Tell You a Definite Answer: Whether Your Data is Tainted During Thread Scheduling

Xiaodong Zhang, Zijiang Yang, *Senior Member, IEEE*, Qinghua Zheng, *Member, IEEE*, Yu Hao, Pei Liu, and Ting Liu, *Member, IEEE*

Abstract—With the advent of multicore processors, there is a great need to write parallel programs to take advantage of parallel computing resources. However, due to the nondeterminism of parallel execution, the malware behaviors sensitive to thread scheduling are extremely difficult to detect. Dynamic taint analysis is widely used in security problems. By serializing a multithreaded execution and then propagating taint tags along the serialized schedule, existing dynamic taint analysis techniques lead to under-tainting with respect to other possible interleavings under the same input. In this paper, we propose an approach called DSTAM that integrates symbolic analysis and guided execution to systematically detect tainted instances on all possible executions under a given input. Symbolic analysis infers alternative interleavings of an executed trace that cover new tainted instances, and computes thread schedules that guide future executions. Guided execution explores new execution traces that drive future symbolic analysis. We have implemented a prototype as part of an educational tool that teaches secure C programming, where accuracy is more critical than efficiency. To the best of our knowledge, DSTAM is the first algorithm that addresses the challenge of taint analysis for multithreaded program under fixed inputs.

Index Terms—Taint analysis, multithreaded programs, symbolic analysis, encoding, guided execution

1 Introduction

Dynamic taint analysis (DTA for short) tracks information flow between sources and sinks during runtime. It has been shown to be effective in dealing with a wide range of security problems such as data leak detection [1], software attack prevention [2], [3], information flow control [4], [5], and malware analysis [6], [7]. Because of its enormous applications, there has been considerable amount of work to improve its efficiency and accuracy. Unfortunately, the existing DTA techniques, when applied on multithreaded programs, lead to severe under-tainting. This is due to the fact that besides inputs, thread scheduling also affects the program executions. Only analyzing one interleaving will miss the report of tainted instances on other interleavings. As a result, DTA is not able to conduct effective taint analysis on multithreaded programs. That is, given a multithreaded program and a fixed input vector, DTA is no longer able to answer users' queries such as whether a particular variable instance is tainted under the input.

Fig. 1 gives an atomicity error [8], which is detailedly introduced in the literature [9]. It allows an attacker to silently trigger a buffer overflow in Moonlight, a Silverlight browser plugin implementation of the Mono open-source .NET framework. The `FastCopy()` method first checks that the types of the destination element and the source element are compatible (Line 3-5). Then, this method performs memory copy if they are compatible (Line 8-9). However, the type check and the copy are not implemented as one atomic step so that an attacker can modify the source or destination array after the type check. For example, the attacker in another thread can modify the source array with a media file. This file cannot get through the type check, but it can lead to a buffer overflow with a crafted modification. If the thread runs the statement at Line 6 after a type check but before a copy action, then buffer overflow will be triggered when the `memcopy` is performed. This kind of attack is called concurrency attack [9]. Through carefully crafted inputs, attackers can artificially control the timing window within which a concurrency error may occur to increase the chance of exploiting the error. When we leverage DTA to detect the attack, we set the crafted media file as untrusted data, that is, taint source. From the code snippet, we can know that if Line 6 happens before Line 4, the type check is failed and `FastCopy` terminates; and if Line 6 happens after Line 9, the untrusted data never be propagated to `memcopy`. During the analysis, only if the interleaving $s = \{...3-4...6...8-9...\}$ is encountered, the attack can be detected. Otherwise, for any interleaving except s, DTA cannot detect the attack and only gives an under-tainting result.

From the example, we know that in order to determine whether an instance is tainted under a given input, all execution traces permissible under that input must be examined. However, in current environment a user has no control over the scheduling of threads. Furthermore, when applying DTA repeatedly on a multithreaded program, the same thread interleavings, with minor variations, tend to be exercised since thread schedulers generally switch among

- *X. Zhang, Q. Zheng, Y. Hao, P. Liu, and T. Liu are with the Ministry of Education Key Lab for Intelligent Networks and Network Security, Xi'an Jiaotong University, Xi'an, Shaanxi 710000, China. E-mail: {xdzhang, yhao, pliu}@sei.xjtu.edu.cn, {qhzheng, tingliu}@mail.xjtu.edu.cn.*
- *Z. Yang is with the Department of Computer Science, Western Michigan University, Kalamazoo, MI 49008. E-mail: zijiang.yang@wmich.edu.*

Manuscript received 26 Oct. 2017; revised 19 Aug. 2018; accepted 2 Sept. 2018. Date of publication 26 Sept. 2018; date of current version 17 Sept. 2020. (Corresponding author: Ting Liu.)
Recommended for acceptance by C. Zhang.
Digital Object Identifier no. 10.1109/TSE.2018.2871666

```
1   bool FastCopy (MonoArray *src, MonoArray* dest, int
      length){
2   \\Checks that the type of dst[i] derive from src[i]
3   for (i = 0; i < length; ++i)
4     if(!safe cast(type of(src[i]), type of(dest[i])))
5       return FALSE;


7   \\directly copy the bytes with memcpy()
8   for (i = 0; i < length; ++i)
9     memcpy(dest[i], src[i], size of(ObjPtr));
10  return TRUE;
  }
```

```
Another Thread


...


6  ERROR: src = A crafted media file;




...
```

Fig. 1. A concurrency attack resulting from an atomicity error in Moonlight.

threads at the same program locations. The net effect of these impediments is that only a few interleavings end up being executed in practice. In the field, complex and largely unpredictable interactions involving external events and other unrelated processes will inevitably trigger an execution sequence of the multithreaded program that escaped detection during controlled monitoring. Even if it were possible to control thread scheduling, it would still be infeasible to explicitly test all interleavings. The number of possible interleavings of a multithreaded program with n threads each executing at most k steps can be as large as $(nk)!/(k!)^n \geq (n!)^k$, a complexity that is exponential in both n and k [10].

In order to address the challenges of taint analysis for multithreaded programs, we develop a synergistic taint analysis framework that integrates symbolic analysis and guided execution to guarantee a systematic and complete traversal of all program behaviors for a test input. The current prototype is part of an educational tool that teaches programmers secure C programming. The most common query raised by students is whether a particular variable instance is tainted under a given input, and if so, an evidence on how the variable becomes tainted. Although there exist a large number of taint analysis tools, such as Dytan [11], TaintCheck [2], BuzzFuzz [12], and BitBlaze [13], none is able to give a definite answer to the query as they do not consider thread scheduling.

There has been very little research on dynamic taint analysis of multithreaded programs. To the best of our knowledge, DTAM [14] is the only research that targets this problem. However, the three approaches proposed by DTAM cause under-tainting, over-tainting,[1] or both. This is not acceptable in applications where precise and definite answers are desired. For example, in our educational software students miss the opportunity to discover the security issues in their programs when under-tainting happens, and they may get confused or misled when over-tainting happens. Besides imprecision, DTAM cannot provide an evidence interleaving when a scheduling-sensitive taint instance is judged. Without a propagation evidence, programmers still know nothing about how the instance is tainted. With these considerations, our algorithm aims to accurately report tainted instances under a fixed input, without false negatives and false positives. Our algorithm not only can give a definite answer whether a variable instance is tainted during thread scheduling, but also provide a propagation evidence for each tainted instance to assist in detecting attack. In summary, this paper makes the following major contributions:

- For multithreaded program, a single input can lead to exponential number of different executions. Existing DTA techniques can only conduct taint analysis on a given execution but not on a given input. This does not meet the expectations of taint analysis for most applications. To the best of our knowledge, We are the first to address the challenge of precise taint analysis for multithreaded program under a given input.
- Under the assumption of an ideal SMT solver, our algorithm is sound and complete in the sense that it can detect the tainted instances under a given input without false positives and false negatives. This is achieved by synergistic integration of symbolic analysis and guided execution. Soundness and completeness are not achievable if there are software artifacts, such as nonlinear expressions, that cause unknown SMT solving result.
- We implement the proposed algorithm in a prototype tool, which generates a propagation evidence for each detected tainted instance. Our evaluation confirms that our algorithm detects more tainted instances than existing approaches that monitor and analyze a single execution.

The remainder of this paper is organized as follows. Section 2 uses an example to illustrate why existing DTA techniques are not sufficient and how our approach works. Section 3 gives a formal presentation of our algorithms. After reporting experimental results in Sections 4 and 5 reviews the related work. Finally, Section 6 concludes the paper.

2 Insufficiency of Existing Approach and Sketch of Our Approach

In this section we first use a more complex example to illustrate the typical procedure of DTA and explain why it is insufficient to handle multithreaded programs. Next we give a sketch of our approach on the same example.

2.1 DTA Not Sufficient

Fig. 2 shows a multithreaded program P, where the main thread T_0 creates two threads T_1 and T_2. There are two shared variables x (accessed by all threads) and z (accessed by T_1 and T_2). With $argv[1]$ being the only (untrusted) input, we aim to detect all the tainted variable instances under a fixed input $argv[1] = 1$. Let an execution of P be $\pi_1 = \langle 1, 2, 3, 6, 7, 8F, 4, 10, 5, 11\rangle$, where T/F after a line number denotes the true/false branch. The trace is depicted as the first column in Fig. 3. The blue, green and red boxes represent the instruction instances of Threads 0, 1 and 2, respectively. Within each node we give the line numbers and corresponding statements. For ease of understanding we

1. An instance that never can be tainted is deemed tainted.

```
Thread 0:
int main(argv[]) {
1  a = argv[1];
2  y = a + 1;
3  create(foo, &y);
4  create(bar, &y);
5  x = 3;}
```

```
Thread 1:
void* foo(int m)
    {
6   x = m - 1;
7   z = x;
8   if (x > z)
9     w = m; }
```

```
Thread 2:
void* bar(int n)
    {
10   x = n + 1;
11   z = x * x;}
```

Fig. 2. A multithreaded program with shared variables x, z.

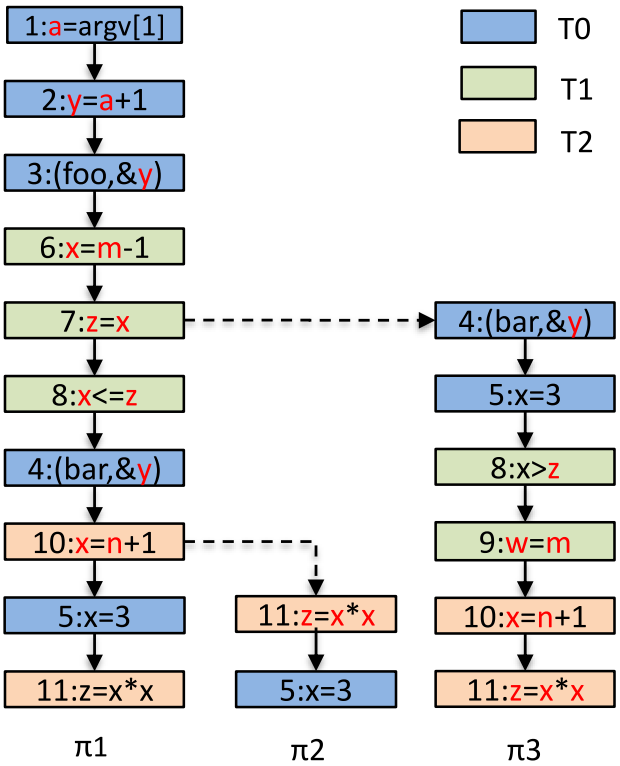


Fig. 3. Three valid executions of the program in Fig. 2 under argv[1]=1.

illustrate the example at the source code level and assume each statement is atomic, although our implementation is at the LLVM bytecode level. DTA monitors the execution and propagates the taint information from the assignment at Line 1. In this example, an assigned variable is tainted if any of the variables on the right hand side of the assignment is tainted. A propagation policy may or may not consider implicit flows caused by control dependency. We choose not to consider implicit flows in this paper. The research on this topic [11], [15], [16], [17] is orthogonal to ours and can be adopted. As shown in Fig. 3, $y@L2$ becomes tainted because a on the right-hand side is tainted; $m@L6$ is tainted because it receives the value of the tainted parameter $y@L3$, which in turn taints $x@L6$; $x@L5$ becomes untainted due to constant assignment. Note that we use variable name and line number to uniquely identify an instance in an execution, although in general this is insufficient. All the tainted instances reported by DTA are marked red in Fig. 3.

