2013

An Empirical Comparison of the Fault-Detection Capabilities of Internal Oracles

Tingting Yu
University of Nebraska-Lincoln, tyu@cse.unl.edu

Witawas Srisaan
University of Nebraska-Lincoln, witty@cse.unl.edu

Gregg Rothermel
University of Nebraska-Lincoln, gerother@ncsu.edu

Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on
Year: 2013
Pages: 11 - 20, DOI: 10.1109/ISSRE.2013.6698900

An Empirical Comparison of the Fault-Detection Capabilities of Internal Oracles

Tingting Yu, Witawas Srisa-an, Gregg Rothermel
Department of Computer Science and Engineering
University of Nebraska - Lincoln
{tyu, witty, grother}@cse.unl.edu

Modern computer systems are prone to various classes of runtime faults due to their reliance on features such as concurrency and peripheral devices such as sensors. Testing remains a common method for uncovering faults in these systems, but many runtime faults are difficult to detect using typical testing oracles that monitor only program output. In this work we empirically investigate the use of *internal* test oracles: oracles that detect faults by monitoring aspects of internal program and system states. We compare these internal oracles to each other and to output-based oracles for relative effectiveness and examine tradeoffs between oracles involving incorrect reports about faults (false positives and false negatives). Our results reveal several implications that test engineers and researchers should consider when testing for runtime faults.

I. Introduction

Modern computer systems ranging from personal computers to consumer electronic devices are becoming increasingly complex. These systems are utilizing high-performance multi-core processors to ensure adequate responsiveness and performance. They also utilize a full array of peripheral devices and sensors to support required features. Competition for market share means that new features are frequently added to these systems, making their product life-cycles last only one to two years. Short life-cycles imply frequent updates to the various runtime systems these systems utilize.

The foregoing characteristics can result in runtime faults that are difficult to identify and correct. While verification techniques such as model checking have been effective for detecting such faults in certain contexts, it is still challenging to use these techniques in practice. For example, model checking can suffer from state-space explosion when used on non-trivial programs. As such, applying it to a system that includes hardware components, one or more operating systems, device drivers, shared libraries, and runtime systems (e.g., virtual machines and dynamically linked libraries) is difficult. We believe that testing is a more practical alternative for assessing and finding faults in these systems.

To effectively test modern systems, software engineers must be able to observe complex interactions between applications, low-level hardware, OS events, and runtime systems. As an illustration, consider a data race between application code and a device driver [1]. This class of fault may *intermittently* result in observable incorrect outputs when a particular interleaving between an application and an interrupt is executed, rendering traditional testing approaches using oracles that examine only execution output unreliable. To detect such a fault, engineers must be able to observe when the interrupt handler and device driver access a shared variable and determine whether the access adheres to a specific correctness property. An access that violates the property is evidence of the presence of a fault, even if that access occurs "silently" without propagating to output. Detecting faults in this manner involves the use of *internal test oracles*: oracles that can examine aspects of a program's internal state either as the program runs or via post-processing following its execution.

The notion of using internal oracles is founded in research on fault-based testing [2], error propagation [3], and testability analysis [4]. Using terminology from [4], suppose a fault f exists in location l in program P. For f to be detected in testing, three conditions must hold: (1) l must be executed by a test, (2) f must erroneously alter ("infect") program state, and (3) the infected state must propagate to observable output. Testing approaches that rely on *output-based oracles* (oracles that inspect externally visible program outputs) require all three of these conditions to occur. However, the absence of incorrect output for a suite of test cases does not indicate that the program does not contain faults; those test cases might have failed to execute faults, or on executing them, failed to cause infection and propagation.

In contrast to output-based oracles, internal oracles monitor aspects of internal program state seeking evidence that infections have occurred, and on finding such evidence, they signal the presence of those infections. Arguably, monitoring internal states rather than outputs should improve fault detection; however, internal oracles can also fail to detect the presence of faults (producing *false negatives*), and signal the presence of anomalies that are *not* faults (producing *false positives*).

Previous research has argued that internal oracles can be effective [5], [6], [7]. There has been little work, however, empirically investigating the relative effectiveness of different internal oracles. For a fault involving incorrect use of synchronization operations, is it more effective to observe proper usage of synchronization operations or to observe memory accesses to detect data races? Does one of these approaches result in more false positives or negatives than the other? Studying the tradeoffs between oracles can provide useful insights.

In prior work [8], we introduced *SimTester*, a testing framework that utilizes virtual machines to provide observability. The framework can observe occurrences of low level events including memory, register and hardware accesses, interrupts, system calls, and execution of binary programs. It can also force hardware events such as interrupts to occur at specific times during testing. The observations gathered by the framework are used by internal oracles to detect the presence of faults that might not otherwise be easily detected.

