both registers and data cache (e.g., register spilling and cache miss), incurring substantial overhead.

To lower the high performance overhead, multiple methods have been proposed to offload taint tracking code to a separate core or a different CPU. The existing work can be roughly classified into two categories. The first category relies on the pervasive multi-core systems to parallelize dynamic taint analysis by logging runtime values that are needed for taint analysis in another core [18, 19, 31, 40]. However, since taint analysis has strong serial data and control dependencies on the program execution, the parallelized taint analysis need to be frequently synchronized for data communication (e.g., control flow directions and memory addresses), either through customized hardware [31, 40] or shared memory [18, 19]. The second category first records the application execution and then replay the taint analysis on a different CPU [15, 42, 45, 48]. Similar to the limitation of the first category, the large online logging data is also a barrier to achieving the expected performance gains.

In this paper, we propose *StraightTaint*, a hybrid static and dynamic method that achieves very lightweight logging, resulting in much lower execution slowdown, while still

The 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016), Singapore, September 3–7, 2016.

permitting us to perform complete offline taint analysis with incomplete inputs. In principle, StraightTaint belongs to the aforementioned second category of decoupling DTA approaches. Therefore, StraightTaint is an ideal fit for *ex post facto* security applications. In StraightTaint, we do not log all runtime values. Instead, we record control flow profiling and execution state when taint seeds are first introduced, which can be very lightweight. Based on the logged branching information, we construct a straight-line code trace for the offline taint analysis. The taint seeds are marked as symbolic variables, and taint propagation is like the symbolic execution on the constructed straight-line code. With the initial execution state and the straight-line code, most addresses of memory load and store operations are computable. Symbolic memory indices can be narrowed down to a small range by solving the path conditions. Compared to a pure static approach, StraightTaint can still deliver a similar level of precision as dynamic taint analysis. For example, we are able to correctly identify the complicated causal relationships among multiple sources and sinks (see Section 6), while static taint analysis fails in such cases.

Restricted by computing resources, conventional DTA exhibits several drawbacks in terms of incomplete taint propagation strategies. First, since multi-tag taint propagation consumes more shadow memory and introduces much higher runtime overhead, most DTA tools choose single-tag propagation as default [12, 20, 30, 35, 40, 54]. However, multi-tag taint analysis is indispensable to many reverse engineering tasks, such as recovering the structure of an unknown protocol format [8] and detecting encoding functions in malware by counting different tainted input bytes [7]. Second, when handling the complicated x86 arithmetic and logic operation instructions (e.g., `add` and `xor`), previous DTA tools typically adopt some simple but conservative propagation strategies for better performance. One example is the prevalent "short circuiting" method: the destination operand is tainted if any of the source operands is tainted. As we will show, these conservative solutions will result in precision loss in many scenarios. As StraightTaint has completely offloaded the taint logic code to the offline analysis, another benefit becomes visible: StraightTaint's offline taint analysis is flexible to support *full-featured* taint propagation strategies. For example, supporting bit-level [48] or multi-tag taint analysis is straightforward in our approach. Each symbolic bit or variable can naturally represent a taint tag with negligible additional overhead. Also, our symbolic execution style taint propagation can faithfully simulate the specific semantics of an instruction. Furthermore, based on symbolic taint analysis on the straight-line code, we introduce a new concept, *Conditional Tainting*; that is, StraightTaint is able to identify precisely the causal data flow relations between sources and sinks, under what *conditions*. In this way, new inputs and runtime values can be mapped to the existing analysis results in certain scenarios so that the new analysis can be more proactive.

We have developed a prototype of StraightTaint, a hybrid taint analysis approach that completely decouples the program execution and taint analysis. Our implementation is based on Pin [25], for the effective parallelization of runtime logging, and BAP [6], for precise offline symbolic taint analysis with incomplete inputs. We have performed comparative studies on a number of applications such as common utility programs, SPEC2006, and real-life software vulnerabilities. The results show that StraightTaint can achieve a similar level of precision as dynamic taint analysis, but with much lower online execution slowdown. The performance experiments show that StraightTaint imposes a small overhead on application execution performance, with up to 3.25 times improvements to SPEC2006 on average. Offline taint analysis takes approximately the same amount of time as an advanced DTA tool. We also demonstrate StraightTaint's value in supporting multi-tag taint propagation and conditional tainting in an attack provenance investigation task. Such experimental evidence shows that StraightTaint can be applied to various large-scale *ex post facto* security applications.

In summary, we make the following contributions:

1. We propose StraightTaint, with a very lightweight logging method to construct straight-line code and thus completely decouple dynamic taint analysis for offline symbolic taint analysis. StraightTaint greatly reduces the program execution slowdown yet can compete with dynamic taint analysis with a similar level of precision.

2. The limitation of previous decoupling taint work is inefficiently collecting and transferring data from the executing application to the analysis module. We demonstrate that StraightTaint offline analysis does not require complete runtime data but can still achieve most tasks.

3. The completely decoupled offline taint analysis allows StraightTaint to perform *full-featured* taint propagation strategies. The symbolic execution style taint propagation can accurately describe the intricate semantics of the x86 instructions, and also naturally support multi-tag and bit-level taint analysis.

4. We introduce a new concept, *Conditional Tainting*, based on the symbolic taint analysis of straight-line code. Conditional tainting not only reports more precise and useful taint results but also opens many new important applications.

We also summarize the main benefits associated with our proposed taint analysis method.

1. Once a log is captured, it can be analyzed by StraightTaint multiple times. This feature is particularly useful when the exact analysis task is hard to anticipate. In our multi-tag taint propagation evaluation, we vary the number of taint tags in each round. StraightTaint only needs to log the required online data once and performs the multiple propagation rounds offline.

2. StraightTaint makes it possible to conduct *ex post facto* logging-based taint analysis in the cloud [32]. Service providers can deploy lightweight online logging in their services, and cloud hosts provide storage space for the logged data. Users can require a service to audit their sensitive data flow offline.

The rest of the paper is organized as follows. Section 2 provides the background information and an overview of our approach. Section 3 describes efficient online logging and our optimization. Offline symbolic taint analysis is discussed in Section 4. Section 5.1 highlights a few of our implementation choices. We present the evaluation of our approach in the rest of Section 5 and demonstrate its applications in Section 6. Related work is presented in Section 7. We conclude the paper and discuss future work in Section 8.

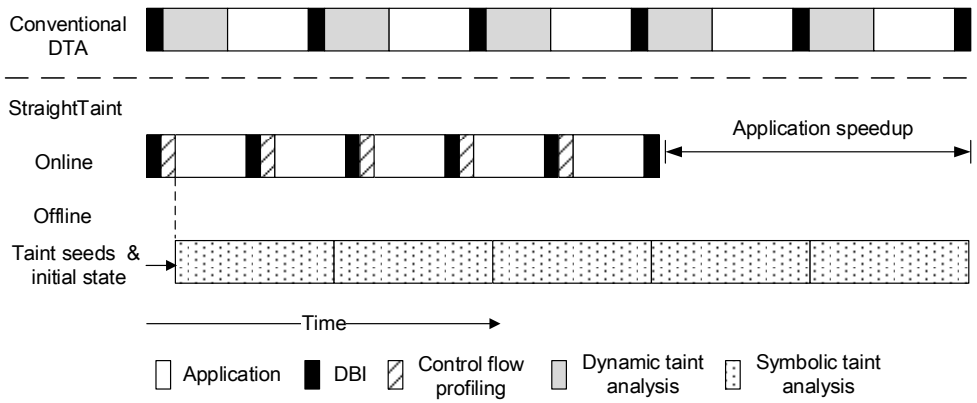


Figure 1: Conventional DTA vs. StraightTaint.