The result obtained by applying DTA is under-tainted. Due to nondeterministic thread scheduling there are

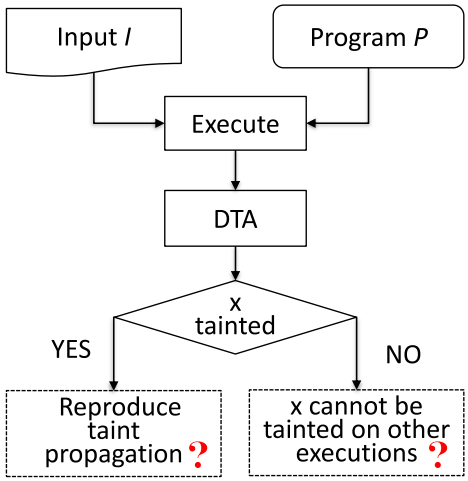


Fig. 4. Existing DTA techniques fail multithreaded programs.

multiple valid executions under the same input. As shown by the second column in Fig. 3, another execution is $\pi_2 = \langle 1, 2, 3, 6, 7, 8F, 4, 10, 11, 5\rangle$, which inverts the last two instructions in π_1. Since Line 11 is executed before the constant assignment at Line 5 in π_2, $z@L11$ is tainted in π_2 but not in π_1. The last column in Fig. 3 gives a third execution $\pi_3 = \langle 1, 2, 3, 6, 7, 4, 5, 8T, 9, 10, 11\rangle$. Unlike π_2 that reorders the instructions in π_1, different thread scheduling in π_3 causes the execution of a different branch at Line 8. The different branch leads to the execution of Line 9 that does not appear in π_1 or π_2. It turns out $w@L9$ is tainted in π_3. The traces π_2 and π_3 illustrate the two reasons for the under-tainting: DTA does not consider the reordering of instructions in a trace; and DTA does not consider the executions of different branches caused by alternative thread schedulings.

Fig. 4 depicts the insufficiency of applying DTA on multithreaded programs. By monitoring a particular execution under a given input, existing DTA techniques have two severe shortcomings. When a variable x is declared not tainted, a future execution under the same input but with different thread schedule may or may not contradict the current result. When a variable x is declared tainted, it is true. However, propagation evidence for such tainting cannot be easily reproduced. Unlike in sequential programs where a re-execution under the same input is sufficient, deterministically replaying an execution is a major challenge for multithreaded programs [18], [19].

2.2 How Our Approach Works

We use the program given in Fig. 2 as a running example to illustrate the basic steps of our approach. As shown by the first column in Fig. 3, existing DTA tools monitor the execution of π_1 and report the set of tainted instances as shown as

$$\Gamma = \{a@L1, y@L2, m@L3, x@L6, z@L7, n@L4, x@L10\}. \quad (1)$$

Starting from π_1, we explain how our approach automatically detects the scheduling-sensitive tainted instances $z@L11$ in π_2 and $w@L9$ in π_3.

Fig. 5 gives the trace of π_1 in Static Single Assignment (SSA) form, where each variable instance gets a unique subscript label. In addition, for each shared variable v, we use superscripts r and w to denote the read and write of v. In addition, we add explicit assignments for parameter passing at Lines $p2$ and $p3$, which are considered as the first

Thread 0:		Thread 1:		Thread 2:	
1	$a_1 = 1;$	$p2$	$m_1 = y_1$	$p3$	$n_1 = y_1$
2	$y_1 = a_1 + 1;$	6	$x_1^w = m_1$	10	$x_4^w = n_1 + 1$
3	$create(foo, y_1);$	7	$z_1^w = x_2^r$	11	$z_2^w = x_5^r * x_5^r$
4	$create(bar, y_1);$	8	$if(x_3^r > z_2^r)$		
5	$x_6^w = 3;$				

Fig. 5. The SSA form of the execution trace π_1.

statements in Threads 2 and 3, respectively. In the following, we present the four types of constraints that, when combined together, symbolically encode all valid ordering of an execution trace.

Partial Order Constraint. The core of the encoding is to relax the total order of π_1 so that other interleavings can be considered. Equation (2) gives the partial order constraint of π_1, where symbolic variable o_i represents the possible position of the statement at Line i in a valid execution, and $o_i < o_j$ means Line i happens before Line j. We only consider the sequential consistency model in this work, therefore the ordering within individual threads cannot be changed. In addition, Line 3 happens before Line $p2$ ($o_3 < o_{p2}$) because the statement at Line 3 creates Thread 1. Similarly, $o_4 < o_{p3}$. On the other hand, the partial order constraint does not specify the relative order of most statements from different threads. For example, Line 10 can happen before Line 6 even the opposite happens in π_1. Of course, for most programs, the order of the statements among different threads cannot be totally arbitrary. We will discuss the encoding of synchronization primitives in Section 3.

$$\begin{aligned}&(o_1 < o_2 < o_3 < o_4 < o_5) \wedge (o_{p2} < o_6 < o_7 < o_8)\\ &\quad \wedge (o_{p3} < o_{10} < o_{11}) \wedge o_3 < o_{p2} \wedge o_4 < o_{p3}.\end{aligned} \tag{2}$$

Program Semantics Constraint. This type of constraint specifies program semantics requirement. Since SSA form already gives unique indices to shared variables, the translation from the trace to such constraint is straight-forward. Equation (3) gives the program semantics constraint for the trace shown in Fig. 5.

$$\begin{aligned}&a_1 = 1 \wedge y_1 = a_1 + 1 \wedge x_6^w = 3\\ &m_1 = y_1 \wedge x_1^w = m_1 \wedge z_1^w = x_2^r \wedge \neg(x_3^r > z_2^r)\\ &n_1 = y_1 \wedge x_4^w = n_1 + 1 \wedge z_2^w = x_5^r * x_5^r.\end{aligned} \tag{3}$$

Taint Propagation Constraint. For a variable instance v_i, we associate with it a symbolic Boolean variable $v_i.tag$ to propagate taint. Fig. 6 gives the encoding of taint propagation, which indicates the variable instance on the left-hand side of an assignment is tainted if one of the operands on the right-hand side is tainted. Of course, in our implementation we need to consider the semantics of an expression. For example, a Boolean `and` with *false* or a Boolean `or` with *true* untaints the assigned variable.

Interleaving Matching Constraint. While a trace in SSA form clearly indicates the data flow of thread local variables, it does not specify the correlation between the reads and writes of a shared variable. For example, $x_2^r@L7$ that may read from either $x_1^w@L6$, $x_6^w@L5$ or $x_4^w@L10$. In order for x_2^r to read from x_1^w, the execution of Line 5 and Line 10 must either occur before Line 6 or after Line 7. In this case, the value of x_2^r is the same as x_1^w and the taint propagates from x_1^w to x_2^r. Similarly, if x_2^r reads x_6^w, the execution of Line 6 and Line 10 must either occur before Line 5 or after Line 7; if x_2^r reads x_4^w, the execution of Line 5 and Line 6 must either occur before Line 10 or after Line 7. The interleaving

1	$a_1 = argv[1];$	$a_1.tag = true$
2	$y_1 = a_1 + 1;$	$y_1.tag = a_1.tag$
3	$create(foo, y_1 \rightarrow m_1);$	$m_1.tag = y_1.tag$
6	$x_1^w = m_1$	$x_1^w.tag = m_1.tag$
7	$z_1^w = x_2^r$	$z_1^w.tag = x_2^r.tag$
8	$if(x_3^r > z_2^r)$	$-$
4	$create(bar, y_1 \rightarrow n_1);$	$n_1.tag = y_1.tag$
10	$x_4^w = n_1 + 1$	$x_4^w.tag = n_1.tag$
5	$x_6^w = 3;$	$x_6^w.tag = false$
11	$z_2^w = x_5^r * x_5^r$	$z_2^w.tag = x_5^r.tag$

Fig. 6. Taint propagation during the execution of π_1.

matching constraint regarding x_2^r is given in Equation (4). The sub-formulas $o_6 > o_7$ in Row 4 and 6 is obviously infeasible, and will be ignored during encoding. For every read of a shared variable, we need a similar constraint.

$$\begin{aligned}&\{x_2^r = x_1^w \wedge x_2^r.tag = x_1^w.tag \wedge o_6 < o_7\\ &\quad \wedge (o_5 < o_6 \vee o_5 > o_7) \wedge (o_{10} < o_6 \vee o_{10} > o_7)\} \bigvee\\ &x_2^r = x_6^w \wedge x_2^r.tag = x_6^w.tag \wedge o_5 < o_7\\ &\wedge \{(o_{10} < o_5 \vee o_{10} > o_7) \wedge (o_6 < o_5 \vee o_6 > o_7)\} \bigvee\\ &\{x_2^r = x_4^w \wedge x_2^r.tag = x_4^w.tag \wedge o_{10} < o_7\\ &\wedge (o_5 < o_{10} \vee o_5 > o_7) \wedge (o_6 < o_{10} \vee o_6 > o_7)\}.\end{aligned} \tag{4}$$

Finally, we combine the four types of constraints that consists of partial order constraint φ_{po}, program semantics constraint φ_{sm}, taint propagation constraint φ_{tp}, and interleaving matching constraint φ_{im}. The formula (5)

$$\varphi_{\pi_1} = \varphi_{po} \wedge \varphi_{sm} \wedge \varphi_{tp} \wedge \varphi_{im} \tag{5}$$

encodes all valid reordering of the instructions in π_1. In order to check if a variable instance, such as z_2^w, is tainted, we can leverage an off-the-shelf SMT solver to check the satisfiability of $\varphi_{\pi_1} \wedge z_2^w.tag$. In our example, the formula is satisfiable, which indicates that z_2^w can be tainted even though it is not in π_1. A replay based on the solution to $\varphi_{\pi_1} \wedge z_2^w.tag$ may give π_2 as an evidence of tainting.

Unfortunately φ_{π_1} cannot be used to check whether $w@L9$ is tainted because Line 9 does not even appear in π_1. Consider the conditional statement at $L8$, where the *false* branch $x \leq z$ is executed in π_1. If there exists a different thread schedule that leads to the execution of the *true* branch, we will have a new base for symbolic analysis. Here, the challenge is to decide whether such a new path is feasible and, if it is feasible, how to enforce its execution during the subsequent guided execution (i.e., make it show up when we run the program). We will discuss the algorithm more formally in Section 3. As for this example, we replace the term $\neg(x_r^3 > z_r^2)$ with $(x_r^3 > z_r^2)$ to mandate that $x > z$ happens in a valid path. Hereon, we do not consider the orderings between branch statements since there is only one branch in the example.