The goal of this work is to *empirically investigate the tradeoffs between internal and output-based oracles, as well as between different internal oracles designed to detect the same classes of faults but utilizing different execution information*. To achieve this goal, we focus on a family of internal oracles that target common faults in modern systems involving lock management, resource management, interrupt management, critical section protection, and buffer management. Such faults can lead to well-known, but intermittent and hard-to-reproduce failures involving data races, deadlock, livelock, critical section violations, and buffer overflow [9], [10]. Our oracles monitor subtle but relevant events (e.g., memory accesses, memory allocations) using existing tools that have been widely used to ease system development and improve testing and debugging processes. By empirically comparing various oracles, our hope is that we can ultimately help engineers make more informed decisions as to whether a given oracle would be more effective for detecting specific classes of faults.

In our empirical study we use the oracles under consideration in the testing of five concurrent programs and a device driver. Two of these are real-world programs that perform data compression and file transfer (BZIP2 and AGET). We also include a program that performs file scanning in parallel (PFSCAN), and commonly used concurrency benchmarks from Inspect (Producer/Consumer) and the Oracle Thread Analyzer (Dining Philosophers) [11], [12] that are representative of common concurrency issues. The device driver is a real-world UART driver from a Linux kernel. All the programs that we use are multithreaded.

We empirically compare the tradeoffs between oracles in terms of effectiveness, false positive rates, and false negative rates. In the cases we study, our internal oracles are more effective at detecting faults than output-based oracles, and the incidence of false-positive reports associated with them is relatively low, especially when compared to the high incidence of false-negative reports associated with output-based oracles. Further analysis of our data reveals several implications for practitioners and researchers wishing to employ and study the use of internal oracles, including specific methods for improving on the oracles we employ.

II. Internal Oracles

In this section, we consider five classes of runtime faults that are common in modern systems and have been the subject of extensive research (as noted in Section V). These include faults involving lock management, resource management, interrupt management, critical section protection, and buffer management; faults that can lead to well-known and common failures involving data races, deadlock, livelock, critical section violations, and buffer overflow. For each such class of faults, we describe internal oracles that can detect them, and explain how we implement them.

A. Faults Involving Lock Management

Lock management faults are common in concurrent programs. This type of fault occurs due to improper use of lock operations that synchronize concurrent access to shared memory. One instance of such a fault occurs when shared data is protected by a lock, but other threads that access this data do not obtain the lock. A second instance involves other threads obtaining some lock other than the one used to protect this shared data. A third instance occurs when shared data is not protected by any locks for threads that access this data. This type of fault can lead to data races.

Data races in applications. Program A provides an example of this type of fault occurring in an application. The lock acquisition operation is missing (statement 1), causing variable `x` to be unprotected. The fault further infects code regions (statements 1-4 and 6-8) by leaving all shared data in the code regions unprotected. The data state of the unprotected data can be infected due to unsynchronized accesses (i.e., `x` is incremented to 1 twice by both threads). Finally, the fault may propagate to output.

Existing dynamic race detectors [13], [14] can detect this type of fault once the shared data in two threads is accessed without proper synchronization. If `input` is 2 and statements 1 and 4 are executed, a data race cannot occur because the fault cannot infect shared variable `x` in T2; thus, the race detector fails to detect the fault. On the other hand, if `input` is -1, the fault can infect `x` (unprotected) in both threads and the race detector detects it. With a specific thread interleaving, i.e., if T1 and T2 concurrently update `x`, the fault further infects the data state of `x` and such an infection may eventually propagate to output.

Oracle implementation. Our internal oracle is based on an existing hybrid race detection algorithm [14] that utilizes both vector-clock and lock-set algorithms. Our oracle evaluates whether the following property holds:

1) Two threads access the same memory location.
2) The two accesses do not hold a common lock.
3) One access does not happen-before the other.
4) At least one of the accesses is a write.

```
T1{                          T2{
1. //lock(); fault           5. lock();
2. x++;                      6. if(input < 0){
3. ...                       7.       x++;
4. //unlock();               8.}
}                            9. unlock();
                             }
```

PROGRAM A. **A fault involving a missing lock**

Our oracle is applied on the race detector provided by ThreadSanitizer [15]. This tool is implemented on top of PIN [16], a widely used binary instrumentation tool that records runtime information (e.g., memory read, write, and synchronization operations such as pthread_mutex_lock and pthread_mutex_unlock) as applications execute. We refer to our implementation of this oracle as O1.

Data races in low-level software. While race detectors can detect the type of data race shown in Program A, they do not work in scenarios in which races occur across different software components such as device drivers and interrupt service routines (ISRs). Device drivers can be a major source of failures in OS kernels [17]. For example, a data race fault involving interactions between the UART driver and an ISR in Linux kernel has been reported [1]. Recent work [8] has shown that races occur when the following property holds:

1) A device driver and an ISR access the same memory location.
2) The associated interrupt is not disabled for the device driver access.
3) At least one of the accesses is a write.

Oracle implementation. We created an internal oracle to detect whether the foregoing property holds. Our oracle was implemented as modules on a widely used virtual platform called Simics [8]. Our implementation monitors shared memory accesses and access types, the interrupt bit (to determine whether interrupts are enabled or disabled), and context information (to determine whether the current access is in the application or interrupt context). Because interrupt related data races are also monitored at the point of shared data accesses, we also refer to our implementation of this oracle as O1.

