TaintPipe: Pipelined Symbolic Taint Analysis

Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu,
The Pennsylvania State University

https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/ming

This paper is included in the Proceedings of the
24th USENIX Security Symposium

August 12–14, 2015 • Washington, D.C.

ISBN 978-1-939133-11-3

TaintPipe: Pipelined Symbolic Taint Analysis

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

Abstract

Taint analysis has a wide variety of compelling applications in security tasks, from software attack detection to data lifetime analysis. Static taint analysis propagates taint values following all possible paths with no need for concrete execution, but is generally less accurate than dynamic analysis. Unfortunately, the high performance penalty incurred by dynamic taint analyses makes its deployment impractical in production systems. To ameliorate this performance bottleneck, recent research efforts aim to decouple data flow tracking logic from program execution. We continue this line of research in this paper and propose *pipelined symbolic taint analysis*, a novel technique for parallelizing and pipelining taint analysis to take advantage of ubiquitous multi-core platforms. We have developed a prototype system called TaintPipe. TaintPipe performs very lightweight runtime logging to produce compact control flow profiles, and spawns multiple threads as different stages of a pipeline to carry out symbolic taint analysis in parallel. Our experiments show that TaintPipe imposes low overhead on application runtime performance and accelerates taint analysis significantly. Compared to a state-of-the-art inlined dynamic data flow tracking tool, TaintPipe achieves 2.38 times speedup for taint analysis on SPEC 2006 and 2.43 times for a set of common utilities, respectively. In addition, we demonstrate the strength of TaintPipe such as natural support of multi-tag taint analysis with several security applications.

1 Introduction

Taint analysis is a kind of program analysis that tracks some selected data of interest (taint seeds), e.g., data originated from untrusted sources, propagates them along program execution paths according to a customized policy (taint propagation policy), and then checks the taint status at certain critical location (taint sinks). It has been shown to be effective in dealing with a wide range of security problems, including software attack prevention [25, 40], information flow control [45, 34], data leak detection [49], and malware analysis [43], to name a few.

Static taint analysis [1, 36, 28] (STA) is performed prior to execution and therefore it has no impact on runtime performance. STA has the advantage of considering multiple execution paths, but at the cost of potential imprecision. For example, STA may result in either under-tainting or over-tainting [32] when merging results at control flow confluence points. Dynamic taint analysis (DTA) [25, 13, 27], in contrast, propagates taint as a program executes, which is more accurate than static taint analysis since it only considers the actual path taken at run time. However, the high runtime overhead imposed by dynamic taint propagation has severely limited its adoption in production systems. The slowdown incurred by conventional dynamic taint analysis tools [25, 13] can easily go beyond 30X times. Even with the state-of-the-art DTA tool based on Pin [20], typically it still introduces more than 6X slowdown.

The crux of the performance penalty comes from the strict coupling of program execution and data flow tracking logic. The original program instructions mingle with the taint tracking instructions, and usually it takes 6–8 extra instructions to propagate a taint tag in shadow memory [11]. In addition, the frequent "context switches" between the original program execution and its corresponding taint propagation lead to register spilling and data cache pollution, which add further pressure to runtime performance. The proliferation of multi-core systems has inspired researchers to decouple taint tracking logic onto spare cores in order to improve performance [24, 31, 26, 15, 17, 9]. Previous work can be classified into two categories. The first category is hardware-assisted approaches. For example, Speck [26] needs OS level support for speculative execution and rollback. Ruwase et al. [31] employ a customized hardware for logging a program trace and delivering it to other idle cores for inspection. Nagarajan et al. [24] utilize a hardware first-in first-out buffer to speed up communication between cores. Although they can achieve an appealing performance, the requirement of special hardware prevents them from being adopted using commodity hardware.

The second category is software-only methods that work with binary executables on commodity multi-core hardware [15, 17, 9]. These software-only solutions rely on dynamic binary instrumentation (DBI) to decouple dynamic taint analysis from program execution. The program execution and parallelized taint analysis have to be properly synchronized to transfer the runtime values that are necessary for taint analysis. Although these approaches look promising, they fail to achieve expected performance gains due to the large amounts of communication data and frequent synchronizations between the original program execution thread (or process) and its corresponding taint analysis thread (or process). Recent work ShadowReplica [17] creates a secondary shadow thread from primary application thread to run DTA in parallel. ShadowReplica conducts an offline optimization to generate optimized DTA logic code, which reduces the amount of information that needs to be communicated, and thus dramatically improves the performance. However, as we will show later, the performance improvement achieved by this "primary & secondary" thread model is fixed and cannot be improved further when more cores are available. Furthermore, in many security related tasks (e.g., binary de-obfuscation and malware analysis), precise static analysis for the offline optimization needed by ShadowReplica may not be feasible.

In this paper, we exploit another style of parallelism, namely pipelining. We propose a novel technique, called TaintPipe, for parallel data flow tracking using *pipelined symbolic taint analysis*. In principle, TaintPipe falls within the second category of taint decoupling work classified above. Essentially, in TaintPipe, threads form multiple pipeline stages, working in parallel. The execution thread of an instrumented application acts as the source of pipeline, which records information needed for taint pipelining, including the control flow data and the concrete execution states when the taint seeds are first introduced. To further reduce the online logging overhead, we adopt a compact profile format and an N-way buffering thread pool. The application thread continues executing and filling in free buffers, while multiple worker threads consume full buffers asynchronously. When each logged data buffer becomes full, an inlined call-back function will be invoked to initialize a taint analysis engine, which conducts taint analysis on a segment of straight-line code concurrently with other worker threads. Symbolic memory access addresses are determined by resolving indirect control transfer targets and approximating the ranges of the symbolic memory indices.

To overcome the challenge of propagating taint tags in a segment without knowing the incoming taint state, TaintPipe performs *segmented symbolic taint analysis*. That is, the taint analysis engine assigned to each segment calculates taint states symbolically. When a concrete taint state arrives, TaintPipe then updates the related taint states by replacing the relevant symbolic taint tags with their correct values. We call this *symbolic taint state resolution*. According to the segment order, TaintPipe sequentially computes the final taint state for every segment, communicates to the next segment, and performs the actual taint checks. Optimizations such as function summary and taint basic block cache offer enhanced performance improvements. Moreover, different from previous DTA tools, supporting bit-level and multi-tag taint analysis are straightforward for TaintPipe. TaintPipe does not require redesign of the structure of shadow memory; instead, each taint tag can be naturally represented as a symbolic variable and propagated with negligible additional overhead.

We have developed a prototype of TaintPipe, a pipelined taint analysis tool that decouples program execution and taint logic, and parallelizes taint analysis on straight-line code segments. Our implementation is built on top of Pin [23], for the pipelining framework, and BAP [5], for symbolic taint analysis. We have evaluated TaintPipe with a variety of applications such as the SPEC CINT2006 benchmarks, a set of common utilities, a list of recent real-life software vulnerabilities, malware, and cryptography functions. The experiments show that TaintPipe imposes low overhead on application runtime performance. Compared with a state-of-the-art inlined dynamic taint analysis tool, TaintPipe achieves overall 2.38 times speedup on SPEC CINT2006, and 2.43 times on a set of common utility programs, respectively. The efficacy experiments indicate that TaintPipe is effective in detecting a wide range of real-life software vulnerabilities, analyzing malicious programs, and speeding up cryptography function detection with multi-tag propagation. Such experimental evidence demonstrates that TaintPipe has potential to be employed by various applications in production systems. The contributions of this paper are summarized as follows:

- We propose a novel approach, TaintPipe, to efficiently decouple conventional inlined dynamic taint analysis by pipelining symbolic taint analysis on segments of straight-line code.
- Unlike previous taint decoupling work, which suffers from frequent communication and synchronization, we demonstrate that with very lightweight runtime value logging, TaintPipe rivals conventional inlined dynamic taint analysis in precision.

- Our approach does not require any specific hardware support or offline preprocessing, so TaintPipe is able to work on commodity hardware instantly.
- TaintPipe is naturally a multi-tag taint analysis method. We demonstrate this capability by detecting cryptography functions in binary with little additional overhead.

The remainder of the paper is organized as follows. Section 2 provides background information and an overview of our approach. Section 3 and Section 4 describe the details of the system design, online logging, and pipelined segmented symbolic taint analysis. We present the evaluation and application of our approach in Section 5. We discuss a few limitations in Section 6. We then present related work in Section 7 and conclude our paper in Section 8.

