DTAM: Dynamic Taint Analysis of Multi-threaded Programs for Relevancy

Malay Ganai
NEC Labs America
malay@nec-labs.com

Dongyoon Lee
University of Michigan
dongyoon@umich.edu

Aarti Gupta
NEC Labs America
agupta@nec-labs.com

Abstract

Testing and debugging multi-threaded programs are notoriously difficult due to non-determinism not only in inputs but also in OS schedules. In practice, dynamic analysis and failure replay systems instrument the program to record events of interest in the test execution, e.g., program inputs, accesses to shared objects, synchronization operations, context switches, etc. To reduce the overhead of logging during runtime, these testing and debugging efforts have proposed tradeoffs for sampling or selective logging, at the cost of reducing coverage or performing more expensive search offline.

We propose to identify a subset of input sources and shared objects that are, in a sense, *relevant* for covering program behavior. We classify various types of relevancy in terms of how an input source or a shared object can affect control flow (e.g., a conditional branch) or dataflow (e.g., state of the shared objects) in the program. Such relevancy data can be used by testing and debugging methods to reduce their recording overhead and to guide coverage.

To conduct relevancy analysis, we propose a novel framework based on dynamic taint analysis for multi-threaded programs, called *DTAM*. It performs thread-modular taint analysis for each thread in parallel during runtime, and then aggregates the thread-modular results offline. This approach has many advantages: (a) it is faster than conducting taint analysis for serialized multi-threaded executions, (b) it can compute results for alternate thread interleavings by generalizing the observed execution, and (c) it provides a knob to tradeoff precision with coverage, depending on how thread-modular results are aggregated to account for alternate interleavings. We have implemented DTAM and performed an experimental evaluation on publicly available benchmarks for relevancy analysis. Our experiments show that most shared accesses and conditional branches are dependent on some program input sources. Interestingly in our test runs, on average, only about 25% input sources and 3% shared objects affect other shared accesses through conditional branches. Thus, it is important to identify such relevant input sources and shared objects for testing and debugging.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

Keywords

Taint Analysis, Relevancy, Generalization

1. Introduction

With the advent of multi-core processors, there is great need to write parallel programs to take advantage of parallel computing resources. However, programming, debugging, and testing concurrent programs are notably difficult because of two types of inherent non-determinism in multiprocessor systems: inputs (i.e., user and system data from non-deterministic sources) and OS schedules (i.e., order of accesses to shared objects). Different inputs and schedules of concurrent threads may affect the shared object state and control-flow of programs. This implies that programmers have to reason about all possible inputs and schedules to understand the behavior of multi-threaded programs.

There are many existing efforts from both software and hardware communities to build useful tools that help program developers to address these challenges. For example, deterministic replay systems [29, 1, 19, 40] enable reproducing concurrency bugs such as data races and atomicity violations, and backward time-travel debugging. Systematic exploration techniques [11, 23] explore all feasible schedules for a given test input. Runtime techniques such as fuzzing-based testing [27, 28] focus on improving the interleaving coverage of some "meaningful" set (using random sleep delays), while trace-based analysis [2, 45, 10, 21, 42, 12, 37] focus on detecting and predicting program errors in generalizations of an observed schedule using either offline search [2, 42, 12, 37] or online analysis [45, 10, 21]. These testing and debugging techniques instrument the program to record events of interest in the test execution, e.g., inputs, accesses to shared objects, synchronization operations, context switches, etc. The recorded information are then used for detecting concurrency-related bugs, for predicting bugs in other thread interleavings, for expanding testing coverage over other thread interleavings, or for reconstructing an observed failure. To reduce the overhead of recording inputs and events during runtime, these testing and debugging efforts have proposed tradeoffs for sampling or selective recording, at the cost of reducing coverage or performing more expensive search offline to find a trace with failure. For example, to reduce the performance overhead of dynamic data-race detection, a sampling-based approach was proposed [21] to process a small percentage of shared accesses based on infrequent visits, thereby, avoiding the need to analyze every shared access executed by the program.

Though many of these previous efforts have addressed the challenges in the space of possible thread interleavings induced by shared object accesses, the space of program inputs has received little attention. For instance, deterministic replay systems [29, 1, 19, 40] record all program inputs, regardless of their effect on program behavior. Deterministic execution systems [26, 7] and systematic testing tools [23] provide determinism or testing coverage guarantee only for a given input. In addition, some concurrency bug reproduction tools [43] assume that they can provide the same input during analysis. The recording of inputs helps to reduce the search space, however at the cost of partial coverage and increased overhead (performance penalty).

To address these issues, we propose relevancy analysis to identify a subset of input sources and shared objects that can affect an execution of the multi-threaded program. We classify relevancy based on how these input sources and shared objects affect the control flow (branches) or data flow (accesses of shared objects) in the program. We also analyze how branches or shared accesses depend on input sources and shared objects.

The relevancy analysis for inputs can be used to reduce the space of inputs considered by other techniques. A possible use-case is

SIGSOFT/FSE'12, November 11-16, 2012, Cary, NC, USA

to perform relevancy analysis on representative test data as a preprocessing step before carrying out time-sensitive testing, debugging, or runtime monitoring to reduce the performance overhead. Some examples of potential benefits provided to other techniques are: (a) A deterministic replay system need not record some input source if it does not impact program behavior and it can be easily generated during replay. Thus, monitoring and recording only relevant input sources would reduce the overhead of such systems. (b) In testing, generating different input stress data only for relevant input sources would improve the test coverage effectively.

Similarly, relevancy analysis for shared objects can also provide notable benefits: (a) It can help to classify data races as malign or benign. For example, if a data race is on a shared object that does not have any effect on conditional branches, one may classify the race as benign, since it is unlikely to affect the program path. (b) It can also empower a data race predictor to relax certain read-after-write ordering constraints during the search for data races. Data race predictors such as [2, 37] use a happens-before analysis, often with a relaxed ordering of lock/unlock as long as read-after-write (RAW) constraints for shared objects are not violated. This avoids reporting false races. However, if a shared object does not affect any conditional branch, it may ignore its associated RAW constraints during race prediction, thereby finding more potential races. (c) It can also help sampling-based race detection such as [21] to focus on only relevant shared objects.

To conduct relevancy analysis effectively, we propose a novel framework based on dynamic taint analysis for multi-threaded programs, called *DTAM*. Taint analysis is shown useful in many applications, including security attack detection and prevention, information-flow policy enforcement, and software testing and debugging. Static taint analysis (e.g., [24, 39]) computes conservatively information-flow within a program, whereas dynamic taint analysis [25, 4, 33, 5, 34, 47] identifies flows that actually occur in one of the observed executions. Static analysis can produce spurious flows due to imprecision in the taint propagation, while dynamic analysis can miss potential flows due to unobserved program executions. In this work, we focus on dynamic taint analysis, and address its shortcoming in its application to multi-threaded programs.

The general idea of dynamic taint analysis is as follows: During runtime, it tags data from an external input source, propagates the taint tag along data and control flows, and then checks if tagged data is used at a target instruction/statement (e.g., target location of a jump instruction). In our DTAM framework, we use a unique id to tag data from each input source and from each shared object. Input data include the return value of system calls and data copied from kernel to user space (e.g., data read by sys_read()). Our runtime system propagates the tag along both data and control flows (described later). We check the tags on instructions corresponding to shared accesses and conditional branches, to identify input sources that are relevant. Although we target instructions corresponding to shared accesses and conditional branches, our DTAM framework is general and one can specify any target location(s) of interest.

Dynamic taint analysis for sequential programs is straightforward, and efficient implementations have been proposed in prior work [25, 4, 33, 5, 47]. However, dynamic taint analysis for multi-threaded programs on multi-core platforms remains a challenge. An implementation needs to carefully handle concurrent accesses to shared objects from multiple threads, and guarantee atomicity of original code and added instrumentation for taint propagation and checking. A naive approach would be to serialize the multi-threaded execution, and simply apply sequential dynamic taint analysis on it. However, this approach would suffer from bad performance due to serialization, i.e., loss of parallelism in the application, and from low coverage due to consideration of only one serialized (observed) trace. To address these problems, our goals in DTAM are two fold: (a) support parallel thread-modular taint analysis at runtime, (b) generalize thread-modular taint analysis results (i.e., to cover alternate OS schedules) in a post-execution (offline) stage. Although thread-modular analysis has been exploited previously, such as for procedure summarization [32, 36], and model checking [13, 18], its use in dynamic taint analysis has not been investigated so far.