B. Faults Involving Resource Management

There are four conditions necessary for deadlock: mutual exclusion, no preemption, hold-and-wait, and circular wait [18]. In typical operating systems including various implementations of Unix and Linux, deadlock avoidance and prevention are not provided as OS features. As such, deadlock can occur in these systems. When a deadlock occurs, we consider the state of the system to be infected because the correct locking discipline has not been exercised. As such, it is possible to detect possible occurrences of deadlock by observing all lock acquire and release operations even on a successful program execution. Therefore, it is helpful to compute potential sources of deadlock and the algorithm used in such a computation can become a internal oracle.

Existing techniques that can identify deadlocks construct graph representations either on-line or off-line using execution traces. The graph is then analyzed to detect whether a certain property holds. Program B provides an example of deadlock and its detection. If `input` is greater than 1, a potential deadlock can be detected by analyzing a successful execution trace. With a specific interleaving such as execution sequence 1, 6, 2, 8 the fault propagates, causing the system to hang. Unfortunately, the analysis is performed dynamically so it can miss sources of "potential deadlock" that are not in the executed paths. For example, if `input` is 0, statement 6 is executed; however, because the other erroneous statement (statement 8) is not executed, the detector cannot detect any potential deadlock.

```
T1 {
1. lock1();
2. lock2();
3. ...
4. unlock2();
5. unlock1();
}
```

```
T2 {
6. lock2(); //fault
7. if(input > 1){
8. lock1();//fault
9. ...
10.unlock1();
11. }
12.unlock2();
}
```

PROGRAM B. **A fault involving resource management**

Oracle implementations. We created an internal oracle to identify potential deadlock by detecting whether a valid cycle can exist. Our implementation uses the improved GoodLock algorithm [19], [20]. In this approach, a directed run-time lock graph is constructed based on traces of application executions. The graph is composed of run-time lock trees for all threads, representing the nested pattern in which locks are acquired and released by each thread. Each node in the graph represents the thread acquiring the lock. A potential deadlock exists if a valid cycle is detected in the run-time lock graph. We refer to our implementation of this oracle as O2.

To detect an actual deadlock, our oracle dynamically constructs a wait-for graph [18] during an execution of an application. The graph is updated each time a lock is acquired. Our oracle reports a deadlock if a circular-wait condition is detected. We refer to our implementation of this oracle as O3.

C. Faults Involving Interrupt Management

Because many low level software components such as interrupt handlers and device drivers are often not preemptible, spin-locks are often used as synchronization primitives. In these software components, improper management of spin-locks can cause applications or operating systems to loop infinitely, making the system unresponsive, yet continuing to consume CPU cycles [21]. In such a case, livelock occurs.

Reports of a fault in a Linux interrupt handler show that one cause of livelock involves improper use of interrupt enabling and disabling functions [22]. In Program C, a watchdog timer application uses a spin lock that is also used by an interrupt handler (statements 3 and 9). To correctly share this lock, the interrupt should be disabled when the timer executes its critical region (statements 3-5); this ensures that the timer will not be preempted while holding the lock. However, the interrupt disable statement is missing prior to the execution of statement 3, and thus, the timer could be preempted while holding the lock. In this case, if `input` is greater than 1, the interrupt handler would attempt to access the lock (statement 9); in doing so, it would be stuck in the spin-lock loop.

An analysis of this fault shows that the system state is infected because the interrupt bit remains set when the application accesses the lock. This infection can be observed

```
void watchdog(...)          ISR{
1:  {                       8: if (input > 1)
2: //cli();  fault          9:    spin_lock(obj);
3:  spin_lock(obj);         10:     ...
4:    ...                   11: }
5:  spin_unlock(obj);
6. //sti();
7:  }
```

PROGRAM C. **A fault involving interrupt management**

after the ISR executes the lock acquisition statement. Then, a potential livelock can be detected by checking this property:

1) The lock is executed by the ISR.
2) The associated interrupt is not disabled in the application during any execution point within the critical section regarding the same lock.

Under certain interrupt interleavings, an infected state can lead to an actual livelock. An actual livelock occurs when:

1) The application is preempted by the ISR while the application is holding a lock.
2) The ISR tries to acquire the same lock and fails.

Oracle implementations. We created internal oracles that detect whether the two foregoing properties hold. The approach is similar to that used to implement oracles O2 and O3; that is, we created an oracle to detect potential sources for deadlocks (we refer to this oracle as O4) and another oracle to dynamically detect actual deadlocks (we refer to this oracle as O5). Both of our oracles are implemented in Simics in the same way in which we implemented oracle O2, by observing lock operations and program context.

D. Faults Involving Critical Sections

A critical section is a code region that should be accessed by only one thread at a time. If this constraint is violated, data races and deadlocks can occur. This type of fault is particularly common in device drivers and embedded software systems that use spin locks and interrupts [23], [24].