2. BACKGROUND AND OVERVIEW

2.1 Dynamic Taint Analysis Optimization

Dynamic taint analysis (DTA) is a form of information flow analysis to trace the *tainted* data along program execution path. Typically, the data derived from untrusted sources are labeled as *tainted* (i.e., taint seeds). The propagation of the tainted data will be tracked according to the taint propagation policy. Then the taint status will be checked at certain critical locations (i.e., taint sinks). DTA has been broadly employed in software security applications. However, an inherent limitation of conventional DTA is that taint logic is *strictly coupled* with program execution. Figure 1 illustrates a conventional DTA tool built on dynamic binary instrumentation (DBI). The taint tracking code is interleaved with program execution, leading to frequent context switches and resource competitions between the application and taint analysis code. As a result, the application under examination is significantly slowed down. Various advanced DTA techniques have been proposed to achieve decent runtime performance [3, 11]. For example, Minemu [3] leverages the x86 SSE registers to provide lightweight taint tracking for 32-bit applications. Unfortunately, they either rely on an ad hoc emulator [3] or cannot work on commodity hardware [11]. Decoupling taint analysis from program execution has been demonstrated as an effective approach. However, due to the heavy data and control-flow dependencies on the application execution, decoupled taint analysis cannot run independently. Intuitively, each memory address and control transfer target have to be delivered to the decoupled taint analysis. Therefore, the large logged data is a barrier to further improving the performance.

As shown in Figure 1, our key insight is that taint analysis can be completely decoupled from program execution, without frequent online communication and synchronization. Offline taint analysis can be performed based on control flow information and very little runtime data (e.g., the initial execution state when taint seeds are introduced). We notice that memory reference operations in x86 architecture are addressed through registers and constant offset calculations. For example, `mov ebx [4*eax+4]` loads the content stored at the address `4*eax+4` to `ebx`. With the initial execution state and the straight-line code, most memory reference addresses can be recovered. The proposed StraightTaint explores this idea.

Note that the execution replay work [13, 33, 47], which records required inputs and replays them on an offline analysis, can be applied to decouple taint analysis as well. Compared to StraightTaint, the logs are smaller, and the online performance could be better. However, the logged data contains little information about execution, making it impossible for direct taint analysis. Furthermore, the offline replay overhead is quite high. For example, Aftersight [13] replays a single-tag taint analysis on a QEMU-based CPU simulator, but the performance is as high as 100X slowdown. Our solution represents a middle ground that balances the performance between online logging and offline taint analysis.

2.2 Incomplete Taint Propagation Strategies

As conventional DTA tools are subject to limited computing resources, typically they have to adopt incomplete taint propagation strategies to achieve acceptable performance. In many cases, such conservative strategies lead to the precision loss. The first drawback comes from the single-tag propagation. Most DTA tools associate each variable with one shadow memory bit or byte to represent the taint status: 1 means tainted and 0 means untainted. Although single-tag works in some simple scenarios, multi-tag taint analysis has much broader security applications. For example, BitFuzz [7] assigns different taint tags to input bytes and then detects encoding functions in malware by identifying high taint degree; iBinHunt [26] utilizes multi-tag taint analysis to reduce the number of possible basic blocks to compare. Furthermore, many arithmetic and logic operation results overlap the operands so that a taint tag may come from multiple sources. Therefore, the multi-tag attribute is essential for accuracy as well. The second limitation is due to the conservative propagation strategies when dealing with the complicated x86 instructions. These simple strategies are fast but neglect the particular instruction semantics that may affect the taint propagation results. In addition to the frequently used "short circuiting" solution, some previous work tracks the taint flow only through unary operations (the output of a binary operation is set as untainted) to achieve better parallelization [40].

Figure 2 presents a snippet of an encoding function, which is frequently used in malware [7]. Figure 2 (a) lists a straight-line code with complicated arithmetic operations. Conventional DTA performs the taint analysis on this code snippet with single-tag and "short circuiting" strategies. Figure 2 (b) shows the propagation results: all variables are tainted. Look carefully at line 3 in Figure 2 (a), the taint tag of variable `w` derives from two taint seeds but conventional DTA just labels it as a single tag. Besides, the variable `d` will always be zero because `c` is the `bitwise NOT` of `a`. However, the "short circuiting" propagation mistakenly label

```
int a, b, c, d, w;
int low_bits = 0x0000ffff;
int high_bits = 0xffff0000;
1: a = read ();
2: b = read ();
3: w = (a ∧ low_bits) ∨ (b ∧ high_bits);
4: c = ~ a;
5: d = a & c;
```
(a)

```
1: Taint (a) = 1;
2: Taint (b) = 1;
3: Taint (w) = 1;
4: Taint (c) = 1;
5: Taint (d) = 1;
```
(b)

```
1: Taint (a) = tag1;
2: Taint (b) = tag2;
3: Taint (w) = (tag1 ∧ low_bits) ∨ (tag2 ∧ high_bits);
4: Taint (c) = ~ tag1;
5: Taint (d) = 0;
```
(c)

Figure 2: Conventional single-tag taint propagation vs. StraightTaint multi-tag symbolic taint propagation: (a) a sequence arithmetic operations; (b) conventional single-tag taint propagation results; (c) StraightTaint multi-tag symbolic taint propagation results.

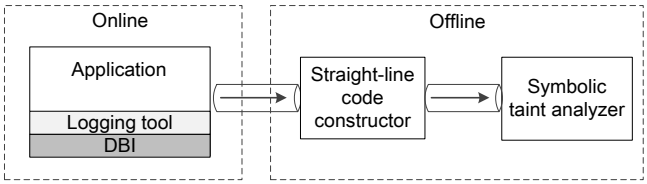


Figure 3: The architecture of StraightTaint.

`d` as tainted, resulting in over-tainting [41]. A nature benefit of StraightTaint's offline taint analysis is that supporting full-featured taint propagation strategies is straightforward, such as multi-tag and bit-level taint analysis. Also, our symbolic taint analysis on the straight-line code can capture intricate details of the x86 instructions. Figure 2 (c) shows the results of StraightTaint multi-tag symbolic taint propagation: `w` and `c` are correctly tainted; the taint tags of `d` are cleaned as expected. StraightTaint avoids the imprecision and over-tainting problems introduced by previous incomplete taint propagation strategies.

2.3 Architecture

Figure 3 illustrates the architecture of StraightTaint, which consists of two stages: online logging and offline analysis. The first stage, as shown in the left part of Figure 3, involves very lightweight online logging to mainly record control flow information. We built a logging tool using dynamic binary instrumentation (DBI), enabling StraightTaint to work with unmodified program binaries directly. The application under examination is executing over the DBI and our logging tool. Our logging tool dynamically instruments each executed basic block to record the execution using tags that are unique for each basic block. The basic block tags are written to a trace buffer and then stored in a disk storage when the buffer is filled up. Careful design of the online logging tool is crucial for achieving better efficiency. Therefore, we propose three guidelines and the details will be discussed in Section 3.

