**Abstract**—We present MagicFuzzer, a novel dynamic deadlock detection technique. Unlike existing techniques to locate potential deadlock cycles from an execution, it iteratively prunes lock dependencies that each has no incoming or outgoing edge. Combining with a novel thread-specific strategy, it dramatically shrinks the size of lock dependency set for cycle detection, improving the efficiency and scalability of such a detection significantly. In the real deadlock confirmation phase, it uses a new strategy to actively schedule threads of an execution against the whole set of potential deadlock cycles. We have implemented a prototype and evaluated it on large-scale C/C++ programs. The experimental results confirm that our technique is significantly more effective and efficient than existing techniques.

**Keywords**—deadlock detection; multithreaded programs.

I. INTRODUCTION

A multithreaded C/C++/Java program may use locks to coordinate its threads. However, some improper uses of locks in the code may lead to concurrency bugs [13][15]. Deadlocks [1][2][9][10][18] are severe problems that lead multithreaded programs (or their components) to fail to make further progress if deadlocks are formed. In general, there are two kinds of deadlocks: resources deadlock [1][10] and communications deadlocks [9]. A resource deadlock occurs when a set of threads is holding some resources and is waiting for the resources which have already been held by the threads in the same set. A communication deadlock occurs when one or more threads wait for some messages/signals from other threads, which are paused and unable to send the required messages or have already sent the messages/signals before a waiting thread starts to wait for the messages/signals. In this paper, we focus on resource deadlocks where locks are resources.

Potential deadlocks can be detected via static analysis [12][20][26], model checking [7], dynamic analysis [2], runtime monitoring [25], or their integration [1][9]. Analyses based on lock order graphs [15] or their integrations with the use of the happens-before relation have been explored [2]. Methods to confirm whether a potential deadlock is real [3][5][10][18] and to avoid or heal deadlocks [11][18][25] have been studied.

It has been well discussed in the above-mentioned references that different categories of techniques complement one another. In general, static detection techniques and model checking for deadlock detection can analyze the whole program including open framework; whereas, dynamic techniques are more precise and more scalable. Dynamic confirmation techniques are valuable to confirm a potential deadlock if it is a real one, but they could not help to rule out a potential deadlock (as a false alarm). Avoidance and healing techniques are often pattern-based, which may not precisely quantify deadlock conditions. They may produce false positive cases, which slow down an execution further, or cannot prevent a deadlock to re-occur.

We observe that the many modern deadlock detection techniques such as MulticoreSDK [15] or DeadlockFuzzer [10] firstly use lockset-based strategies to predict potential deadlocks. Once a potential deadlock has been found, deadlock confirmation, avoidance, or healing strategies can be applied. However, without analyzing an execution successfully, such a technique cannot report any potential deadlocks for the subsequence steps to take actions.

Many large-scale applications such as OpenOffice [19], Chromium [4], Firefox [6], MySQL [16], SQLite [22] and Thunderbird [24] are widely-used. A deadlock bug in such a program may affect millions of users. However, due to the sheer sizes of large-scale programs, the probabilities of a run from exhibiting a thread holding a lock for a particular deadlock (because there are many locks in a program), that for such a lock occurred a right time to trigger a deadlock, and that of all such locks simultaneously occurred in the run can be all low. It poses challenges to dynamic deadlock detections.

In this paper, we present our technique, which is known as MagicFuzzer. MagicFuzzer consists of three phases. In Phase I (see Section IV.A), it executes a given program $p$, monitors the critical events (i.e., thread creation as well as lock acquisition and release), and generates a log consisting of a series of lock dependencies (see Section III.B for definition). This log can be viewed as a lock dependency relation $D$. In Phase II, it uses the Magiclock algorithm (see Section IV.B) to find potential deadlock cycles from $D$. Magiclock firstly classifies all the locks appearing in $D$ into four sets using an innovative and highly efficient algorithm. In particular, after our iterative classification, one (which is called cyclic-set) of the four sets must contain all the target lock dependencies (i.e., all the locks that may occur in any potential deadlock cycles in the monitored execution). We interestingly observe that (1) each thread can only occur once in a cycle, (2) multiple threads form an order in every permutation of a cycle, and (3) detecting one permutation of the same cycle suffices to represent the cycle. Magiclock explores this insight, and constructs a set of thread-specific
lock-dependency relations based on the locks in cyclic-set. In Magiclock, we propose a novel depth-first-search algorithm to traverse every such thread-specific lock-dependency relation to find cycles. All such cycles will form a set (denoted by CycleSet) of potential deadlock cycles. In Phase III (see Section IV.C) MagicFuzzer accepts CycleSet as an input, and actively executes p with the aim of triggering the occurrence of one or multiple potential deadlock cycles in CycleSet in single execution. If a real deadlock occurs in this phase, MagicFuzzer report it.

The main contribution of this paper is three-fold. First, we propose a novel and elegant technique MagicLock to detect potential deadlock cycles from an execution. Second, we present MagicFuzzer. Unlike existing active scheduling strategies for deadlock detection, it can schedule threads against a set of cycles, with the aim of improving the probability of finding a match between a cycle and an execution. Third, we have implemented MagicFuzzer as a C++ tool, and shows that the tool can analyze executions of a suite of widely-used and large-scale C/C++ programs efficiently with very manageable memory consumption (compared to the other techniques in the experiment).

The rest of this paper is organized as follows. Section II shows a motivating example. Section III presents the basic terminology. Our MagicFuzzer technique will be presented in Section IV. Section IV presents our experiment to validate MagicFuzzer, followed by a discussion on related work in Section V. Section VI concludes this paper.

II. MOTIVATING EXAMPLE

Example A: We motivate our work via the example adapted from [15] as shown in Figure 1. The example includes two functions doubleLock and tripleLock, three threads (t₁, t₂ and t₃), and seven locks (l₁–l₇). The thread t₁ calls doubleLock twice, the thread t₂ accesses l₆ and l₇ in a nested manner, and the thread t₃ calls doubleLock followed by calling tripleLock twice.

Suppose that during the call to doubleLock(h, b), t₁ acquires l₁ at s₁ followed by t₂ acquiring l₁ at s₁₀. Then, t₁ wants to acquire l₆ at s₁₀, which is blocked by t₂. Similarly, t₂ wants to acquire l₇ at s₁₀, which is blocked by t₃. These form a deadlock. In the rest of the example, t₁ invokes doubleLock(h, h). However, t₁ cannot acquire l₁ successfully because t₃ is holding l₁. The entire execution ceases to proceed further.

In a lock order graph [1][2], a node represents a lock. For instance, the two nodes labeled as h₁ and h₉ present the two lock h₁ and h₉ in Figure 1, respectively. The directed edge from node h₁ to node h₉ is associated with a set of labels (e.g., t₁ as a label), representing that, during the above execution, the thread t₁ acquires the lock h₁ while holding the lock m₁. For instance, t₁ is holding m₁ when it acquires h₁, and so, there is an edge from node t₃ to node t₁. For simplicity, we do not show the other information on an edge in the rest of the paper.