2 Background

In this section, we discuss the background and context information of the problem that TaintPipe seeks to solve. We start by comparing TaintPipe with the conventional inlined taint analysis approaches, and we then present the differences between the previous “primary & secondary” taint decoupling model and the pipelined decoupling style in TaintPipe.

2.1 Inlined Analysis vs. TaintPipe

Figure 1 (“Inlined DTA”) illustrates a typical dynamic taint analysis mechanism based on dynamic binary instrumentation (DBI), in which the original program code and taint tracking logic code are tightly coupled. Especially, when dynamic taint analysis runs on the same core, they compete for the CPU cycles, registers, and cache space, leading to significant performance slowdown. For example, “context switch” happens frequently between the original program instructions and taint tracking instructions due to the starvation of CPU registers. This means there will be a couple of instructions, mostly inserted per program instruction, to save and restore those register values to and from memory. At the same time, taint tracking instructions themselves (e.g., shadow memory mapping) are already complicated enough. One taint shadow memory lookup operation normally needs 6–8 extra instructions [11].

Our approach, analogous to the hardware pipelining, decouples taint logic code to multiple spare cores. Figure 1 (“TaintPipe”) depicts TaintPipe’s framework, which consists of two concurrently running parts: 1) the instrumented application thread performing lightweight online logging and acting as the source of the pipeline; 2) multiple worker threads as different stages of the pipeline to perform symbolic taint analysis. Each horizontal bar with gray color indicates a working thread. We start online logging when the predefined taint seeds are introduced to the application. The collected profile is passed to a worker thread. Each worker thread constructs a straight-line code segment and then performs taint analysis in parallel. In principle, fully parallelizing dynamic taint analysis is challenging because there are strong serial data dependencies between the taint logic code and application code [31]. To address this problem, we propose *segmented symbolic taint analysis* inside each worker thread whenever the explicit taint information is not available, in which the taint state is symbolically calculated. The symbolic taint state will be updated later when the concrete data arrive. In addition to the control flow profile, the explicit execution state when the taint seeds are introduced is recorded as well. The purpose is to reduce the number of fresh symbolic taint variables.

We use a motivating example to introduce the idea of segmented symbolic taint analysis. Figure 2 shows an example for symbolic taint analysis on a *straight-line code segment*, which is a simplified code snippet of the `libtiff` buffer overflow vulnerability (CVE-2013-4231). Assume when a worker thread starts taint analysis on this code segment (Figure 2(a)), no taint state for the input data (“size” and “num” in our case) is defined. Instead of waiting for the explicit information, we treat the unknown values as taint symbols (`symbol1` for “size” and `symbol2` for “num”, respectively) and summarize the net effect of taint propagation in the segment. The symbolic taint states are shown in Figure 2(b). When the explicit taint states are available, we resolve the symbolic taint states by replacing the taint symbols with their real taint tags or concrete values (Figure 2(c)). After that, we continue to perform concrete taint analysis like conventional DTA. Note that here we show pseudo-code for ease of understanding, while TaintPipe works on binary code.

Compared with inlined DTA, the application thread under TaintPipe is mainly instrumented with control flow profile logging code, which is quite lightweight. Therefore, TaintPipe results in much lower application runtime overhead. On the other hand, the execution of taint logic code is decoupled to multiple pipeline stages running in parallel. The accumulated effect of TaintPipe’s pipeline leads to a substantial speedup on taint analysis.

2.2 “Primary & Secondary” Model

Some recent work [15, 17, 9] offloads taint logic code from the application (primary) thread to another shadow (secondary) thread and runs them on separate cores. At the same time, the primary thread communicates with

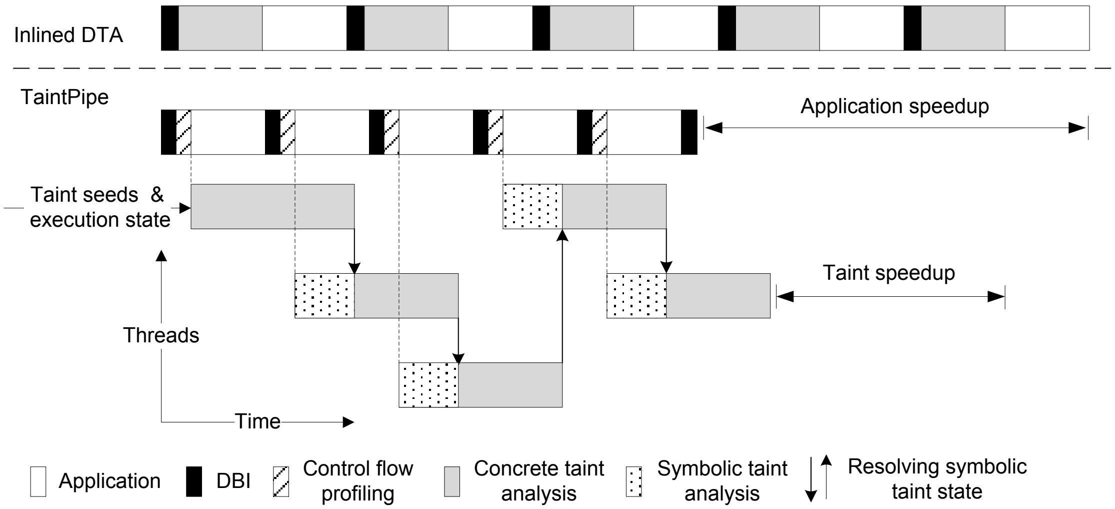

Figure 1: Inlined dynamic taint analysis vs. TaintPipe.

```
size = getc(infile);
A = -1;
B = size + 1;
C = (1 << size) - 1;
D = num & C;
```

(a) Code segment

Output	Taint state
A	0
B	symbol1 + 1
C	(1 << symbol1) - 1
D	symbol2 & ((1 << symbol1) - 1)

(b) Symbolic taint state

Output	Taint state
A	0
B	tag1 + 1
C	(1 << tag1) - 1
D	(1 << tag1) - 1

(c) Resolving symbolic taint state

Figure 2: An example of symbolic taint analysis on a code segment: (a) code segment; (b) symbolic taint states, the input value `size` and `num` are labeled as `symbol1` and `symbol2`, respectively; (c) resolving symbolic taint states when `size` is tainted as `tag1` and `num` is a constant value (num = 0xffffffff).

the secondary thread to convey the necessary information (e.g., the addresses of memory operations and control transfer targets) for performing taint analysis. However, this model suffers from frequent communication between the primary and secondary thread. In principle, every memory address that is loaded or stored has to be logged and transferred. Due to the frequent synchronization with the primary thread and the extra instructions to access shadow memory, taint logic execution in the secondary thread is typically slower than the application execution. As a result, the delay for each taint operation could be accumulated, leading to an delay proportional to the original execution. ShadowReplica [17] partially addresses this drawback by performing advanced offline static optimizations on the taint logic code to reduce the runtime overhead. However, in many security analysis scenarios, precise static analysis and optimizations over taint logic code are not feasible, e.g., reverse engineering and malware forensics. In such cases, program static features such as control flow graphs are possibly obfuscated.

In TaintPipe, we record compact control flow information to reconstruct *straight-line* code, in which all the targets of direct and indirect jumps have been resolved. However, we do not record or transfer the addresses of memory operations. Our key observation is that most addresses of memory operations can be inferred from the straight-line code. For example, if a basic block is ended with an indirect jump instruction `jmp eax`, we can quickly know the value of `eax` from the straight-line code. In this way, all the other memory indirect access calculated through `eax` (before it is updated) can be determined. For instance, we can infer the memory load address for the instruction: `mov ebx, [4*eax + 16]`. Even when the index of a memory lookup is a symbol, with the taint states and path predicates of the straight-line code, we can often narrow down the symbolic memory addresses to a small range in most cases.

Since TaintPipe's data communication is lightweight, TaintPipe can achieve nearly constant delay given enough number of worker threads. The upper limit number of worker threads is also bounded, which equals roughly the ratio of the taint analysis execution time over the application thread execution time for each segment.