The generated log data is passed to the offline taint analysis (the right component of Figure 3). This stage first reconstructs the straight-line code trace from the log data, and then lifts the x86 instructions to BIL [6], an RISC-like intermediate language. The core of our symbolic taint analyzer is an abstract taint analysis processor. Similar to the shadow memory in DTA, StraightTaint maintains a *context* structure to store symbolic taint variables and concrete values. Our offline taint analyzer is able to carry out both forward taint tracking to detect the effect of an intrusion, and backward tracing to identify attack provenance. Even without complete runtime data information, StraightTaint can achieve comparable precision as dynamic taint analysis, which will be discussed in detail in Section 4.

3. EFFICIENT ONLINE LOGGING

StraightTaint applies a lightweight logging to lower the impact on application performance. Since not all the instructions executed are of interest, we invoke online trace logging when pre-defined taint seeds are first introduced. In StraightTaint, a user can set the input data from keyboard, file, network or function return value as taint seeds. To avoid symbolic taint variables explosion in the offline analysis, we leverage the concrete execution state when the taint seeds are introduced to constrain fresh symbolic taint variables. We collect an execution state by performing a process dump. Beyond that, the executed control flow information is logged to reconstruct the straight-line code later. Nondeterministic variables (e.g., random numbers and time) that may affect control flow are recorded as well.

The logged data are first stored in a memory buffer and then dumped to disk storage when the buffer is filled up. Three design goals guide us to achieving low online logging overhead: 1) the logged data representation should be compact so that trace buffer holds as much data as possible; 2) the application (i.e. producer) should not be blocked when the full buffers are being consumed, that is, processing the buffers asynchronously; 3) instrumentation overhead should be minimized. We meet the first requirement by extending an advanced trace profiling format [53]. To address the second challenge, we propose an n-way fast buffering scheme on multi-cores to parallelize profile consumption. At last, we carefully design our instrumentation code to favor code inlining and avoid frequent context switches. In Section 5.1, we will introduce other Pin specific optimizations we adopted to achieve enhanced performance gains.

3.1 Trace Profiling

Application's straight-line trace can be represented as a sequence of basic blocks executed. A basic block is a straight-line sequence of code with one entry point and one exit. A naive approach is to record each basic block's entry address. On a 32-bit machine, a 4-byte tag is needed to label a basic block. However, a full 4-byte tag is an excessive use and would take up too much space. Zhao et al. [53] proposed an efficient method, *Detailed Execution Profile* (DEP). DEP uses only 2-byte tags to record most basic blocks and handles special cases with extra escape bytes. DEP splits a 4-byte address into 2 high bytes for *H-tag* and 2 low bytes for *L-tag*. During control flow profiling, if two sequential basic blocks share the same H-tag, only L-tag of each basic block is logged into the profile buffer. If the two H-tags are different, an escape tag 0x0000 followed by the new H-tag will be entered into the buffer. Our trace profiling design is based on DEP with a number of optimizations.

Certain x86 string instructions (`MOVS`, `LODS`, `STOS`, `CMPS` and `SCAS`) with `REP`-prefix execute repeatedly. DBI tools [5, 25] usually treat `REP`-prefixed instructions as implicit loops. If

a REP-prefixed instruction iterates more than once, iterations after the first will cause a single instruction basic block to be generated. In such case, we'll see much more basic blocks than we expect. To address this issue, we inspect the first loop of REP-prefixed instructions and configure Pin to disable unrolling following loops. And then we encode REP-prefixed instructions with two consecutive escape values 0xffff, followed by an iteration number.[1]

We justify here why we choose to encode the basic block executed rather than control flow branching decision or Pin trace. First, it is possible to use a single bit to log a basic block by recording the binary decision of conditional jump [37], which leads to a much denser log data. However, encoding 1-bit does not favor Pin code inline, which introduces more instrumentation overhead. Also, recovering straight-line code from 1-bit encoding is time-consuming. Second, the single-entry, multi-exits property of Pin trace makes the trace size cannot be uniquely decided. Third, static program analysis [2, 19] can be used to remove the redundant instrumentation points. However, recall that StraightTaint works in an adversarial environment, in which the accurate static features such as control flow graphs are typically not available. Our design choice enables StraightTaint to analyze obfuscated binaries.

3.2 Multithreaded Fast Buffering Scheme

In this section, we introduce our generic scheme that supports concurrent buffering data on the multi-core platform. We exploit underutilized computing resources to alleviate the disk I/O bottleneck. The center of our design is a buffering thread pool, in which multiple buffers enable the instrumented application to continue executing and filling up free buffers while multiple Pin-tool internal threads process full buffers asynchronously. Figure 4 illustrates how the buffering thread pool works, and the processing steps are as follows.

1) When a program starts running, the application (i.e., *producer*) allocates a number of free buffers (8 buffers in Figure 4).

2) Simultaneously, multiple Pin-tool internal threads are spawned. We call them *worker threads* (8 worker threads in Figure 4). The worker thread takes a buffer from the full-buffer queue and dump buffer data to disk storage. Multiple worker threads access a full buffer exclusively by acquiring the buffer's lock.

3) The application first fills one free buffer. When this buffer becomes full, a callback function, *BufferFull* will be called to perform two tasks: 1) enqueue the full buffer to the global full-buffer queue and wake up one worker thread to process it, 2) return the next available free buffer to the application.

We bias the implementation of our buffering scheme to lower the impact on the application execution. Specifically, we create enough worker threads to ensure a full buffer can be processed immediately by worker threads. Besides, we dynamically adjust the number of buffers allocated and the number worker threads created to optimize the synchronization and load balancing. The availability of unused cores and the size of a profile buffer have a great impact on the runtime performance. In Section 5.2, we will discuss how to tune these two factors.

[1] The maximum REP-prefixed loop count in our evaluation comes from *gcc* benchmark, which is 1770, far less than two-byte number limit.

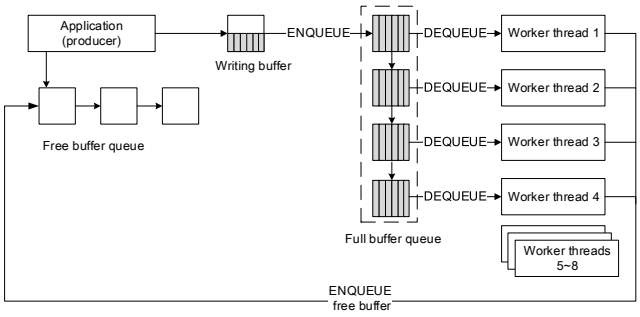


Figure 4: Buffering thread pool.

4. OFFLINE SYMBOLIC TAINT ANALYSIS

4.1 Reconstruction of Straight-line Code

Given the trace collected from the online logging, reconstructing a complete sequence of 4-byte starting addresses of basic blocks is quite straightforward. The beginning of the trace profile should be a special value 0x0000, followed by an H-tag. Each basic block 4-byte entry address is the concatenation of its corresponding H-tag and L-tag. Then the x86 instructions of each basic block are extracted from the application's disassembly code. An elaborate knowledge of the x86 ISA is required to accurately track taint propagation at binary level. However, the cumbersome x86 ISA makes it an extremely tedious work. For example, previous work such as libdft [20] contains more than $5,000$ lines of code to handle the x86 ISA complexity. Figure 2(a) shows such an example involving complicated arithmetic operations. Even worse, some instructions with implicit side effects only propagate taint conditionally according to the contents of EFLAGS (e.g., CMOVcc). To get rid of the intricate details of the x86 ISA, we lift up x86 instructions to BIL [6], an RISC-like intermediate language. BIL leaves us only 25 instructions that we need to analyze carefully for accurate taint tracking. Note that with control flow information, we have resolved all indirect control flow targets and conditional jump directions in the straight-line IL code.