Goodlock [1][2]: To detect a deadlock in the above execution, Goodlock firstly constructs a lock order graph to detect whether there is any cycle on the graph. The lock order graph for the example is shown in Figure 2(a). We also highlight a detected cycle using dotted edges.

Following [10], in the rest of this paper, we refer to such a cycle as a potential deadlock cycle (or simply cycle).

Directly checking on a traditional lock order graph for large-scale program is impractical. For instance, Luo et al. [15] reported that such a graph for the ITCAM application contained over 300K nodes and 600K edges, and Goodlock spent 48 hours and 13.6 GByte memory to traverse it to find cycles if they exist [15].

MulticoreSDK [15] is a most recent technique based on lock order graph. It employs a two-phase strategy to address the scalability problem. It firstly groups the locks being held by different threads at the same code location in the same group, and then merges multiple groups into the same group whenever they have at least one shared lock, resulting in a location-based lock order graph (see Figure 2(b)), on which MulticoreSDK locates whether any cyclic dependencies among these groups exist. In Figure 2(b), Groups A and B form a cycle. Then, MulticoreSDK only consider the locks in these groups (i.e., l₁, l₂, l₃, l₄, and l₅) in its second phase, where it constructs a traditional lock order graph (Figure 2(c)).

Finding all cycles on a digraph has been well-researched such as applying the Tarjan algorithm [23] (which is also

---

**Figure 1. Example program (adapted from [15])**

**Figure 2. Lock order graph example (S, in (b) presents the code line x where the corresponding lock is acquired; edges in cycles are shown in dotted lines)**
optimal. As highlighted by the above experience on ITCAM in
Luo et al. [15], one key challenge is to generate a small
digraph (as small as possible) to apply such an algorithm on
it. From Figure 2(c), we observe that the graph used by
MulticoreSDK to search for cycles is far from optimal. For
instance, none of $l_1$, $l_2$, and $l_3$ on the graph has any outgoing
edge — they cannot be involved in any cycle, and yet they
appear on the graph. In Figure 2(b), they belong to Group $A$,
which also contains $l_2$. It has no information to eliminate
these locks from $A$ to reduce the graph in Figure 2(c) further.

**DeadlockFuzzer** [10]: $iGoodlock$ is the core component of
DeadlockFuzzer to identify cycles. It however searches for
cycles on the full permutations of the whole set of lock
dependencies [10] generated from an execution trace (with a
heuristic pruning strategy), and only suppresses the detected
but duplicated cycles (rather than preventing them by
design). In our experiment (see Section V), $iGoodlock$ is
found to consume all the memory that a Linux process is
allowed to consume, and crashes before returning any cycle,
making the phase two of DeadlockFuzzer even unable to
start because no input (i.e., cycles annotated with their object
abstractions as contexts) has been generated by $iGoodlock$.

Once a set of cycles has been identified by $iGoodlock$,
DeadlockFuzzer selects cycles one by one, and actively (but
biased-randomly) schedules a run to confirm whether the
selected cycle is a real deadlock. We observe that its
probability of successfully matching a cycle with an
execution depends on (1) not only how the algorithm
schedules the execution (2) but also whether a cycle that can
match with the execution has been selected to check against
the execution (which is fixed before an invocation of the
algorithm is started). If the probability of producing an
execution that matches any potential deadlock cycle in the
identified cycle set is not high, and there are many cycles in
the cycle set, the probability of “hitting” a right combination
is, intuitively, low.

**Our technique (this paper):** To find cycles, Magiclock
of our technique iteratively removes the lockset $\{l_1, l_2, l_3\}$
and their edges followed by removing $\{l_1, l_2\}$ and their
edges, resulting in a set of lock dependencies that precisely
represents the lock order graph as shown in Figure 2(d). Note
that, after the first round of graph pruning to remove $\{l_1, l_2, l_3\}$,
this intermediate lock order graph is already smaller than the
corresponding result of MulticoreSDK. Moreover, to
reduce the size of the set of lock dependencies for a cycle
detection algorithm to work on, Magiclock executes this step
by a new thread-specific strategy (See Section IV.B.2). To
improve the probability of hitting a “match” to address the
above active scheduling problem, our algorithm works at the
cycle set level rather than merely picking one cycle to pair
with the execution subject to active thread scheduling.

III. PRELIMINARIES

In this section, we revisit the basic definitions.

A. Monitoring Events and Execution Trace

Given an execution of a multithreaded program $p$, we use
t $\in$ Tid to identify a thread and $m \in$ Lock to identify a lock in
the execution. A lockset $L$ is defined as $\{m \mid m \in$ Lock$\}$,
representing a set of locks. We also denote the set of thread
identifiers and the set of locks in $D$ by $D$.Tid and $D$.Lock,
respectively. Similar to [10][15], MagicFuzzer monitors the
following three kinds of critical events:

- create($t$): a new thread $t$ is created;
- acquire($t$, $m$): the thread $t$ acquires the lock $m$;
- release($t$, $m$): the thread $t$ releases the lock $m$.

An execution trace $\sigma_p$ is a sequence of such acquire($t$, $m$)
and release($t$, $m$) events.

B. Lock Dependency Relation

DeadlockFuzzer [10] uses a lock dependency relation to
model an execution trace. The phase one of our technique
also uses a kind of lock dependency relation to describe an
execution. Our lock dependency relation is as follows:

A lock dependency relation $D$ for $\sigma_p$ is a set of lock
dependencies on $\sigma_p$. A lock dependency $\tau = \langle t$, $m$, $L$ $\rangle$
is a triple that contains a thread $t$, a lock $m$, and a lockset $L$ such
that the thread $t$ acquires a lock $m$ while holding all the locks
in the lockset $L$. In Example A, at the execution step where
$T$ acquires the lock $L$ at line $2$ while holding the lockset $\{T1\}$
at line $1$ via calling doubleLock($T$, $L$), the corresponding lock
dependency is $\langle T1$, $L$, $\{T1\} \rangle$.

Given a lock dependency $\langle t$, $m$, $L$ $\rangle$, from the perspective
of lock order graph [15], a lock $m$ in $L$ represents an edge
from node $n$ to node $m$ on such a graph. A lock dependency
$\langle t$, $m$, $L$ $\rangle$ has a correspondence with the set of edges from $n_i$
(for all $n_i \in L$) to $m$ in a lock order graph. The cardinality of
this set of edges is the same as that of the lockset $L$. We
simply refer to the cardinality of $L$ as $|L|$. Note that in
general, a lock order graph may contain multiple sets of
nodes (say $L_1$ and $L_2$) that each forms a lock dependency
with $m$ (where $t$ is a label of such an edge), and they contain
the same node. It is understandable because during an execution,
a thread may hold different sets of locks when it acquires the
same lock.