We refer to the approach with serialization execution as *DTAM-serial*, and implemented it within our DTAM framework. It is built over Dytan [5], a taint analysis generic framework for sequential programs. In addition to DTAM-serial, we also propose two new approaches named *DTAM-parallel* and *DTAM-hybrid*. They comprise two main stages: (1) an online stage where each thread performs taint analysis locally, in parallel with other threads at runtime, and (2) an offline stage where the thread-modular results are aggregated, to capture taint propagations across threads. To enable thread-modular taint analysis in the first stage, we treat a shared read event by a thread as another type of input and generate a pseudo taint tag for that event. Furthermore, whenever a thread executes a shared write event, we log the shared object and the propagated taint tags, if any, for further propagation during the second stage. In the second stage, an offline merger collects the taint results for each thread and aggregates them for the multi-threaded program. It recursively replaces the pseudo taint tags on shared reads with the taint tags on shared writes (to the same object) from remote threads, until convergence.

There are two advantages of using DTAM-parallel compared to DTAM-serial. First, it can take advantage of parallelism by performing thread-modular taint analysis. Second, it can provide generalized results corresponding to many schedules. Note that dynamic taint analysis on a serialized execution (DTAM-serial) inherently follows the observed schedule, and it may require exploring a large number of serialized schedules to get adequate coverage. On one hand, DTAM-parallel implicitly captures the effects of many possible schedules when merging the thread-modular results offline. On the other hand, depending on the precision of the inter-thread propagation, there is a tradeoff involved in using this parallel approach. In addition to a one-time additional cost of an offline merge stage, DTAM-parallel approach may lead to over-tainting if it conservatively propagates the taint tag to all remote threads when there exists a shared write-read pair to the same object, even though such pairing may not be feasible due to synchronization.

To remove the imprecision of this over-approximation, we propose DTAM-hybrid which considers must-happens-before relationships due to synchronization operations. In this approach, we allow a taint tag to be propagated from a shared write in one thread to a shared read (on the same object) in another thread, only if there is no must-happens-before ordering between the read and the write. By considering synchronization operations, DTAM-hybrid approach enables us to collect generalized results (i.e., not limited to the observed schedule) in comparison to DTAM-serial, while addressing some over-tainting issues in DTAM-parallel. While these ideas are similar to techniques for static dataflow analysis and predictive analysis for multi-threaded programs, we use them to provide a flexible framework for dynamic taint propagation that can tradeoff precision and coverage.

The proposed technique DTAM can give both false positives (i.e., spurious relevancy due to offline merging) and false negatives (i.e, missing relevancy due to unobserved program paths). However, as mentioned earlier, it can still be useful in many best-effort applications such as debugging, stress testing, runtime monitoring, to reduce logging overhead and improve coverage. To summarize, the contributions of this paper are as follows:

- We propose a dynamic relevancy analysis for multi-threaded programs to identify a subset of input sources and shared objects that can affect shared-object state or control flow of the

programs. Relevancy is classified in terms of their effect on branches and on shared accesses. Such an analysis can supplement other techniques to provide the following benefits: reduce recording overhead, improve testing coverage, improve data race prediction, and help classify benign/malign data races.

- We present a dynamic taint analysis framework for multi-threaded programs (DTAM). It uses parallel thread-modular taint analysis, with an offline aggregation for propagating taints across threads. It provides flexibility to tradeoff precision and coverage, ranging from results over only the observed serialized execution to generalized results over other thread schedules. Although we focus here on its use for performing relevancy analysis, it is a general framework that can be used for other applications of taint analysis for multi-threaded programs.

The rest of the paper is organized as follows. We start with background and notation in Section 2. Section 3 presents the main ideas in relevancy analysis. Section 4 describes our proposed dynamic taint analysis framework, which we use to perform relevancy analysis, and Section 5 presents its implementation. We discuss an empirical evaluation in Section 6. Finally, we compare our work with other related efforts in Section 7 and conclude in Section 8.

2. Background

A multi-threaded program consists of a set of concurrently executing threads. The threads communicate with shared objects, some of which are used for synchronization such as locks and signals. A trace π of a program is a total ordered sequence of observed events corresponding to various thread operations such as shared accesses and external API calls. Each event e of the sequence, i.e., $e \in \pi$ is carried out by some thread denoted as $tid(e)$ at a thread program location $loc(e)$. These events include the following:

- `write/read(t,x)`: write/read by t on a shared object x
- `nd(t)`: a non-deterministic external input API call by t (including hardware interfaces, system calls)
- `branch(t)`: a conditional branch taken by t
- `wait/notify(t,s)`: wait/notify by t on a signal s
- `fork(t,t')/thread_start(t')`: t forks a thread t'
- `join(t,t')/thread_end(t')`: t waits until t' ends

A *shared instruction* is a $loc(e)$ where a `write/read` occurs.

Happens-before. Given a trace π of a program, and events $e, e' \in \pi$, we say e *happens-before* e', i.e., $e \preceq e'$, if e is *observed* before e' in the trace. We say e *must happens-before* e' (i.e., causally-ordered), denoted as $e \prec e'$, if $e \preceq e'$ holds and one of the following holds:

- e, e' belong to the same thread
- $e = notify(t, s)$ and $e' = wait(t', s), t \neq t'$
- $e = fork(t, t')$ and $e' = thread\_start(t'), t \neq t'$
- $e = thread\_end(t)$ and $e' = join(t', t), t \neq t'$
- $\exists e_1 \in \pi.\ (e \prec e_1 \preceq e')$ or $\exists e_2 \in \pi.\ (e \preceq e_2 \prec e')$

A must happens-before relation can be maintained easily using vector clocks [22, 9]. A vector clock of a thread, denoted as $VC(t)$, records the clocks of all threads. Whenever a non-locking synchronization occurs, vector clocks are updated. Each event is time stamped with a vector clock. (Implementation details of vector clocks can be found [30].)

Effect Chains. We now define source, sink and conduit in a multi-threaded program w.r.t. effect chains.

Source: A source is a non-deterministic external input API or a shared object associated with the event `nd` or `read`, respectively. We refer to the former source as *input* and the latter source as *shared object*. (In the sequel, we use *inputs* to denote input sources.)

Sink: A sink is a $loc(e)$, where event e is either a `read/write` on a shared object, or a `branch`. We refer to the former sink as *shared access* and the latter sink as *branch*.

Conduit: A conduit determines propagation of the effect from a source to a sink through a read-after-write dependency. If an effect propagates through a write, followed by a read (on the same shared object), we refer to the conduit, i.e., the pair of events, as *shared accesses*. If an effect propagates through a branch, we refer to the conduit as *branch*.

Let $A, B \in$ {*source*, *conduit*, *sink*}.
$A \rightarrow B$: A affects or propagates an effect to B
$A \not\rightarrow B$: A does not affect or propagate an effect to B.
$\not A \rightarrow B$: B does not get an effect propagated through A.
An effect propagation chain is shown as: *source* $\rightarrow$ {*conduit*}* $\rightarrow$ *sink*. For the above chain, we also say that the conduit/sink is *dependent* on the source.

3. Relevancy Analysis

We discuss relevancy of two sources of non-determinism in multi-threaded programs, i.e., inputs and shared objects, with respect to how they impact other shared accesses and branches (in Sections 3.1-3.2). Going in the other direction (from sinks to sources), we also discuss the dependency of sinks (shared accesses and branches) on various sources (in Section 3.3).

3.1 Relevant Inputs (Sources)

In multi-threaded programs, inputs can affect shared object state, control flow, or both. The inputs correspond to hardware interfaces such as network, disk, user keyboard, and non-deterministic system API such as `gettimeofday`, `sys_read`, `rdtsc` instructions, etc.

We propose six possible relevancy types for inputs, based on how they influence branches or shared accesses: i_irrel, $i\_\bar{b}s\bar{b}$, $i\_\bar{s}b\bar{s}$, $i\_sb\bar{s}$, $i\_\bar{s}bs$, and i_sbs. These types are described in Table 1. The key to understanding the mnemonics for these types is as follows: i stands for inputs, b stands for branches, and s stands for shared objects or accesses. Here, the positive forms (b, s) mean that there is influence, while the negative forms ($\bar{b}$, $\bar{s}$) mean that there is no influence. For example, $i\_\bar{b}s\bar{b}$ denotes that the input does not influence any branch, either directly or through a shared access. The type i_irrel is shorthand to denote inputs that do not influence any branch or any shared access. The remaining types denote cases where an input affects a branch. Among these, we distinguish cases where the branch is (or is not) affected through a shared access (mnemonically, b is preceded by s or $\bar{s}$, respectively). We also distinguish cases where the branch itself affects (or does not affect) a shared access (mnemonically, b is followed by s or $\bar{s}$, respectively). This leads to four types, denoted mnemonically as $i\_\bar{s}b\bar{s}$, $i\_sb\bar{s}$, $i\_\bar{s}bs$, and i_sbs.