4.2 Symbolic Taint Analysis

By labeling the stream bytes of taint seeds as symbolic variables, StraightTaint offline taint propagation is a kind of symbolic execution on the straight-line code. Since each taint seed byte can be associated with a fresh symbol, multi-tag taint propagation is natural for StraightTaint. The core of our symbolic taint analysis engine (as shown in Figure 5) is an abstract processor, which maintains a *context* structure as the execution state. The context structure consists of a program counter pc, a variable context V and a memory context M. For conciseness, we represent the state of the abstract processor with the tuple $s = (pc, V, M)$. The variable context V contains all symbolic register values (e.g., general purpose registers and bits of EFLAGS) and temporaries. The temporaries are the expressions used in the static single assignment form of BIL. We also explicitly represent the return value of a function as a special variable to facilitate detecting buffer overflow vulnerabilities. The memory context M, with a structure analogous to the two-level architecture of x86 virtual addressing, is a mapping from memory addresses to their symbolic variables. By interpreting the current IL at pc, a state of the abstract processor $s = (pc, V, M)$ is translated into a new state

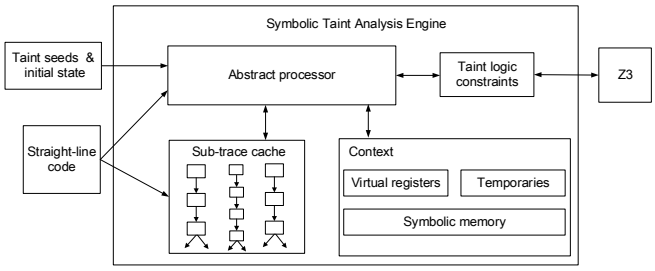


Figure 5: Symbolic taint analysis engine.

```
1: neg   reg
2: sbb   reg, reg
3: and   reg, (val1 - val2)
4: add   reg, val2
```
(a)

```
1: if (reg)  cf = 1;
   else      cf = 0;
2: reg = reg - reg - cf;
3: reg = reg | (val1 - val2);
4: reg = reg + val2;
```
(b)

```
if (reg)  reg = val1;
else      reg = val2;
```
(c)

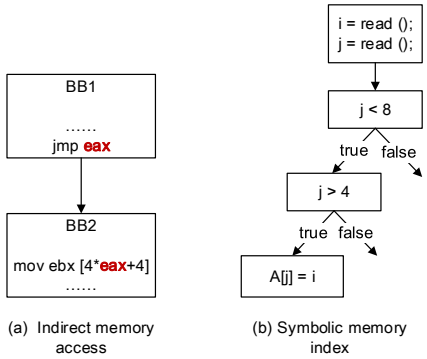
Figure 6: Example: branchless logic code (`reg` stands for register; `val1` and `val2` are two tainted variables).

$s' = (pc', V', M')$ and V' and M' are updated according to the semantic of the IL. At the same time, StraightTaint checks whether a location of interest (i.e., taint sink) is tainted by checking whether its value is a symbolic expression. After the last IL is simulated, pc is set to `halt` and V and M are not updated anymore.

We start offline taint analysis when the pre-defined taint seeds are first introduced to the application. Besides the taint seeds, there could be other uninitialized variables such as the stack pointer and memory contents. In principle, we can assign a fresh symbolic variable to each uninitialized variable. However, symbolic taint analysis with an unconstrained initial state can quickly reach the memory capacity and lead to the problem of "over-tainting" [41] as well. Our solution is to leverage a process dump to assign other uninitialized variables with concrete values, only leaving the taint seeds as symbolic variables. Here we use another common example to show the value of symbolic execution style taint propagation. To reduce the number of conditional jumps, some compiler optimization options translate conditional instructions into a sequence of arithmetic operations. Figure 6 (a) shows such an example we find in our test cases. Figure 6 (b) lists the semantics for each instruction. The net result of the sequence of arithmetic operations is presented in Figure 6 (c), which is actually a branch condition. The taint tag of `reg` is either from `val1` or `val2`. StraightTaint successfully propagate taint tags for this tricky case, while previous tools such as Temu [52], libdft [20], and FlowWalker [15] all fail.

4.3 Memory Reference Address Resolution

Another feature of StraightTaint's offline taint analysis is that we do not record memory reference addresses, which are typically calculated through general registers and constant offsets. Our observation is that, with the initial execution state and the straight-line code, most memory reference addresses can be decided along the symbolic taint analysis. Figure 7 (a) shows how we resolve an indirect memory access. Since we have resolved each indirect jump target in the



(a) Indirect memory access

(b) Symbolic memory index

Figure 7: Example: memory reference address resolution.

straight-line code (See Section 4.1), the memory indirect access through `eax` in `BB2` can be determined. To solve a memory address `address_a` that is cannot be computed accurately (e.g., heap memory allocation), we allocate memory on-the-fly. Inspired by micro execution [16], we use return value of `malloc(1)` as `address_a`, which guarantees that `address_a` would not conflict with an existing address. Then we assign a symbolic variable to represent the content of `address_a`, and subsequent reading at `address_a` will return the same symbolic value. A symbolic index happens when a symbolic variable is used as the index of a memory lookup, such as the conversion function of ASCII to Unicode, to_lower, and to_upper. Intuitively a symbolic memory index could point to any memory slot. We deal with this problem by solving path conditions. As shown in Figure 7 (b), the path conditions along the straight-line code restrict the range of symbolic memory index `j` within $4 < j < 8$. Then we conservatively label all the possible memory values as tainted. For the example in Figure 7 (b), A[5], A[6], and A[7] will be tainted.

4.4 Conditional Tainting

As x86 conditional control transfer instructions typically depend on the value of the `EFLAGS` register (e.g., jz and jo) our virtual registers also keep track of bit-level symbolic variables for `EFLAGS`. When a symbolic expression is used in a conditional jump instruction, we collect it as a branch condition. After a complete symbolic taint propagation run, the conjunction of all branch conditions is the *Taint Logic Constraints*. Thus, the values that satisfy the taint logic constraints are the concrete taint seeds that would lead the program to execute the same taint tracking operations as the one symbolically tainted. With taint logic constraints, which are solved by a theorem prover (e.g., Z3 [29]), previously taint analysis results can be mapped to new inputs and runtime values without DTA again!

4.5 Optimization

Like Pin's block cache to save the overhead of frequently executed basic block retranslation, we take a similar approach to speed up our offline symbolic taint analysis. We call it "sub-trace cache" (see "sub-trace cache" component in Figure 5). We merge sequential basic blocks that have one predecessor and one successor as a sub-trace, which can be viewed as an extended basic block. We represent the input-output relations of a sub-trace as a set of symbolic formulas and maintain a lookup table in the sub-trace unit. Therefore, the successive runs can directly reuse previous results, without the need for recomputing them. Another

primary optimization we adopt is function summary. Most well-known library functions have explicit semantics (e.g., C strings manipulation functions defined in string.h), and many of them even do not affect taint propagation (e.g., strcmp). Therefore, we turn off symbolic taint analysis at the boundary of these functions and update context according to their semantics summaries. For a sequence of adjacent memory access introduced by `REP`-prefixed instructions, we recover the number of repetitions from trace profile and perform batch processing instead of byte by byte operations.