We also present three elementary definitions below to
relate a lock dependency relation to a lock order graph. We
note that the following definitions of indegree and outdegree
are the same as the definitions of indegree and outdegree2 of
a digraph in graph theory.

- indegree($m$) is the sum of $|L|$ for all $n_i \in L \ | \langle t$, $m'$, $L' \rangle \in D \land m = m'$.
  Intuitively, indegree($m$) represents the indegree of the node $m$ on the lock order graph.

- outdegree($n$) is the cardinality of the set $\{ \langle t$, $m'$, $L' \rangle \ | \langle t$, $m'$, $L' \rangle \in D \land n \in L \land m = m' \}$. Intuitively, it represents
  the outdegree of the node $n$ on the corresponding lock order graph.

- edgesFromTo($m$, $n$) is the cardinality of the set $\{ \langle t$, $m'$, $L' \rangle \ | \langle t$, $m'$, $L' \rangle \in D \land n \in L \land m = m' \}$. Intuitively, it
  represents the number of edges from $n$ to $m$ on a lock

1 In set theory, the cardinality of a set $A$ is defined as the number of elements of the set, and is denoted by $|A|$.
2 In graph theory, the indegree and outdegree of a node $n$ are the number of incoming edges to $n$ and that of outgoing edges from $n$, respectively.
C. Lock Dependency Chain

Given a sequence of $k$ (where $k > 1$) lock dependencies $D = \langle \langle t_1, m_1, L_1 \rangle, \ldots, \langle t_i, m_i, L_i \rangle \rangle$, if $m_i \in L_j \forall i, j \leq k (i \neq j)$, we refer to $D$ as a lock dependency chain. In particular, if $m_i \in L_i$, $D$ is a cyclic lock dependency chain. A cyclic dependency chain represents a potential deadlock cycle.

For example, the lock dependency chain for the dotted edges in Figure 2 (a) is $\langle \langle t_1, L_2, \{l_1\}, \langle t_2, l_1, \{l_2\} \rangle \rangle$. This chain also forms a deadlock as illustrated in the running example.

IV. ALGORITHM

Our technique MagicFuzzer consists of three phases.

A. Phase I: Generation of Execution Trace

This phase is a pre-processing step to construct a log based on the critical events occurred in an execution of a multithreaded program $p$. Given a program $p$, we firstly collect the set of critical events from an execution of the program. The detail is as follows:

Suppose that a log $w$ is an empty sequence initially. Whenever an event create($t$) occurs, we allocate a new thread identifier and an empty lockset $L_t$ for the thread $t$. Also, whenever an event acquire($t$, $m$) occurs, we firstly append the triple ($t$, $m$, $L_t$) to $w$, and then add $m$ to $L_t$ (i.e., $L_t := L_t \cup \{m\}$). Whereas, whenever a release($t$, $m$) event occurs, we only remove the lock $m$ from $L_t$ (i.e., $L_t := L_t \setminus \{m\}$) without affecting $w$.

To identify a reentrant lock, which can be acquired by the same thread multiple times before the thread releases the lock, we set up a counter for each lock $m$, and increment (and decrement, respectively) it by 1 on an acquire event (and a release event, respectively). After an increment/decrement, only when this counter becomes 1/0, the above triple for lock acquisition/release is appended to $w$. The generated log $w$ is used by Phase II.

B. Phase II: Magiclock

We firstly recall that in general, on a lock order graph $G$, a node may have no incoming or outgoing edge. However, for a node participating into a potential deadlock cycle, the node must have both incoming and outgoing edges.

Based on the above observation, suppose that we have a lock order graph $G$. A node that has no incoming edge or outgoing edge cannot be on any cycle in $G$. Hence, it is safe to remove all such nodes and their outgoing and incoming edges from $G$ without the worry of removing any cycle in $G$.

Our first insight is that after such a removal, the generated graph (say $G_1$) may contain nodes that each has no incoming edge or outgoing edge. Such a node (say $n$) however must have at least one edge on $G$ because $n$ must have at least one edge connected it with a removed node; otherwise, $n$ must have been removed from $G$ already.

Magiclock iteratively applies such a removal strategy until no more node can be removed. This iterative process must be terminating because it only removes nodes and edges from a graph without adding any new node or edge. The resultant graph should contain only nodes, each of which has both incoming and outgoing edges.

The second insight is that in applying the above strategy, we only need to know the indegree and outdegree of each node to determine whether a node should be removed. The net result is that Magiclock needs not to construct or maintain any lock order graph explicitly at all, but only iteratively subtracts the indegree and outdegree of each node from those outdegree and indegree of the removed nodes, respectively, and marks whether a node has been removed during the inference.

A cycle having $v$ nodes is a sequence, but there are in total $v$ permutations of the nodes to represent the same cycle. Detecting one permutation suffices to represent the cycle.

The third insight (Thread-Specificity) is, as follows, in a cycle, each thread (as an edge to connect two nodes in the cycle) can only occur once. Our definition of cyclic lock dependency chain in Section III.C reflects this insight. More importantly, because (1) each thread can only occur once in a cycle, (2) multiple threads form an order in every permutation of a cycle, and (3) detecting one permutation of the same cycle is sufficient to represent the cycle, we observe that we can use a thread-driven approach to search for cycles.

Magiclock firstly partitions the set of lock dependencies by threads, sorts the partitions in the ascending order of their thread identifiers to align its search sequence among the partitions with the permutation of every potential cycle that a thread with a smaller identifier always appears first in the permutation. Because each thread can only occur once in a cycle, Magiclock further employs a depth-first-search to avoid exploring any subtree if any node in the path from the root node to the current node in the search tree has the thread identifier of the root node of the subtree.

In the rest of this section, we present Phase II in detail.

1) Lock Classification

This is an iterative step. In each iteration, Magiclock aims at categorizing all the lock dependencies $D$ into four sets iteratively:

- **independent-set:** contains all the locks, each (say $m$) of which satisfies the following condition: $\text{indegree}(m) = 0 \land \text{outdegree}(m) = 0$.
- **intermediate-set:** contains all the locks, each (say $m$) of which satisfies the following condition: $\text{indegree}(m) = 0 \lor \text{outdegree}(m) = 0 \land \neg \text{indegree}(m) = 0 \land \text{outdegree}(m) = 0$.
- **inner-set:** contains all the locks, each (say $m$) of which satisfies the following condition: either (1) for all $(t, m, L) \in D$ and for all $n \in L$, $n$ must be an element of intermediate-set $\lor$ inner-set; or (2) for all $(t, n, L) \in D$ and for all $m \in L$, $n$ must be an element of intermediate-set $\lor$ inner-set.

At the final iteration, if there are still locks that do not belong to any one of the above three sets, the algorithm classifies them into the fourth set: cyclic-set.