Table 1: Relevancy Types for Inputs

Type	Description
i_irrel	Affects neither any branch nor any shared access.
$i\_\bar{b}s\bar{b}$	Affects some shared access but not any branch.
$i\_\bar{s}b\bar{s}$	Affects some branch but not any shared access.
$i\_sb\bar{s}$	Affects some branch through some shared access, but the branch does not affect any shared access.
$i\_\bar{s}bs$	Affects some branch not through any shared access, and the branch affects some shared access.
i_sbs	Affects some branch through some shared access, and the branch affects some shared access.

These six input types are also shown pictorially in Figure 1(a), where the circles show the different domains (I:Inputs, BR:Branches, SH:Shared Accesses). Note that I are the sources, and BR and SH can be either conduits or sinks, and various intersections denote how inputs influence them. The inputs that do not affect any shared access and do not affect any branch, denoted by type i_irrel, are regarded as *irrelevant*, and the rest are regarded as *relevant*.

The goal of relevancy analysis is to identify relevant inputs which may affect shared-object state and control flow of multi-threaded programs. Information about relevant inputs can have many potential uses in monitoring and analyzing multi-threaded programs. For example, deterministic replay systems [29, 1, 19, 40] need not monitor irrelevant inputs for which natural (unlogged) input during replay can be used since they would not affect the shared-object state of the program nor the program path (e.g., taken branches). Similarly, deterministic execution systems [26, 7], systematic schedule testing tools [23], and concurrency bug reproduction tools [43] also can be augmented with system support for monitoring only relevant inputs. Relevant input analysis can also have a significant effect on testing, as one need not explore the space of irrelevant inputs. Given limited testing time, one can selectively alter only relevant inputs to provide improved test coverage.

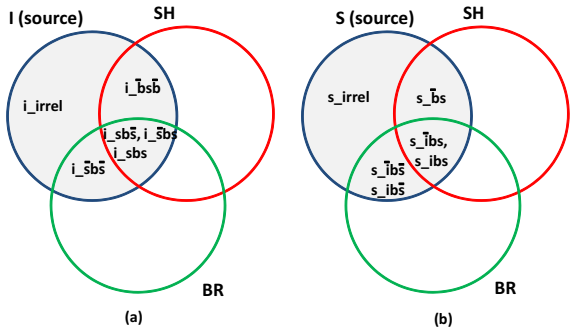

Figure 1: Relevancy Types for (a) Inputs (I), (b) Shared objects (S), based on their effect on branches (BR) and shared accesses (SH).

The relevant input types of interest may vary for different applications. We highlight some of the potential uses:

- One may be interested in the subset of inputs which affect branches only but not any shared object, i.e., those that do not influence the interleaving coverage (ordering of shared accesses). Such inputs, identified by the type $i\_\bar{s}b\bar{s}$, are included in the intersection of I and BR.
- Similarly, one may be interested in the subset of inputs that do not affect branches but only some shared accesses, i.e., that do not influence any program path (but may be just the outputs). Such inputs, identified by the type $i\_\bar{b}s\bar{b}$, are included in the intersection of I and SH. From the point of view of deterministic replay, such inputs may also be irrelevant.
- One may want to obtain a subset of inputs which affect some shared accesses that are control-dependent on inputs, i.e., those that may influence the local branches but not the branches of the other threads. Such inputs, identified by the type $i\_\bar{s}bs$, are included in the intersection of I, BR and SH.

Based on above potential uses, we also identify some useful groups of relevant types, as shown in Table 2. We discuss the significance of the grouping shortly, but first we provide illustrations of the types of inputs as shown in Table 3. For each type (shown in Column 1), we show the effect propagation chain in Column 2. In Column 3, we use two threads T1 and T2, communicating with shared objects S and S', and local objects x, y, z, and highlight the corresponding input types with code snippets. In the rest of the columns, we identify the membership of each type in the groups.

The groups of input types we consider are as follows:

- $I\_\bar{s}b$: inputs that affect some branches not through any shared access
- I_bs: inputs that affect some shared accesses through some branches
- I_b: inputs that affect some branches (shared object may or may not be a conduit)
- I_s: inputs that affect some shared accesses (branch may or may be a conduit)
- I_b_s: inputs that affect either some shared accesses or some branches

Note that both I_b and I_s include I_bs, and I_b_s subsumes all relevant input types. Our experimental evaluation (described later) uses these groups.

Table 2: Input Relevancy Groups

Group	i_irrel	$i\_\bar{b}s\bar{b}$	$i\_\bar{s}b\bar{s}$	$i\_sb\bar{s}$	$i\_\bar{s}bs$	i_sbs
$I\_\bar{s}b$			X		X	
I_bs					X	X
I_b			X	X	X	X
I_s		X		X	X	X
I_b_s		X	X	X	X	X

Table 3: Types and Groups of Inputs Relevancy

Inputs affecting shared accesses (SH), and/or branch (BR)		Thread Examples (x,y,z: local; S,S': shared)		Input Relevancy (Groups of types)				
Types	Effect Chains	T1	T2	I_s̄b	I_bs	I_b	I_s	I_b_s
i_irrel	I→ ~~SH~~, ~~BR~~							
i_b̄sb̄	I→ ~~BR~~ → SH → ~~BR~~	x:=nd(); S:=x;	y:=S;				X	X
i_s̄bs̄	I→ ~~SH~~ → BR → ~~SH~~	x:=nd(); if(x)		X		X		X
i_sbs̄	I→ SH → BR→ ~~SH~~	x:=nd(); S:= x;	y:=S; if(y){z:=1}			X	X	X

i_s̄bs	I→ ~~SH~~ → BR→ SH	x:=nd();
		if(x){
		S:=x; }
			y:=S;	X	X	X	X	X

i_sbs	I→ SH → BR → SH	x:=nd(); S:= x; S':=0;	y:=S; if(y) {z:=S'}		X	X	X	X

3.2 Relevant Shared Objects (Sources)

Like inputs, *shared objects* can influence other shared-object state, or control flow, or both.

Table 4: Relevancy Types for Shared Objects

Types	Description
s_irrel	Affects neither a branch nor another shared access.
$s\_\bar{b}s$	Affects some shared access but not any branch.
$s\_\bar{i}b\bar{s}$	Affects some branch without propagating the effect of any input, but the branch does not affect any shared access.
$s\_ib\bar{s}$	Affects some branch with propagating the effect of some input, but the branch does not affect any shared access.
$s\_\bar{i}bs$	Affects some branch without propagating the effect of any input, and the branch affects some shared access.
s_ibs	Affects some branch with propagating the effect of some input, and the branch affects some shared access.

Table 5: Shared Object Relevancy Groups

Group	s_irrel	$s\_\bar{b}s$	$s\_\bar{i}b\bar{s}$	$s\_ib\bar{s}$	$s\_\bar{i}bs$	s_ibs
$S\_\bar{i}b$			X		X	
S_bs					X	X
S_b			X	X	X	X
S_s		X			X	X
S_b_s		X	X	X	X	X

We propose six possible relevancy types for shared objects (as sources), based on how they influence branches or shared accesses: s_irrel, $s\_\bar{b}s$, $s\_\bar{i}b\bar{s}$, $s\_ib\bar{s}$, $s\_\bar{i}bs$, and s_ibs. These are described in Table 4, and the mnemonics used here are similar to those for inputs (as sources) described earlier. Also, these types are shown pictorially in Figure 1(b). The shared objects that do not affect

Table 6: Types and Groups for Shared Object Relevancy

Shared objects affecting shared accesses (SH), branches (BR)		Thread Examples (x,y,z: local; S,S': shared)		Shared Object Relevancy (Groups of types)				
Types	Effect Chains	T1	T2	S_$\bar{i}$b	S_bs	S_b	S_s	S_b_s
s_irrel	~~SH→SH, BR~~							
s_$\bar{b}$s	SH→~~BR~~ → SH	x:=2; S:=x;	y:=S;				X	X
s_$\bar{i}$b$\bar{s}$	~~I→~~ SH → BR → ~~SH~~	x:=S; if(x)	S:=2	X		X		X
s_ib$\bar{s}$	I→ SH → BR→ ~~SH~~	x:=nd(); S:= x;	y:=S; if(y){z:=1}			X		X

s_$\bar{i}$b$\bar{s}$	~~I→~~ SH → BR→ SH	x:=S;
		if(x){
		S:=x; }
			S:=2;	X	X	X	X	X

s_ibs	I→ SH → BR→ SH	x:=nd(); S:= x; S':=0;	y:=S; if(y) {z:=S'}		X	X	X	X

any shared accesses and do not affect any branches, indicated by type s_irrel, are regarded as *irrelevant*. The remaining types are regarded as *relevant*.