Due to TaintPipe's pipelining design, it is possible that TaintPipe may detect an attack some time after the real attack has happened. However, this trade-off does not prevent TaintPipe from practically supporting a broad variety of security applications, such as attack forensic investigation and post-fact intrusion detection, which do not require strict runtime security enforcement. It is worth noting that different from ShadowReplica, TaintPipe does not depend on extensive static analysis to reduce data communication. Therefore, TaintPipe has a wider range of applications in speeding up analyzing obfuscated binaries, as static analysis of obfuscated binaries is of great challenge.

3 Design

3.1 Architecture

Figure 3 illustrates the architecture of TaintPipe. We have built the pipelining framework on top of a dynamic binary instrumentation tool, enabling TaintPipe to work with unmodified program binaries. The steps followed by TaintPipe for pipelining taint analysis are:

1. TaintPipe takes in a binary along with the taint seeds as input. The instrumented application thread starts execution with lightweight online logging for control flow and other information (Section 4.1.1).

2. Then the instrumented program is executed together with a multithreaded logging tool to efficiently deliver the logged data to memory (Section 4.1.2).

3. When the profile buffer becomes full, a taint analysis engine will be invoked for online pipelined taint analysis (Section 4.2.1).

4. The generated log data are then used to construct straight-line code, which helps to solve many precision loss problems in static taint analysis. In this stage, we generate a segment of executed code blocks for each logged data buffer. The memory addresses that are accessed through indirect jump targets are also resolved (Section 4.2.2).

5. The taint analysis engine will further translate straight-line code to taint operations, which avoid precision loss and support both multi-tag and bit-level taint analysis (Section 4.2.3).

6. With the constructed taint operations, TaintPipe performs pipelined symbolic taint analysis. When a thread finishes taint analysis with an explicit taint state, it synchronizes with its following thread to resolve the symbolic taint state (Section 4.2.4).

3.2 Segmented Symbolic Taint Analysis

In this section, we analyze symbolic taint analysis from a theoretical point of view to justify the correctness of our pipelining scheme. In order to formalize *segmented symbolic taint analysis*, we use the following notations:

1. Let σ denote a taint state, which maps variables to their taint tags.

2. Let $\mathscr{A}(\sigma, S)$ denote a symbolic taint analysis $\mathscr{A}$ on a straight-line code segment S, with an initial taint state σ. We use $\mathscr{A}_\sigma(S)$ for convenience.

Note that the straight-line code segment S has no control transfer statement. Conceptually, S only contains one type of statements, namely assignment statements. Of course, from the implementation point of view, there may be other types of statements, but they can all be regarded as assignment statements. For example, as we will show in Section 4.2.3, our taint operations contain assignment operations, laundering operations, and arithmetic operations. The latter two operations can be derived from taint assignment operations.

Based on the semantics of assignment statements, we define symbolic taint analysis for an assignment statement as follows:

$$\mathscr{A}_\sigma(x := e) = \sigma[x \mapsto e^t] \tag{1}$$

where e^t denotes the taint tag of e, and $[\cdot]$ is the taint state update operator. If x is a new variable, the taint state σ is extended with a new mapping from x to its taint. If x occurs in the taint state σ, for the variables in the domain of σ whose symbolic taint expressions depend on x, their symbolic taint expressions will be updated or recomputed with the new taint value of x.

Assume $\sigma_1 = \mathscr{A}_\sigma(i_1)$ for a statement i, then the symbolic taint analysis for two sequential statements $i_1; i_2$ is:

$$\mathscr{A}_\sigma(i_1; i_2) = \mathscr{A}_{\sigma_1}(i_2) \tag{2}$$

Assume straight-line code segment $S_1 = (i_1; S_1')$. We can then deduce the symbolic taint analysis on two sequential segments $S_1; S_2$ as follows:

$$\begin{aligned} &\mathscr{A}_\sigma(S_1; S_2) \\ =&\mathscr{A}_\sigma((i_1; S_1'); S_2) \\ =&\mathscr{A}_\sigma(i_1; (S_1'; S_2)) \\ =&\cdots \\ =&\mathscr{A}_{\mathscr{A}_\sigma(S_1)}(S_2) \end{aligned} \tag{3}$$

That is, given $\mathscr{A}_\sigma(S_1) = \sigma_1$ and $\mathscr{A}_\varepsilon(S_2) = \sigma_2$, where ε is an empty taint state, Eq. 3 leads to:

$$\mathscr{A}_\sigma(S_1; S_2) = \sigma_2[\sigma_1] \tag{4}$$

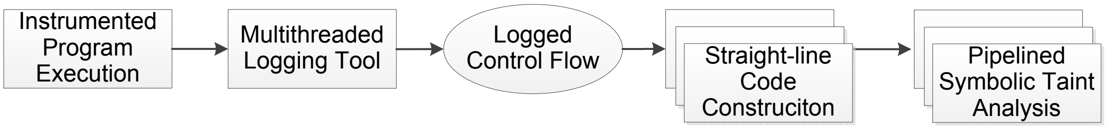
Instrumented Program Execution
Multithreaded Logging Tool
Logged Control Flow
Straight-line Code Construciton
Pipelined Symbolic Taint Analysis

Figure 3: Architecture.

Here, we misuse the taint state update operator $[\cdot]$ and apply it to a taint state map, instead of a single taint variable update. With Eq. 4, we can perform segmented taint analysis in parallel or in a pipeline style. For two segments $S_1;S_2$, assume the starting taint state is σ_0. We start two threads, one compute $\mathscr{A}_{\sigma_0}(S_1)$ and the other computes $\mathscr{A}_{\varepsilon}(S_2)$, where ε is an empty taint state. Assume the result of the first thread analysis is σ_1 and the result of the second is σ_2. The symbolic taint analysis of $S_1;S_2$ is $\sigma_2[\sigma_1]$, that is, the right hand side of Eq. 4. Eq. 4 forms the foundation of our segmented taint analysis in a pipeline style.

4 Implementation

To demonstrate the efficacy of TaintPipe, we have developed a prototype on top of the dynamic binary instrumentation framework Pin [23] (version 2.12) and the binary analysis platform BAP [5] (version 0.8). The online logging and pipelining framework are implemented as Pin tools, using about 3,100 lines of C/C++ code. The taint operation constructors are built on BAP IL (intermediate language). TaintPipe's taint analysis engine is based on BAP's symbolic execution module, using about 4,400 lines of Ocaml and running concurrently with Pin tools. We utilize Ocaml's functor polymorphism so that taint states can be instantiated in either concrete or symbolic style. All of the functionality implemented in taint analysis engine are wrapped as function calls. To support communication between Pin tools and taint analysis engine, we develop a lightweight RPC interface so that each worker thread can directly call Ocaml code. The saving and loading of the taint cache lookup table is implemented using the Ocaml Marshal API, which encodes IL expressions as sequences of compact bytes and then stores them in a disk file.

Dynamic binary instrumentation tools tend to inline compact and branch-less code to the final translated code. For the code with conditional branches, DBI emits a function call instead, which introduces additional overhead. Therefore, we carefully design our instrumentation code to favor DBI's code inlining. To fully reduce online logging overhead, we also utilize Pin-specific optimizations. We leverage Pin's fast buffering APIs for efficient data buffering. For example, the inlined `INS_InsertFillBuffer()` writes the control flow profile directly to the given buffer; the callback function registered in `PIN_DefineTraceBuffer()` processes the buffer when it becomes full or thread exits. Besides, we force Pin to use the fastcall x86 calling convention to avoid emitting stack-based parameter loading instructions (i.e., push and pop). Currently Pin-tools do not support the Pthreads library. Thus we employ Pin Thread API to spawn multiple worker threads. We also implement a counting semaphore based on Pin's locking primitives to assist thread synchronization. Additionally, TaintPipe can be extended to support multithreaded applications with no difficulty by assigning one taint pipeline for each application thread.

4.1 Logging

TaintPipe's pipeline stages consist of multiple threads. The thread of instrumented application (producer) serves as the source of pipeline, and a number of Pin internal threads act as worker threads to perform symbolic taint analysis on the data collected from the application thread. Note that unlike application threads, worker threads are not JITed and therefore execute natively. One of the major drawbacks of previous dynamic taint analysis decoupling approaches is the large amount of information collected in the application thread and the high overhead of communication between the application thread and analysis thread. To address these challenges, TaintPipe performs lightweight online logging to record information required for pipelined taint analysis. The logged data comprise control flow profile and the concrete execution state when taint seeds are first introduced, which is the starting point of our pipelined taint analysis. The initial execution state, consisting of concrete context of registers and memory, (e.g., CR0∼CR4, EFLAGS and addresses of initial taint seeds), is used to reduce the number of fresh symbolic taint variables.