5. IMPLEMENTATION AND EVALUATION

5.1 Implementation

To demonstrate the idea of StraightTaint, we implemented a prototype including online logging based on the Pin DBI framework [25] (version 2.12) with 2,660 lines of code in C/C++, and offline symbolic taint analysis engine on top of BAP [6] (version 0.8) with 4,540 lines of OCaml code. We rely on BAP to convert assembly instructions to IL and convert IL expressions to CVC formulas. We use Z3 [29] as our constraint solver. The saving and loading of sub-trace cache lookup table are implemented using the OCaml Marshal API, which encodes arbitrary data structures as sequences of bytes and then store them in a disk file.

When implementing the Pin-tool, we create thread-local storage (TLS) slot to store and retrieve per-thread buffer structure. Note that Pin-tools are unable to work with either pthreads library or Win32 threading API. We utilize the Pin thread API to spawn worker threads and implement a counting semaphore using Pin's own binary semaphore. To make the best use of Pin's code cache effect, we enlarge the maximum number of basic blocks per Pin trace from 3 to 8. We also use GCC's built-in macro "`__builtin_expect()`" to provide the compiler with the branch prediction. Furthermore, we perform low-overhead buffering of data through Pin's fast buffering APIs, which support inlining a callback function when a buffer becomes full. We also force Pin to use fastcall calling convention to pass arguments via registers to avoid emitting stack access instructions (i.e., `push` and `pop`). StraightTaint's efficient multithreaded control flow profiling Pin-tool is available at https://github.com/s3team/bincfp.

Our testbed contains two machines. One is a server machine, which is equipped with two Intel Xeon E5-2690 processors (16-core with 2.9GHz) and 128GB of RAM. Another is a desktop, consisting of Intel Core i7-3770 processor (Quad Core with 3.40GHz) and 8GB memory. Both are running Ubuntu 12.04. The data presented throughout this section are all mean values. We calculate them by running five repetitions of each experiment case.

5.2 Buffer Size and Worker Threads

We studied two factors that may affect StraightTaint online logging performance: 1) the buffer size of control flow profile; 2) the number of available worker threads. We first survey the impact of various buffer size. In order to achieve enough parallelism, the number of worker threads is set to 16 and 4, respectively. The total buffer sizes are therefore the number of worker threads × single buffer size. We choose SPEC CPU2006 with *test* workload as the training set. As shown in Figure 8, roughly the overhead decreases as the buffer size is increased. This is mainly due to the reduction of free/full buffer switches, and worker threads spend less time on synchronization. As the buffer size is beyond a certain

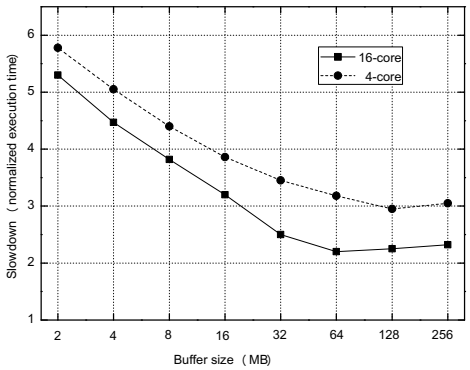


Figure 8: Normalized slowdown on 16-core and 4-core systems when profile buffer size varies.

point (64MB for the 16-core system and 128MB for the 4-core system), the slowdown is increased a little. We attribute this to the large total buffer sizes (e.g., 16 × 256MB) interfering with the application's working set. Then we fix the buffer size to 64MB for the 16-core system and 128MB for the 4-core system and alter the number of worker threads. In general, the performance is better as more worker threads are added. Due to the maximum parallelism and the tuned buffer size, 16 worker threads with 64MB buffer size achieve the optimum result. We set these two parameters as default configuration and conduct the following experiments on the 16-core system.

5.3 StraightTaint vs. libdft

We first compare StraightTaint with libdft [20], a state-of-the-art inlined DTA tool built on Pin ("libdft" bar). In aid of evaluating the application performance slowdown imposed exclusively by StraightTaint, we develop a simple tool (nullpin) to measure Pin's environment runtime overhead, which runs a program under Pin without any form of analysis ("nullpin" bar). We also measure the logging overhead without buffering the profile data to disk ("online-no I/O" bar). Under this configuration, the application never stalls to wait for free profile buffers, which can represent the upper bound of performance improvement attainable by StraightTaint. Viewed from a different angle, "online-no I/O" bar also indicates the overhead introduced by Pin's instrumentation. All runtime data[2] presented in this section are normalized to application native execution time (without running Pin).

Figure 9 shows the normalized overhead of running SPEC CPU2006 int benchmark suite with *reference* workload. Since the reference workload is CPU-intensive, we expect that these results can estimate the worst case scenarios. On average, StraightTaint's online logging exhibits a 3.06X slowdown to native execution, while libdft lags behind as much as 9.96X, indicating that StraightTaint speeds up application execution by a factor of 3.25. It is noteworthy that if taking *nullpin* as the baseline, the slowdown exclusively introduced by StraightTaint is only 1.97X while for libdft is 6.43X. This number is in line with the observations by the previous work [12, 39]; that is, performing one taint propagation operation normally needs six extra instructions. The overhead incurred by StraightTaint's online instrumentation is 2.16X ("online-no I/O" bar), compared to Pin's environment runtime overhead

[2]The "online" bar is calculated by counting wall-clock time because we have to consider the I/O time introduced by our buffering scheme. Other bars are calculated by counting CPU time.

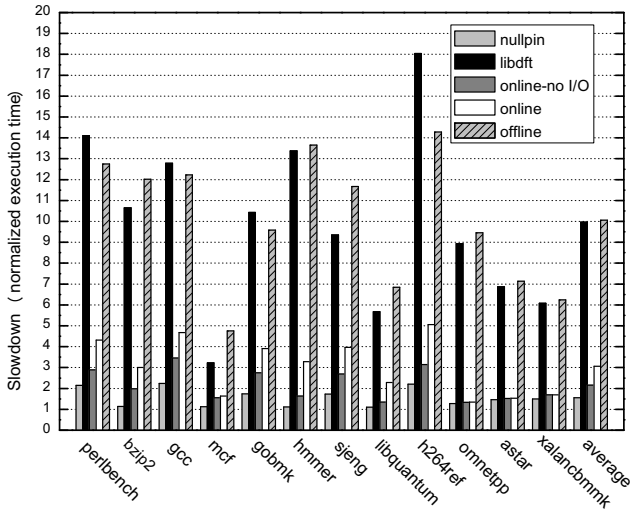


Figure 9: StraightTaint vs. libdft: slowdown on SPEC CPU2006.