Some potential applications of relevancy analysis for shared objects are as follows:

- One may be interested to know if a shared object can influence a branch but not another shared access. Such shared objects, identifiable by types $s\_\bar{i}b\bar{s}$ and $s\_ib\bar{s}$, are included in the intersection of S (source) and BR (sink).
- One may like to know if a shared object can not influence any branch. Such shared objects, identifiable by type role $s\_\bar{b}s$, are included in the intersection of S (source) and SH (sink). For example, if a race is on a shared object that does not have any effect on control flow, one may classify the race as benign; otherwise, potentially malign.
- It can also empower a data-race detector to relax certain read-after-write ordering constraints while predicting data races. Data race detectors such as [2, 37] use happens-before analysis, often with a relaxed ordering of lock/unlock as long as read-after-write (RAW) constraints are not violated. Such constraints are enforced to avoid potentially false (infeasible) data races. However, if a shared object does not affect any branch, it may ignore those constraints during race prediction to find more potential races.
- A sampling-based race detection such as [21] can also benefit by focusing on a smaller set of shared objects. (In our experimental data we found that, on average, about 12% shared objects affect some branch).

Again, based on potential uses, we have grouped the shared types into the following five groups, also shown in Table 5.

- $S\_\bar{i}b$: shared objects that affect some branches without propagating the effect of any input
- S_bs: shared objects that affect some other shared accesses through some branches
- S_b: shared objects that affect some branches (input can be a source)
- S_s: shared objects that affect some other shared accesses (branch can be a conduit, input can be a source)
- S_b_s: shared objects that affect either some other shared accesses or some branches

Like types for inputs, we also illustrate the various types for shared objects using small examples, as shown in Table 6.

3.3 Dependencies of Sinks

So far we have addressed relevance of sources (Inputs, Shared Objects) by following chains of influence from these sources to sinks (Branches, Shared Accesses). Going in the other direction, we can also identify dependencies of sinks, i.e. how shared accesses and branches depend on inputs/branches/shared objects.

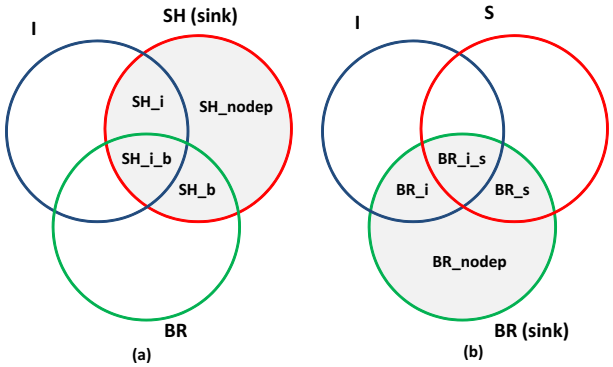

Figure 2: Dependencies of Sinks: (a) Shared accesses (SH), (b) Branches (BR) based on their dependency on inputs(i)/branches(b)/shared objects(s).

We identify four types of shared access dependencies, as shown in Figure 2(a).

- SH_i_b: shared accesses that are dependent on some branches and on some inputs
- SH_i: shared accesses depending only on some inputs (not dependent on any branch)
- SH_b: shared accesses depending only on some branches (not dependent on any input)
- SH_nodep: shared accesses that depend neither on any input nor on any branch

Similarly, we identify four types of branches dependencies, as shown in Figure 2(b).

- BR_i_s: branches that depend on some inputs and some shared objects
- BR_i: branches depending only on some inputs
- BR_s: branches depending only on some shared objects
- BR_nodep: branches that depend neither on any input nor on any shared object

4. Dynamic Taint Analysis for Multi-threaded Programs

In this section, we describe how dynamic taint analysis can be used for relevancy analysis and describe some challenges in multi-threaded programs in Section 4.1. Then, we provide our proposed dynamic taint analysis approaches for multi-threaded programs, namely: DTAM-serial, DTAM-parallel, and DTAM-hybrid.

4.1 Taint Analysis for Relevancy Analysis

There are three main steps in traditional sequential dynamic taint analysis: (1) *tagging*, i.e., identifying data from external inputs and marking them as tainted, (2) *propagating* the taint tag along the data and control flow through the program, and (3) *checking* whether tainted data is used unsafely. We use this dynamic taint tag propagation and check scheme to enable relevant analysis. We also generate a unique taint tag on data from each input and from each shared object reads, and propagate it along data-flow and control-flow. However, in our relevancy analysis, we do not use taint analysis to focus on security policy or detect security attacks; instead, we perform runtime checks on shared accesses and/or branches. For example, when a taint tag from input data propagates to a shared access, through a branch or directly, we say that the input affects shared-object state. We classify it according to our relevancy types.

Extending a sequential dynamic taint analysis to multi-threaded programs is non-trivial. One needs to consider concurrent accesses to a shared object with additional guarantee that a shared instruction and instrumented code for taint propagation are executed atom-

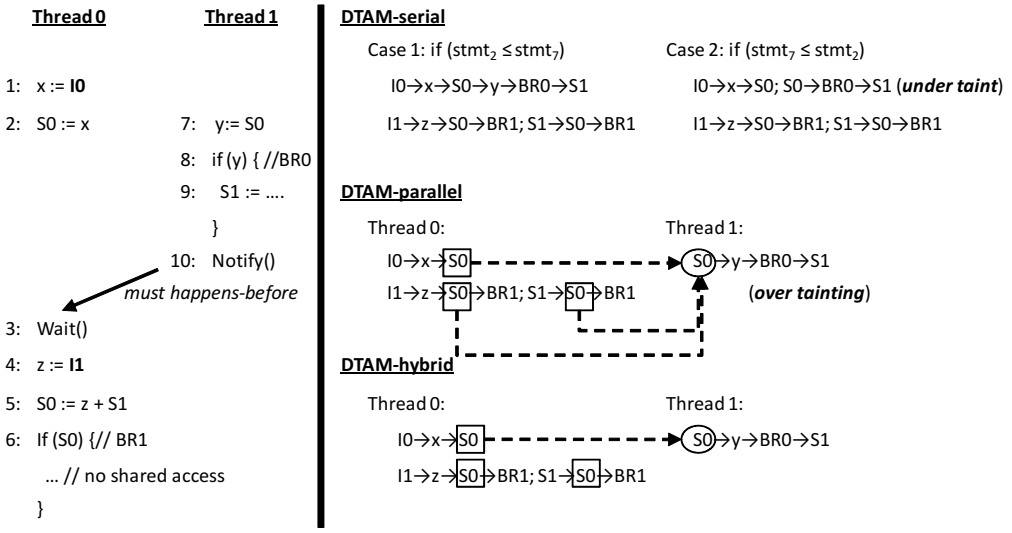

Figure 3: Example of DTAM Analysis. On the left code example, I0 and I1 are program inputs; x, y, and z are local objects; $S0$ and $S1$ are shared objects where statements 2 and 7 have a data-race (i.e., statement 2 can happen before ($\prec$) statement 7, or vice-versa); and there exists a must-happens-before order between statements 10 and 3 enforced by Notify() and Wait(). On the right, we show the results of DTAM-serial, DTAM-parallel, and DTAM-hybrid approaches. DTAM-serial depends on the serialized schedule, so it may lead to under-tainting as in case 2. For DTAM-parallel and DTAM-hybrid, the dotted arrows represent the result of offline analysis. Since DTAM-parallel does not consider a must-happens-before order constructed by synchronization operations, it may result in over-tainting.

ically. In addition to thread-local data and control flow, one also need to consider *inter-thread flow* for taint propagation, where in a taint tag can propagate from one thread to another through read-after-write (RAW) shared object dependency.

4.2 DTAM-serial

One straightforward solution for multi-threaded programs is to serialize the multi-threaded execution, and then propagate taint tags along the serialized schedule while ensuring atomicity between a shared instruction and instrumented code for taint propagation. We call this approach DTAM-serial.

In DTAM-serial approach, there exists a single taint map for shared objects which holds propagated taint tags for each shared object. This map is shared by all threads so that one thread can see the effect of taint propagation by the other. In addition, each thread has its own taint map for registers which contains thread local context information. Though DTAM-serial is easy to reason about and implement, it has two drawbacks. First, it cannot exploit application-level parallelism and therefore, it is much slower than normal executions even without considering instrumentation costs. Second, DTAM-serial is carried out on the observed serialized schedule, so it may lead to *undertainting* w.r.t. other possible interleavings on the same inputs.

We illustrate the above issue in Figure 3. On the left code example, I0 and I1 are program inputs; x, y, and z are local objects; S0 and S1 are shared objects. Note that statements 2 and 7 have a data-race. On the right, we show the result of DTAM-serial approach based on two different schedules. If statement 2 happens-before statement 7 (Case 1), then taint tag for I0 is propagated to y, BR0, and $S1$ in Thread 1 via read-after-write (RAW) shared accesses of S0. However, if statement 7 happens before statement 2 (Case 2), then the taint tag for I0 would not be propagated to Thread 1. Thus, serializing an execution potentially leads to undertainting.