Let the revised formula be φ'_{π_1} as shown in as

$$\varphi'_{\pi_1} = \varphi_{\pi_1} | REMOVE(\neg x_r^3 > z_r^2) \wedge x_r^3 > z_r^2. \tag{6}$$

If it is unsatisfiable, the branch at Line 8 cannot be negated branch in any execution. Otherwise, its solution gives a schedule to a to-be-explored path. In our example, the new schedule is $s : \langle 1, 2, 3, 6, 7, 4, 5, 8T, 10, 11, 4, 5\rangle$.

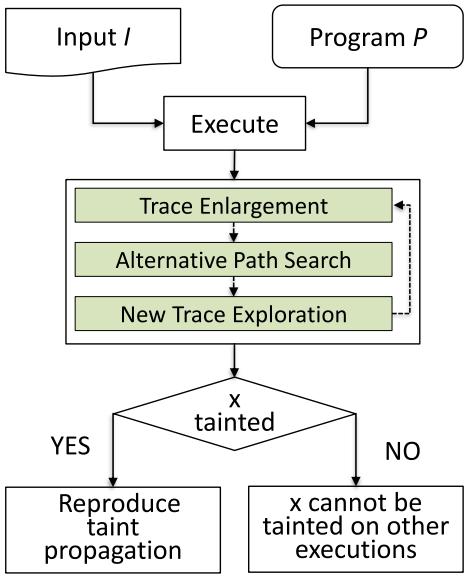


Fig. 7. Dynamic and symbolic taint analysis for multithreaded programs (DSTAM).

Once a new schedule is obtained, a concrete execution has to be performed under the guidance of thread schedule s. The reason that symbolic analysis alone is not sufficient is because the thread schedules generated by symbolic analysis are only partially valid. The schedule in s is not correct after $8T$ because $L9$ must be executed. Since $L9$ is not considered by φ'_{π_1}, the symbolic analysis gives a random solution after $L8$ in s. However, the prefix $s' = \langle 1, 2, 3, 6, 7, 4, 5, 8T, \rangle$ is not only valid but also guarantees the execution of $true$ branch at $L8$. If we enforce a guided execution following s' we obtain $\pi_3 : \langle 1, 2, 3, 6, 7, 4, 5, 8T, 9, 10, 11\rangle$. The new trace will trigger further symbolic analysis. That is, we will re-encode a new constraint φ_{π_3} for execution π_3 since φ_{π_1} is invalid for π_3 anymore.

The above example illustrates, in a nutshell, how our approach works. Fig. 7 depicts our synergistic taint analysis framework that analyzes and explores program executions instead of just observing an execution as DTA does. Our tool includes a `Trace Enlargement` component (for symbolic analysis), an `Alternative Path Search` component (for symbolic analysis), and a `New-Trace Exploration` component (for guided execution). The entire framework forms a synergistic loop, where symbolic analysis and guided execution reinforce each other, to guarantee the systematic and complete traversal of all program behaviors for a given test input.

Specifically, in the `Trace Enlargement` component, we capture valid partial order of a given execution trace using a quantifier-free first-order logic (FOL) formula and conduct predictive symbolic analysis to infer new program behaviors. In the `Alternative Path Search` component, we detect the branch sequences that are not yet visited by the previous executions, and compute a thread scheduling that enables the execution with new branches. In the `New-Trace Exploration` component, we perform guided execution, which enforces the newly computed thread schedules in executing the program. Such execution, in turn, leads to new execution traces to be analyzed by the symbolic analysis. The loop terminates when no new distinctive program path is found.

3 Algorithms for Taint Analysis of Multithreaded Programs

In this section, we give a more formal presentation, including the encoding with synchronization primitives and the issues with negating branches. We will also explain in more detail the integration of symbolic analysis and guided execution, whose combined efforts enumerate all possible thread interleavings under a fixed test input. Our approach integrates explicit executions with symbolic analysis to systematically explore program behavior under fixed inputs. Its pseudocode, shown in Algorithm 1, consists of four components: `New-Trace Exploration`, `Potential Taint Detection`, `Trace Enlargement`, and `Alternative Path Search`.

Algorithm 1. DSTAM(Program P, Input I)

```
1: ScheduleSet S = {⟨⟩};
2: TaintedSet Γ = ∅;
3: TestedSet Π = ∅;
4: while S ≠ ∅ do
5:    π = NewTraceExploration(P, I, S.remove(), Γ);
6:    if π.abstract ∉ Π then
7:       Π.add(π.abstract);
8:       PotentialTaintSet ϒ_π =DetectTaint(π, Γ);
9:       φ_π = TraceEnlargement(π, Γ, ϒ_π);
10:      AltPathSearch(φ_π, S);
11:   end if
12: end while
```

We maintain a set of to-be-explored schedules S, initially with one item (an empty vector). The algorithm terminates when S becomes empty, which indicates that all possible behaviors under the fixed input I have been explored. `New-Trace Exploration` enforces an execution to follow a predefined thread schedule prefix s removed from S. An execution that goes beyond s becomes random. Note that the initial schedule prefix is an empty vector $\langle\rangle$, so the first trace is a random execution from the beginning. Each `New-Trace Exploration` produces a recorded trace.

We say a trace is *explicitly* explored if it is produced by `New-Trace Exploration`, and is *implicitly* explored if it is considered by `Trace Enlargement`. Since a trace π produced by `New-Trace Exploration` may have been implicitly explored, we conduct further analysis only if π is not in a set Π that contains all the explicitly and implicitly explored paths. In Section 3.4 we explain why and how to maintain Π.

We analyze a trace π in two steps. The first step conducts an offline analysis of π and records the tainted instances in Γ according to standard taint policies. In addition, it collects the set of instances Υ_π in π that can be *potentially* tainted. `Trace Enlargement` encodes π and confirms if any instance in Υ_π but not in Γ can actually be tainted. The encoding of a trace is incapable of predicating the program behavior involving different set of instruction instances. The purpose of `Alternative Path Search` component is to compute thread schedules that lead to executions with a branch sequence different from previously explored paths.

3.1 New-Trace Exploration

We have implemented a thread scheduler to enforce `New-Trace Exploration`. There are two pieces of critical

```
Thread 1:
1  int x=a;
2  if (x>0)
3   y = y+1;
   else
4   y = y-1;
```

```
Thread 2:
5  int y=b;
6  if (y>0)
7    x = x+1;
   else
8    x = x-1;
```

Fig. 8. Code snippet with input ($a = 1, b = 0$) and shared variables x, y.

Thread 0:		Thread 1:		Thread 2:	
1	$a_1.tag = true;$	$p2$	$m_1.tag = pt$	$p3$	$n_1.tag = pt$
2	$y_1.tag = a_1.tag;$	6	$x_1^w.tag = m_1.tag$	10	$x_4^w.tag = n_1.tag$
3		7	$z_1^w.tag = pt$	11	$z_2^w.tag = pt$
4					
5	$x_6^w.tag = false;$				

Fig. 9. An example for detecting potentially tainted set.

information in a schedule item $s[i]$: the thread id $s[i].tid$ and the instruction $s[i].ins$, which demands an execution to execute $s[i].ins$ of thread $s[i].tid$ at ith step. The thread scheduling is random when the execution is beyond schedule s. Each thread follow the control flow until the end of execution.

We have to maintain a set Π of explored traces to avoid repeated exploration of the same traces. Consider the code snippet in Fig. 8 with test input ($a = 1, b = 0$) and shared variables x, y. Let the initial execution be $\pi_1 = \langle 1, 2T, 3, 5, 6F, 8\rangle$. A search for alternative branches confirms that the branch at Line 2 can be negated, which leads to a schedule prefix $\langle 1, 5, 6F, 8, 2F\rangle$. An execution following the prefix results in the second execution $\pi_2 = \langle 1, 5, 6F, 8, 2F, 4\rangle$. The branch at Line 2 in π_2 can be negated as well, with a schedule prefix of $\langle 1, 2T\rangle$. Following such schedule prefix we may execute $\langle 1, 2T, 3, 5, 6F, 8\rangle$, same as π_1. Without Π Algorithm 1 may not terminate.

There are two challenges to maintain the set of explored traces. First, the number of traces, even under a fixed test input, can be exponential to the number of instruction instances. Therefore recording all the explored traces can be extremely expensive. Second, recording only explicitly executed traces is not sufficient. Π has to include the traces implicitly explored by SMT solvers as well. To address both issues, we abstract an execution trace π to avoid recording complete traces and at the same time cover all the implicit traces derived from π. Let π^t be a subsequence by projecting π onto thread t. We partition π into a set $\pi = \{\pi^t | 1 \leq t \leq N\}$, where N is the number of threads. Let $\pi_B^t = \langle b_1^t b_2^t \dots b_k^t\rangle$ be the subsequence of branches in π^t. The trace abstract of π is defined as $\pi.abstract = \{\pi_B^t | 1 \leq t \leq N\}$. During guided execution, we need to keep abstracts of only those traces that are explicitly explored. This is sufficient because according to our algorithm: (1) explicitly explored traces must be different at some branches, and (2) traces with different interleaving but the same branch instances are covered by the same SMT solving procedure. A set Π based on abstracts not only keeps much shorter sequences of individual traces but also gives exponential reduction in the number of traces.

3.2 Detection of Potentially-Tainted Set

There are two main steps in traditional sequential dynamic taint analysis: (1) tagging, i.e., identifying data from external inputs and marking them as tainted, and (2) propagating the taint tag along the data flow (some taint policies consider control flow as well) through the program. Given a trace π obtained from a guided execution, we apply the traditional approach to compute a set of tainted instances Γ. This is achieved by serializing π and then propagating taint tags along the serialized schedule. As shown in Section 2.1, predictive symbolic analysis can find more tainted instances in π by determining if $\varphi_\pi \wedge v_i.tag$ is satisfiable for each candidate instance v_i. Since SMT solving is computationally expensive, we need to carefully compute a set of candidates to avoid unnecessary computation. We call this set as potentially tainted set Υ_π. In order to compute Υ_π, each thread maintains its own taint map for shared objects and performs thread modular taint propagation. When a thread performs a shared read or passes a parameter to another thread, it creates a pseudo taint tag [14] 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 real taint tags.

We use an example to illustrate the computation of the potentially tainted set. Consider the execution trace π_1 given in Fig. 6. As shown in Fig. 9, we associate a pseudo taint tag pt with $m@Lp2, z@L7, n@Lp3$ and $z@L11$ because of shared variable read or parameter passing. The pseudo taint tags are propagated in each thread separately. During offline analysis, once we replace pseudo taint tags with $true$, we get $\Upsilon_{\pi_1} = \{a@L1, y@L2, m@Lp2, x@L6, z@L7, n@Lp3, x@L10,\ z@L11\}$. The actual taint set for serialized π_1 is $\Gamma_{\pi_1} = \{a@L1,\ y@L2, m@Lp2, x@L6, z@L7, n@Lp3, x@L10\}$. In this case there is only one instance, $\Upsilon_{\pi_1} - \Gamma_{\pi_1} = \{z@L11\}$ that needs to be considered in `Trace Enlargement` component for π_1.

3.3 Trace Enlargement

`Trace Enlargement` first transforms an execution trace π into a quantifier free first order logic formula

$$\varphi_\pi = \varphi_{po} \wedge \varphi_{sm} \wedge \varphi_{tp} \wedge \varphi_{im}, \tag{7}$$

where $\varphi_{po}, \varphi_{sm}$, φ_{tp} and φ_{im} denote the partial order constraint, program semantics constraint, taint propagation constraint, and interleaving matching constraint, respectively. For each variable instance v_i that is potentially tainted but not already tainted, i.e., in the set of $\Upsilon_\pi - \Gamma$, we check if $\varphi_\pi \wedge v_i.tag$ is satisfiable. If so, its solution confirms that v_i is tainted and gives a schedule that our tool can follow to deterministically replay the taint propagation. Here, the encoding method is not our contribution, but we still illustrate it for integrity.