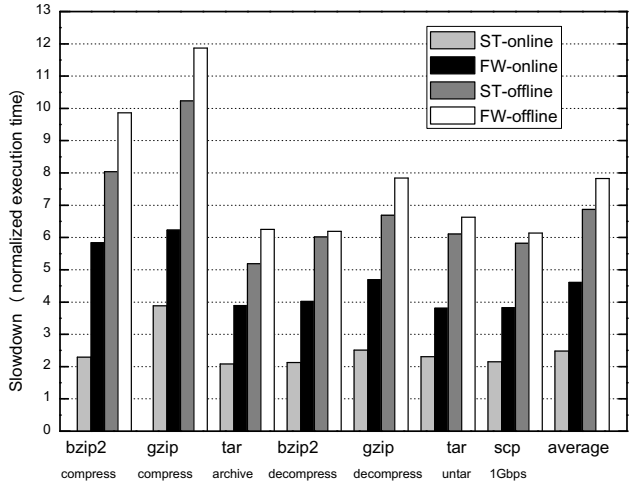


Figure 10: StraightTaint vs. FlowWalker: slowdown on common Linux utilities.

(1.55X), 39.4% extra performance penalty added. Due to the CPU bounded test suite, StraightTaint has to put more efforts to deal with large amount of I/O. Therefore, additional 41.7% overhead to "online-no I/O" version is introduced.

On average, StraightTaint generates about 2.8GB of raw trace profiling data for SPEC2006's reference workload. Compared to the raw 4-byte tag profile size, the relative size of StraightTaint is only 49.2%. In general, StraightTaint outperforms DEP's encoding [53] by 5 percentages in terms of smaller profiling data size. It is worth mentioning that we see a significant size reduction for the `h264ref` benchmark, from DEP's 4.8GB to 2.1GB. The reason is `h264ref` intensively utilizes `REP`-prefixed instructions, which are very well handled by StraightTaint's optimization.

The last bar for each application in Figure 9 presents the performance of symbolic taint analysis, which is normalized to native execution as well. Since we have decoupled taint tracking from program execution, offline symbolic taint analysis avoids the overhead introduced by DBI's environment and computing resource competitions. On the other hand, symbolic taint analysis engine is in fact an interpreter for each IL, which is much slower than native execution. To alleviate this issue, we have applied a number of optimization methods (discussed in Section 4.5). The net result is that our offline symbolic taint analysis takes approximately the same amount of time as libdft (10.06X for StraightTaint and 9.96X for libdft on average). In several cases (e.g., perlbench and h264ref), StraightTaint's offline part outperforms libdft. Considering that StraightTaint is aiming to shift dynamic taint analysis cost to the offline analysis phase, this degree of slowdown is tolerable. In Section 8, we will discuss several possible ways to further accelerate offline taint analysis.

5.4 StraightTaint vs. FlowWalker

FlowWalker [15] is perhaps the closest work to StraightTaint in its goals: we are both offline taint analysis in *record and replay* style. Similar to StraightTaint, FlowWalker also records limited CPU context on top of Pin to calculate the memory address offline. However, FlowWalker lacks fine-grained optimizations in both online logging and offline taint analysis (see Section 7). In this experiment, we evaluate StraightTaint (short for ST) and FlowWalker (short for FW) on four common Linux utilities that represent three kinds of workload.[3] The program `tar` is I/O bounded, whereas `bzip2` and `gzip` are CPU intensive program, and `scp` represents a middle level between these two cases. We use `tar` to archive and extract GNU Core utilities 8.13 package (∼50MB). And then we apply `bzip2` and `gzip` to compress and decompress the archive file of Core utilities. For `scp`, we copy the archive file of Core utilities over 1Gbps link. We achieve a similar improvement with the SPEC CPU2006 experiment. As shown in Figure 10, StraightTaint imposes a average 2.48X slowdown to native execution, with a 1.86 times speed up to FlowWalker. Besides, StraightTaint's offline taint analysis is faster than FlowWalker with a factor of 1.14. We attribute this to our sub-trace cache and function summary optimizations.

5.5 Offline Symbolic Taint Analysis

Next we evaluate the accuracy of our offline symbolic taint analysis in the task of software attack detection. To this end, we test ten recent software vulnerabilities using a set of exploits listed in Table 1. These test cases are chosen from CVE vulnerability data source[4] with two criteria: 1) It is easy to mark the locations of taint sinks in the binary code so that we can count the tainted bytes at the same place; 2) we have exploits that can trigger these vulnerabilities (not all the CVE vulnerabilities have related exploits). All of these applications are compiled with the option "gcc -O2". Taking these exploits as inputs, we apply StraightTaint on each application and check taint tags at various taint sinks (e.g., function return value). In all cases, StraightTaint successfully detects the attacks without false negatives. At the same time, we count the number of tainted (or symbolic) bytes at the end of taint analysis. We compared StraightTaint with *Log-all* and *Pure SE*. Log-all means recording complete runtime data (e.g., each memory address and control transfer target) during online logging, and then use the data for offline taint analysis. Log-all represents vanilla decoupled offline taint analysis, but its result is accurate. Pure SE does symbolic taint analysis but without concrete execution state initialization (see Section 4.2) and memory reference address resolution (see Section 4.3). As shown in Table 1, the taint

[3]SPEC2006's reference workload is too huge for FlowWalker to work out the result in reasonable time.
[4]http://www.cvedetails.com/

Table 1: StraightTaint successfully detects various intrusions with the listed exploits.

Program	Vulnerability	CVE ID	# Taint (Symbolic) Bytes		
			Log-all	StraightTaint	Pure SE
nginx	validation bypass	CVE-2013-4547	45	45	1,035
mini_httpd	validation bypass	CVE-2009-4490	66	66	2,706
libpng	denial of service	CVE-2014-0333	72	80	2,256
gzip	integer underflow	CVE-2010-0001	94	94	6,490
tiny server	validation bypass	CVE-2012-1783	125	131	12,171
coreutils	buffer overflow	CVE-2013-0221	252	272	–
libtiff	buffer overflow	CVE-2013-4231	268	280	–
waveSurfer	buffer overflow	CVE-2012-6303	384	384	–
grep	integer overflow	CVE-2012-5667	608	644	–
regcomp	validation bypass	CVE-2010-4052	1,124	1,186	–

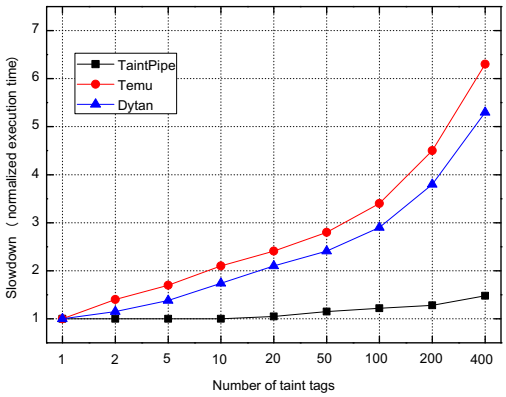


Figure 11: Normalized slowdown when the number of taint tags varies.

bytes added by StraightTaint is quite close to the Log-all. StraightTaint introduces additional taint bytes to 6 cases, but no one is beyond 5%. Most likely, our conservative approach to dealing with symbolic memory indices results in the small additional taint bytes. In contrast, symbolic taint analysis with a completely unconstrained initial state (pure SE) incurs taint variable explosion. Pure SE fails in the last 5 test cases due to quickly reaching the memory capacity. Note that we also identify 14 code segments which can fail DTA tools with incomplete taint propagation strategies. One such example has been shown in Figure 6. In contrast, StraightTaint's full-featured offline taint analysis succeeds in all cases.