When a critical section is not properly protected, various types of concurrency faults such as data races, livelock, and deadlock can ensue. Program D illustrates how a data race can occur due to critical section violations. The critical section is supposed to be protected by disabling interrupts. However, function `foo()` incidentally enables interrupts while executing inside the critical section. The *original infection* occurs after the unsafe function (statement 2) in the critical section is executed. The fault further infects the shared variable `g` in the unprotected critical section (statements 3-8), and such an infection can be detected when it is accessed by both T1 and ISR. If an interrupt occurs between statements 4 and 6 in this unprotected critical section, the data state of `g` is infected. Finally, this fault propagates to output by generating an error event (statement 7).

To detect this type of fault, two different internal oracles can be applied. The first has already been introduced as oracle O1. The second oracle checks for proper protection of a critical section. It detects a fault at its *original infection* state, i.e., when the fault is executed, by checking the property:

```
T1{                         ISR{
1. local_irq_disable();     10. g++;
2. foo(); //fault           11. ...
3. if (input > 0)           }
4.      g = 0;
5.      ...                 foo(){
6.      if(g != 0){         12.  ...
7.        error;            13. local_irq_enable();
8.      }                   }
9. local_irq_enable();
}
```

PROGRAM D. **A fault involving critical section protection**

1) For each critical section entry there is not a matching critical section exit in the same thread.
2) A critical section is entered twice without being exited in between.
3) A critical section is exited twice without being entered in between.

Oracle implementation. We created a internal oracle to detect whether the foregoing property holds. The oracle is implemented on PIN for applications and on Simics for low-level software. Our implementation captures critical section protection operations used by programs; these are analyzed dynamically for violations. We refer to this oracle as O6.

Next, we provide an example of a critical section violation that can lead to livelock. If interrupts are incidentally enabled while the application is still holding a lock, it is possible for an interrupt to occur before the lock is released. At that time, if the ISR tries to acquire the same lock, livelock can occur.

Program E illustrates a scenario in which this can happen. The *original infection* occurs after the unsafe operation in the critical section is executed. The infection (the interrupt bit is set in `foo()`) occurs when statement 11 is executed. This infection can be detected by oracle O6. The fault further propagates to the code region between the unsafe operation and the exit of the critical section. Once the lock acquisition statement (statement 8) is executed by the ISR with input greater than zero, this infection can be detected by oracle O4, which can identify potential deadlocks. The actual deadlock will not occur until this interrupt preempts T1 at some point within the unprotected code region. If actual deadlocks occur, oracle O5 can be used to detect such occurrences.

E. Faults Involving Buffer Management

Weak string functions (e.g., strcpy, fprintf, strcat, memcpy, etc.) in the C library do not automatically perform bounds

```
T1{                         ISR{
1. spin_lock_irq();         7. if(input > 0)
2. ...                      8.      spin_lock();
3. foo(); //fault           9. ...
4. ...                      }
5. spin_unlock_irq();
6. ...                      foo(){
}                           10. ...
                            11. local_irq_enable();
                            }
```

PROGRAM E. **A second fault involving critical section protection**

```
int main(int argc, char *argv[]){
1. long distance;
2. char *buf1 = (char *)malloc(16);
3. char *buf2 = (char *)malloc(16);
4. distance = 5;
5. memset(buf2+distance,'A', 10);
6. sprintf(buf1, argv[1]); //fault
7. printf("buf1=\%s", buf1);
8. printf("buf2=\%s", buf2+distance);
9. free(buf1);
10. free(buf2);
11. return 0;
}
```

PROGRAM F. **A heap buffer overflow fault**

checks; as such they are vulnerable to buffer overflow. When a statement that causes a buffer to overflow is executed, the system state is infected, because data is written to memory outside of the allocated buffer, causing the content of this memory cell to be incorrect. The fault can further infect the data state of another variable if it writes the overflowed data into the memory that is allocated for that variable. Finally, this infection may propagate to output.

Program F illustrates an example of buffer overflow, that occurs when statement 6 is executed with an input string longer than 16 characters. Statement 6 writes beyond the boundary of `buf1`, causing system state to be infected. This type of buffer overflow can be detected by memory safety detection tools such as Valgrind at the infection point. However, this infection does not propagate to output (statement 8) unless the length of the input is greater than or equal to 21, such that the input data overwrites the content in the memory allocated for `buf1`. As such, we can detect buffer overflows when a thread attempts to write to a memory cell (e.g., a variable or a field of a data structure) and the write operation overflows the size of the targeted cell.

Oracle implementation. We use a memory safety tool, Valgrind, to detect heap buffer overflow. Valgrind intercepts malloc calls during application execution to put *memory guards* around allocated blocks. A buffer overflow occurs when a program reads or writes outside an allocated block. We refer to this oracle as O7.