Algorithm 2. TraceEnlargement(Trace π, TaintSet Γ, PotentialTaintSet Υ_π)

```
1: Translate π into a quantifier free first order logic formula φ_π;
2: for each v ∈ Υ_π − Γ do
3:    if φ_π ∧ v.tag is satisfiable then
4:       Γ.add(v);
5:    end if
6: end for
7: return φ_π;
```

Partial Order Constraint (φ_{po}). This constraint specifies the potential ordering of the instructions in an execution trace π. In this paper, we consider sequential consistency memory model only. If instruction i happens before instruction j in π and both belong to the same thread, we enforce $o_i < o_j$.

The inter-thread ordering is guarded by synchronization primitives. In multithreaded programs, the most popular synchronization operations are *lock/unlock* and *wait/signal*.

Consider two *lock/unlock* pairs on the same mutex. The following constraint mandates that one pair must be executed either before or after another:

$$\varphi_{po}^{L[m]} = \bigwedge_{l_i/u_i, l_k/u_k \in L[m]} o(u_i) < o(l_k) \vee o(u_k) < o(l_i), \quad (8)$$

where $L[m]$ denotes the set of *lock/unlock* pairs on mutex lock m, and $o(x)$ eepresents the order of synchronization operation x.

Given a condition variable cd, let WT be the set of wait operations on cd, and SG the set of signal operations on cd. The constraint for *wait/signal* is:

$$\varphi_{po}^{W[cd]} = \left\{ \bigwedge_{w \in WT} \bigvee_{s \in SG} (o_w < o_s < o_{w'} \wedge m_s^w = 1) \right\} \bigwedge \varphi_{SG}^{WT} \bigwedge \varphi_{WT}^{SG}, \quad (9)$$

where $o_{w'}$ denotes the next event of wait on cd immediately after w in the same thread, $o_w < o_s < o_{w'}$ indicates that a signal operation s must be executed between w and w', and $m_s^w = 1$ flags that s is mapped to w. Equation (10) defines φ_{SG}^{WT} and φ_{WT}^{SG}, in which φ_{SG}^{WT} enforces that each wait operation w needs to map to at least one signal operation, and φ_{WT}^{SG} restricts that each signal operation s signals at most one wait operation.

$$\varphi_{SG}^{WT} = \bigwedge_{w \in WT} \left\{ \left\{ \sum_{s \in SG} m_s^w \right\} \geq 1 \right\}$$
$$\varphi_{WT}^{SG} = \bigwedge_{s \in SG} \left\{ \left\{ \sum_{w \in WT} m_s^w \right\} \leq 1 \right\}, \quad (10)$$

Constraints on other types of synchronization primitives are modeled similarly. The conjunction of these intra- and inter-thread constraints relaxes the total order observed in an execution trace π. Its encoding method is derived from our previous work [20].

Program Semantics Constraint (φ_{sm}). The constraint maps executed individual instructions to corresponding formula. We skip detailed presentation as it requires mapping rules for complete LLVM syntax. However, the mapping rules are straightforward. Note that φ_{sm} enforces the same control flow for all encoded thread interleavings as a derivation leads to an execution with instructions unknown to a trace. This encoding is also derived from our previous work [20].

Taint Propagation Constraint (φ_{tp}). The taint propagation constraint specifies the taint status for data derived from tainted or untainted operands. For each instance v_i, we associate with it a Boolean value tag $v_i.tag$ that indicates whether v_i is tainted. If v_i is a tainted source such as untrusted input, $v_i.tag = true$. If v_i is a constant, $v_i.tag = false$. Let π^t be the subtrace of π that is projected on Thread t. Taint is then propagated through π^t in a straightforward manner, e.g., the result of a binary operation such as + is tainted if either operand is tainted, an assigned variable is tainted if the right-hand side value is tainted, and so on. Different applications of taint analysis can use different policy decisions, and our encoding can be adapted to accommodate different taint policies. Note that the constraint only handles taint propagation within an individual thread. The inter-thread taint propagation is handled by interleaving matching constraint.

Interleaving Matching Constraint (φ_{im}). The program semantics and taint propagation constraints consider each thread individually. In multithreaded programs different threads communicate data via shared variables. The complexity of multithreaded programs is due to the non-deterministic nature of such communication. The purpose of interleaving matching constraint is to enumerate all possible matchings between read and write instructions of shared variables. Consider a shared variable v. Let $R(v)$ and $W(v)$ be the sets of reads and writes on v, respectively. We use v_r to denote the read of v at instruction r, and v_w the write of v at instruction w. In addition, let o_r and o_w be the order variables of r and w, and $v_r.tag$ and $v_w.tag$ be the taint tags. The interleaving matching constraint on v is:

$$\varphi_{im}^{v} = \bigwedge_{r \in R(v)} \bigvee_{w \in W(v)} \{(v_r = v_w \wedge v_r.tag = v_w.tag \wedge o_w < o_r) \wedge \bigwedge_{x \neq w \in W(v)} (o_x < o_w \wedge o_r < o_x)\}. \quad (11)$$

The constraint above describes that, v_r matches v_w if r is executed after w, and there are no other writes to v in between. If v_r matches v_w, the value read at r is the same as the value written at w, and taint propagates though this read-after-write chain. We name the component $v_r = v_w$ as *value matching* term and $v_r.tag = v_w.tag$ as *taint matching* term. Finally, let V be the set of shared variables, the interleaving matching constraint is:

$$\varphi_{im} = \bigwedge_{v \in V} \varphi_{im}^{v}. \quad (12)$$

Note that in our implementation value matching and taint matching terms do not always appear together. If a shared variable access instance does not depend on any tainted source, there is no need to include it in the interleaving matching constraint. If the purpose of encoding is to locate alternative branches, there is no need to include a value matching term if the corresponding variable instance does not depend on any branches. The data and control dependencies in sequential programs are well-defined. These classical definitions can be used for intra-thread dependence analysis. In order to address the inter-thread dependence relation, we compute vector clocks on a multi-threaded trace. For a read instance r and a write w instance on the same shared variable, if the vector clock does not mandate that r must happen before w, we consider RAW (read-after-write) may happen and there is a inter-thread data flow from w to r. A shared variable access instance depends on a tainted source or a branch if there exists any transitive intra- and inter-thread dependency. Our analysis is an over-approximation, which affects performance (unnecessary taint matching terms are added) but not correctness. Consider the code snippet shown in Fig. 10, where the set of shared variable access instances is $\{y@L3, x@L4, x@L6, y@L7\}$. For locating alternative branches, the set of instances that need be considered for value matching is $\{y@L3, y@L7\}$. It is disjoint from the set of instances $\{x@L4, x@L6\}$ that need to be considered for taint matching.

3.4 Alternative Path Search

The pseudo-code for `Alternative Path Search` is given in Algorithm 3. The formula φ_π, i.e., $\varphi_{sm} \wedge \varphi_{po} \wedge \varphi_{tp} \wedge \varphi_{im}$, is obtained from component `Trace Enlargement`. Two types of constraints are not needed for the search of alternative paths. Any constraints related to taint are not necessary, which include the taint propagation φ_{tp} and the taint

```
1  int a = taint;
2  create(foo, &a)
3  y = 4;
4  x = 1;
```

```
5  void* foo(a){
6  x = a
7  if(y > 3)
     ...}
```

Fig. 10. Code snippet with tainted a and shared variables x, y.

matching terms in the format of $v_r.tag = v_w.tag$. The second type of constraints are those terms that encode a branch condition. They must be removed, otherwise they cause conflicts with newly added branch constraints. For each $c_i \in C$, we check whether it can be the *first* branch in π that can be negated. That is, any branches before c_i must produce the same outcomes as in π. The potential path can be represented as

$$\varphi_\pi^{c_i} = \varphi'_\pi \wedge \neg c_i \wedge \bigwedge_{c_j \neq c_i} (o_j < o_i \rightarrow c_j). \tag{13}$$

In the above formula, φ'_π denotes φ_π with aforementioned two types of constraints removed, and o_i represents the order of c_i. If $\varphi_\pi^{c_i}$ is satisfiable, we extract its solution that gives the schedule up to c_i. Note that the schedule after c_i is invalid because the negation of c_i leads to unknown behavior that cannot be determined statically. However, the schedule prefix s_i up to c_i is valid and we save it as a to-be-explored schedule in S. Note that when conducting symbolic analysis on the new paths generated from s_i, we need to re-encode new constraints for the paths.

Algorithm 3. AltPathSearch(Formula φ_π, ScheduleSet S)

1: Remove branch constraints $C = \{c_1, c_2, \ldots, c_n\}$ from φ_π;
2: Remove constraints about taint propagation from φ_π;
3: **for** each c_i in C **do**
4: $\varphi_\pi^{c_i} = \varphi_\pi \wedge \neg c_i \wedge \bigwedge_{c_j \neq c_i}(o_j < o_i \rightarrow c_j)$.
5: **if** $\varphi_\pi^{c_i}$ is satisfiable **then**
6: Obtain a schedule s_i up to branch c_i from the solution of $\varphi_\pi^{c_i}$.
7: S.add(s_i);
8: **end if**
9: **end for**

For sequential programs, exhaustive search algorithms such as depth-first search (DFS) can iterate all possible execution traces by backtracking. Unfortunately such a strategy is not valid for multithreaded programs, otherwise valid executions can be missed. This is illustrated in Fig. 11 that shows an execution with multiple branches belonging to two threads t_1 and t_2. In the figure a solid circle denotes the branch being negated and dotted arrow indicates a new execution caused by the negation. Let Trace 1 be the initial execution with branch sequence $\langle b_1, b_2, b_3, b_4, b_5, b_6\rangle$. Assuming b_6, b_5 and b_4 cannot be negated, the first negation of b_3 leads to a Trace 2 with branch sequence $\langle b_1, b_2, \neg b_3, b_k, b_m, b_n\rangle$. If b_n in Trace 2 cannot be negated, the negation of b_m leads to Trace 3 with branch sequence $\langle b_1, b_2, \neg b_3, b_k, \neg b_m, b_h\rangle$. Suppose no other branches can be negated after Trace 3, the traversal terminates. This is a typical DFS procedure for exploring paths of sequential programs. Such traversal has an implicit assumption, which is the fixed order of the branches. That is, when a branch is negated, the relative order of the branches before the selected branch remain the same. The assumption is no longer valid for multithreaded programs. Consider a valid execution Trace 4, where b_k is negated. In order to negate b_k in thread $t1$, b_m in $t2$ has to happen *before* b_k. Note that in DFS b_k never

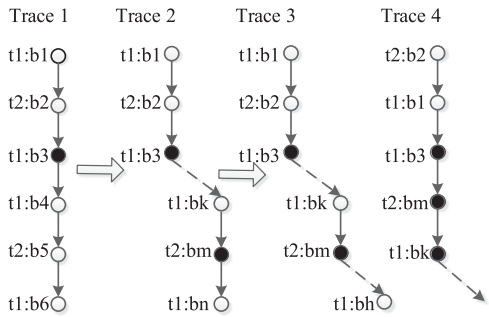


Fig. 11. Counter-example for depth-first search.

happens before b_m. This example shows when deciding whether a branch can be negated, we have to consider all the branches instead of only those branches that are executed before this branch. In addition, the positions of b_1 and b_2 are switched in Trace 4 as well. Therefore, even for those branches that are executed before the branch under consideration, their relative order may be changed. As a result, in our encoding when a branch b_i is selected, we only fix the outcomes of the branches before b_i to ensure b_i is the first negated branch in the new execution. We allow any branch to happen before b_i, and the order of the branches before b_i is arbitrary.