4.3 DTAM-parallel

To overcome the two limitations of DTAM-serial, i.e., serialization and undertainting, we propose a parallel dynamic taint analysis technique, namely DTAM-parallel. In addition to thread-local data and control flows, a taint tag can be propagated from one thread to another through write and read accesses to the same shared object in multi-threaded programs (RAW dependency). Therefore, it is necessary to deal with this additional inter-thread flow in dynamic taint analysis for multi-threaded programs. For each thread, the shared writes are the points where a local taint tag can be potentially propagated to the other threads, and the shared reads are the ones where a remote taint tag can be propagated into a thread.

Like DTAM-serial, DTAM-parallel marks each input data with a unique *input taint* tag. However, in DTAM-parallel, each thread maintains its own taint map for shared objects and performs thread-modular taint propagation. When a thread performs a shared read, it creates a unique *pseudo taint tag* and propagates it as if the shared read was treated as an external input.During offline analysis, this pseudo taint tag will be replaced by taint tags, which can be propagated via shared writes (performed on the same object) by remote threads. To obtain this additional information, when a thread performs a shared write, it logs the shared object identifier (such as memory address) and corresponding taint tags. On completion of thread-modular taint propagation, an offline analysis collects the taint results of each thread and aggregates them by replacing the pseudo taint tags on shared reads with the propagated taint tags on shared writes (performed on the same object) by remote threads.

There is a performance-precision tradeoff in using the DTAM-parallel approach. By performing taint analysis for each thread, it can take advantage of parallelism, but the offline merging step may lead to over-tainting because it does not consider must-happens-before relation due to synchronization operations. It conservatively propagates the taint tag from one thread to another whenever there exists a shared write-read pair to the same shared object. However, if the shared write happens after the shared read, the taint tag should not propagate from write to read.

Although not precise, this conservative propagation provides generalization over other interleavings on the same inputs. The proposed DTAM-parallel approach implicitly captures the effects of many possible interleavings when merging the thread-modular results offline; however, this comes at the cost of over-tainting. In the following section, we discuss our final technique, namely DTAM-hybrid, which takes into account the must-happens-before relationship enforced by synchronization operations.

Figure 3 shows this tradeoff. In DTAM-parallel, Thread 0 logs

taint tag I0 and I1 on shared writes (on statements 2 and 5, respectively), and Thread 1 treats the shared read as a pseudo input on statement 7. This allows two threads to perform thread-modular taint analysis in parallel. Then, the offline analysis merges the results by replacing pseudo taint tags with the propagated input tags on shared accesses to the same object, as represented by the dotted arrows in the example. However, DTAM-parallel does not consider a must-happens-before order between statements 10 and 3 enforced by Notify() and Wait(), thus it could result in over-tainting, i.e., it may conclude that I1 can also affect y, BR0, and S1.

4.4 DTAM-hybrid

DTAM-hybrid tracks synchronization operations and takes into account the must-happens-before relationship to address the above over-tainting problem. Such types of synchronization prevent local taint tags from being propagated from a shared write to a shared read, if the shared read must happen before the shared write. We also record vector clocks for each read and write event, in addition to the shared events and tags. Such information is used for determining the must-happens-before relationship during aggregation step (as discussed in the next section).

Figure 3 shows the difference between DTAM-parallel and DTAM-hybrid. DTAM-hybrid is aware of the must-happens-before order between Notify() in Thread 1 and Wait() in Thread 0. Therefore, during the offline merge stage, it does not allow taint tag of I1 in Thread 0 to be propagated to Thread 1.

In Table 7, we present the results of taint analysis in identifying various relevant inputs and shared objects for the example in Figure 3. Similarly, in Table 8, we present the results of taint analysis in identifying various dependencies of shared accesses and branches. Note, all branches and shared access have dependencies on inputs and/or shared objects.

As one would expect, DTAM-hybrid gives the most precise[1] result compared to DTAM-serial and DTAM-parallel.

Table 7: Input and Shared object relevancy

dtam	input groups					shared objects groups				
	$I\_\bar{s}b$	I_bs	I_b	I_s	I_b_s	$S\_\bar{i}b$	S_bs	S_b	S_s	S_b_s
serial (1)		I0	I0,I1	I0,I1	I0,I1		S0	S0,S1	S0,S1	S0,S1
serial (2)			I1	I0,I1	I0,I1	S0		S0,S1	S0,S1	S0,S1
parallel		I0,I1	I0,I1	I0,I1	I0,I1		S0,S1	S0,S1	S0,S1	S0,S1
hybrid		I0	I0,I1	I0,I1	I0,I1		S0	S0,S1	S0,S1	S0,S1

Table 8: Shared access and branch dependency

dtam	shared access			branch		
	SH_i_b	SH_i	SH_b	BR_i_s	BR_i	BR_s
serial (1)	S1	S0		BR0,BR1		
serial (2)		S0	S1	BR1		BR0
parallel	S1	S0		BR0,BR1		
hybrid	S1	S0		BR0,BR1		

4.5 Aggregation of thread-modular results

In this section, we describe how we perform the offline merging. We first introduce some notation:

- I: a set of all input taint tags (i.e., inputs)
- S: a set of all pseudo taint tags (i.e., shared object reads)
- t: an input or a pseudo taint tag, i.e., $t \in I \cup S$
- $t.r$: read access event corresponding to pseudo taint tag t
- $t.obj$: shared object corresponding to pseudo taint tag t
- $t.lw$: last observed thread local write event to $t.obj$ ($t.lw \preceq t.r$)
- $t.gw$: last observed global write event to $t.obj$ ($t.gw \preceq t.r$)
- $t.A$: a set of all shared write events by other threads to $t.obj$
- $w.T$: a set of taint tags propagated (during thread-modular taint analysis) to write event w

The goal of aggregation is to obtain a transitive dependency on inputs taint tags (or pseudo taint tags) for a given set of taint tags propagated at sinks in a thread-modular taint analysis.

For a given relevancy group R (e.g., I_b, S_b), let $R.d$ denote a set of pseudo taint and input taint tags that were relevant for R as obtained from the thread-modular taint analysis.

We then obtain a set of all input taint tags that are relevant for R, denoted as $R.I$, as follows:

$$R.I = \bigcup_{t \in R.d} t.I \tag{1}$$

where $t.I$ is the set of all input taint tags that can possibly affect t (computed as described below).

Similarly, we obtain a set of all pseudo taint tags that are relevant for R, denoted as $R.S$, as follows:

$$R.S = \bigcup_{t \in R.d} t.S \tag{2}$$

where $t.S$ is the set of all pseudo taint tags that can possibly affect t.

We now describe how we obtain $t.I$ and $t.S$ for a tag t. Let $t.W$ denote a set of all write accesses that can possibly affect the read of $t.obj$. In the following, we define $t.W$ in such a way that we can uniformly handle DTAM-serial/parallel/hybrid.

$$t.W = \begin{cases} \{t.gw\} & serial \\ \{t.lw\} \cup t.A & parallel \\ \{t.lw\} \cup \{w | w \in t.A \wedge (t.r \not\prec w)\} & hybrid \end{cases} \tag{3}$$

We then compute $t.I$ and $t.S$ recursively using $t.W$ as follows:

$$t.I = \begin{cases} \{t\} & \text{if } t \in I \\ \bigcup_{w \in t.W} (\bigcup_{t' \in w.T} t'.I) & \text{if } t \notin I \end{cases} \tag{4}$$

$$t.S = \begin{cases} \{\} & \text{if } t \in I \\ \bigcup_{w \in t.W} (\bigcup_{t' \in w.T} t'.S) & \text{if } t \notin I \end{cases} \tag{5}$$

5. Implementation

In this section, we describe our implementation for relevancy analysis based on dynamic taint analysis. We first discuss a profile-based approach to identify shared instructions in Section 5.1 and then discuss the implementation of DTAM approaches in Section 5.2.

5.1 Profiling Shared Instructions

For a given program, DTAM needs to identify a set of shared instructions (i.e., program locations of read/write events) for two reasons: First, the DTAM approaches proposed in Section 4 rely on shared instructions for inter-thread data flow based (RAW dependency) taint propagation. For example, DTAM-serial ensures atomicity between shared accesses and instrumented code. For DTAM-parallel and DTAM-hybrid, the pseudo taint tags are generated at shared read accesses, and the taint tags are logged during shared write accesses. Second, the relevant analysis perform taint checks on the shared accesses as we are interested in the inputs and shared objects that can affect the outcome of the shared accesses. The process of identifying shared instructions can be done through profiling or static analysis. In our implementation, we use

[1] DTAM-hybrid may still have overtainting problem if the causal-ordering ($\prec$) permits infeasible permutation of trace events. Although one may avoid the problem by using maximal-causal models [35], one may incur low coverage.