Algorithms 1 and 2 show our lock classification algorithms. In the algorithms, indegree and outdegree are arrays that each maps a lock (as an index) to a number, denoting the values of indegree and outdegree of the lock;
Algorithm 1: InitClassification(D)

1 for each \( m \in D.\text{Lock} \) do
2 \( \text{indegree}(m) := 0 \)
3 \( \text{outdegree}(m) := 0 \)
4 end for
5 for each pair of locks \( (m, n) \) in \( D.\text{Lock} \) such that \( \exists \langle t, n, L \rangle \in D \) and \( m \in L \) do
6 \( \text{edgesFromTo}(m, n) := 0 \)
7 end for
8 for each lock dependency \( \langle t, m, L \rangle \in D \) do
9 \( \text{indegree}(m) := \text{indegree}(m) + 1 \)
10 \( \text{outdegree}(m) := \text{outdegree}(m) + 1 \)
11 \( \text{edgesFromTo}(n, m) := \text{edgesFromTo}(n, m) + 1 \)
12 end for
14 end for

Algorithm 2: LockClassification(D)

1 Stack \( S := \emptyset \); independent-set := \emptyset; intermediate-set := \emptyset; cyclic-set := \emptyset;
2 for each lock \( m \in D.\text{Lock} \) do
3 if \( \text{indegree}(m) \neq 0 \) and \( \text{outdegree}(m) = 0 \) then
4 add \( m \) to independent-set // keep in independent-set
5 else
6 if \( \text{indegree}(m) = 0 \) or \( \text{outdegree}(m) = 0 \) then
7 add \( m \) to intermediate-set // keep in intermediate-set
8 push \( m \) into \( S \)
9 end if
10 end if
11 end for
12 while \( S \) is non-empty do
13 pop \( m \) from \( S \)
14 if \( \text{indegree}(m) = 0 \) then
15 for each \( n \in D.\text{Lock} \) and \( n \neq m \) do
16 \( \text{indegree}(n) := \text{indegree}(n) - \text{edgesFromTo}(m, n) \)
17 \( \text{outdegree}(n) := \text{outdegree}(n) - \text{edgesFromTo}(m, n) \)
18 \( \text{edgesFromTo}(n, m) := 0 \)
19 end if
20 push \( n \) into \( S \)
21 add \( n \) into intermediate-set // keep in intermediate-set
22 end if
24 end if
25 if \( \text{outdegree}(m) = 0 \) then
26 for each \( n \in D.\text{Lock} \) and \( n \neq m \) do
27 \( \text{outdegree}(n) := \text{outdegree}(n) - \text{edgesFromTo}(n, m) \)
28 \( \text{indegree}(n) := \text{indegree}(n) - \text{edgesFromTo}(n, m) \)
29 \( \text{edgesFromTo}(n, m) := 0 \)
30 end if
31 push \( n \) into \( S \)
32 add \( n \) into intermediate-set // keep in intermediate-set
33 end if
34 end for
35 end while
36 for each lock \( m \in D.\text{Lock} \) do
37 if \( m \in \text{independent-set} \cup \text{intermediate-set} \cup \text{inner-set} \) then
38 add \( m \) to cyclic-set // keep in cyclic-set
39 end if
40 end if
41 end for

\( \text{edgesFromTo} \) is a two-dimensional array (a sparse matrix), where an entry \( \text{edgesFromTo}(n, m) \) represents the number of edges to go from \( n \) to \( m \). An entry \( \text{isTraversed}(i) \) keeps whether the thread \( i \) has been completed its traversal or not.

\( \text{InitClassification} \) (Algorithm 1) initializes the indegree, outdegree, and \( \text{edgesFromTo} \) associated with each lock as those values for the corresponding node on a corresponding lock order graph.

\( \text{LockClassification} \) (Algorithm 2) firstly identifies all the locks that belong to independent-set by checking, for each lock \( m \), whether the indegree\( (m) \) and outdegree\( (m) \) of the lock \( m \) are both zero (lines 3–4). Then, it further identifies all the locks that belong to intermediate-set by checking, for each lock \( m \), both whether both \( m \) does not belong to independent-set and whether one of its indegree\( (m) \) and outdegree\( (m) \) is zero (lines 6–7). Such an identified lock must have either no incoming edge or no outgoing edge. Hence, all such locks and their edges can be removed from the subsequent consideration of deadlock detection. Then, for each lock that belongs to intermediate-set, \( \text{LockClassification} \) also pushes the lock into a stack \( S \) (line 8).

The algorithm then enumerates the stack \( S \) and identifies the locks in the independent-set, intermediate-set, and cyclic-set.

### Example B: Take the lock order graph in Figure 2 (a) for illustration purpose. Table 1 shows the indegree and outdegree of each lock (node) for the graph in Figure 2 (a). For instance, for lock \( l_1 \), the table shows that the lock has three outgoing edges and 1 incoming edge, which match the situation presented in Figure 2 (a). Other entries can be interpreted similarly.

<table>
<thead>
<tr>
<th>Lock instance</th>
<th>( l_1 )</th>
<th>( l_2 )</th>
<th>( l_3 )</th>
<th>( l_4 )</th>
<th>( l_5 )</th>
<th>( l_6 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>indegree</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>outdegree</td>
<td>3</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

After the initialization of indegree, outdegree, and \( \text{edgesFromTo} \) for every node, \( \text{LockClassification} \) aims to classify nodes to independent-set, but, as shown in Table 1, no lock has 0 in both (indegree and outdegree) rows. Hence, the set independent-set is empty. Then, it classifies \( l_1 \), \( l_3 \), and \( l_2 \) into intermediate-set because each of them has a value 0 in its outdegree row, and the algorithm pushes these three locks into the stack \( S \) (initially empty). Readers may refer to Figure 2 (a) that if the three locks (i.e., nodes) have been removed, the five edges connected to them can be removed. Correspondingly, on processing the three lock in \( S \), \( \text{LockClassification} \) decrements the values in the outdegree row for \( l_1 \), \( l_2 \), \( l_4 \), and \( l_6 \) by 1, 2, 1, and 1, respectively. The
values in the outdegree row for \( l_4 \) and \( l_6 \) become zeros. Hence, LockClassification further classifies \( l_4 \) and \( l_6 \) into inner-set, and pushes them into \( S \). Note that the values in the outdegree row for the locks \( l_1-l_3 \) are now 2, 3, 0, 0, 0, 0, and 0, respectively. The algorithm then handles these two locks in \( S \), and finds that 1 outgoing edge connected to \( l_1 \) and 2 outgoing edges connected to \( l_2 \) are associated with the classified \( l_4 \) and \( l_6 \). The algorithm then deducts the values in the outdegree row for \( l_4 \) and \( l_6 \) by 2 and 1, respectively. The row for the seven locks becomes 1, 1, 0, 0, 0, 0, and 0, respectively. The iteration stops because the stack \( S \) is now empty. Both the indegree and outdegree rows for either \( l_1 \) or \( l_2 \) are non-zeros. The algorithm classifies \( l_1 \) and \( l_2 \) into cyclic-set (which, incidentally, precisely reduces the set of locks to show the cycle for Example A).