We take major two steps to reduce the application thread slowdown: First, we adopt a compact profile structure so that the profile buffer contains logged data as much as possible, and it is quite simple to recover the entry address of each basic block as well. Second, we apply the "one producer, multiple consumers" model and N-way buffering scheme to process full buffers asynchronously, which allows application to continue execution while pipelined taint analysis works in parallel. We will discuss each step in the following sub-sections.

4.1.1 Lightweight Online Logging

Besides the initial execution state when the taint seeds are introduced, TaintPipe collects control flow information, which is represented as a sequence of basic blocks executed. Conceptually, we can use a single bit to record the direction of conditional jump [29], which leads to a much more compact profile. However, reconstruction straight-line code from 1 bit profile is more complicated to make it fit for offline analysis. Zhao et al. [47] proposed *Detailed Execution Profile* (DEP), a 2-byte profile structure to represent 4-byte basic block address on x86-32 machine. In DEP, a 4-byte address is divided into two parts: *H-tag* for the 2 high bytes and *L-tag* for 2 low bytes. If two successive basic blocks have the same H-tag, only L-tag of each basic block enters the profile buffer; otherwise a special tag 0x0000 followed by the new H-tag will be logged into the buffer.

We extend DEP's scheme to support `REP`-prefix instructions. A number of x86 instructions related to string operations (e.g., `MOVS`, `LODS`) with `REP`-prefix are executed repeatedly until the counter register (ecx) counts down to 0. Dynamic binary instrumentation tools [23, 4] normally treat a `REP`-prefixed instruction as an implicit loop and generate a single instruction basic block in each iteration. In our evaluation, there are several cases that unrolling such `REP`-prefix instructions would be a performance bottleneck. We address this problem by adding additional escape tags to represent such implicit loops. Figure 4 presents an example of the control flow profile we adopted. The left part shows a segment of straight-line code containing 1028 basic blocks, and 1024 out of them are due to `REP`-prefixed instruction repetitions. Our profile (the right side of Figure 4) encodes such case with two consecutive escape tags (0xffff), followed by the number of iterations (0x0400).

We note that it is usually unnecessary to turn on the logging all the time. For example, when application starts executing, many functions are only used during loading. At that time, no sensitive taint seed is introduced. Therefore we perform on-demand logging to record control flow profile when necessary. As application starts running, we only instrument limited functions to inspect the various input channels that taint could be introduced into the application (taint seeds). Such taint seeds include standard input, files and network I/O. Besides, users can customize other values as taint seeds, such as function return values or parameters. When the pre-defined taint seeds are triggered, we turn on the control flow profile logging. At the same time, we save the current execution state to be used in the pipelined taint analysis. Many well-known library functions have explicit semantics, which facilitates us to selectively turn off logging inside these functions and propagate taint correspondingly at function level. We will discuss this issue further in Section 4.2.3.

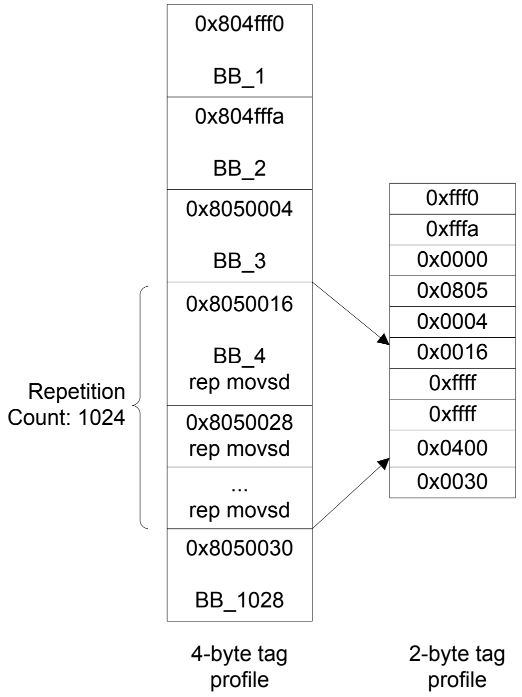

Figure 4: An example of 2-byte tag profile.

4.1.2 N-way Buffering Scheme

Since TaintPipe's online logging is lightweight, application (producer) thread's execution speed is typically faster than the processing speed of worker threads. To mitigate this bottleneck, we employed "one producer, multiple consumers" model and *N-way buffering scheme* [46]. At the center of our design is a thread pool, which is subdivided into n linked buffers, and the producer thread and multiple worker threads work on different buffers simultaneously. More specifically, when the instrumented application thread starts running, we first allocate n linked empty buffers ($n > 1$). At the same time, n Pin internal threads (worker threads) are spawned. Each worker thread is bound to one buffer and communicates with the application thread via semaphores. When a buffer becomes full, the application thread will release the full buffer to its corresponding worker thread and then continue to fill in the next available empty buffer. Given a full profile buffers, a worker thread will send it to a taint analysis engine to perform concrete/symbolic taint analysis in parallel. After that, the worker thread will release the profile buffer back to the application thread and wait for processing the next full buffer.

It is apparent that the availability of unused worker threads and the size of profile buffer will affect overall performance of TaintPipe (both application execution time and pipelined taint analysis) significantly. In Section 5.1, we will conduct a series of experiments to find the optimal values for these two factors.

4.2 Symbolic Taint Analysis

4.2.1 Taint Analysis Engine

When the application thread releases a full profile buffer, a worker thread is waked up to capture the profile buffer and then communicates with a taint analysis engine for pipelined taint analysis. The taint analysis engine will first convert the control flow profile to a segment of straight-line intermediate language (IL) code and then translates the IL code to even simpler taint logic operations. The translations are cached for efficiency at taint basic block level. The key components of taint analysis engine are illustrated in Figure 5.

The core of TaintPipe's taint analysis engine is an abstract taint analysis processor, which simulates a segment of taint operations and updates the taint states accordingly. The taint state structure contains two contexts: virtual registers keeping track of symbolic taint tags for register, and taint symbolic memory for symbolic taint tags in memory. The taint symbolic memory design is like the two-level page table structure and each page of memory consists of symbolic taint formulas rather than concrete values. After the initialization of the symbolic taint inputs, the engines perform taint analysis either concretely or symbolically in a pipeline style.

4.2.2 Straight-line Code Construction

Given the control flow profiles, recovering each basic block's H-tag and L-tag is quite straightforward. A basic block's entry address is the concatenation of its corresponding H-tag and L-tag [47]. The taint analysis engine should only execute the instructions required for taint propagation. Otherwise, the work thread may run much slower than the application thread. On the other side, due to the cumbersome x86 ISA, precisely propagating taint for the complex x86 instructions is an arduous work, especially for some instructions with side effect of conditional taint (e.g., `CMOV`). To achieve these two goals, we first extract the x86 instructions sequence from the application binary and then lift them to BIL [5], a RISC-like intermediate language. Since we know exactly the execution sequence, the sequence is a straight-line code. We have removed all the direct and indirect control transfer instructions and substituted them with control transfer target assertion statements.

After resolving an indirect control transfer, we go one step further to determine all the memory operation addresses which depend on this indirect control transfer target. For example, after we know the target of `jmp eax`, we continue to trace the use-def chain of `eax` for each memory load or store operation whose address is calculated through this `eax`. With the initial execution state (containing addresses of taint seeds) and indirect control transfer target resolving, we are able to decide most of the memory operation addresses.

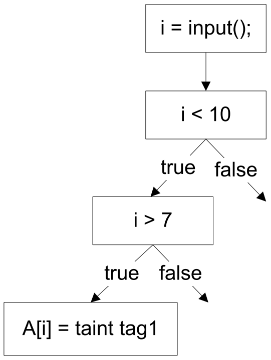

Figure 6: A path predicate constrains symbolic memory access within the boundary of $7 < i < 10$.