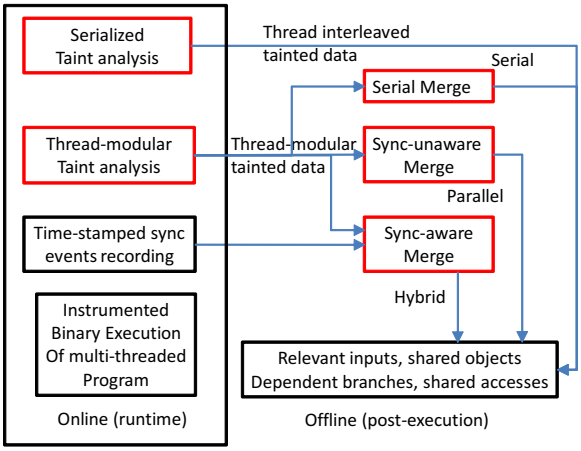

Figure 4: DTAM overview

a profile-based approach, where we execute the programs multiple times with representative test data and collect the set of shared instructions, along with its type of operation (write or read).

We used PIN [20], a dynamic binary translation tool to instrument and profile application x86 executables. We maintain metadata for each byte of location address, which represents the access history of each thread. An instruction is marked *shared* if the instruction accesses a shared object which has been accessed by other threads. Instead of using instruction address, we name each instruction the tuple of loaded image name and offset so that we can refer to each instruction uniformly for different profiled runs (e.g., a dynamically linked library may be loaded into different addresses).

5.2 DTAM

We implemented the proposed DTAM approaches (DTAM-serial, DTAM-parallel, and DTAM-hybrid) based on Dytan [5] which provides a generic framework for dynamic taint analysis. Dytan supports dynamic taint analysis for an x86 executable by instrumenting it on-the-fly to produce an instrumented executable, using the PIN [20] tool. Dytan maintains taint tags for each byte of location address and supports allocating virtually any number of taint markings, which allows us to specify different taint marking for different input sources. Dytan also supports both explicit (data-flow based) and implicit (control-flow based) information flow. For control-flow based taint propagation, Dytan performs static analysis on the binary to find immediate post-dominator of a conditional branch. All the statements that belong to the paths starting at the conditional branch and ending with its immediate postdominator are conservatively marked as tainted if the conditional branch depends on the tainted data. In addition to leveraging the default functionalities in Dytan, we extended the framework to support DTAM as follows.

First, Dytan was originally developed for supporting sequential programs. For sequential programs, there is no inter-thread taint propagation. However, concurrent accesses to shared location locations and corresponding metadata updates should be carefully handled in multi-threaded programs. In DTAM-serial mode, we ensure atomicity between shared accesses and taint propagation using locks, as all threads share a single taint map for shared objects. In DTAM-parallel and DTAM-hybrid, each thread maintains its own taint map for shared objects and performs thread-modular taint propagation. Note that inter-thread taint propagations are handled by the offline merge stage. On the thread-modular data, we can also do a serialized merging (as shown in Eq 3) to obtain DTAM-serial data. Figure 4 gives an overview of the implementations of the various DTAM approaches.

Second, as our goal is to identify relevant inputs, DTAM taints all the program inputs. This is not true for traditional dynamic taint analysis tools including Dytan, as most of them refer to some (not all) inputs that stem from external sources such as network, disk, user keyboard, etc. to be tainted. (Traditional DTA tools were primarily used to track untrusted user input data.) For example, DTAM considers the return value of system calls such as `gettimeofday, rdtsc` to be tainted because its value can have an effect on shared state or control flow of the program. In contrast, Dytan clears the taint tag for the target registers.

Third, DTAM supports precise control-flow based taint propagation at the flag (bit) granularity of status register (e.g. EFLAGS). In Dytan, like other registers, taint tags are maintained for the whole EFLAGS status register, thus Dytan cannot differentiate the taint tag for each flag bit such as CF(carry flag), PF(parity flag), and ZF(zero flag). However, most x86 instructions do not affect all the flags in the status register, so flag-sensitive taint propagation for the status register can lead to better precision.

6. Experimental Results

We now report evaluation of DTAM approaches for relevancy analysis. We begin with describing our evaluation methodology. Then, we compare the result of relevancy analysis for different DTAM approaches. Last, we show the distribution of shared accesses and branches in terms of their dependency on inputs, branches, and shared objects.

Experimental Setup. We ran all experiments on a 2.8GHz 2-core Xeon processor with 4GB of RAM running Linux 2.6.9 kernel. We used two sets of benchmarks. The first set consists of six desktop applications from open source repositories [17, 16], namely: aget, pfscan, pnscan, pbzip2, fastspy, and axel. The test setup for each application is as follows: *aget* downloads a 47KB file in parallel; *pfscan* searches in parallel for the string "debug" in a directory with 30KB of sources files; *pnscan* scans TCP port in parallel to find running web servers; *pbzip2* compresses a 0.8KB log file in parallel; *fastspy* scans port 10 to 20 for a given IP address; and *axel* downloads a 47KB file. The second set consists of three representatives of scientific application with different characteristics from PARSEC-2.1 suite [31] [2]: *blackscholes*, *canneal*, and *streamcluster*. We used *simsmall* input sets for the evaluations.

Table 9: Characteristics of multi-threaded applications.

Apps (LOC, Lang)	N	E	I_{dyn}	I_{sc}	SH	S	ON	OFF
`aget` (1.1K, C)	4	84K	318	65	35K	8.5K	9	18
`pfscan` (1K, C)	3	47K	197	49	37K	869	17	30
`pnscan` (1.5K, C)	2	6K	776	66	2K	931	32	0.1
`pbzip2` (5K, C++)	3	5K	252	55	712	1K	28	0.05
`blackscholes` (800, C)	3	2K	192	39	68	369	5	0.05
`canneal` (1.7K, C)	3	7K	204	43	4K	717	5	0.1
`streamcluster` (1.2K, C++)	5	20K	2K	44	10K	804	20	4
`fastspy` (1.2K, C)	12	73K	740	54	28K	7K	9	71
`axel` (4K, C)	7	64K	1568	65	32K	11.5K	23	4

In Table 9, we report the run time characteristics for the tests conducted for the applications listed in Column 1. The number of threads (N, Column 2) was between 2 to 12, and the number of observed events was between 2K to 84K (E, Column 3). We use I_{dyn} to denote the number of `nd` events (Column 4), ranging from 192 to 2K. We count each instance of system calls (return value, data copied into user-level process) and nondeterministic instructions (e.g., rdtsc) as input source. The static count of inputs (Column 5, I_{sc}) is the count of $loc(e)$ where e is `nd` event, ranging from 39 to 66. It is obtained by combining the effect of multiple dynamic instances (i.e., `nd` events). We use SH to denote the number of `read/write` events (Column 6), ranging from 68 to 37K.

[2] Blackscholes has coarse-grained data- parallel parallelization with low sharing; Canneal has fine-grained unstructured parallelism with high frequent sharing pattern; and Streamcluster has medium-grained and medium level of sharing characteristics.

The number of shared objects (Column 7, S) refers to the distinct objects, where each object was accessed by at least two threads. We ran each application 20 times first to profile a set of shared instructions. The online thread-modular taint analysis (Column 8, ON) took between 5 to 32 minutes, about 16 minutes on average per application. The offline aggregation of the thread-modular taint results (Column 9, OFF) took between a few seconds to 70 minutes, 15 minutes on average per application. These times (shown in min.) can be improved further with a better implementation.

Results of Relevancy Analysis. For comparing different DTAM approaches on the same observed trace, we use the same thread-modular data for aggregation for serial/parallel/hybrid. The left half of Figure 5 shows the results of relevant input analysis based on DTAM-hybrid. Results for DTAM-serial and DTAM-parallel show similar trends, and are not shown. Each bar represents the ratio of relevant inputs to the total number of inputs (I_{sc}) for a given relevant input type. For aget, $I\_\bar{s}b$ is 9% (=6/65), I_bs is 28% (=18/65), I_b is 28% (=18/65), I_s is 43% (=28/65) and I_b_s is 43% (=28/65), resp. The average of each bar is shown rightmost.