4 Empirical Study

We have implemented the proposed method in a software tool that targets multithreaded C programs implemented with the POSIX thread library. The tool is developed on top of LLVM [21], KLEE [22] and Z3 [23]. Note that KLEE does not by itself support multi-threading. Although Cloud9 [24] has extended KLEE to support a limited number of POSIX thread routines, it does not attempt to cover all feasible thread interleavings. Indeed, Cloud9 allows for thread context switches only before certain POSIX thread synchronizations but not before shared variable reads/writes. We have developed a customized thread scheduler to support multithreading. The scheduler is able to guide program executions under a given thread interleaving specification. We have also implemented a listener to collect executed instruction instances and an encoder to translate a trace into a logic formula. In particular, if the parameters of system functions and libraries are tainted, their return values are viewed as being tainted. In this paper, we don't consider control flow. In our evaluation, we do not handle the loops over shared variables. Though there is no this type of loops in our evaluation, it really be a threat to the completeness of proactive debugging.

In addition, we need to introduce the identification of shared variables in KLEE. For each variable, KLEE sets a corresponding memory object, one of whose fields labels whether the variable is global. When KLEE running a multithreaded program under a concrete program, we identify shared variables according to the labeling field. For we all know, KLEE provides no false negatives in identifying shared variables during concrete execution.

4.1 Experimental Setup

There has been very little research on dynamic taint analysis of multithreaded programs. The one that is closest to ours is DTAM [14]. In order to compare our approach against theirs we have re-implemented the three approaches proposed in DTAM. DTAM-serial (DS) serializes a multithreaded

execution, and then propagates taint tags along the serialized schedule. This approach basically ignores threading and leads to under-tainting, as discussed in Section 1. DTAM-parallel (DP) propagates a taint tag from one thread to another through write and read accesses to the same shared objects. Although DP implicitly captures the effects of many possible interleavings when merging the thread-modular results offline, the approach causes over-tainting because it ignores the happens-before relation. For example, if there are two statements from different threads and `y=x` must happen before `x = input()` through thread synchronization, DTAM-parallel still considers `y` tainted because of the shared variable `x`. DTAM-hybrid (DH) tracks synchronization operations and takes into account the must-happens-before relation to address the above over-tainting problem. It 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 relation during the aggregation step. The paper [14] shows that DH can be both over- and under-tainting, although it is more accurate than DP. Note that DP and DH can't generate interleaving schedules to reproduce the detected tainted instances. To conduct taint analysis on a single trace, our approach is reduced to predictive symbolic analysis without guided execution as there is no need to find alternative branches. That is, our approach becomes Symbolic Taint Analysis for Multithreaded Programs (STAM).

Since DTAM approaches (including DS, DP and DH) do not explore multiple traces, we integrate the three DTAM algorithms within our framework to conduct experiments on taint analysis under a single test input. In particular, we utilize the capability of `Alternative Path Search` (APS) to guide multiple paths, and then for each trace we apply DTAM algorithms instead of predictive symbolic analysis. We name the three DTAM+APS algorithms APS-DS, APS-DP, APS-DH, respectively. In summary, we have a total of eight algorithms to conduct taint analysis. Among them DS, DP, DH and STAM consider a single trace, while APS-DS, APS-DP, APS-DH and DSTAM consider all the feasible traces under a single input vector.

Our empirical study is conducted on 13 benchmarks obtained from application suites including SPLASH2 [26] and PARSEC [25]. Due to the scalability issue we are not able to complete all the SPLASH2 and PARSEC benchmarks within a 2-hour time limit. In order to conduct a sufficient comparison we add testing programs from other sources. Table 1 gives basic statistics regarding the benchmarks. Column *Prog* shows the names of the benchmarks and where they are obtained. Column *LOC* lists the lines of source code in a program. Note that our analysis is based on execution traces so *LOC* only gives indirect measurement of the size of our experiments. Column #*I* gives the average number of instructions in a trace. An assumption of our approach is program termination. A program with infinite loops cannot be analyzed with our offline approach. The number of threads is given in Column Th. Column Π gives the total number of paths under the given input. Column #*B* lists the number of branches on an interleaving on average. Column *NB* gives the percentage of flippable branches in an interleaving on average. The data show that only a small portion, with an average of 5.5 percent, of branch instances are scheduling-sensitive. For programs such as blacksholes, fft, lu-c and lu-nc, the portion is not more than 1 percent.

TABLE 1
Benchmarks

Prog.	LOC	Th	#I	Π	#B	NB
blackscholes [25]	639	5	16,914	1	167	0.0%
fft [26]	1,482	3	33,646	2	977	0.2%
lu-c [26]	1,401	3	22,594	2	854	0.2%
lu-nc [26]	1,182	3	10,265	2	2,020	0.1%
radix [26]	1,547	3	6,947	3	363	3.3%
aget [14]	1,157	3	5,744	2	355	6.7%
pbzip [14]	7,890	4	69,718	27	1,594	1.4%
pfscan [14]	998	3	6,206	30	54	17.2%
pcopy [27]	579	2	1,725	6	53	5.4%
pnscan [14]	1,190	2	588	1	10	0.0%
swarm [28]	2,286	5	44,064	1	766	0.0%
queue [29]	155	3	924	2	17	5.9%
stack [29]	109	3	486	33	24	31%
Avg.	1,095.6	3.2	16,909	7.8	484	5.5%

Table 2 provides the time usage of all approaches. Meanwhile, DSTAM computes a database that includes the propagation evidences of all the tainted instances. Each of them can be replayed in the prototype of DSTAM. Column *APS* lists the time consumption by alternative path search. Column *TE* stands for the time usage by trace enlargement. Column *Run* gives the time when executing instructions in KLEE. We can find that the time cost of constraint solving accounts for 99.7 percent of the total time cost on average. Hereon, we provide the time usage to admit that scalability is an issue with our approach. In terms of time cost, the DTAM approaches are more efficient. If an ideal SMT solver is provided, we believe that DSTAM can greatly improve scalability. In the rest of this section we only focus on the accuracy comparison.

4.2 Taint Analysis on a Single Trace

For approaches DS, DP, DH and STAM, they conduct taint analysis on a single trace. As shown in Table 3, we compare the four approaches in terms of accuracy when analyzing taint on a single trace. Columns with heading VI list the number of tainted instances. There are two group of numbers in each column. The numbers after the slash give the quantity of tainted instances in an executed trace. The numbers before the slash give the data after mapping the instances in a trace to source code. Since multiple instances in a trace can map to the same source code, the numbers in front of the slash are smaller. A variable in a statement is considered tainted if there exists at least one trace where the instance of the variable is tainted. Columns with headings FP and FN give the percentile of false positives and false negatives, respectively.

Without considering the limits of SMT solvers, such as the incapability of handling non-linear expressions, STAM is able to accurately detect all tainted instances of a multi-threaded trace (i.e., all the valid permutations of the instructions in the trace). The statistics for false positives and false negatives are relative to STAM. As illustrated in the last row, on average DS produces 14.6 and 16.4 percent false negatives in terms of source code and trace instances, respectively. DP produces 18.2 and 31.2 percent false positives, and DH produces 12.9 and 16.2 percent false positives. Note that in theory DS does not produce any false positives and DP does not

TABLE 2
Time Usage of Eight Algorithms

Prog.	DS	DP	DH	STAM	APS-DS	APS-DP	APS-DH	DSTAM	APS	TE	Run
blackscholes	0.042	0.044	0.045	2.046	15.924	15.926	15.926	17.970	15.839	2.046	0.085
fft	0.160	0.166	0.168	972.555	1953.068	1953.081	1953.086	3898.178	1952.270	1945.110	0.798
lu-c	0.060	0.063	0.064	1070.155	3465.131	3465.136	3465.138	5605.441	3464.830	2140.310	0.301
lu-nc	0.029	0.030	0.030	72.178	29.070	29.073	29.074	173.426	28.928	144.356	0.143
radix	0.025	0.026	0.026	11.602	105.958	105.961	105.962	140.765	105.463	34.807	0.495
aget	2.155	2.155	2.156	0.405	10.426	10.428	10.428	11.237	3.962	0.810	6.464
pbzip	0.271	0.335	0.337	0.052	1212.857	1214.573	1214.641	1214.271	1193.890	1.414	18.967
pfscan	0.018	0.019	0.019	0.274	35.597	35.620	35.627	43.823	32.314	8.227	3.283
pcopy	0.056	0.057	0.057	21.492	1065.827	1065.831	1065.832	1194.779	1065.150	128.952	0.677
pnscan	0.019	0.019	0.019	0.004	0.039	0.039	0.039	0.043	0.020	0.004	0.019
swarm	0.086	0.100	0.102	18.840	22.152	22.165	22.167	40.992	22.066	18.840	0.086
queue	0.002	0.002	0.003	0.027	0.086	0.087	0.087	0.136	0.077	0.049	0.007
stack	0.002	0.002	0.002	0.631	81.895	81.904	81.908	102.727	81.412	20.832	0.483
AVG	0.225	0.232	0.233	166.943	615.233	615.371	615.378	957.214	612.786	341.981	2.447

TABLE 3
Accuracy Comparison on Single Trace

Prog.	DS			DP			DH			STAM
	VI	FN	FP	VI	FN	FP	VI	FN	FP	VI
blackscholes	18/152	0.0%/0.0%	0	18/160	0	0.0%/5.3%	18/160	0	0.0%/5.3%	18/152
fft	42/582	10.6%/7.8%	0	52/780	0	10.6%/23.6%	49/680	0	4.3%/7.8%	47/631
lu-c	29/154	19.4%/14.0%	0	47/284	0	30.6%/58.7%	40/210	0	11.1%/17.3%	36/179
lu-nc	7/16	65.0%/68.0%	0	33/151	0	65.0%/202.0%	26/91	0	30.0%/82.0%	20/50
radix	24/71	11.1%/6.6%	0	32/107	0	18.5%/40.8%	32/103	0	18.5%/35.5%	27/76
aget	14/16	22.2%/23.8%	0	20/22	0	11.1%/4.8%	20/22	0	11.1%/4.8%	18/21
pbzip	8/9	0.0%/0.0%	0	8/9	0	0.0%/0.0%	8/9	0	0.0%/0.0%	8/9
pfscan	25/34	7.4%/5.6%	0	38/47	0	40.7%/30.6%	36/45	0	33.3%/25.0%	27/36
pnscan	19/22	0.0%/0.0%	0	19/22	0	0.0%/0.0%	19/22	0	0.0%/0.0%	19/22
pcopy	12/22	0.0%/8.3%	0	12/25	0	0.0%/4.2%	12/25	0	0.0%/4.2%	12/24
swarm	13/224	0.0%/0.0%	0	15/272	0	15.4%/21.4%	15/256	0	15.4%/14.3%	13/224
queue	2/8	33.3%/11.1%	0	4/10	0	33.3%/11.1%	4/10	0	33.3%/11.1%	3/9
stack	5/10	16.7%/67.7%	0	7/32	0	16.7%/3.2%	7/32	0	16.7%/3.2%	6/31
Avg.	17/109	14.6%/16.4%	0	23/148	0	18.2%/31.2%	22/128	0	12.9%/16.2%	20/113

produce any false negatives. That is, DS is under-tainting and DP is over-tainting.[2]

STAM considers only and all the feasible permutations of the events in a trace. We illustrate via some code snippets how imprecision happens using DS, DP and DH. Fig. 12 shows a false negative when applying DS on program *stack*. Since DS does not consider permutations of the trace $\langle 1, \ldots 4, 5 \ldots \rangle$, the variable instances of *top* are not claimed tainted at Lines 2 and 3. The result is not correct as they are tainted if Line 1 executes after Line 5.