For some applications such as word processing, a symbolic taint input may be used as a memory lookup index. Without any constraint, a symbolic memory index could point to any memory cell. Inspired by the index-based memory model proposed by Cha et al. [8], we attempt to narrow down the symbolic memory accesses to a small range with symbolic taint states and path predicates. We first leverage value set analysis [2] to limit the range of a symbolic memory access and then refine the range by querying a constraint solver. The path predicate along the straight-line code usually limits the scope of symbolic memory access. Figure 6 shows such an example where the path predicate restricts the symbolic memory index i within a range such that $7 < i < 10$. When propagating a taint tag to the memory cell referenced by i, we conservatively taint all the possible memory slots, that is, `A[8]` and `A[9]` in Figure 6 will be tainted as `tag1`. In Section 5.3, we will demonstrate that our symbolic memory index solution only introduces marginal side effects.

4.2.3 Taint Operation Generation

Based on BIL statements, we construct taint operations. Taint operations inside a basic block are formed as "taint basic block" [37], which are cached for efficiency. To make the best of cache effect, we merge the basic blocks with only one predecessor and one successor. Since BIL explicitly reveals the side effect of intricate x86 instructions, it is easy to perform intra-block optimizations to get rid of redundant taint operations. Therefore, our taint

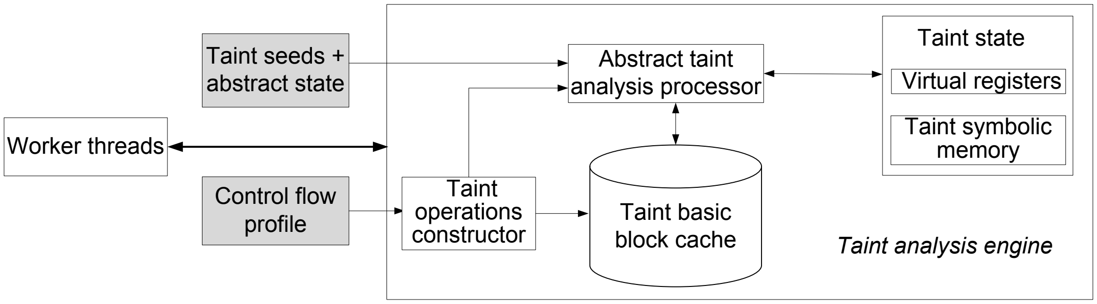

Figure 5: The structure of taint analysis engine.

operations are simple and accurate. Currently our taint operations mainly consist of three types of operations for tracking taint data:

1. Assignment operations: The operations in this category are involved in copying values between registers and registers/memory. We simply assign the taint tag of source operand to the destination operand.

2. Laundering operations: The operations are used to clean the taint tag of the destination operand. For example, `xor eax,eax` will clean the taint result of `eax`. We identify all laundering operations in taint basic blocks and substitute them with assignment operations.

3. Arithmetic and logic operations: This category of operations are the most difficult to handle. We emulate arithmetic and logic operations on the taint tags to capture their real semantics.

Figure 7 presents an example of taint operations for a basic block. TaintPipe's symbolic taint operations outperform conventional DTA approaches in three ways. First and foremost, multi-tag taint analysis is straightforward for TaintPipe. Each symbolic variable can naturally represent a taint tag (see Line 1 and Line 2 in Figure 7). Second, previous DTA tools mostly adopted a "short circuiting" method to handle arithmetic operations, that is, the destination is tainted if at least one of the source operands is tainted regardless of the real semantics. However, in many scenarios, it will lead to precision loss. Check the code at Line 4 and 5 in Figure 7, value *d* will always be zero since *b* is the negation of *a*. Unfortunately, some previous work may label *d* as tainted incorrectly [41]. Third, different from related work [20, 17], TaintPipe supports bit-level taint for EFLAGS register, representing whether a bit of the EFLAGS is tainted or not due to side effects. Recent work has demonstrated the value of bit-level taint in binary code de-obfuscation [42].

```
int a, b, c, d;
1: a = read ();       1: Taint (a) = tag1;
2: c = read ();       2: Taint (c) = tag2;
3: c = c xor c;       3: Taint (c) = 0;
4: b = ~ a;           4: Taint (b) = ~ tag1;
5: d = a & b;         5: Taint (d) = 0;

(a) a basic block     (b) a taint basic block
```

Figure 7: Example: taint operations.

Table 1: Function summary.

Category	Function
No tainting	strcmp, strncmp, memcmp, strlen, strchr, strstr, strpbrk, strcspn, qsort, rand, time, clock, ctime
Function level	strcat, strncat, strcpy, strncpy, memcpy, memmove, strtok, atoi, itoa, abs tolower, toupper

Another major optimization we adopt is so called "function summary". As many well-known library functions have explicit semantics (e.g., atoi, strlen), we generate a summary of each function and propagate taint correspondingly at function level. Table 1 lists two types of function summary TaintPipe supports: 1) Functions within "no tainting" category do not have any side effect on taint state. We can safely turn off logging when executing them. 2) Some functions do propagate taint from an input parameter to output. We still turn off logging and update taint state correspondingly when these functions return.

4.2.4 Symbolic Taint State Resolution

In TaintPipe's pipeline framework, a worker thread may perform taint analysis concretely or symbolically in parallel. When a worker thread completes taint analysis with concrete taint tags, the final taint state it maintains is deterministic. Then it synchronizes with the subsequent worker thread to resolve the symbolic taint state maintained by the latter. Taint states are allocated in a shared memory area so that multiple threads can access them easily. Basically, given the concrete taint state at the beginning of a code segment, we replace a symbolic taint tag with the appropriate starting value (either a taint tag or a concrete value). We further update all the symbolic formula containing that symbolic taint tag. For example, the logic AND formula in Figure 2(b) will be simplified to a single taint tag. After that, the subsequent thread switches to the concrete taint analysis and continue processing the left segment code.

In this order, the taint states of each segment will be resolved and updated one by one. The defined taint policy (e.g., a function return value should not be tainted) is checked along the concrete taint analysis as well. A tainted sink is identified if it contains a symbolic formula; multiple tags are determined by counting the number of different symbols in the formula. Note that a previous hardware-assisted approach [31] utilized a separate "master" processor to update each segment's taint status sequentially. However, as pointed out by the paper, when there are more than a few "worker" processors, the master processor will become the bottleneck. Our approach amortizes the workload of the master processor to each worker thread.

5 Experimental Evaluation

We conducted experiments with several goals in mind. First, we wanted to choose optimal values for two factors that may affect TaintPipe's performance, namely control flow profile buffer size and the number of worker threads. Then we studied overall runtime overhead when running TaintPipe on the SPEC2006 int benchmarks and a number of common utilities. We also compared TaintPipe with a highly optimized inline dynamic taint analysis tool. At the same time, we wanted to make sure TaintPipe is effective in speeding up various security analysis tasks and can compete with conventional inlined dynamic taint analysis in precision. To this end, we demonstrated three compelling applications: 1) detecting software attacks; 2) tracking information flow in obfuscated malicious programs; and 3) identifying cryptography functions with multi-tag propagation.

5.1 Experiment Setup

Our experiment platform contains two Intel Xeon E5-2690 processors, 128GB of memory and a 250GB solid state drive, running Ubuntu12.04. Each processor is equipped with 8 2.9GHz cores, 16 hyper threads and 20MB L3 cache. The performance data reported in this section are all mean values with 5 repetitions.

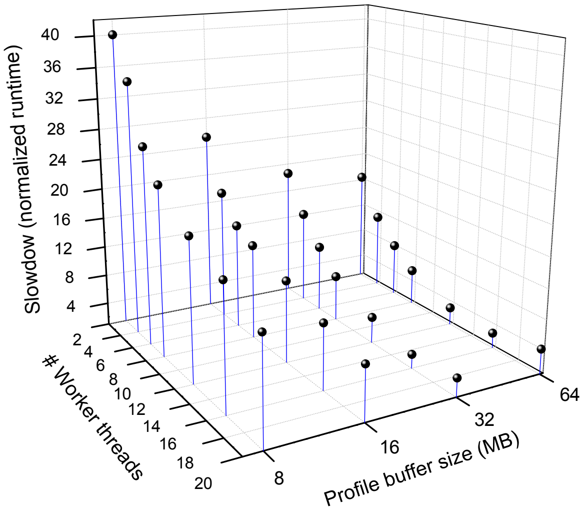

Figure 8: Optimal buffer size and number of worker threads.