For DTAM-hybrid (and similarly for parallel and serial), on average, about 15% of inputs are categorized into relevant group $I\_\bar{s}b$ that affects branches not through any shared access, about 25% of inputs are categorized into group I_bs (and I_b). The percentage increases to 40% on average for I_s (and I_b_s) on our benchmarks. This implies that about 15% inputs ($I\_b\_s - I\_s$) that affect shared accesses do not have control over the program paths, and 60% inputs are irrelevant. We found that (1) many `rdtsc` instructions (e.g., 10/49 in pfscan) are just used for logging time stamp of events (e.g., thread creation) and never affect branching, (2) many return values from system calls (e.g., `sys_close`, `sys_fstat64`, `sys_write`, `sys_mprotect` etc.) are not used, whereas some return value matters (e.g., `sys_open`).

For comparing different DTAM approaches, we show results for group I_bs as shown in the right half of Figure 5. For three applications (i.e., aget, fastspy, and axel) DTAM-parallel and DTAM-hybrid demonstrate the generalization effect, resulting in larger relevancy ratios compared to serial. DTAM-hybrid addresses the over-tainting issue in DTAM-parallel, and undertainting issue in DTAM-serial, as observed for *aget* in particular. We observe that the ratio of shared objects (S) to that of inputs (I_{sc}) for these three applications is about 100, while that of the rest is about 10. Such a large ratio indicates more pronounced generalization.

The left half of Figure 6 shows the results of relevant shared object analysis based on DTAM-hybrid. (Results for DTAM-serial and DTAM-parallel are similar, and are not shown.) Each bar represents the ratio of relevant shared objects to the total number of shared objects for a given relevant shared type. The average of each bar is shown rightmost. We observe that relevancy ratio $S\_\bar{i}b$ for these applications is 0%, i.e., *all* shared objects that affect some branch also propagate the effect of some input to the branch. Further, the relevancy ratio for groups S_bs and S_b is about 3% each on average, indicating that shared objects have very small impact on the control over the program paths. The ratio increases to 13% on average for S_s and S_b_s each, indicating that the 10% shared objects that affect other shared accesses do not control the program, and about 87% shared objects are irrelevant. Comparing all DTAM approaches, hybrid gives better precision, as observed for *axel*.

Disclaimer: The relevancy ratios for inputs and shared objects reflect the trends observed on the test runs of the benchmarks used in our experiments. It may hold for other multi-threaded benchmarks that were not used, but may not hold in general.

Sink Dependencies. We now evaluate the dependency for sinks (shared accesses and branches) i.e., (1) how many shared accesses depend on inputs and/or branches and (2) how many branches rely on inputs and/or shared objects. We chose DTAM-hybrid to represent the results, as other methods give similar results.

The left half of Figure 7 shows the distribution of dynamic instances of shared accesses that are dependent on inputs and/or branches among all tested applications. Except for blackscholes, we observe that more than 95% of shared accesses turn out to be dependent upon inputs (SH_i and SH_i_b) at runtime, and most of them depend on both inputs and branches.

The right half of Figure 7 shows the distribution of dynamic instances of branches that are dependent upon inputs and shared objects. Similar to previous result, most of branches depend on inputs (BR_i) or both inputs and shared objects (BR_i_s).

These results highlight that most shared accesses and branches are dependent on some inputs. This implies that without properly monitoring such relevant inputs, it is extremely difficult to reconstruct shared state or control flow of multi-threaded programs, which justifies our motivation of relevancy analysis.

7. Related Work

Our work is related to dynamic taint analysis and runtime multi-threaded program monitoring.

Dynamic Taint Analysis. In the last few years, there have been many proposals to build efficient dynamic taint analysis tools from both software and hardware communities. Overall, the previous proposals can be categorized into three approaches based on underlying infrastructure used: dynamic binary translation (DBT) based approaches [25, 4, 33, 44, 5, 47], whole-system emulation based approaches [8, 46, 15], and hardware-assisted systems [38, 6, 41].

Dynamic binary translation has been largely used for implementing dynamic taint analysis, as it works easily on a given executable binary, and most previous works have focused on reducing performance overhead of taint propagation and runtime checks for sequential programs with better instrumentation techniques. For example, TaintTrace [4], based on DynamoRio, leverages one-to-one table mapping between data and taint metadata for fast lookup, and also implements a fast switch between original code and instrumentation using dead register analysis and eflag liveness analysis. Similarly, LIFT [33], implemented by StarDBT, proposes a conditional branching mechanism between a fast path (w/o instrumentation) and a slow path (with instrumentation) to skip redundant runtime checks. It also implements an efficient instrumentation mechanism without stack switches. Recently, TaintEraser [47], based on PIN, proposes to leverage user-annotated function summaries to speed up runtime taint propagation and checks. Even though we also use the dynamic binary translation tool PIN, our focus is not on improving performance using better instrumentation like previous works. Instead, we focus on enabling parallel taint analysis for multi-threaded programs. All the previous DBT-based works either do not consider multi-threaded programs, or simply assume serialized executions (like DTAM-serial). On the other hand, previous whole-system emulation based approaches enable dynamic taint analysis for multi-threaded programs, but they suffer from high performance overhead and require support from operating systems or hardware for practical implementation.

Runtime Monitoring for Parallel Programs. With the advent of multicore processors, runtime systems that exploit extra cores to monitor multi-threaded programs have been proposed. Log-Based Architecture (LBA) [3] supports a hardware event queue which efficiently collects runtime execution events (e.g. memory operations). The logged information can be dequeued by lifeguards, which run in parallel with the original program on spare cores and perform runtime checks. For example, Bufferfly analysis [14] has been built on top of LBA for efficient memory bound analysis and taint analysis for multi-threaded programs.

Respec [19] proposes a software-only system that enables decoupled runtime checks of multi-threaded programs based on an online deterministic replay technique. Respec records non-deterministic events of the original process and reproduces them for the replayed

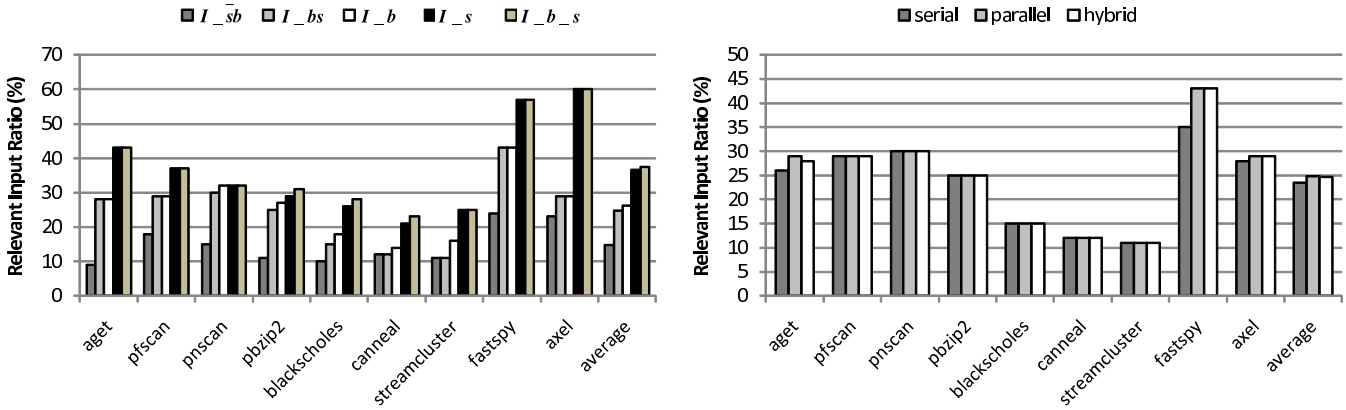

Figure 5: (L) Ratio of relevant inputs to total inputs (I_{sc}) for DTAM-hybrid; (R) Comparison of serial/parallel/hybrid for I_bs.

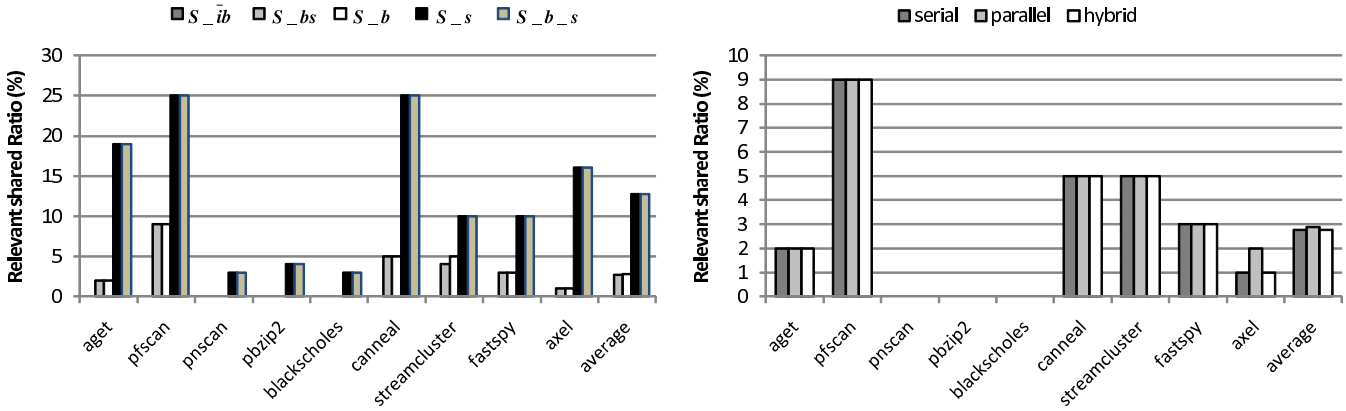

Figure 6: (L) Ratio of relevant shared objects to total shared objects (S) for DTAM-hybrid; (R) Comparison of serial/parallel/hybrid for S_bs.