To ease readers to follow, Figure 2 (d) shows the result of cyclic-set with the edges associated with their lock dependencies such that both nodes of an edge are elements in cyclic-set.

2) Cycle Detection Algorithm

In this step, Magiclock constructs one thread-specific lock dependency relation \( D_i \) for each thread \( t_i \) by CycleDetection (Algorithm 3) as a partition mentioned in the “Thread Specificity” insight. Lines 2–10 in CycleDetection show the partitioning process. Note that, as explained above, Magiclock only needs to examine the lock dependencies for the locks that each of them is in cyclic-set (lines 7–9).

Then, CycleDetection iteratively (lines 17 and 33) search the sequences of thread-specific lock dependency relations via a depth first search strategy in such a way that when visiting the partition \( D_i \), it only further explores \( D_j \) for \( 1 \leq j \leq k \), where \( k \) is the number of threads in \( D \) (i.e., \( |D.Tid| \)) (lines 12 and 22), skipping those visited (line 23). It also prunes a branch when a cycle is detected (line 29).

3) Discussion

Compared with iGoodlock in DeadlockFuzzer, Magiclock has several innovations:

First, Magiclock uses a thread-specific lock dependency relation (denoted by thread-specific ldr) for each thread instead of mixing all them in the same ldr as iGoodlock does. Every thread in every lock dependency in a lock dependency chain can only occur once. Hence, if a technique puts all available lock dependencies in the same ldr, the technique cannot tell whether two lock dependencies in this ldr share the same thread identifier, unless the technique compares the thread identifiers of the two lock dependencies. However, to use a thread-specific ldr, Magiclock can actively select a particular set of lock dependencies (i.e., a partition mentioned above) with the required thread identifier without doing any comparison later.

Second, Magiclock employs a new depth-first-search algorithm to traverse \( D_i \) for each thread \( t_i \). This is different from the iGoodlock algorithm in DeadlockFuzzer. iGoodlock uses the transitive closure to iteratively find cycles. A noticeable limitation in iGoodlock is that iGoodlock has to keep all intermediate results, which consumes a lot of memory [10]. For Magiclock, a key parameter of its overhead is the traversal depth, which is at most the same as the total number of threads in an execution. On the other hands, iGoodlock may require a shorter period of time on reporting a cyclic lock dependency chain with, say, length = 2 because Magiclock has to traverse all possible depths for a given ldr before traversing another one at the same depth. Our technique has compensated this disadvantage by using the innovative thread-specific strategy as discussed above.

Third, iGoodlock suffers from an overhead of suppressing the report of \( v \) occurrences of the same cyclic lock dependency chain where \( v \) is the length of the chain. For example, given a cyclic lock dependency chain \( \langle t_1, m_1, L_1 \rangle, \langle t_2, m_2, L_2 \rangle, \langle t_3, m_3, L_3 \rangle \rangle \) with \( v = 3 \), there are two other cyclic lock dependency chains: \( \langle t_2, m_2, L_2 \rangle, \langle t_3, m_3, L_3 \rangle, \langle t_1, m_1, L_1 \rangle \rangle \) and \( \langle t_3, m_3, L_3 \rangle, \langle t_1, m_1, L_1 \rangle, \langle t_2, m_2, L_2 \rangle \rangle \). They all represent the same cycle. iGoodlock addresses this problem by suppressing the report of all but one occurrence of each cyclic dependency chain. However, it can only do so after the repeated occurrences of the same cyclic dependency chain have been detected. The Algorithm 3 of Magiclock

---

3 Of course, with a slight adaption, Magiclock can be also configured to detect cyclic lock dependency chains with depth = 2 only.

---

Algorithm 3: CycleDetection(cyclic-set, \( D \))

```
1 k := \(|D.Tid|\)
2 for each \( i \) from 1 to \( k \) do
3   isTraversed(i):=False
4   \( D_i := \emptyset \)
5 end for
6 for each lock dependency \( \tau = (t, m, L) \in D \) do
7   if \( m \in \text{cyclic set} \) then
8     \( \tau \) into \( D_i \)
9   end if
10 end for
11 Stack \( S := \emptyset \)
12 for each \( i \) from 1 to \( k \)
13   \( \text{visiting}:=\emptyset \) //repeated cycles elimination
14 for each \( \tau = (t, m, L) \in D \) do
15   isTraversed(i):=True //mark thread identifier \( t \)
16   \( \tau \) into \( S \)
17   \( t \) := DFS_Traverse(\( \text{visiting}, S \))
18   pop \( \tau \) from \( S \)
19 end for
20 end for
21 Function DFS_Traverse(\( \text{visiting}, S \))
22 for each \( j \) from \( \text{visiting}+1 \) to \( k \) do
23   if isTraversed(j):=False then //otherwise, skip all visited \( D_i \)
24     for each \( \tau \in D_i \) do
25       \( \text{visiting}:=\emptyset \)
26       \( \tau \) into \( \text{visiting} \)
27       if \( \tau \) forms a dependency chain then
28         \( \text{report} \) as a potential deadlock cycle
29       else
30         isTraversed(j):=True
31       end if
32       \( \tau \) into \( S \)
33       \( \text{DFS} \) Traverse(\( \text{visiting}, S \))
34       pop \( \tau \) from \( S \)
35       isTraversed(j):=False
36     end if
37   end for
38 end if
39 end if
40 end for
41 end Function
```
uses the Thread Specificity insight and an elegant depth-first-search strategy to prevent any traversal that the search visits a thread partition with a larger thread identifier before visiting a thread partition with a smaller thread identifier.

C. Phase III: deadlock confirmation

1) Object Abstraction

To confirm whether a cyclic lock dependency chain is a real deadlock [9], we need to map the locks on every potential deadlock cycle provided by Phase II to the locks of an execution in this phase. DeadlockFuzzer uses a lightweight indexing algorithm [10], which computes an abstraction for each thread or lock for Java programs. For each object o in a Java program, lightweight indexing is computed according the thread-local CallStack and the thread-local Counter. A Counter is an integer mapped from three keys: a thread identifier t, the depth of CallStack d (precisely, half of the depth), and a label c (e.g., code line number) where the object o is created.

MagicFuzzer adapts this object abstraction approach as lightweight indexing in DeadlockFuzzer to compute an abstraction for each thread and lock so that it can work on C/C++ programs. There are two differences, however. First, unlike a Java program, in a C/C++ program, not all locks are dynamically initialized. For instance, developers may statically initialize a block of memory as the initialization of a particular lock via a call to PTHREAD_MUTEX_INITIALIZER in the Pthread library. In the implementation, MagicFuzzer uses pintool [14] to monitor execution events, which cannot provide events to our tool about this kind of memory allocations in the Probe mode (see Section V.A). Therefore, MagicFuzzer works around to compute an abstraction for every statically initialized lock by checking whether a lock has been created in its first lock acquisition, and if its creation has not been recorded, MagicFuzzer approximates the acquisition site of the lock in the code as the creation site of the lock. Second, the data structure CallStack in DeadlockFuzzer is maintained by DeadlockFuzzer itself on function call and return as well as on creation of a new object. MagicFuzzer directly uses the call stack of the C/C++ program runtime to retrieve any required call stack directly to precisely represent the actual situation and optimize its performance.