TaintPipe's performance is affected by the size of the profile buffers and the number of worker threads. Generally, the more worker threads and larger profile buffers, the less possibility that an application is suspended to wait for a free buffer. On the other hand, our taint analysis engine has to take longer time to process larger segment code. We conducted a series of tests with the SPEC CPU2006 int benchmarks, under different settings of these two variables. We dynamically adjust the number of worker threads from 2 to 20 (2, 4, 6, 8, 12, 16 and 20), and profile size from 8MB to 64MB (8MB, 16MB, 32MB and 64MB). Figure 8 displays the experimental results. Roughly, as number of worker threads and buffer size increase, the application slowdown reduces. That is mainly because large buffer sizes allow application thread continue to fill up and worker threads spend less time on synchronization. After a certain point (buffer size $\geq$ 32MB and number of worker threads $>$ 16), overhead increases slightly. Two factors prevent TaintPipe from achieving more speedup. First, taint analysis engine slows down when processing large code segment. Second, more worker threads introduce larger communication latency when resolving symbolic taint states. According to the results, we set the two factors as their optimal values (32MB buffer size and 16 worker threads), which will be used in the following experiments.

5.2 Performance

To evaluate the performance gains achieved by pipelining taint logic, we compared TaintPipe with a state-of-the-art tool, libdft [20], which performs inlined dynamic taint analysis based on Pin ("libdft" bar). In addition, we developed a simple tool to measure the slowdown im-

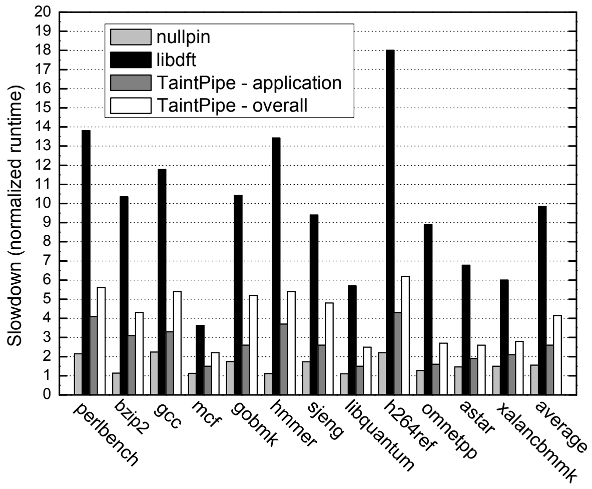

Figure 9: Slowdown on SPEC CPU2006.

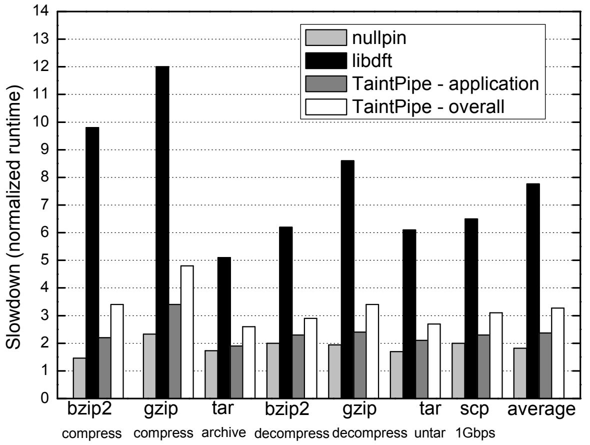

Figure 10: Slowdown on common Linux utilities.

posed exclusively by TaintPipe. It runs a program under Pin without any form of analysis ("nullpin" bar). The "TaintPipe - application" bar represents the running time of instrumented application thread alone, and "TaintPipe - overall" corresponds to the overall overhead when both the application thread and pipelined worker threads are running. The major reason we reported "TaintPipe - application" and "TaintPipe - overall" time separately is to show the two improvements, namely "Application speedup" and "Taint speedup" (see Figure 1). Since the application thread typically runs faster than the worker threads, the "TaintPipe - overall" time is actually dominated by the worker threads. Therefore, usually the "TaintPipe - overall" time represents the relative time spent by worker threads as well. The times reported in this section are all normalized to native execution, that is, application running time without dynamic binary instrumentation.

SPEC CPU2006. Figure 9 shows the normalized execution times when running the SPEC CPU2006 int benchmark suite under TaintPipe. On average, the instrumented application thread enforces a 2.60X slowdown to native execution, while the overall slowdown of TaintPipe is 4.14X. If we take Pin's environment runtime overhead ("nullpin" bar) as the baseline, we can see TaintPipe imposes 2.67X slowdown ("TaintPipe - overall" / "nullpin") and libdft introduces 6.4X slowdown—this number is coincident to the observation that propagating a taint tag normally requires extra 6–8 instructions [30, 11]. In summary, TaintPipe outperforms inlined dynamic taint analysis drastically: 2.38X faster than the inlined dynamic taint analysis, and 3.79X faster in terms of application execution. In the best case (*h264ref*), the application speedup under TaintPipe exceeds 4.18X.

Utilities. We also evaluated TaintPipe on four common Linux utilities, which were not chosen randomly. These four utilities represent three kinds of workloads: I/O bounded (`tar`), CPU bounded (`bzip2` and `gzip`), and the case in-between (`scp`). We applied `tar` to archive and extract the GNU Core utilities package (version 8.13) (∼50MB), then we employed `bzip2` and `gzip` to compress and decompress the archive file. Finally we utilized `scp` to copy the archive file over a 1Gbps link. As shown in Figure 10, TaintPipe reduced slowdown of dynamic taint analysis from 7.88X to 3.24X, by a factor of 2.43 on average.

Effects of Optimizations. In this experiment, we quantify the effects of taint logic optimizations we presented in Section 4.2, which are paramount for optimized TaintPipe performance. Figure 11 shows the impact of these optimizations when applied cumulatively on SPEC CPU2006 and the set of common utilities. The "unopt" bar approximates an un-optimized TaintPipe, which does not adopt any optimization method. The "O1" bar indicates the optimization of function summary, reducing application slowdown notably by 26.6% for SPEC CPU2006 and 25.0% for the common utilities. The "O2" bar captures the effect of taint basic block cache, leading to a further reduction by 19.0% and 22.9% for SPEC and utilities, respectively. Intra-block optimizations, denoted by "O3", offer further improvement, 12.0% with SPEC and 11.6% with the utilities).

Table 2: Tested software vulnerabilities.

Program	Vulnerability	CVE ID	# Taint Bytes		
			libdft	Temu	TaintPipe
Nginx	Validation Bypass	CVE-2013-4547	45	45	45
Micro_httpd	Validation Bypass	CVE-2014-4927	80	85	80
Tiny Server	Validation Bypass	CVE-2012-1783	125	126	125
Regcomp	Validation Bypass	CVE-2010-4052	1,124	1,148	1,180
Libpng	Denial Of Service	CVE-2014-0333	72	72	72
Gzip	Integer Underflow	CVE-2010-0001	94	112	96
Grep	Integer Overflows	CVE-2012-5667	608	682	653
Coreutils	Buffer Overflow	CVE-2013-0221	252	260	256
Libtiff	Buffer Overflow	CVE-2013-4231	268	286	290
WaveSurfer	Buffer Overflow	CVE-2012-6303	384	418	406
Boa	Information Leak	CVE-2009-4496	164	164	164
Thttpd	Information Leak	CVE-2009-4491	328	328	328

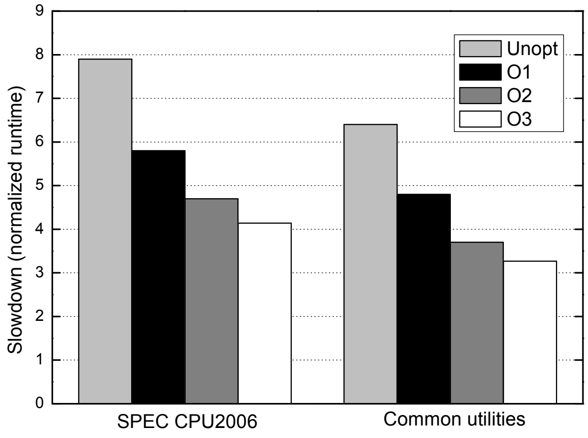

Figure 11: The impact of optimizations to speed up TaintPipe when applied cumulatively: O1 (function summary), O2 (O1 + taint basic block cache), O3 (O2 + intra-block optimizations).