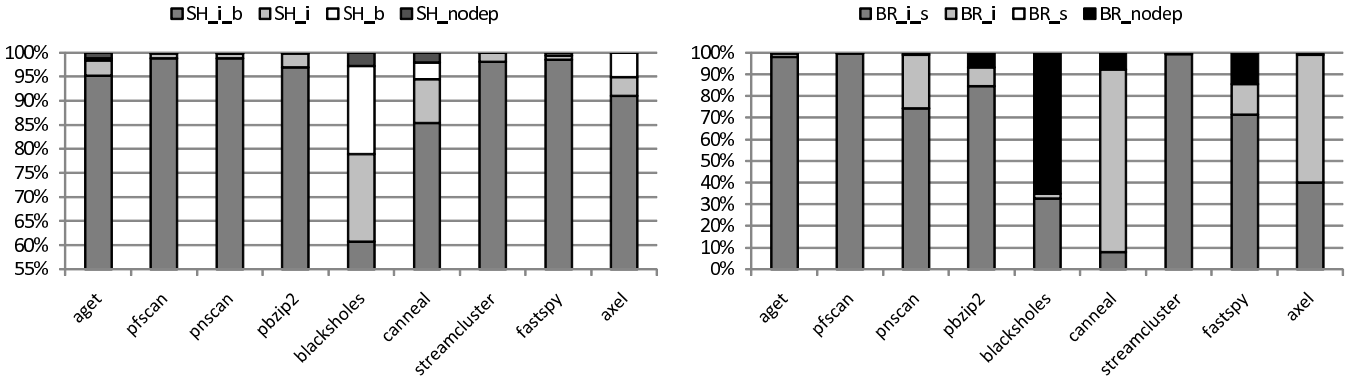

Figure 7: Distribution of dynamic instances of (left) shared accesses that depend on inputs and/or branches, (right) branches that depend on inputs and/or shared objects.

process, which run in parallel on extra cores. As replayed execution is guaranteed to be same as the original execution, it allows users to perform heavy-weight runtime checks on replayed executions without pausing the original executions. As an extension, Doubleplay [40] timeslices multi-threaded executions into so-called epochs, and runs each epoch in uniprocessor concurrently with original execution. As each epoch is executed in uniprocessor, Doubleplay enables using sequential version of monitoring tools (such as bound checks, taint checks, etc.) without modification and its runtime cost can be parallelized.

In contrast, our system DTAM monitors each thread independently (i.e., performs thread-modular taint analysis) and considers the effect of shared accesses later in an offline analysis.

8. Conclusions

We presented a thread-modular dynamic taint analysis for multi-threaded programs which can provide generalized taint analysis from a single observed execution. We used this analysis to identify a smaller set of inputs and shared objects in a multi-threaded program. In future work, we plan to evaluate the benefits of relevancy analysis to a replay system for multi-core platforms. Based on potential uses, we also introduced various relevancy types and groups for inputs and shared objects. Such relevancy analysis can be used to improve testing, verification, debugging, and program understanding for multi-threaded programs during development, or for failure diagnosis after deployment.

References

[1] G. Altekar and I. Stoica. ODR: Output-deterministic replay for multicore debugging. In *Symposium on Operating Systems Principles*, 2009.

[2] F. Chen, T. F. Serbanuta, and G. Rosu. jPredictor: A predictive runtime analysis tool for Java. In *Proc. of ICSE*, 2008.

[3] 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 *Proc. of ISCA*, 2008.

[4] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In *Proc. of ISCC*, 2006.

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

[6] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: a flexible information flow architecture for software security. In *Proc. of ISCA*, 2007.

[7] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. Dmp: deterministic shared memory multiprocessing. In *Proc. of ASPLOS*, 2009.

[8] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic spyware analysis. In *Proc. of USENIX ATC*, 2007.

[9] J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Australian Computer Science Conference*, 1988.

[10] C. Flanagan and S. N. Freund. FastTrack: Efficient and precise dynamic race detection. In *Proc. of PLDI*, 2009.

[11] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proc. of POPL*, 2005.

[12] M. Ganai. Scalable and precise symbolic analysis for atomicity violations. In *Proc. of ASE*, 2011.

[13] M. K. Ganai and A. Gupta. Efficient modeling of concurrent systems in BMC. In *Proc. of SPIN Workshop*, 2008.

[14] M. L. Goodstein, E. Vlachos, S. Chen, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. Butterfly analysis: adapting dataflow analysis to dynamic parallel monitoring. In *Proc. of ASPLOS*, 2010.

[15] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical taint-based protection using demand emulation. In *Proc. of EUROSYS*, 2006.

[16] G. Inc. Freshmeat. *http://freshmeat.net*.

[17] G. Inc. SourceForge. *http://sourceforge.net*.

[18] A. Lal and T. Reps. Reducing concurrent analysis under a context bound to sequential analysis. In *Proc. of CAV*, 2008.

[19] D. Lee, B. Wester, K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy. Respec: Efficient online multiprocessor replay via speculation and external determinism. In *Proc. of ASPLOS*, 2010.

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

[21] D. Marino, M. Musuvathi, and S. Narayanasamy. Literace: effective sampling for lightweight data-race detection. In *Proc. of PLDI*, 2009.

[22] F. Mattern. Virtual time and global states of distributed systems. In *Workshop on Parallel and Distributed Algorithms, France*, 1988.

[23] M. Musuvathi and S. Qadeer. Chess: systematic stress testing of concurrent software. In *Proc. of LOPSTER*, 2007.

[24] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. of POPL*, 1999.

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

[26] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *Proc. of ASPLOS*, 2009.

[27] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *Proc. of FSE*, 2008.

[28] S. Park, S. Lu, and Y. Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *Proc. of ASPLOS*, 2009.

[29] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: Probabilistic replay with execution sketching on multiprocessors. In *Symposium on Operating Systems Principles*, 2009.

[30] E. Pozniansky and A. Schuster. MultiRace: Efficient on-the-fly data race detection in multithreaded C++ programs. In *Proc. of Concurrency and Computation: Practice and Experience*, 2007.

[31] Princeton. The parsec benchmark suite. *http://parsec.cs.princeton.edu/*.

[32] S. Qadeer, S. K. Rajamani, and J. Rehof. Summarizing procedures in concurrent programs. In *Proc. of POPL*, 2004.

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

[34] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symposium on Security and Privacy*, 2010.

[35] T. F. Serbanuta, F. Chen, and G. Rosu. Maximal causal models for multithreaded systems. Technical Report UIUCDCS-R-2008-3017, University of Illinois at Urbana-Champaign, 2008.

[36] N. Sinha and C. Wang. Staged concurrent program analysis. In *Proc. of FSE*, 2010.

[37] Y. Smaragdakis, J. M. Evans, C. Sadowski, J. Yi, and C. Flanagan. Sound predictive race detection in polynomial time. In *Proc. of POPL*, 2012.

[38] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proc. of ASPLOS*, 2004.

[39] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: effective taint analysis of web applications. In *Proc. of PLDI*, 2009.

[40] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. DoublePlay: Parallelizing sequential logging and replay. In *Proc. of ASPLOS*, 2011.

[41] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. Flexitaint: A programmable accelerator for dynamic taint propagation. In *Proc. of HPCA*, 2008.

[42] C. Wang, R. Limaye, M. Ganai, and A. Gupta. Trace-based symbolic analysis for atomicity violations. In *Proc. of TACAS*, 2010.

[43] D. Weeratunge, X. Zhang, and S. Jagannathan. Analyzing multicore dumps to facilitate concurrency bug reproduction. In *Proc. of ASPLOS*, 2010.

[44] W. Xu, E. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proc. of USENIX Security Symposium*, 2006.

[45] J. Yi, C. Sadowski, and C. Flanagan. SideTrack: Generalizing dynamic atomicity analysis. In *Proc. of PADTAD*, 2009.

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

[47] D. Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. Tainteraser: protecting sensitive data leaks using application-level taint tracking. *SIGOPS Oper. Syst. Rev.*, 45:142–154, February 2011.