At last, we show that StraightTaint can support multi-tag taint analysis naturally. We test a lightweight web server, `thttpd`,[5] with a 400-byte size HTTP request as input. The X-axis numbers in Figure 11 represent different taint tags we assigned: 1 taint tag indicates the whole 400 bytes are labeled as a single taint tag; 2 taint tags means that the first 200 bytes are labeled as one taint tag and the next 200 bytes are labeled as another one; 400 taint tags means each input byte is associated with a different taint tag. Following the similar style, we vary the number of taint tags in each round. At the same time, we compare two DTA tools (Temu [52] and Dytan [14]) which also support multi-tag taint analysis. The baseline for each tool is their single-tag version. As shown in Figure 11, it is apparent that as the number of taint tags increases, both Temu and Dytan imposes high additional overhead; while StraightTaint only introduce 1.48X showdown in the worst case. Please note that this evaluation demonstrates StraightTaint's another notable feature: *once a log is captured, it can be analyzed multiple times.* In our multiple round testing, StraightTaint only needs to record the required data once and performs the different multi-tag propagation rounds on top on the straight-line code. By contrast, both Temu and Dytan have to rerun at each round.

6. CASE STUDY: ATTACK PROVENANCE ANALYSIS

Because of the offline analysis property, StraightTaint is an ideal fit for *ex post facto* security applications. In this section, we demonstrate the merit of StraightTaint with a case study of attack provenance investigation. The goal is to reveal the provenance of intrusions or suspicious events (e.g., information leaks). The previous work [22, 24] did this by generating causal graph linking root causes and suspicious events. Certainly DTA can be utilized to precisely generate causal dependence between taint source and taint sink. We show that StraightTaint is able to get a similar level of precision as DTA with multi-tag backward propagation. The test case is `wget`,[6] an open source tool for retrieving files from web. We execute `wget` with the command "`wget www.google.com www.bing.com`".

As shown in Figure 12, `wget` receives two URLs as command line arguments and then downloads their respective `index.html` files (`index1.html` is from `www.google.com` and `index2.html` is from `www.bing.com`). Supposing we have already got these two downloaded files, an interesting question is "which exact URL are they derived from?" "google, bing or both?" Apparently DTA can precisely identify such mappings by forward taint tracking with multiple tags. Please note the pseudo-code of Figure 12: two files are generated subsequently when the loop is unrolling. As a result, static taint analysis, without runtime information, fails to identify causal relations between sources and sinks.

We take the input buffer of `fwrite`, which is used to generate HTML file, as symbolic taint sinks. Then we apply StraightTaint for backward tainting along the straight-line trace. Of course without runtime values and inputs, StraightTaint is unable to exactly correlate the concrete URL to its corresponding file. However, compared to pure static approaches, StraightTaint catches *conditional* causal relationships between two sinks and sources: the first downloaded file is derived from the first URL input and the second one is related to the second URL. Another benefit of StraightTaint's *conditional tainting* is that we are possible to directly map previous taint analysis results to new inputs and runtime values. For example, supposing new command for *wget* is "`wget www.bing.com www.google.com`", with the previous conditional causal relationship, we can get the exact mappings immediately without running DTA again.

[5] http://acme.com/software/thttpd/

[6] http://www.gnu.org/software/wget

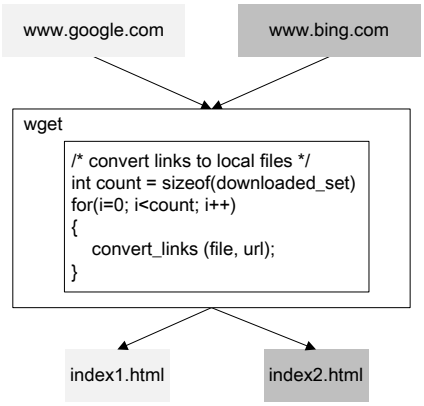


Figure 12: Causal relationship between two sinks and two sources.

7. RELATED WORK

Decoupling Dynamic Taint Analysis. To address the performance bottleneck of dynamic taint analysis (DTA), two major approaches have been proposed to decouple taint analysis from program execution. The first category parallelizes dynamic taint analysis by delivering the needed runtime values to another core, in which the taint analysis is running [18, 19, 31, 40, 27]. DECAF [18] extends Temu [52] to support asynchronous heavyweight taint propagation. However, DECAF does not show the performance gains introduced by its asynchronous tainting. TaintPipe [27] parallelizes DTA in a pipeline style. Because of the strict synchronization requirement, some tools in the first category adopt incomplete taint propagation strategies to catch up the application execution. The second direction, like StraightTaint, first records the application execution and then replay the taint analysis on a different CPU [15, 42, 45, 48]. The most related work to StraightTaint is FlowWalker [15], which also uses Pin to record CPU context, and then performs a multi-tag assembly level taint propagation offline. However, StraightTaint reveals two distinct advantages. First, we design a more compact profile structure and multithreaded fast buffering scheme to parallelize the runtime data logging. Second, our offline taint analysis is performed on a side-effect free intermediate language instead of cumbersome x86 instructions. As demonstrated in our evaluation, StraightTaint outperforms FlowWalker with better performance and accuracy.

As we have pointed out, due to the large amount data in exchange, the two approaches mentioned above may not achieve the expected performance improvements. Recently, ShadowReplica [19] alleviates such communication overhead by performing an advanced static analysis to remove redundant taint logic code. As a result, it achieves a decent performance in the evaluation. Our work differs from ShadowReplica in that StraightTaint does not depend on fine-grained static analysis of binary code. Therefore, StraightTaint can be applied to reverse engineering tasks such as malware analysis [7, 51] and code deobfuscation [49].

Dynamic Symbolic Execution. Another related area to StraightTaint's offline taint analysis is dynamic symbolic execution, namely *concolic testing* [4, 10, 9, 17], a method of combining concrete execution with symbolic execution. StraightTaint is similar to the concolic testing in that we map symbols to taint seeds and then perform the symbolic taint analysis along a recorded execution trace. Also, StraightTaint can benefit from symbolic execution optimization work to speed up taint analysis, such as memoized symbolic execution [50]. However, we have different goals. Dynamic symbolic execution is mainly for automatic input generation to explore more paths while our primary interest lies in accurate taint analysis on the straight-line code. In addition, concolic testing relies on complete runtime information while StraightTaint only depends on limited runtime information. The recent work, Hercules [34], also mentions the idea of using symbolic execution for precise taint tracking. However, StraightTaint has a strikingly different purpose with Hercules. Hercules is for reproducing crashes in benign application binaries; while StraightTaint is designed to speed up reverse engineering tasks on binary code.

8. DISCUSSIONS AND CONCLUSION

StraightTaint is a prototype to demonstrate that completely decoupling dynamic taint analysis is feasible. The performance of online logging and offline taint analysis can be further improved. Currently, the upper bound of online logging performance that we can achieve is restricted by Pin's environment runtime overhead. One of our future work is to leverage the advanced binary reassembling development toolkits such as Uroboros [43] so that we can insert the taint tracking code directly into the disassembled code and then compile it to the binary code again. In this way, we can remove DBI's environment overhead. StraightTaint's offline taint analysis is as fast as, but not faster than, DTA on average, since in StraightTaint the semantics of taint operations are simulated. One future work to speed up offline taint analysis is to construct a recompilable straight-line program from execution trace. As a result, we can apply another round of DTA directly on the straight-line program. Currently, StraightTaint works on sequential programs. To support taint analysis for multi-threaded programs, we have to carefully handle the complicated inter-thread taint propagation, such as concurrent accesses to shared locations and corresponding taint tag updates. We plan to explore these directions in future.