Fig. 13 illustrates a false positive when applying DP on program *fft*. The variable instance $x[2]$ at Line 5 is tainted due to the potential data flow $2 \rightarrow 4 \rightarrow 5$. DP further claims that $x[2]$ at Line 8 is tainted. However, $x[2]$ at Line 8 can never read from $x[2]$ at Line 5.

One of the false positive reported by DH on program *lu-c* is illustrated in Fig. 14, where the variable instance $MyNum$ at Line 6 is actually not tainted. If the false branch at Line 2 is taken, $Global \rightarrow id$ at Line 3 is not executed at all, so $MyNum$ at Line 6 cannot be tainted by $Global \rightarrow id$ at Line 6 when $Global \rightarrow id$ at Line 6 reads the value assigned at Line 3. If the true branch at Line 2 is taken, Line 2 must execute after Line 6. Thus, it is impossible for $Global \rightarrow id$ at Line 6 to read from the value assigned at Line 3. However, DH reports that the variable instance $Global \rightarrow id$ at Line 6 is tainted because it fails to recognize this with happens-before implemented by vector clock.

2. DP can still be under-tainting under fixed inputs instead of fixed executions.

4.3 Taint Analysis For Single Input

In Table 4 we compare accuracy among the four approaches on all possible executions under a given input. Since reporting tainted instances on multiple traces is difficult to correlate instances from different runs, we map the instances to source code and report the statistics on tainted statements in a program only. Column *SV* gives the number of tainted static variables. Again, without considering the limits of SMT solvers, DSTAM is able to precisely detect all tainted instances under a given input. Therefore, the statistics for FP and FN in Table 4 are relative to DSTAM. As illustrated in the last row, on average APS-DS reports 10.3 percent false negatives, APS-DP reports 15.1 percent false positives, and APS-DH reports 10 percent false positives. The false positive rate for APS_DS, and the false negative rate for APS_DP and APS_DH are not included in the table as they are all zero.

4.4 Single Trace versus Single Input

Our approach enhances the capability of taint analysis by not just considering a single execution, but multiple executions

```
1 pthread_mutex_lock(&m);
2 if (top>0)
3    pop(arr); \\top--
4 pthread_mutex_unlock(&m);
```

```
5 top = tainted var;
6 pthread_mutex_lock(&m);
7 j=push(arr,55);\\top++
8 pthread_mutex_unlock(&m);
```

Fig. 12. Code snippet in *stack* with shared variables top, arr.

```
1  BARRIER(Global->start, P);
2  x[0] = tainted var;
   ...
3  BARRIER(Global->start, P);
4  x_r = x[0];
   ...
5  x[2] = omega_r*x_c + omega_c*x_r;
6  BARRIER(Global->start, P);
```

```
7  BARRIER(Global->start, P);
8  x_r = x[2];
   ...
9  x[3] = omega_r*x_c + omega_c*x_r;
10 BARRIER(Global->start, P);
   ...
   ...
11 BARRIER(Global->start, P);
```

Fig. 13. Code snippet in *fft* with shared variables x.

under a single input. To the best of our knowledge, we are the first to seek completeness for multithreaded programs under fixed inputs. In this section we compare the accuracy with and without multiple trace exploration. Without considering multiple traces, all four methods DS, DP, DH and STAM are under-tainting, even though they may infer other possible interleavings with the same set of instruction instances. The experimental results are given in Table 5. Column Δ gives the difference in the number of tainted statements between each pair of methods on single trace and multiple traces. Note that the tainted instances are mapped to statements in source code and the data are all referring to source code. Since there is only one trace under the given input for programs *blackscholes*, *pnscan* and *swarm*, APS-Dx methods and DSTAM are not able to find more tainted statements. In addition, scientific computation programs, including *fft*, *luc*, and *lunc*, have very few paths so the change of control flow has little impact on taint analysis. As illustrated in last row, on average DS, DP, DH, and STAM detect 12.3, 5.1, 5.1, and 6.8 percent less tainted statements than their counterparts for the 13 benchmarks. Note that the data may not reflect the actual difference between the methods as the tainted instances are mapped to static source code. The difference between the number of dynamic tainted instances is obviously greater.

4.5 DSTAM versus DS over a Same Time Bugdet

In this section, we compare DSTAM with DS on the number of tainted shared variables over a same time budget. The evaluation is conducted on two random programs. As shown in the Fig. 15, the *X*-axis and *Y*-axis stand for time consumption and the number of tainted variables, respectively. In this experiment, we repeatedly and randomly run programs and collect the new tainted shared variables in each run with DS. The total running time is same as that of DSTAM. For *radix*, both DSTAM and Random collect 24 tainted shared variables in the first execution. Later, Random does not encounter any new taint and DSTAM finds 3 more taints by performing symbolic analysis. For *stack*, both DSTAM and Random explore 5 tainted shared variables in the first execution. In the later exploration, Random finds 3 more taints and then keeps the same result.

```
1  ...
2  if(MyNum == 1)
3    Global->id = tainted
     var;
4  ...
```

```
5  LOCK(Global->idlock)
6  MyNum = Global->id;
7  Global->id ++;
8  UNLOCK(Global->idlock)
```

Fig. 14. Code snippet in *lu-c* with shared variables $Global, MyNum$, where $Global \rightarrow id = 1$ initially.

TABLE 4
Accuracy Comparison on Single Input (Multiple Traces)

Prog.	APS-DS		APS-DP		APS-DH		Ours
	SV	FN	SV	FP	SV	FP	SV
blackscholes	18	0.0%	18	0.0%	18	0.0%	18
fft	42	10.6%	52	10.6%	49	4.3%	47
lu-c	33	8.3%	47	30.6%	40	11.1%	36
lu-nc	12	42.9%	33	57.1%	26	23.8%	21
radix	26	3.7%	32	18.5%	32	18.5%	27
aget	15	21.1%	20	5.3%	20	5.3%	19
pbzip	8	0.0%	8	0.0%	8	0.0%	8
pfscan	26	7.1%	39	39.3%	37	32.1%	28
pnscan	19	0.0%	19	0.0%	19	0.0%	19
pcopy	12	0.0%	12	0.0%	12	0.0%	12
swarm	13	0.0%	15	15.4%	15	15.4%	13
queue	4	20.0%	6	20.0%	6	20.0%	5
stack	8	20.0%	10	0.0%	10	0.0%	10
Avg.	18	10.3%	24	15.1%	22	10.0%	20

DSTAM gets 5 more taints. From this result, we can conclude that DSTAM is not better than Random running in prophase, but out-performs it in post phase.

4.6 Discussion

From the above evaluation, we find that DTAM approaches are more efficient in terms of time cost, and DSTAM performs better in terms of precision. Furthermore, DSTAM has two advantages that DTAM and even current DTA methods do not have. On one hand, DSTAM offers the capability to answer queries such as whether a variable instance is tainted under an input while existing approaches cannot. On the other hand, DSTAM provides a witness interleaving for each tainted instance, which may contribute to exploit a concurrency attack. Given a multithreaded program and an input, our prototype tool creates a tainted instance database so the cost of multiple queries can be amortized.

Additionally, failing to solve the satisfiable constraint is a common issue in a great number of SMT solving problems. When a satisfiable constraint is too huge, the SMT solver may give an unsatisfiable or unknown result. We plan to handle the real-world programs in our future work. One possible solution is to set up the time budge to guarantee efficiency. Meanwhile, reducing the unnecessary constraint formulas, simplifying constraint, or over-approximation encoding, would be useful to improve the ability of SMT solving.

4.7 Validate Completeness

Our approach not only can compute an interleaving schedule to replay a scheduling-sensitive taint instances, but also can decide that an instance never can't be tainted under a given input. To validate the correctness and completeness of our approach, we compare it with the three methods of DTAM on the detection rate of injected malicious code.

TABLE 5
Comparison between Single Trace and Multiple Traces

Prog.	DS		APS-DS	DP		APS-DP	DH		APS-DH	STAM		DSTAM
	SV	Δ	SV	SV	Δ	SV	SV	Δ	SV	SV	Δ	SV
blackscholes	18	0.0%	18	18	0.0%	18	18	0.0%	18	18	0.0%	18
fft	42	0.0%	42	52	0.0%	52	49	0.0%	49	47	0.0%	47
lu-c	29	12.1%	33	47	0.0%	47	40	0.0%	40	36	0.0%	36
lu-nc	7	41.7%	12	33	0.0%	33	26	0.0%	26	20	4.8%	21
radix	24	7.7%	26	32	0.0%	32	32	0.0%	32	27	0.0%	27
aget	14	6.7%	15	20	0.0%	20	20	0.0%	20	19	0.0%	19
pbzip	8	0.0%	8	8	0.0%	8	8	0.0%	8	8	0.0%	8
pfscan	25	3.8%	26	38	2.6%	39	36	2.7%	37	27	3.6%	28
pnscan	19	0.0%	19	19	0.0%	19	19	0.0%	19	19	0.0%	19
pcopy	12	0.0%	12	12	0.0%	12	12	0.0%	12	12	0.0%	12
swarm	13	0.0%	13	15	0.0%	15	15	0.0%	15	13	0.0%	13
queue	2	50.0%	4	4	33.3%	6	4	33.3%	6	3	40.0%	5
stack	5	37.5%	8	7	30.0%	10	7	30.0%	10	6	40.0%	10
Avg.	16.8	12.3%	18.2	23.5	5.1%	23.9	22.0	5.1%	22.5	19.6	6.8%	20.2

These malicious codes are designed to achieve leaking sensitive information, totally including three types shown as in Figs. 16, 17 and 18. The details are as follows.

Type I gives a real leakage that avoids the detection of DS sometimes. When execution order in Fig. 16 is 1-3-4-2, the sensitive data is leaked. DS can detect the leakage under the execution. So the leakage is real. But DS fails to detect the leakage under execution 1-2-3-4. Type II gives an infeasible leakage that DP reports. Although only execution 1-2-3-4 is valid in Fig. 17, DP still falsely views `sendData(y)` as a leakage. Type III gives an infeasible leakage that DH report. Besides the synchronization statement in Type II, program semantics also can constrain the execution order between threads. In Fig. 18, if Line 3 takes True branch, Line 2 must happen before Line 3. Thus, Line 1 happens before Line 4. Since DH is incapable of analyzing semantics, it will view `sendData(y)` as a leaking statement. However, we believe that our approach can give a definite answer to whether there is an information leakage. The reason is that symbolic analysis is capable of predicting multiple interleavings and computing schedules that lead to the execution of new paths.

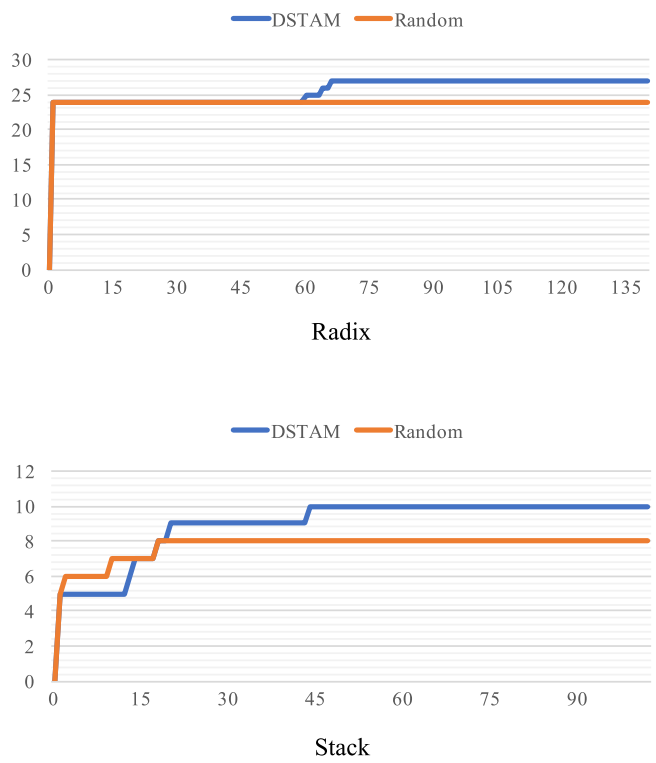


Fig. 15. DSTAM versus random running on the number of explored taints.