Table 3: Malware samples and taint graphs.

Sample	Type	Taint Graph		Control Flow Obfuscation
		Node #	Edge #	
Svat	Virus	90	62	
RST	Virus	154	82	
Agent	Rootkit	624	402	✓
KeyLogger	Trojan	554	368	✓
Subsevux	Backdoor	1648	764	✓
Tsunami	Backdoor	734	534	✓
Keitan	Backdoor	618	482	✓
Fireback	Backdoor	1038	620	✓

5.3 Security Applications

Software Attack Detection. One important application of taint analysis is to define taint policies, and ensure they are not violated during taint propagation. We tested TaintPipe with 12 recent software exploits listed in Table 2, which covers a wide range of real-life software vulnerabilities. For example, the vulnerabilities in `nginx`, `micro_httpd`, and `tiny server` allow remote attackers to bypass input validation and crash the program. The `libtiff` buffer overflow vulnerability leads to an out of bounds loop limit via a malformed gif image. Both `boa` and `thttpd` write data to a log file without sanitizing non-printable characters, which may be exploited to execute arbitrary commands. Since we have detailed vulnerability reports, we can easily mark the locations of taint sinks in the straight-line code and set corresponding taint policies.

In our evaluation, TaintPipe did not generate any false positives and successfully identified taint policy violations while incurring only small overhead. At the same time, we evaluated the accuracy of TaintPipe. To this end, we counted the total number of tainted bytes in the taint state when taint analysis hit the taint sinks. Column 4 $\sim$ 6 of Table 2 show the number of taint bytes when running libdft, Temu [44] and TaintPipe, respectively. Compared with the inlined dynamic taint analysis tools (libdft and Temu), TaintPipe's symbolic taint analysis achieves almost the same results in 8 cases and introduces only a few additional taint bytes in the other 4 cases. We attribute this to our conservative approach to handling of symbolic memory indices. The evaluation data show that TaintPipe does not result in over-tainting [32] and rivals the inlined dynamic taint analysis at the same level of precision.

Table 4: Cryptographic function detection time.

Algorithm	TaintPipe (s)	Temu (s)
TEA	3.8 (<1.1X)	15.2 (2.2X)
AES-CBC	12.3 (1.2X)	125.6 (3.8X)
Blowfish	4.5 (<1.1X)	21.4 (2.5X)
MD5	7.4 (<1.1X)	35.1 (2.6X)
SHA-1	8.8 (1.1X)	40.2 (3.3X)

Generating Taint Graphs for Malware. We ran 8 malware samples collected from `VX Heavens`[1] with TaintPipe.[2] Similar to Panorama [43], we tracked information flow and generated a *taint graph* for each sample. In a taint graph, nodes represent taint seeds or instructions operating on taint data, and a directed edge indicates an explicit data flow dependency between two nodes. Taint graph faithfully describes intrinsic malicious intents, which can be used as malware specification to detect suspicious samples [12]. The statistics of our testing results are presented in Table 3. It is worth noting that 6 out of 8 malware samples are applied with various control flow obfuscation methods (the fifth column), such as opaque predicates, control flow flattening, obfuscated control transfer targets, and call stack tampering. As a result, the control flow graphs are heavily cluttered. For example, malware samples `Keitan` and `Fireback` have a relatively high ratio of indirect jumps (e.g., `jmp eax`). Typically it is hard to precisely infer the destination of an indirect jump statically. Thus, the taint logic optimization methods that rely on accurate control flow graph [17, 18] will fail. In contrast, our approach does not rely on control flow graph and therefore we analyzed these obfuscated malware samples smoothly.

Cryptography Function Detection. Malware authors often use cryptography algorithms to encrypt malicious code, sensitive data, and communication. Detecting cryptography functions facilitates malware analysis and forensics. Recent work explored the *avalanche effect* to quickly identify possible cryptography functions by observing the input-output dependency with multi-tag taint analysis. That is, each byte in the encrypted message is dependent on almost all bytes of input data or key [7, 21, 48]. However, multi-tag dynamic taint analysis normally has to sacrifice more shadow memory and imposes much higher runtime overhead than single-tag dynamic taint analysis. Recall that multi-tag propagation is handled transparently in TaintPipe. In this experiment, we applied TaintPipe to detect such avalanche effects in binary code. We utilized the test case suite of Crypto++ library[3] and tested 5 cryptography algorithms. Each byte of the plain messages was labeled as a different taint tag. We compared TaintPipe with Temu [44], which supports multiple byte-to-byte taint propagation as well.[4] The detection time is shown in Table 4. We also reported the ratio of multi-tag's running time to single-tag's. The results show that TaintPipe is able to detect cryptographic functions with little additional overhead (less than 1.1X on average), while Temu's multi-tag propagation imposes a significant slowdown (2.9X to single-tag propagation on average).

6 Discussions and Limitations

Since TaintPipe's pipelining design leads to an asynchronous taint check, TaintPipe may detect a violation of taint policy after the real attack happens. One possible solution is to provide *synchronous* policy enforcement at critical points (e.g., indirect jump and system call sites). In that case, we can explicitly suspend the application thread, and wait for the worker threads to complete. Our current design spawns worker threads in the same process of running both Pin and the application. In the future, we plan to replace the worker threads with different processes to increase isolation.

As TaintPipe may perform symbolic taint analysis when explicit taint states are not available, TaintPipe exhibits similar limitations as symbolic execution of binaries. Recent work MAYHEM [8] proposes an advanced index-based memory model to deal with symbolic memory index. We plan to extend our symbolic memory index handling in the future. TaintPipe recovers the straight-line code by logging basic block entry address. However, with malicious self-modifying code, the entry address may not uniquely identify a code block. To address this issue, we can augment TaintPipe by logging the real executed instructions at the expense of runtime performance overhead.

Our focus is to demonstrate the feasibility of pipelined symbolic taint analysis. We have not fully optimized the symbolic taint analysis part which we believe can be greatly improved in terms of performance based on our current prototype. As our taint analysis engine simulates the semantics of taint operations, the speed of taint analysis is slow. One future direction is to execute concrete taint analysis natively like micro execution [16] and switch to the interpretation-style when performing symbolic taint analysis. Currently TaintPipe requires large share memory to reduce communication overhead between different pipeline stages. Therefore, our approach is more suitable for large servers with sufficient memory.

[1] http://vxheaven.org

[2] All these 8 samples are not packed. To analyze packed binaries, we can start TaintPipe when the unpacking procedure arrives at the original entry point.

[3] http://www.cryptopp.com/

[4] libdft does not support multi-tag taint analysis.

7 Related Work

In this section we first present previous work on static and dynamic taint analysis. Our work is a hybrid of these two analyses. Then we introduce previous efforts on taint logic code optimization, which benefits our taint operation generation. Finally, we describe recent work on decoupling taint tracking logic from original program execution, which is the closest to TaintPipe's method.

Static and Dynamic Taint Analysis. Since static taint analysis (STA) is performed prior to execution by considering all possible execution paths, it does not affect application runtime performance. STA has been applied to data lifetime analysis for Android applications [1], exploit code detection [36], and binary vulnerability testing [28]. Dynamic taint analysis (DTA) is more precise than static taint analysis as it only propagates taint following the real path taken at run time. DTA has been widely used in various security applications, including data flow policy enforcement [25, 40, 27], reversing protocol data structures [33, 38, 6], malware analysis [39] and Android security [14]. However, an intrinsic limitation of DTA is its significant performance slowdown. Schwartz et al. [32] formally defined the operational semantics for DTA and forward symbolic execution (FSE). Our approach is in fact a hybrid of these techniques. Worker thread conducts concrete taint analysis (like DTA) whenever explicit taint information is available; otherwise symbolic taint analysis (like STA and FSE) is performed.

Taint Logic Optimization. Taint logic code, deciding whether and how to propagate taint, require additional instructions and "context switches". Frequently executing taint logic code incurs substantial overhead. Minemu [3] achieved a decent runtime performance at the cost of sacrificing memory space to speed up shadow memory access. Moreover, Minemu utilized spare SSE registers to alleviate the pressure of general register spilling. As a result, Minemu only worked on 32-bit program. TaintEraser [49] developed function summaries for Windows programs to propagate taint at function level. Libdft [20] introduced two guidelines to facilitate DBI's code inlining: 1) tag propagation code should have no branch; 2) shadow memory updates should be accomplished with a single assignment. Ruwase et al. [30] applied compiler optimization techniques to eliminate redundant taint logic code in hot paths. Jee et al. [19] proposed *Taint Flow Algebra* to summarize the semantics of taint logic for basic blocks. All these efforts to generate optimized taint logic code are orthogonal and complementary to TaintPipe.