2) MagicFuzzer Scheduler

We firstly recall that by the non-deterministic nature of a multithreaded program, executing a program over an input may probabilistically exhibit a real deadlock in an execution if the deadlock can be formed. Our insight here is that such an execution may also probabilistically produce an object abstraction of a potential deadlock cycle. To actively guide a run to produce a deadlock, it relies on the probability of producing such an object abstraction in the run and the probability of selecting a potential deadlock cycle that contains the same object abstraction. It is possible that the same object abstraction may exist in an execution multiple times, but a corresponding deadlock cycle may not be the focus of the current monitoring run, or the right occurrence of the abstraction has been accidently missed in active thread scheduling, hence, missing an opportunity to confirm a deadlock in the execution. DeadlockFuzzer suffers from this problem because before an execution produces any object abstraction that may match with any potential deadlock cycle, a particular potential deadlock cycle (which may not match with the object abstraction in question) has been chosen as the only “suspect” to be confirmed for the run.

MagicFuzzer uses an active random scheduler to check against a set of cycles (denoted as CycleSet) reported by Magiclock with each execution. To ease our presentation, we firstly define the following notations: CycleSet is a set of cycles reported by Magiclock. ToBePaused is a set of

\begin{algorithm}
\begin{tabular}{l}
1 ToBePaused := \{t \mid t = (t, m, L) \text{ and } D \in \text{CycleSet}, \text{ such that } t \in D\} \\
2 Pause := \emptyset \\
3 \text{Lockset}(t) := \emptyset \text{ for each thread } t \\
4 \text{Enable} := \{t \mid t \in p\} \\
5 \text{while Enable} \neq \emptyset \text{ do} \\
6 \qquad r := \text{a random thread in Enable} \\
7 \qquad \text{stmt} := \text{next statement to be executed by } t \\
8 \qquad \text{if } t \notin \text{ToBePaused} \text{ then} \\
9 \qquad \qquad \text{execute(stmt)} \\
10 \qquad \text{else} \\
11 \qquad \qquad \text{stmt} := \text{acquire}(t, m) \text{ then} \\
12 \qquad \qquad \quad \text{call CheckDeadlock} \\
13 \qquad \qquad \quad \text{if CheckAndPause}(t, m) \text{ returns a Cycle} \text{ then} \\
14 \qquad \qquad \qquad \text{pause}(t) \\
15 \qquad \qquad \quad \text{else} \\
16 \qquad \qquad \qquad \text{Lockset}(t) := \text{Lockset}(t) \cup \{m\} \\
17 \qquad \qquad \quad \text{execute(stmt)} \\
18 \qquad \text{end if} \\
19 \qquad \text{else} \\
20 \qquad \quad \text{stmt} := \text{release}(t, m) \text{ then} \\
21 \qquad \quad \text{Lockset}(t) := \text{Lockset}(t) \setminus \{m\} \\
22 \qquad \quad \text{execute(stmt)} \\
23 \qquad \text{end if} \\
24 \text{end if} \\
25 \text{end if} \\
26 \text{end if} \\
27 \text{end while} \\
28 \text{if } \text{Enable} \neq \emptyset \text{ then} \\
29 \quad \text{print } ‘\text{System Stalls!}' \\
30 \text{end if} \\
31 \text{end Function CheckAndPause(m)} \\
32 \text{Function CheckDeadlock} \\
33 \quad \text{if } \{l_1, \ldots, l_n\} \subseteq \text{ToBePaused and Cycle } \in \text{CycleSet such that } \langle l_1, \text{ToAcquire}(l_1), \text{Lockset}(l_1)\rangle, \ldots, \langle l_n, \text{ToAcquire}(l_n), \text{Lockset}(l_n)\rangle = \text{Cycle} \text{ then} \\
34 \quad \quad \text{print } ‘\text{a real deadlock detected on } '+ \text{ToString(Cycle)} \\
35 \quad \text{end if} \\
36 \text{end Function CheckDeadlock} \\
\end{tabular}
\end{algorithm}

\footnote{Note that this CallStack is maintained by DeadlockFuzzer itself and slightly different from that we usually said call stack at its content. It contains one more item Counter on each function call event.}
threads with each thread existing in some cycles in CycleSet. ToAcquire$t$ represents a lock that $t$ wants to acquire in its next statement. Paused is a set of pairs of a thread $t$ and a Cycle, which denotes that, when executing $p$, a thread will be paused and added into Paused if $(t, ToAcquire(t), Lockset(t))$ belongs to a Cycle. Enable is a set of threads that each has not terminated yet. We use stmt to denote a high-level instruction, such as an acquire or release operation. We denote a call to execute a statement stmt by execute(stmt). The functions pause(t) and resume(t) represent the actions to pause and resume $t$, respectively.

Algorithm 4 shows our MagicScheduler active random scheduler.

Given a program $p$ and a CycleSet, MagicScheduler firstly identifies ToBePaused set by extracting all identical threads abstractions from each Cycle in CycleSet (at line 1). It then initializes Paused to be empty (at line 2), Lockset($t$) to be empty for each thread $t$ (at line 3), and Enable to contain all threads in $p$ (at line 4).

When executing $p$, if $t$ is not in ToBePaused, MagicScheduler allows $t$ to execute statements. Otherwise, if the next statement of $t$ is a lock acquisition statement (acquire($t$, $m$)), just before executing this statement, MagicScheduler checks whether any real deadlock may occur if $t$ acquires $m$ by call CheckDeadlock. The function CheckDeadlock (lines 36–40) checks whether a real deadlock occurs, and reports a deadlock if there exists a cyclic lock dependency chain as defined in Section IV.B. No matter a deadlock occurs or not, MagicScheduler then calls CheckAndPause to determine whether or not the current thread should be paused. If CheckAndPause returns a Cycle, MagicScheduler pauses $t$ and adds the pair $(t, Cycle)$ into Paused; otherwise, MagicScheduler calls execute(stmt) to execute the statement, and updates the lockset of $t$. If the statement stmt is a lock release statement, MagicScheduler updates the lockset of $t$ and calls execute(stmt). All other statements will be directly executed without the interception by MagicFuzzer.