III. Empirical Study

Our goal is to evaluate the foregoing internal oracles, comparing them to one another and to output-based oracles. We consider the following research questions:

RQ1: How do internal oracles compare, in terms of abilities to detect faults, to output-based oracles and each other?

RQ2: How do internal oracles compare, in terms of tendencies to produce false-positive or false-negative fault indications, to output-based oracles and each other?

A. Objects of Analysis

As objects of analysis we chose five concurrent programs and one device driver. Table I lists these programs and the numbers of lines of non-comment code they contain (other columns are described later). Three of the concurrent programs are from the Inspect benchmark suite [11]: BBUF is an implementation of a producer and consumer problem, AGET is a multithreaded FTP client, and PFSCAN is a parallel file scanner. The fourth program, BZIP2SMP, is a multithreaded compression program. The fifth program, DININGPHILOSOPHER, is an example from the Oracle Thread Analyzer [12]. The sixth program, UART, is a UART device driver from the Linux kernel. We selected these programs for two reasons: (1) they are each applicable to one or more of the classes of faults described in Section II; (2) they include real-world applications from a widely used benchmark suite, commercial tools, and a Linux device driver.

TABLE I. Object Program Characteristics

Program	*NLOC*	*Fault Class*				
		I	II	III	IV	V
BBUF	256	4(1)	0	0	4	0
DIN.PHIL.	104	3	2(1)	0	1	0
AGET	846	4(1)	0	0	4	6(3)
PFSCAN	752	6	0	0	6	0
BZIP2SMP	4232	11	0	0	6	0
UART	1654	4(1)	0	6	12	0

Our object programs were not equipped with test cases. For these programs, a test case must include three components: (1) test input data and other relevant parameters, (2) specified thread interleavings [25], and (3) execution locations at which to invoke interrupts [8], [26]. To create a large number of test cases representative of those that might be created in practice, for each program, we first generated descriptions of valid user input values and command options. We used these to generate 1000 unique test inputs. (e.g., numbers of threads ranging from 2 to 100, a file, or strings sent via a UART console). For each of these test inputs, we assigned a thread interleaving by randomly selecting a set of program locations at the granularity of instructions. To run the first five user applications with the specified interleavings, we added yield points at the selected locations; this has a high probability of achieving determinism [25]. To run the OS level program (UART) with the specified interleavings, we forced interrupts to occur at particular execution locations that contain data shared between applications and interrupt handlers by raising the associated interrupt pins. By generating large numbers of potential test inputs, this approach lets us compare different oracles across a large range of such inputs.

To address our research questions we also required faulty versions of our object programs. Since the programs could have contained faults initially, we first ran all of the test cases we had created on each original program using all of our internal and output-based oracles. These oracles did detect seven potential faults in the programs, two of which, we discovered, were false positives. We corrected these faults in the source code, but saved them for later use in the study.

Five faults are not enough to let us address our research questions, so we next created additional seeded faults of the classes described in Section II. To perform this task, we asked a staff programmer with experience developing embedded systems, but no knowledge of this study, to create faults from the fault classes of interest, relying on his own practical experience

with such faults. For each program, our staff programmer first identified the statements for which a given fault class is applicable, and then created potential seeded faults related to that fault class in those statements. (At this stage, we cannot assert that the faults being injected can actually be revealed by some input to the given program; thus, we refer to the faults as "potential faults".)[1]

Given our test cases and sets of potential faults, we eliminated any faults that were never executed by any test cases. These faults cannot be detected by any oracle in the context of this study, and thus provide no insights into our research questions. This process left us with the numbers of faults reported in Table I. The numbers in parentheses denote faults found in the original programs prior to fault injection.

B. Variables and Measures

Independent Variable. Our independent variable is the oracle approach used. We consider output-based and internal oracles. Internal oracles were implemented based on those described in Section II. To provide output-based oracles, we implemented output checkers appropriate to the various programs. On BBUF, we utilized automatable checks of expected output; namely, (1) "the number of items must be increased by one for the producer;" (2) "the number of items must be decreased by one for the consumer;" (3) "the number of items must be greater or equal than 1 but less than or equal to the size of buffer;" (4) "deterministic portions of output strings must match expected contents." On AGET, if the test case is a valid website, we checked whether the file has been successfully downloaded; otherwise we used a differencing tool on the outputs of the initial and faulty versions of the program to determine, for each test case t and fault f, whether t revealed f. The same differencing approach applied on PFSCAN, UART, and BZIP2SMP. On BZIP2SMP, if the test input is a valid command, we checked whether the compressed zip file is valid by decompressing it and comparing it with the original unzipped file. Finally, for DININGPHILOSOPHER we checked the following: (1) "program terminates properly (i.e., no deadlock)"; (2) "two adjacent philosophers were not in the eating state simultaneously"; and (3) "deterministic portions of output strings must match expected contents".

Dependent Variables. As dependent variables we measure *oracle effectiveness*, *false positives*, and *false negatives*. We gathered and we report these metrics for each oracle class independently in order to assess the classes independently.