Decoupling Dynamic Taint Analysis. A number of researchers have considered the high performance penalty imposed by inlined dynamic taint analysis. They proposed various solutions to decouple taint tracking logic from application under examination [24, 31, 26, 15, 17, 9], which are close in spirit to our proposed approach. Speck [26] forked multiple taint analysis processes from application execution to spare cores by means of speculative execution, and utilized record/replay to synchronize taint analysis processes. Speck required OS level support for speculative execution and rollback. Speck's approach sacrifices processing power to achieve acceleration. Similar to TaintPipe's segmented symbolic taint analysis, Ruwase et al. [31] proposed *symbolic inheritance tracking* to parallelize dynamic taint analysis. TaintPipe differs from Ruwase et al.'s approach in three ways: 1) Their approach was built on top of a log-based architecture [10] for efficient communication with idle cores, while TaintPipe works on commodity multi-core hardware directly. 2) To achieve better parallelization, they adopted a relaxed taint propagation policy to set a binary operation as untainted, while TaintPipe performs full-fledged taint propagation so that we provide stronger security guarantees. 3) They used a separate "master" processor to update each segment's taint status sequentially, while TaintPipe resolves symbolic taint states between two consecutive segments. Our approach could achieve better performance when there are more than a few "worker" processors.

Software-only approaches [15, 17, 9] are the most related to TaintPipe. They decouple dynamic taint analysis to a shadow thread by logging the runtime values that are needed for taint analysis. However, as we have pointed out, these methods [15, 9] may suffer from high overhead of frequent communication between the application thread and shadow thread. Recent work ShadowReplica [17] ameliorates this drawback by adopting fine-grained offline optimizations to remove redundant taint logic code. In principle, it is possible to remove redundant taint logic by means of static offline optimizations. Unfortunately, even static disassembly of stripped binaries is still a challenge [22, 35]. Therefore, the assumption by ShadowReplica that an accurate control flow graph can be constructed may not be feasible in certain scenarios, such as analyzing control flow obfuscated software. We take a different angle to address this issue with lightweight runtime information logging and segmented symbolic taint analysis. We demonstrate the capability of TaintPipe in speeding up obfuscated binary analysis, which ShadowReplica may not be able to handle. Furthermore, ShadowReplica does not support bit-level and multi-tag taint analysis, while TaintPipe handles them naturally.

8 Conclusion

We have presented TaintPipe, a novel tool for pipelining dynamic taint analysis with segmented symbolic taint analysis. Different from previous parallelization work on taint analysis, TaintPipe uses a pipeline style that relies on straight-line code with very few runtime values, enabling lightweight online logging and much lower runtime overhead. We have evaluated TaintPipe on a number of benign and malicious programs. The results show that TaintPipe rivals conventional inlined dynamic taint analysis in precision, but with a much lower online execution slowdown. The performance experiments indicate that TaintPipe can speed up dynamic taint analysis by 2.43 times on a set of common utilities and 2.38 times on SPEC2006, respectively. Such experimental evidence demonstrates that TaintPipe is both efficient and effective to be applied in real production environments.

9 Acknowledgments

We thank the Usenix Security anonymous reviewers and Niels Provos 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) grant N00014-13-1-0175. Liu was also partially supported by ARO W911NF-09-1-0525.

References

[1] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., and McDaniel, P. 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] Balakrishnan, G., and Reps, T. WYSINWYX: What You See Is Not What You eXecute. *ACM transactions on programming languages and systems 32*, 6 (2010).

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

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

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

[6] Caballero, J., Poosankam, P., Kreibich, C., and Song, D. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of the 16th ACM Conference on Computer and Communication Security (CCS'09)* (2009).

[7] Caballero, J., Poosankam, P., McCamant, S., Babi ć, D., and Song, D. 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] Cha, S. K., Avgerinos, T., Rebert, A., and Brumley, D. Unleashing mayhem on binary code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (2012).

[9] Chabbi, M., Periyanayagam, S., Andrews, G., and Debray, S. Efficient dynamic taint analysis using multicore machines. Tech. rep., The University of Arizona, May 2007.

[10] Chen, S., Gibbons, P. B., Kozuch, M., and Mowry, T. C. Log-based architectures: Using multicore to help software behave correctly. *ACM SIGOPS Operating Systems Review 45*, 1 (2011), 84–91.

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

[12] Christodorescu, M., Jha, S., and Kruegel, C. Mining specifications of malicious behavior. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE'07)* (2007).

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

[14] Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 2010 USENIX Symposium on Operating Systems Design and Implementation (OSDI'10),* (2010).

[15] Ermolinskiy, A., Katti, S., Shenker, S., Fowler, L. L., and McCauley, M. Towards practical taint tracking. Tech. rep., EECS Department, University of California, Berkeley, Jun 2010.

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

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

[18] Jee, K., Portokalidis, G., Kemerlis, V. P., Ghosh, S., August, D. I., and Keromytis, A. D. A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware. In *Proceedings of the 19th Internet Society (ISOC) Symposium on Network and Distributed System Security (NDSS)* (2012).

[19] Jee, K., Portokalidis, G., Kemerlis, V. P., Ghosh, S., August, D. I., and Keromytis, A. D. A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware. In *Proceedings of the 2012 Network and Distributed System Security Symposium (NDSS'12)* (2012).

[20] Kemerlis, V. P., Portokalidis, G., Jee, K., and Keromytis, A. D. 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] Li, X., Wang, X., and Chang, W. CipherXRay: Exposing cryptographic operations and transient secrets from monitored binary execution. *IEEE Transactions on Dependable and Secure Computing 11*, 2 (2014).

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

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

[24] Nagarajan, V., Kim, H.-S., Wu, Y., and Gupta, R. Dynamic information flow tracking on multicores. In *Proceedings of the 2008 Workshop on Interaction between Compilers and Computer Architectures* (2008).

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

[26] Nightingale, E. B., Peek, D., Chen, P. M., and Flinn, J. 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).

[27] Qin, F., Wang, C., Li, Z., seop Kim, H., Zhou, Y., and Wu, Y. 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).

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

[29] Renieris, M., Ramaprasad, S., and Reiss, S. P. Arithmetic program paths. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-13)* (2005).

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

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

[32] Schwartz, E. J., Avgerinos, T., and Brumley, D. 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 2010 IEEE Symposium on Security and Privacy* (2010).

[33] Slowinska, A., Stancescu, T., and Bos, H. Howard: A dynamic excavator for reverse engineering data structures. In *Proceedings of the 2011 Network and Distributed System Security Symposium (NDSS'11)* (2011).

[34] Vachharajani, N., Bridges, M. J., Chang, J., Rangan, R., Ottoni, G., Blome, J. A., Reis, G. A., Vachharajani, M., and August, D. I. RIFLE: An architectural framework for user-centric information-flow security. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'37)* (2004).

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

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

[37] Whelan, R., Leek, T., and Kaeli, D. Architecture-independent dynamic information flow tracking. In *Proceedings of the 22nd international conference on Compiler Construction (CC'13)* (2013).

[38] Wondracek, G., Comparetti, P. M., Kruegel, C., and Kirda, E. Automatic network protocol analysis. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)* (2008).

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

[40] Xu, W., Bhatkar, S., and Sekar, R. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the 15th Conference on USENIX Security Symposium (USENIX'06)* (2006).

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

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

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

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

[45] Yip, A., Wang, X., Zeldovich, N., and Kaashoek, M. F. Improving application security with data flow assertions. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 291–304.

[46] Zhao, Q., Cutcutache, I., and Wong, W.-F. PiPA: Pipelined profiling and analysis on multi-core systems. In *Proceedings of the 2008 International Symposium on Code Generation and Optimization (CGO'08)* (2009).

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

[48] Zhao, R., Gu, D., Li, J., and Zhang, Y. Automatic detection and analysis of encrypted messages in malware. In *Proceedings of the 9th China International Conference on Information Security and Cryptology (INSCRYPT'13)* (2013).

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