CheckAndPause differentiates MagicScheduler from DeadlockFuzzer. DeadlockFuzzer only checks a predetermined cycle for each invocation. However, CheckAndPause checks all the cycles in CycleSet and returns a Cycle if $(t, ToAcquire(t), Lockset(t))$ belongs to at least one cycle; otherwise, it returns $\emptyset$. In such way, MagicScheduler is able to check and confirm multiple cycles to be real deadlocks in the same run. If not all cycles are confirmed by MagicScheduler in a run, then MagicScheduler can proceed to confirm the remaining cycles at the next run iteratively until all cycles have been confirmed, or it reaches a certain number $N$ (e.g., 100, which is inputted by a user) of runs.

3) Thrashing

Thrashing [10] may occur due to improper pausing a set of threads. Both DeadlockFuzzer and MagicScheduler suffer from thrashing. When a thrashing occurs, MagicScheduler selects a thread randomly, and resumes it (lines 28–32).

### Table 2. Descriptive Statistics of Benchmarks (a-r events refer to acquisition and release events; NA means no bug ID available)

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Bug ID</th>
<th>SLOC</th>
<th># of threads</th>
<th># of locks</th>
<th>Trace size</th>
<th>File size</th>
<th># of a-r events</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite</td>
<td>#1672</td>
<td>74.0k</td>
<td>10</td>
<td>4</td>
<td>732Bytes</td>
<td>460</td>
<td></td>
</tr>
<tr>
<td>MySQL</td>
<td>#37080</td>
<td>1,093.6k</td>
<td>27</td>
<td>127</td>
<td>23.8KB</td>
<td>4,986</td>
<td></td>
</tr>
<tr>
<td>Chromium</td>
<td>NA</td>
<td>3,577.5k</td>
<td>21</td>
<td>1,363</td>
<td>4.1MB</td>
<td>1,325,202</td>
<td></td>
</tr>
<tr>
<td>Firefox</td>
<td>NA</td>
<td>3,315.4k</td>
<td>22</td>
<td>912</td>
<td>5.7MB</td>
<td>4,165,230</td>
<td></td>
</tr>
<tr>
<td>OpenOffice</td>
<td>NA</td>
<td>5,445.8k</td>
<td>7</td>
<td>1,349</td>
<td>4.1MB</td>
<td>1,357,696</td>
<td></td>
</tr>
<tr>
<td>Thunderbird</td>
<td>NA</td>
<td>2,751.2k</td>
<td>10</td>
<td>915</td>
<td>2.6MB</td>
<td>1,601,456</td>
<td></td>
</tr>
</tbody>
</table>

### V. Experiment

A. Implementation and Benchmark

**Implementation.** We have implemented MagicFuzzer using Pin 2.9 [14], a dynamic instrumentation analysis tool, running in its Probe-based mode. The Probe-based mode supports high-level instrumentation so that the instrumented program runs almost natively [14]. MagicFuzzer has been implemented for C/C++ programs using Pthreads libraries on a Linux system. For each thread or lock, MagicFuzzer maintains a shadow memory location to store its data, such as a lockset for a thread, and an integer heldCounter for a lock (where heldCounter is used to handle the acquisitions of a reentrant lock).

MagicFuzzer instruments a program to produce an execution trace as described in Section IV.A. It also generates a location for each lock acquisition event for MulticoreSDK as this technique needs it. To compare with our tool, we also faithfully implemented DeadlockFuzzer [10] and MulticoreSDK [15] on pin based on their papers and downloadable artifacts because their original tool can handle Java programs only. However, to compute an abstraction for each thread and lock, we directly search CallStack (through stack pointer $sp$ via pin) rather than maintaining a CallStack as in [10].

**Benchmarks**. We selected a set of widely-used C/C++ open source programs, including SQLite [22], MySQL [16], Firefox [6], Chromium [4], Thunderbird [24], and Open Office [19]. Because SQLite is an embedded database, we wrote a simple test harness program with two threads to concurrently call it. Originally, we intended to use benchmarks that have been published, but there is virtually no such benchmark with large-scale C/C++ programs with test cases that can repeat the occurrences of deadlocks. For SQLite and MySQL, we use the test cases adapted from their bug reports [22][16]. For Firefox, Chromium, and Open Office, we simple start them, and then close them when their user interface appears. For Thunderbird, we configure it to get two emails from a Gmail account.

Our experiment was performed on the Ubuntu Linux 10.04 configured with a 3.16GHz Duo 2 processor and 3.25GB physical memory. We use the time command (a Linux utility) to collect the time consumption and read the

---

5 The suite of benchmarks and MagicFuzzer can be downloaded at http://www.cs.cityu.edu.hk/~51948163/magicfuzzer/.
memory usage from /proc/<benchmark processing ID>/statm to compute the maximum amount of memory used for each run on a benchmark. Following the experiment in [10], we reported the average perform on 100 runs on each tool.

Table 2 shows the descriptive statistics of the benchmarks we selected. The first three columns show the name, Bug ID (where NA means no bug ID available), and code size (SLOC [21]) of each benchmark. The fourth and fifth columns show the number of threads and the number of locks, respectively. The last two columns show the execution trace size in forms of the trace file size and the number of lock acquisitions and releases (denoted as a-r events).

### B. Result Analysis

Table 3 summarizes the overall comparisons among iGoodlock, MulticoreSDK (denoted as MSDK), and Magiclock in aspects of the memory consumption (under the column Memory) in Megabytes (MB) (or GB for Gigabytes if the memory consumption is large than 1024MB), the time consumption (under column Time) in second (s) or “h” for hour if the time is larger than 60 minutes), and the number of cycles (under the column # of cycles). The last column shows the number of real deadlocks among the detected cycles. Due to the out of memory error of iGoodlock, we cannot collect its data in full. We mark these cells with “ND” indicating where no data is collected and with “>>” indicating that the value in the cell is just the value before it crashed. We also use these two marks in Table 4 and Table 5 for the same purpose.

From Table 3, we observe on SQLite, the three algorithms performed similarly in memory and time consumption. They also reported the same number of cycles.

### C. Comparisons

#### 1) Comparison between iGoodlock and Magiclock

Because both iGoodlock and Magiclock used the lock dependencies relation implementations to find cycles, we compared the number of lock dependencies produced by the two algorithms as shown in the second and the third columns in Table 4. Besides, iGoodlock uses an iterative algorithm to find all cycles and has to store all intermediate results (see Section IV.B.2 and [10]), Table 4 also shows the intermediate results for each benchmark produced by iGoodlock in the last three columns (denoted by DFx). However, except on SQLite, iGoodlock consumed the most memory and the most time among the three algorithms and run out of memory on MySQL, Chromium, and Firefox. MulticoreSDK consumed up to hundreds of Megabyte memory. Magiclock consumed the least memory; and on all benchmarks, it consumed less than ten Megabytes memory. On time consumption, MulticoreSDK consumed two to six times than that consumed by Magiclock except on Chromium and Open Office. On Chromium and Open Office, both iGoodlock and MulticoreSDK did not finish (iGoodlock run out of memory and, for MulticoreSDK, we have killed its process after the reported time in Table 3 has elapsed).