To measure *oracle effectiveness* for a given oracle O, program P, and fault f of the fault class targeted by O, we count the number of test cases in the set of test cases T for P that caused O to report the presence of f. Based on these observations we calculate the effectiveness of O in terms of the percentage of the 1000 test cases in T that cause O to report faults. Reporting percentages across the large sets of test cases provides a finer-grained assessment of oracle performance than, say, simply reporting whether or not the entire sets revealed faults. Note that, while we calculate effectiveness measures for all faults, when reporting the overall effectiveness of specific oracles we do so relative only to "actual faults"; that is, faults that are not false-positives.

To measure *false positives* and *false negatives*, we first needed to determine which of our seeded potential faults were in fact actual faults. This required us to determine whether, for program P and potential fault f, there exists an input to P in the domain D for P that can reveal f. If one of the test inputs t in the set of test cases T for P revealed f according to the output oracle for P, we verified by inspection that the fault was in fact an actual fault. For any faults not thus revealed, we inspected P and f to determine whether some fault-revealing input not in T exists. If we could find such an input, we know that f is an actual fault. If we could not, f is a non-fault.

Given this fault classification, we say that oracle O for program P reports a false positive result for potential fault f if f is a non-fault, and if there exists one or more test cases t in T for which O reports the existence of f. Further, we say that O reports a false negative result for potential fault f if f is an actual fault, and if there exists no test case t in T for which O reports the existence of f.

C. Study Operation

We executed our test cases on all of the faulty versions of each object program, with one fault activated on each execution (to avoid fault-interactions and masking effects, and to allow us to accurately determine whether each fault was indeed detected) with all oracles that pertain to that fault class and that monitor behavior at run-time active in the program. We then applied oracle O4, which does not monitor behavior at run-time, to the test execution log for each execution.

D. Threats to Validity

The primary threat to external validity for this study involves the representativeness of our programs, faults, and test cases (including inputs and thread interleavings). Other systems may exhibit different behaviors and cost-benefit tradeoffs, as may other forms of test cases. Furthermore, our study considers only a specific set of internal oracles, and specific instantiations of output-based oracles.

The primary threat to internal validity for this study is possible faults in the implementation of our approach and in the tools that we use to perform evaluation. We controlled for this threat by extensively testing our tools and verifying their results against a smaller program for which we can determine correct results. We also chose to use some commonly used tools (e.g., Valgrind and Simics).

Where construct validity is concerned, our measurements of false positives and negatives necessarily involve the test cases being utilized, and thus, our results are relative to those test cases. Further, while fault detection, false positives, and false negatives are important metrics, they are not the only possible ones. Metrics such as the costs of applying techniques and the human costs of investigating oracle outputs are also of interest.

[1]Due to space limitations we cannot provide details on all of the fault types seeded; however, interested readers can obtain all of the programs, faults, test cases, and oracles by contacting the first author.

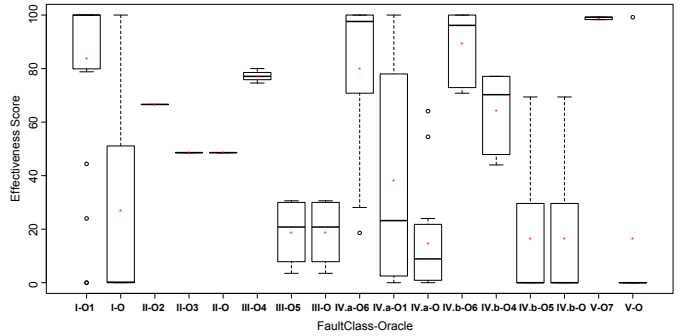

FIGURE 1. **Oracle effectiveness per oracle**

TABLE II. FAULT CLASS I (LOCK MANAGEMENT)

Prog.	*Fault*	*O1*			*O*		
		eff.	FP	FN	eff.	FP	FN
B.B	**1**(a)	97.8	T		0.0		
	2	80.9			0.0		T
	3	44.4			0.0		T
	4	100.0			0.0		T
D.P	1	100.0			0.0		T
	2	100.0			0.0		T
	3	100.0			0.0		T
AGET	**1**	78.8			76.7		
	2	79.8			0.0		T
	3	78.9			73.7		
	4	80.0			31.5		
PFSCAN	1	40.0	T		0.0		
	2	100.0			0.0		T
	3	98.6			0.0		T
	4	100.0	T		0.0		
	5	100.0			40.0		
	6	100.0			100.0		
B.S	1	100.0			19.8		
	2	100.0			20.6		
	3	100.0			100.0		
	4	100.0			100.0		
	5	100.0			62.2		
	6	100.0			72.8		
	7	100.0			0.0		T
	8	100.0			0.0		T
	9	100.0			0.0		T
	10	100.0			0.0		T
	11(c)	0.0		T	0.2		
UART	**1**	24.0			9.8		
	4(d)	87.1	T		0.0		
	5(d)	80.2	T		0.0		
	7(c)	0.0		T	22.9		