We have presented StraightTaint, a novel technique for completely decoupling dynamic taint analysis for offline symbolic taint analysis. Unlike previous approaches, StraightTaint does not rely on complete runtime values or inputs, which enables very lightweight logging and much lower online execution slowdown. StraightTaint can also support full-featured, multi-tag, and bit-level taint analysis with low extra overhead. We have evaluated StraightTaint on a set of applications such as utility programs, SPEC2006, and real-life software vulnerabilities. The results show that StraightTaint can rival dynamic taint analysis at a similar level of precision, but with a much lower online execution slowdown and more flexible functionalities. The experimental evidence indicates that StraightTaint can be applied to speed up various *ex post facto* security applications with full-featured offline taint analysis.

9. ACKNOWLEDGMENTS

We thank the ASE 2016 anonymous reviewers for their valuable feedback. This research was supported in part by the National Science Foundation (NSF) grants CNS-1223710 and CCF-1320605, and the Office of Naval Research (ONR) grants N00014-13-1-0175, N00014-16-1-2265, and N00014-16-1-2912. Liu was also supported by ARO W911NF-13-1-0421 and NSF CNS-1422594.

10. REFERENCES

[1] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*, 2014.

[2] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(4), 1994.

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

[4] E. Bounimova, P. Godefroid, and D. Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *Proceedings of the International Conference on Software Engineering (ICSE'13)*, 2013.

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

[6] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. BAP: A binary analysis platform. In *Proceedings of the 23rd international conference on computer aided verification (CAV'11)*, 2011.

[7] J. Caballero, P. Poosankam, S. McCamant, D. Babi ć, and D. Song. Input generation via decomposition and re-stitching: Finding bugs in malware. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*, 2010.

[8] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07)*, 2007.

[9] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, 2008.

[10] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: Automatically generating inputs of death. In *Proc. of the ACM Conference on Computer and Communications Security (CCS'06)*, 2006.

[11] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos. Flexible hardware acceleration for instruction-grain program monitoring. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA'08)*, 2008.

[12] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige. TaintTrace: Efficient flow tracing with dynamic binary rewriting. In *Proceedings of the 11th IEEE Symposium on Computers and Communications (ISCC'06)*, 2006.

[13] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *Proceedings of the USENIX Annual Technical Technical Conference (ATC'08)*, 2008.

[14] J. Clause, W. P. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'07)*, 2007.

[15] B. Cui, F. Wang, T. Guo, and G. Dong. A practical off-line taint analysis framework and its application in reverse engineering of file format. *Computers & Security*, 51(C), June 2015.

[16] P. Godefroid. Micro execution. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*, 2014.

[17] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, 2008.

[18] A. Henderson, A. Prakash, L. K. Yan, X. Hu, X. Wang, R. Zhou, and H. Yin. Make it work, make it right, make it fast: Building a platform-neutral whole-system dynamic binary analysis platform. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA'14)*, 2014.

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

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

[21] G. A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'73)*, 1973.

[22] S. T. King and P. M. Chen. Backtracking intrusions. In *Proceedings of the 9th ACM symposium on Operating systems principles (SOSP'03)*, 2003.

[23] S. Krishnan, K. Z. Snow, and F. Monrose. Trail of Bytes: Efficient support for forensic analysis. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*, 2010.

[24] K. H. Lee, X. Zhang, and D. Xu. High accuracy attack provenance via binary-based execution partition. In *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS'13)*, 2013.

[25] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI'05)*, 2005.

[26] J. Ming, M. Pan, and D. Gao. iBinHunt: Binary hunting with inter-procedural control flow. In *Proceedings of the 15th Annual International Conference on Information Security and Cryptology (ICISC'12)*, 2012.

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

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

[29] L. D. Moura and N. Bjørner. Z3: an efficient smt solver. In *Proceedings of the 14th International Conference on*

Tools and Algorithms for the Construction and Analysis of Systems, 2008.

[30] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'05)*, 2005.

[31] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. Parallelizing security checks on commodity hardware. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08)*, 2008.

[32] V. Pappas, V. P. Kemerlis, A. Zavou, M. Polychronakis, and A. D. Keromytis. CloudFence: Data flow tracking as a cloud service. In *Proceedings of the 16th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'13)*, 2013.

[33] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. PinPlay: A framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'10)*, 2010.

[34] V.-T. Pham, W. B. Ng, K. Rubinov, and A. Roychoudhury. Hercules: Reproducing crashes in real-world application binaries. In *Proceedings of the 37th International Conference on Software Engineering (ICSE'15)*, 2015.

[35] F. Qin, C. Wang, Z. Li, H. seop 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'06)*, 2006.

[36] S. Rawat, L. Mounier, and M.-L. Potet. Static taint-analysis on binary executables. http://stator.imag.fr/w/images/2/21/Laurent_Mounier_2013-01-28.pdf, 2011.

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

[38] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'95)*, 1995.

[39] O. Ruwase, S. Chen, P. B. Gibbons, and T. C. Mowry. Decoupled lifeguards: Enabling path optimizations for dynamic correctness checking tools. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'10)*, 2010.

[40] O. Ruwase, P. B. Gibbons, T. C. Mowry, V. Ramachandran, S. Chen, M. Kozuch, and M. Ryan. Parallelizing dynamic information flow tracking lifeguards. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'08)*, 2008.

[41] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2010.

[42] C.-W. Wand and S. W. Shieh. SWIFT: Decoupled system-wide information flow tracking and its optimizations. *Journal of Information Science and Engineering*, 31(4), 2015.

[43] S. Wang, P. Wang, and D. Wu. Reassembleable disassembling. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security'15)*, 2015.

[44] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu. STILL: Exploit code detection via static taint and initialization analyses. In *Proceedings of the 24th Annual Computer Security Applications Conference (ACSAC'08)*, 2008.

[45] R. Whelan, T. Leek, and D. Kaeli. Architecture-independent dynamic information flow tracking. In *Proceedings of the 22nd International Conference on Compiler Construction (CC'13)*, pages 144–163, 2013.

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

[47] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weissman. ReTrace: Collecting execution trace with virtual machine deterministic replay. In *Proceedings of the Workshop on Modeling, Benchmarking and Simulation*, 2007.

[48] B. Yadegari and S. Debray. Bit-level taint analysis. In *Proceedings of the 14th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'14)*, 2014.

[49] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray. A generic approach to automatic deobfuscation of executable code. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, 2015.

[50] G. Yang, C. S. Păsăreanu, and S. Khurshid. Memoized symbolic execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA'12)*, 2012.

[51] H. Yin, D. S. amd M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *ACM Conference on Computer and Communications Security (CCS'07)*, 2007.

[52] H. Yin and D. Song. TEMU: Binary code analysis via whole-system layered annotative execution. Technical Report UCB/EECS-2010-3, EECS Department, University of California, Berkeley, Jan 2010.

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

[54] D. Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. TaintEraser: protecting sensitive data leaks using application-level taint tracking. *ACM SIGOPS Operating Systems Review*, 45:142–154, January 2011.