To validate that, we inject the malicious codes into the 13 benchmarks and assume that the taint source is sensitive information. If shared variable `y` at statement `sendData(y)` is tainted by taint source, we think the sensitive information is leaked. Therefore, we just need to check whether `y` at statement `sendData(y)` in the injected codes is tainted or not. We conduct comparison on the modified benchmarks between DSTAM and DTAM.

Table 6 gives the comparison results. Column # gives the number of seeded leakages for the corresponding type. Column *Dx* and *Ours* give the number of to-be-verified leakages detected by method D*x* and DSTAM, respectively. As we expected, DS just detects 23 percent Type I leakages. The concrete number of detection result is not most concerned to us since the execution of DS is random. Here, we just show the completeness of our method. DP and DH falsely treat the injected codes of Type II and Type III as real leakages, respectively. Our approach finds all of leakages completed by Type I codes. Meanwhile, it also definitely judge that both Type II and Type III can't achieve leaking information.

5 Related Work

In the past few years, most work on dynamic taint analysis has focused on either overhead reduction or accuracy improvement. Beside DTA techniques, we also describe related work on predictive analysis.

5.1 Overhead Reduction

The performance overhead mainly comes from two sources. First, as typical DTA tools [1], [14], [30], [31], [32], [33] are built on dynamic binary instrumentation (DBI) frameworks such as PIN [34], Valgrind [35], and DynamoRio [36], there are switches between original source code and instrumented code. Second, DTA tools need to track taint propagation during the runtime.

```
1  x = sensitive data;
2  x = safe data;
```

```
3  y = x;
4  sendData(y);
```

Fig. 16. Type I—False negative for DS.

```
1  sendData(y);
2  barrier();
```

```
3  barrier();
4  y = sensitive data;
```

Fig. 17. Type II—False positive for DP.

```
1  sendData(y);
2  x = 1;
```

```
3  if(x>0)
4    y = sensitive data;
```

Fig. 18. Type III—False positive for DH.

TABLE 6
Validate the Correctness of Our Approach

Prog.	Type I			Type II			Type III		
	#	DS	Ours	#	DP	Ours	#	DH	Ours
blackscholes	2	0	2	2	2	0	2	2	0
fft	2	1	2	2	2	0	2	2	0
lu-c	2	1	2	2	2	0	2	2	0
lu-nc	2	0	2	2	2	0	2	2	0
radix	2	1	2	2	2	0	2	2	0
aget	2	0	2	2	2	0	2	2	0
pbzip	2	1	2	2	2	0	2	2	0
pfscan	2	1	2	2	2	0	2	2	0
pnscan	2	0	2	2	2	0	2	2	0
pcopy	2	0	2	2	2	0	2	2	0
swarm	2	0	2	2	2	0	2	2	0
queue	2	1	2	2	2	0	2	2	0
stack	2	0	2	2	2	0	2	2	0
Total	26	6	26	26	26	0	26	26	0

Chang et al. propose the system that performs static inter-procedural flow analysis to determine points in a program where an attack could take place [37]. It then performs taint tracking instrumentation of these locations. Similarly, in [38], Lam et al. propose a compiler that performs taint instrumentation and requires source code. Our approach requires source code as well, so the optimization proposed in these tools can potentially be adopted to simplify constraint construction, which leads to performance enhancement.

TaintTrace [30] uses DynamoRIO instrumentation on binaries to minimize register spilling by making use of dead registers to store taint values. TaintEraser [1], based on PIN, increases the speed of taint propagation by using user annotated function summary. Ng et al. [39] propose to propagate taint information at the basic-block instead of at instruction level. Another approach to improve DTA performance is to parallelize taint tracing. In [40], Ruwase et al. present techniques on how to execute taint tracking code in parallel by relaxing the rules of taint propagation. Since our taint detection is mainly conducted by symbolic analysis, it is challenge to exploit such runtime optimization.

5.2 Accuracy Improvement

DTA techniques have several challenges in achieving accurate analysis results. Schwartz et al. [41] point out several fundamental challenges including under-tainting and over-tainting. A major cause of under-tainting is implicit flows caused by control dependencies, which have been studied since at least the 1970s. Examples of recent systems that attempt this include DTA++ [15] and Dytan [11], which take a subset of control dependencies into consideration and propagate the taint with those dependencies. Bao et al. [16] propose a concept of Strict Control Dependency to avoid false alarms. Taint Dependency Sequences Calculus [17] can be used to calculate a set of paths that need to be analyzed. These tools can find more tainted variables without causing the explosion of taints because they do not consider all the control dependencies. We do not consider implicit flows in DSTAM. However, this research is orthogonal to our work and can be added to DSTAM.

There has been very little research on taint analysis of multithreaded programs. The one closest to ours is DTAM [14] that considers the impact of thread scheduling on taint propagation. However, DTAM considers a single execution, which does not cover all the possible executions under a single input. Mounier et al. [42] propose an approach to predict the taint values on multithreaded program by performing sliding window based static analysis on a fragment of an execution trace. This approach requires more data to be collected during the runtime. In our approach we consider complete trace, although by sacrificing completeness we can adopt sliding window based static analysis. Butterfly Analysis [43], a model to formalize multithreaded program, is built on a log based architecture [44] to perform taint analysis on multithreaded program. Our work considers not only the impact of scheduling non-determinism on a trace, but also its impact on branches that leads to new traces. To the best of our knowledge, we are the first to attempt a sound and complete taint analysis for multithreaded programs under a fixed input.

5.3 Predictive Analysis

There is also a large body of work on predictive analysis of the execution traces of a concurrent program. Wang et al. [45], [46] introduce the first method for *symbolic predictive analysis*, where they leverage SMT solvers to conduct trace-based predictive analysis symbolically. Subsequently, Farzan et al. [47] develop a tool called ExceptioNULL, which leverages constraint solving based techniques to detect concurrency bugs that result in null-pointer dereferences. PECON [48] is a pattern-directed tool for searching a partial and temporal order graph extracted from an execution trace, to detect general access anomalies and produce the corresponding thread schedules. CLAP generates a thread schedule to reproduce concurrent bugs by encoding failed executions [49]. Huang et al. leverage constraint solving to detect data race on a trace, and take into account the control flow information for a higher race detection power [50]. Some researchers have focused on how to fix concurrency bug with constraint solving. For example, Wang et al. use unsatisfiability core to automatically locate the root causes of the failing executions, and then compute the potential repairs [51]. In order to fix atomicity violation bug, Shi et al. verify whether synchronization is sufficient in the failing executions with the SMT solver [52]. However, these aforementioned methods and other predictive analysis based techniques known to us only conduct analysis on the logged execution traces without inferring new traces with different instructions. In contrast, our new method leverages predictive symbolic analysis to guide the new trace exploration, and then leverages guided execution to

drive the predictive symbolic analysis, to construct a mutually benefiting feedback loop.

Another tool that is related to our tool is TAME [53], which aims at identifying schedule-sensitive branches in concurrent programs, i.e., branches whose decision may vary depending on the actual thread schedule. However, it does not have the synergistic composition of symbolic analysis and guided execution, which is the main contribution of this work.

5.4 Systematic Testing

Systematic testing technique aims to investigate a unique scheduling in each run by controlling thread scheduling with a predefined coverage information. It terminates when the predefined coverage is reached. Generally speaking, this technique has been developed in two major categories, namely, coverage-driven [54], [55], [56], [57], [58], [59] and stateless model checking [29], [60], [61], [62]. For example, Wang et al. learn access patterns of shared variables on good runs to capture the already tested concurrency scenarios and cover the untested scenarios with the guide of the tested scenarios [59]. Musuvathi et al restricted preemptive context switches by using a small bounded number to significantly alleviate the state explosion [60], [61]. It has become an influential technique in practice since many concurrency bugs can be triggered by interleavings with few context switches. If DTA is combined with systematic testing, we can capture more diverse data flow, which leads to the detection of more tainted instances. Unfortunately, since the number of possible interleavings can be enormous (w.r.t. a given test input), explicitly exhausting all the interleavings is a nearly impossible mission [59]. Even if these techniques often stop after a specified coverage criterion or context switch bound is reached, it is still possible for some deeply hidden to-be-tainted instances to remain undetected. However, via explicitly and implicitly investigating the interleavings under a given input, DSTAM can analyze every tainted instance in the whole interleaving space under a given input.

6 Conclusion

We have presented a new taint analysis method called DSTAM for multithreaded programs. By integrating symbolic analysis with dynamic execution, our method is able to offer systematic and complete coverage of the program behaviors under a given input. If a tainted instance is scheduling-sensitive, DSTAM can not only detect it but also generate a schedule to reproduce it. The experimental evaluation proves that DSTAM over-performs DTAM [14] on accuracy comparison. To the best of our knowledge, DSTAM is the first tool that is able to give a definite answer to whether an instance is tainted under a fixed input in a concurrent program. In fact, the challenges and problems in parallel software are intrinsically complex and are impossibly solved by one step. This paper just proposes a promising solution for taint analysis of multithreaded programs. In order to handle scalable software, we plan to design some strategies to enhance scalability in the future work, such as reducing the explosive search space. In addition, we will evaluate the effectiveness of DTA + systematic testing for scalability, such as coverage-driven testing.

Acknowledgments

This work was supported by the National Key R&D Program of China (2016YFB1000903), National Natural Science Foundation of China (61632015, 61772408, U1766215, U1736205, 61721002, 61472318, 61532015), Fok Ying-Tong Education Foundation (151067), Ministry of Education Innovation Research Team (IRT_17R86), and Project of China Knowledge Centre for Engineering Science and Technology.

References

[1] 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.*, vol. 45, no. 1, pp. 142–154, Feb. 2011.

[2] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proc. Netw. Distrib. Syst. Security Symp.*, 2005.

[3] W. Xu, S. Bhatkar, and R. Sekar, "Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks," in *Proc. 15th Conf. USENIX Security Symp.*, 2006, pp. 121–136.

[4] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Improving application security with data flow assertions," in *Proc. ACM SIGOPS 22nd Symp. Operating Syst. Principles*, 2009, pp. 291–304.

[5] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August, "Rifle: An architectural framework for user-centric information-flow security," in *Proc. 37th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2004, pp. 243–254.

[6] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: Capturing system-wide information flow for malware detection and analysis," in *Proc. 14th ACM Conf. Comput. Commun. Security*, 2007, pp. 116–127.

[7] M. Fan, J. Liu, X. Luo, K. Chen, Z. Tian, Q. Zheng, and T. Liu, "Android malware familial classification and representative sample selection via frequent subgraph analysis," *IEEE Trans. Inf. Forensics Security*, vol. 13, no. 8, pp. 1890–1905, Aug. 2018.

[8] "Cve-2011-0990," [Online]. Available: http://www.cvedetails.com/cve/CVE-2011-0990, Apr. 2011.

[9] J. Yang, A. Cui, S. J. Stolfo, and S. Sethumadhavan, "Concurrency attacks," in *Proc. 4th USENIX Workshop Hot Topics Parallelism*, 2012, Art. no. 15.