TABLE III. FAULT CLASS II (RESOURCE MANAGEMENT)

Prog.	*Fault*	*O2*			*O3*			*O*		
		eff.	FP	FN	eff.	FP	FN	eff.	FP	FN
D.P	**1**(b)	1.8	T		0			0		
	2	66.6			48.6			48.6		

TABLE IV. FAULT CLASS III (INTERRUPT MANAGEMENT)

Prog.	*Fault*	*O4*			*O5*			*O*		
		eff.	FP	FN	eff.	FP	FN	eff.	FP	FN
UART	1	77.1			30.6			30.6		
	2	77.1			3.5			3.5		
	3(d)	87.1	T		0.0			0.0		
	4	74.6			29.4			29.4		
	5	80.0			12.2			12.2		
	6(d)	87.1	T		0.0			0.0		

E. Results and Analysis

Figure 1 presents oracle oracle effectiveness data. Labels on the horizontal axis are of the form FaultClass-OracleName, where FaultClass is one of "I", "II", "III", "IV", and "V" and OracleName is "O" for the output-based oracle and "O_i" ($1 \leq i \geq 7$) indicating an internal oracle applicable to the given fault class. Fault class IV has two types of faults, IV.a and IV.b, because each instance of a critical section violation can lead to either a race (IV.a) or a deadlock (IV.b). Each box reflects the effectiveness scores measured across all programs and faults of the given fault class. (The effectiveness scores reported correspond only to actual faults, not faults signaled erroneously by an oracle — i.e., false-positives.) Asterisks reflect means and dark horizontal lines reflect medians.

Tables II – VII list all actual data points measured. To clarify how to read the tables we refer to Table II as an example. This table contains results pertinent to Fault Class I. Results are partitioned per program by horizontal lines, with the program name given (in some cases abbreviated) in Column 1. For each program, the faults of Fault Class I that were present in the program are noted in Column 2; numbers shown in bold represent the seven faults reported in the original programs, while others are seeded faults. Lowercase letters appearing in parentheses following some numbers provide a mechanism for referencing those later in order to explain specific results.

For each fault listed in the table, columns to the right provide data on each oracle applied. For each oracle there are three columns, reflecting the effectiveness score for that oracle on each fault (.eff), whether the oracle produced a false positive (FP), and whether the oracle produced a false negative (FN). Table II reports results for a case in which only one internal oracle was applied, but several other tables report results for more than one internal oracle.

1) RQ1: Relative Effectiveness of Oracles: Inspection of Figure 1 and the data provided in the tables suggests that in all but three cases (Fault Class II with oracle O3, Fault Class III with oracle O5 and Fault Class IV.b with oracle O5), internal oracles had higher effectiveness scores than output-based oracles. In fact, for each fault class, there exists at least one internal oracle that outperformed the corresponding output-based oracle. In two cases (Fault Classes IV.a and IV.b), there are multiple internal oracles that outperformed the corresponding output-based oracle, with the internal oracles placed at potential infection points close to the point at which the fault exists exhibiting greater effectiveness than those placed at infection points farther away.

To assess whether these observed differences were statistically significant, we applied Wilcoxon tests to the data sets, comparing each pair of oracles within each fault class, except for Fault Class 2 whose data set is too small. Table VIII provides p-values from each comparison. In all but three cases, the oracles were statistically significantly different for $\alpha = 0.05$. Two of these cases (marked "NA") were cases mentioned above in which effectiveness scores were equivalent.

2) RQ2: False Positives and False Negatives: As we anticipated, internal oracles produced more false-positives than output-based oracles. For the faults our study considered, our output-based oracles produced no false positives. In contrast, considering just the most effective internal oracle utilized for

TABLE V. FAULT CLASS IV.A (CRITICAL SECTIONS)

Prog.	*Fault*	*O6*			*O1*			*O*		
		eff.	FP	FN	eff.	FP	FN	eff.	FP	FN
B.B	1	28.1			8.5			1.0		
	2	29.3			10.2			0.9		
	3	98.1			2.5			21.8		
	4	100.0	T		0.0			0.0		
D.P	1	97.7	T		0.0			0.0		
AGET	1	98.3			25.5			24.0		
	2	96.4			76.6			54.5		
	3	97.6			78.0			64.1		
	4	97.8	T		0.0			0.0		
PFSCAN	1	100.0			100.0			6.6		
	2	58.9			0.0		T	0.0		T
	4	67.2	T		40.0	T		0.0		
	5	18.6			0.0		T	0.0		T
	6	19.2	T		0.0			0.0		
	7	19.2	T		0.0			0.0		
B.S	1	100.0			100.0			0.0		
	2	88.8			80.6			10.6		
	3	100.0			28.3			20.6		
	4	100.0			100.0			0.0		T
	6	98.9	T		0.0			0.0		
	7	98.3	T		0.0			0.0		
UART	1(c)	100.0			0.0		T	22.9		
	2(c)	100.0			0.0		T	10.8		
	3	70.8			19.1			3.6		
	4	77.1			23.2			8.9		
	5	92.3	T		0.0			0.0		
	6(d)	100.0	T		80.2	T		0.0		