On the reported numbers of cycles, MulticoreSDK and Magiclock reported the same number of cycles; iGoodlock also reported the same number as that reported by MulticoreSDK and Magiclock except on those benchmarks that ran out of memory (and crashed).

We find that Magiclock is better than iGoodlock and MulticoreSDK in terms of memory and time consumption. In the following subsection V.C, we compare Magiclock with iGoodlock and MulticoreSDK in more details.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Memory(MB)</th>
<th>Time(s)</th>
<th># of cycles</th>
<th># of real deadlocks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>iGoodlock</td>
<td>MS DK</td>
<td>Magiclock</td>
<td></td>
</tr>
<tr>
<td>SQLite</td>
<td>1.05MB</td>
<td>1.05MB</td>
<td>1.05MB</td>
<td>0.002s</td>
</tr>
<tr>
<td>MySQL</td>
<td>&gt;2.8GB</td>
<td>&gt;2.8GB</td>
<td>&gt;2.8GB</td>
<td>&gt;2m5s</td>
</tr>
<tr>
<td>Chromium</td>
<td>&gt;2.8GB</td>
<td>&gt;2.8GB</td>
<td>&gt;2.8GB</td>
<td>&gt;1h47m</td>
</tr>
<tr>
<td>Firefox</td>
<td>&gt;2.8GB</td>
<td>&gt;2.8GB</td>
<td>&gt;2.8GB</td>
<td>&gt;10m40s</td>
</tr>
<tr>
<td>OpenOffice</td>
<td>245.20MB</td>
<td>&gt;2.8GB</td>
<td>&gt;2.8GB</td>
<td>&gt;1h64m</td>
</tr>
<tr>
<td>Thunderbird</td>
<td>298.83MB</td>
<td>298.83MB</td>
<td>298.83MB</td>
<td>16m13s</td>
</tr>
</tbody>
</table>

However, except on SQLite, iGoodlock consumed the most memory and the most time among the three algorithms and run out of memory on MySQL, Chromium, and Firefox. MulticoreSDK consumed up to hundreds of Megabyte memory. Magiclock consumed the least memory; and on all benchmarks, it consumed less than ten Megabytes memory. On time consumption, MulticoreSDK consumed two to six times than that consumed by Magiclock except on Chromium and Open Office. On Chromium and Open Office, both iGoodlock and MulticoreSDK did not finish (iGoodlock run out of memory and, for MulticoreSDK, we have killed its process after the reported time in Table 3 has elapsed).

On the reported numbers of cycles, MulticoreSDK and Magiclock reported the same number of cycles; iGoodlock also reported the same number as that reported by MulticoreSDK and Magiclock except on those benchmarks that ran out of memory (and crashed).

We find that Magiclock is better than iGoodlock and MulticoreSDK in terms of memory and time consumption. In the following subsection V.C, we compare Magiclock with iGoodlock and MulticoreSDK in more details.

#### C. Comparisons

#### 1) Comparison between iGoodlock and Magiclock

Because both iGoodlock and Magiclock used the lock dependencies relation implementations to find cycles, we compared the number of lock dependencies produced by the two algorithms as shown in the second and the third columns in Table 4. Besides, iGoodlock uses an iterative algorithm to find all cycles and has to store all intermediate results (see Section IV.B.2 and [10]), Table 4 also shows the intermediate results for each benchmark produced by iGoodlock in the last three columns (denoted by DFx). However, except on SQLite, iGoodlock consumed the most memory and the most time among the three algorithms and run out of memory on MySQL, Chromium, and Firefox. MulticoreSDK consumed up to hundreds of Megabyte memory. Magiclock consumed the least memory; and on all benchmarks, it consumed less than ten Megabytes memory. On time consumption, MulticoreSDK consumed two to six times than that consumed by Magiclock except on Chromium and Open Office. On Chromium and Open Office, both iGoodlock and MulticoreSDK did not finish (iGoodlock run out of memory and, for MulticoreSDK, we have killed its process after the reported time in Table 3 has elapsed).

On the reported numbers of cycles, MulticoreSDK and Magiclock reported the same number of cycles; iGoodlock also reported the same number as that reported by MulticoreSDK and Magiclock except on those benchmarks that ran out of memory (and crashed).

We find that Magiclock is better than iGoodlock and MulticoreSDK in terms of memory and time consumption. In the following subsection V.C, we compare Magiclock with iGoodlock and MulticoreSDK in more details.
VI. RELATED WORK

Techniques on deadlock detection can be classified into static ones and dynamic ones. We have compared our MagicFuzzer with two dynamic techniques DeadlockFuzzer and MulticoreSDK, and indirectly compared with Goodlock.

Many static techniques ([1][17][20][26]) analyze the source code and infer lock order graphs to find potential deadlock cycles. They have an advantage to apply for software that is not close such as the Java library. These techniques however suffer from high false positives. For example, an early work [26] reports that 1,000 deadlocks and only 7 are real deadlocks. More recently, Naik et al. [17] combines a suite of static analysis techniques to reduce the false positive rates. However, problems like conditional variables and scalability are still the concerns on using static techniques. MagicFuzzer never reports a false positive due to its confirmation of each potential deadlock cycle.

Joshi et al. [9] monitors annotated conditional variables to produce a trace program containing only thread and lock operations as well as the values of conditionals. Then they apply a model checker (Java Pathfinder) to check all abstracted execution paths of the trace program for deadlocks. This technique suffers from needing manual effort to add annotations and scalability to handle large-scale programs. Bensalem et al. [2][3] use the happens-before relation to improve the precision of cycle detection and use a guided scheduler to confirm deadlocks. Ur and colleagues [5][18] propose ConTest that uses a Goodlock algorithm to identify cycles, and actively introduces noise to increase the probability of deadlock occurrence [5].

Like [10], MagicFuzzer uses object abstractions to relate locks and threads to overcome the cross-execution reference problem, and guides executions to work toward cycles.

Deadlock Immunity [11] prevents the second occurrence of a deadlock by maintaining a database containing all patterns of occurred deadlock and using online monitoring. It does not have an active schedule or potential deadlock cycle detection component. Gadara [25] statically detects deadlocks and inserts deadlock avoidance code right before the positions of the lock acquisitions in detected deadlocks. When executing the inserted code, Gadara is called to analyze the state of lock acquisition and insert a gate lock acquisition dynamically to prevent the occurrence of the corresponding deadlock. Gadara however may report both false positives and false negatives when detecting deadlocks.

VII. CONCLUSION

Existing dynamic potential deadlock techniques are not scalable enough to detect potential deadlock cycles. We have presented MagicFuzzer, a novel technique to detect potential deadlocks and confirm them as real ones. The experiment confirms that it is highly efficient and effective to tackle the challenges in handling the executions of large-scale, widely-used, and open-source multithreaded C/C++ programs.
REFERENCES