[10] Z. Tian, T. Liu, Q. Zheng, E. Zhuang, M. Fan, and Z. Yang, "Reviving sequential program birthmarking for multithreaded software plagiarism detection," *IEEE Trans. Softw. Eng.*, vol. 44, no. 5, pp. 491–511, Mar. 2018.

[11] J. Clause, W. Li, and A. Orso, "Dytan: A generic dynamic taint analysis framework," in *Proc. Int. Symp. Softw. Testing Anal.*, 2007, pp. 196–206.

[12] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *Proc. 31st Int. Conf. Softw. Eng.*, 2009, pp. 474–484.

[13] D. Song, D. Brumley, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," in *Proc. 4th Int. Conf. Inf. Syst. Security*, 2008, pp. 1–25.

[14] M. Ganai, D. Lee, and A. Gupta, "DTAM: Dynamic taint analysis of multi-threaded programs for relevancy," in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng.*, 2012, pp. 46:1–46:11.

[15] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, "Dta++: Dynamic taint analysis with targeted control-flow propagation," in *Proc. Netw. Distrib. Syst. Security Symp.*, 2011.

[16] T. Bao, Y. Zheng, Z. Lin, X. Zhang, and D. Xu, "Strict control dependence and its effect on dynamic information flow analyses," in *Proc. 19th Int. Symp. Softw. Testing Anal.*, 2010, pp. 13–24.

[17] D. Ceará, L. Mounier, and M.-L. Potet, "Taint dependency sequences: A characterization of insecure execution paths based on input-sensitive cause sequences," in *Proc. 3rd Int. Conf. Softw. Testing Verification Validation Workshops*, 2010, pp. 371–380.

[18] D. Lee, M. Said, S. Narayanasamy, Z. Yang, and C. Pereira, "Offline symbolic analysis for multi-processor execution replay," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2009, pp. 564–575.

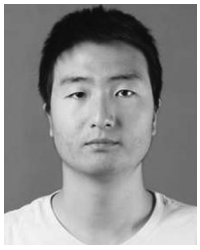
[19] D. Lee, M. Said, S. Narayanasamy, and Z. Yang, "Offline symbolic analysis to infer total store order," in *Proc. 17th Int. Conf. High-Perform. Comput. Archit.*, 2011, pp. 357–358.
[20] X. Zhang, Z. Yang, Q. Zheng, Y. Hao, P. Liu, L. Yu, M. Fan, and T. Liu, "Debugging multithreaded programs as if they were sequential," in *Proc. Int. Conf. Softw. Anal. Testing Evol.*, Nov. 2016, pp. 78–83.
[21] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Generation Optimization: Feedback-Directed Runtime Optimization*, 2004, Art. no. 75.
[22] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. 8th USENIX Conf. Operating Syst. Des. Implementation*, 2008, pp. 209–224.
[23] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proc. Theory Practice Softw. 14th Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2008, pp. 337–340.
[24] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea, "Cloud9: A software testing service," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 4, pp. 5–10, Jan. 2010.
[25] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proc. 17th Int. Conf. Parallel Archit. Compilation Tech.*, 2008, pp. 72–81.
[26] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," in *Proc. 22nd Annu. Int. Symp. Comput. Archit.*, 1995, pp. 24–36.
[27] "pcopy," [Online]. Available: ftp://ftp.lysator.liu.se/pub/unix/pcopy, Dec. 2003.
[28] N. Jalbert and K. Sen, "A trace simplification technique for effective debugging of concurrent programs," in *Proc. 18th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2010, pp. 57–66.
[29] L. Cordeiro and B. Fischer, "Verifying multi-threaded software using smt-based context-bounded model checking," in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, pp. 331–340.
[30] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige, "Tainttrace: Efficient flow tracing with dynamic binary rewriting," in *Proc. 11th IEEE Symp. Comput. Commun.*, 2006, pp. 749–754.
[31] J.-W. Min, Y.-H. Choi, J.-H. Eom, and T.-M. Chung, "Explicit untainting to reduce shadow memory usage and access frequency in taint analysis," in *Computational Science and Its Applications*. Berlin, Germany: Springer, 2013, pp. 175–186.
[32] J. Ma, P. Zhang, G. Dong, S. Shao, and J. Zhang, "Twalker: An efficient taint analysis tool," in *Proc. 10th Int. Conf. Inf. Assurance Security*, 2014, pp. 18–22.
[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. 39th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2006, pp. 135–148.
[34] 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. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2005, pp. 190–200.
[35] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proc. 28th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2007, pp. 89–100.
[36] D. L. Bruening, "Efficient, transparent, and comprehensive runtime code manipulation," Ph.D. dissertation, Dept. of Electrical Engineering and Computer Science, Massachusetts Inst. Technol., Cambridge, MA, 2004.
[37] W. Chang, B. Streiff, and C. Lin, "Efficient and extensible security enforcement using dynamic data flow analysis," in *Proc. 15th ACM Conf. Comput. Commun. Security*, 2008, pp. 39–50.
[38] L. C. Lam and T.-C. Chiueh, "A general dynamic information flow tracking framework for security applications," in *Proc. 22nd Annu. Comput. Security Appl. Conf.*, 2006, pp. 463–472.
[39] B. H. Ng, E. Fernandes, A. Aluri, D. Velazquez, Z. Yang, and A. Prakash, "Beyond instruction level taint propagation," in *Proc. 6th ACM Eur. Workshop Syst. Security*, 2013.
[40] O. Ruwase, P. B. Gibbons, T. C. Mowry, V. Ramachandran, S. Chen, M. Kozuch, and M. Ryan, "Parallelizing dynamic information flow tracking," in *Proc. 20th Annu. Symp. Parallelism Algorithms Architectures*, 2008, pp. 35–45.
[41] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Proc. IEEE Symp. Security Privacy*, 2010, pp. 317–331.
[42] L. Mounier and E. Sifakis, "Dynamic information-flow analysis for multi-threaded applications," in *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*. Berlin, Germany: Springer, 2012, pp. 358–371.
[43] 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," *ACM SIGARCH Comput. Archit. News*, vol. 38, no. 1, pp. 257–270, 2010.
[44] 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," *ACM SIGARCH Comput. Archit. News*, vol. 36, no. 3, pp. 377–388, 2008.
[45] C. Wang, S. Kundu, M. Ganai, and A. Gupta, "Symbolic predictive analysis for concurrent programs," in *Proc. 2nd World Congr. Formal Methods*, 2009, pp. 256–272.
[46] C. Wang, R. Limaye, M. Ganai, and A. Gupta, "Trace-based symbolic analysis for atomicity violations," in *Proc. 16th Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2010, pp. 328–342.
[47] A. Farzan, P. Madhusudan, N. Razavi, and F. Sorrentino, "Predicting null-pointer dereferences in concurrent programs," in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng.*, 2012, pp. 47:1–47:11.
[48] J. Huang and C. Zhang, "Persuasive prediction of concurrency access anomalies," in *Proc. Int. Symp. Softw. Testing Anal.*, 2011, pp. 144–154.
[49] J. Huang, C. Zhang, and J. Dolby, "Clap: Recording local executions to reproduce concurrency failures," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 141–152, 2013.
[50] J. Huang, P. O. Meredith, and G. Rosu, "Maximal sound predictive race detection with control flow abstraction," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2014, pp. 337–348.
[51] S. Khoshnood, M. Kusano, and C. Wang, "Concbugassist: Constraint solving for diagnosis and repair of concurrency bugs," in *Proc. Int. Symp. Softw. Testing Anal.*, 2015, pp. 165–176.
[52] Q. Shi, J. Huang, Z. Chen, and B. Xu, "Verifying synchronization for atomicity violation fixing," *IEEE Trans. Softw. Eng.*, vol. 42, no. 3, pp. 280–296, Mar. 2016.
[53] J. Huang and L. Rauchwerger, "Finding schedule-sensitive branches," in *Proc. 10th Joint Meet. Found. Softw. Eng.*, 2015, pp. 439–449.
[54] H. Wang, T. Liu, X. Guan, C. Shen, Q. Zheng, and Z. Yang, "Dependence guided symbolic execution," *IEEE Trans. Softw. Eng.*, vol. 43, no. 3, pp. 252–271, Mar. 2017.
[55] S. Hong, J. Ahn, S. Park, M. Kim, and M. J. Harrold, "Testing concurrent programs to achieve high synchronization coverage," in *Proc. Int. Symp. Softw. Testing Anal.*, 2012, pp. 210–220.
[56] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam, "Maple: A coverage-driven testing tool for multithreaded programs," in *Proc. ACM Int. Conf. Object Oriented Program. Syst. Lang. Appl.*, 2012, pp. 485–502 .
[57] J. Yu and S. Narayanasamy, "A case for an interleaving constrained shared-memory multi-processor," in *Proc. 36th Annu. Int. Symp. Comput. Archit.*, 2009, pp. 325–336.
[58] M. Fan, J. Liu, W. Wang, H. Li, Z. Tian, and T. Liu, "Dapasa: Detecting android piggybacked apps through sensitive subgraph analysis," *IEEE Trans. Inf. Forensics Security*, vol. 12, no. 8, pp. 1772–1785, Aug. 2017.
[59] C. Wang, M. Said, and A. Gupta, "Coverage guided systematic concurrency testing," in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, pp. 221–230.
[60] M. Musuvathi and S. Qadeer, "Iterative context bounding for systematic testing of multithreaded programs," in *Proc. 28th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2007, pp. 446–455.
[61] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing heisenbugs in concurrent programs," in *Proc. 8th USENIX Symp. Operating Syst. Des. Implementation*, 2008, pp. 267–280.
[62] K. E. Coons, S. Burckhardt, and M. Musuvathi, "Gambit: Effective unit testing for concurrency libraries," *ACM SIGPLAN Notices*, vol. 45, no. 5, pp. 15–24, 2010.

Xiaodong Zhang received the BS degree in network engineering from Southwest University, China, in 2011. He is currently working toward the PhD degree in the Department of Computer Science and Technology, Xi'an Jiaotong University, China. His research interests include trustworthy software and analysis of multithread programs.

Zijiang Yang received the BS degree from the University of Science and Technology of China, the MS degree from Rice University, and the PhD degree from the University of Pennsylvania. He is a professor of computer science with Western Michigan University. Before joining WMU, he was an associate research staff member at NEC Labs America. He was also a visiting professor with the University of Michigan from 2009 to 2013. His research interests include software engineering with the primary focus on the testing, debugging, and verification of software systems. He is a senior member of the IEEE.

Qinghua Zheng received the BS degree in computer software in 1990, the MS degree in computer organization and architecture in 1993, and the PhD degree in system engineering in 1997 from Xi'an Jiaotong University, China. He was a postdoctoral researcher at Harvard University in 2002. He is currently a professor in Xi'an Jiaotong University. His research areas include computer network security, intelligent e-learning theory and algorithm, multimedia e-learning, and trustworthy software. He is a member of the IEEE.

Yu Hao received the BS degree in automation science and technology from Xi'an Jiaotong University, China, in 2015. He is currently working toward the PhD degree in the Department of Computer Science and Technology, Xian Jiaotong University, China. His research interests include trustworthy software and analysis of multithread programs.



Pei Liu received the BS degree in computer science and technology from Sichuan University, China, in 2014. He is currently working toward the MS degree in the Department of Computer Science and Technology, Xi'an Jiaotong University, China. His research interests include trustworthy software and analysis of multithread programs.

Ting Liu received his BS degree in information engineering and PhD degree in system engineering from Xi'an Jiaotong University, Xi'an, China, in 2003 and 2010, respectively. Currently, he is an associate professor in Xi'an Jiaotong University. His research interests include CPS security and software security. He is a member of the the IEEE.