TABLE VI. FAULT CLASS IV.B (CRITICAL SECTIONS)

Prog.	*Fault*	*O6*			*O4*		
		eff.	FP	FN	eff.	FP	FN
UART	1	100.0			77.1		
	2	100.0			77.1		
	3	70.8			47.9		
	4	72.9			44.0		
	5	92.3			63.4		
	6	100.0			77.1		
Prog.	*Fault*	*O5*			*O*		
		eff.	FP	FN	eff.	FP	FN
UART	1	69.4			69.4		
	2	29.6			29.6		
	3	0.0		T	0.0		T
	4	0.0		T	0.0		T
	5	0.0		T	0.0		T

each fault class, relative to reported faults, we see that:

- Fault Class I: 5 of 32 faults were false positives;
- Fault Class II: 1 of 2 faults were false positives;
- Fault Class III: 2 of 6 faults were false positives;
- Fault Class IV.a, 10 of 27 faults were false positives;
- Fault Class IV.b, no false positives among 6 faults;
- Fault Class V, no false positives among 6 faults;

As an additional observation, on Fault Class IV.a, internal oracle O1 (a less effective oracle than oracle O6 for that fault class) produced just two rather than ten false positives.

Turning our attention to false-negatives we see different results. Internal oracles failed to report actual faults in only six cases overall (four of these involving the weaker of two internal oracles on Fault Class IV.a, oracle O1). Output-based oracles, in contrast, failed to report 25 of the 60 actual faults present in the programs. All 25 of these faults were reported by at least one internal oracle.

IV. DISCUSSION AND IMPLICATIONS

We now discuss the results of our study further, focusing on implications for practitioners and researchers, and specific instructive instances observed in the study.

TABLE VII. FAULT CLASS V (BUFFER MANAGEMENT)

Prog.	*Fault*	*O7*			*O*		
		eff.	FP	FN	eff.	FP	FN
AGET	**1**	99.2			0.0		T
	2	99.2			0.0		T
	3	98.3			0.0		T
	4	99.2			99.2		
	6	99.2			0.0		T
	8	98.3			0.0		T

TABLE VIII. P-VALUES FROM COMPARISONS OF EFFECTIVENESS

Class I	*Class III*			*Class IV.a*		
O1-O	O2-O3	O2-O	O3-O	O6-O1	O6-O	O1-O
3.702e-05	0.0041	0.0041	NA	0.0011	0.0003	0.0500

Class IV.b						*Class V*
O6-O4	O4-O5	O6-O5	O6-O	O4-O	O5-O	O7-O
0.0340	0.0360	0.0360	0.0360	0.0360	NA	0.0533

A. Implications for Practitioners

Our results show that internal oracles can be effective. Moreover, on every fault class considered, one or more internal oracle was more effective than output-based oracles.

Where false-positives are concerned, our internal oracles reported 18, whereas output-based oracles reported none. This is a potential cause for concern, because engineers forced to consider fault reports that are false positives may waste time, and may lose confidence in the oracles. Further exploration of our data revealed, however, that six of the false-positives flagged by internal oracles involved potential faults which, though not capable of propagating to output in the given program, could propagate if the program were used in conjunction with different interrupt handlers. Arguably, these potential faults should be corrected.

Conversely, the relative ineffectiveness of output-based oracles resulted in 25 false-negatives – cases in which output-based oracles missed faults, all of which were reported by internal oracles. For example, in one case, a data race was detected by an internal oracle, and one thread won the race and overwrote the data written by another thread, but with the same value. This is not a benign race because a different input could cause the output to be affected. In a second case, a potential deadlock was detected by an internal oracle but not exposed given the thread interleavings utilized by our test cases; however, different interleavings could cause the program to fail. In cases such as these, internal oracles are better able to detect faults that might otherwise remain hidden.

Finally, further inspection of our results reveals that oracle performance can differ across classes of systems. For example, consider two of the internal oracles that did not increase fault detection effectiveness compared to output-based oracles: the thread based deadlock (Table III) and interrupt related live lock (Table IV) oracles. When using these oracles when deadlock occurred, none of the threads proceeded further. However, for infinite spin-lock loops occurring in embedded software with preemptive scheduling or running on multiprocessors, it is possible for some threads in an application to be "stuck" while other threads continue to execute, making the application appear to be responsive, so the actual deadlock and livelock oracles should still be useful in this case.

B. Implications for Researchers

Our results show that internal oracles located close to the points at which errors occur are, in general, more successful at detecting faults than oracles located at points later in program execution. The former oracles are also, in general, more likely than the latter to produce false positives. Our results also show, however, that the differences between oracles are more complex than one might imagine.

For example, algorithms for detecting potential deadlock that operate earlier in program execution do not automatically identify user-defined synchronization primitives and thus may miss faults, whereas algorithms for detecting actual deadlock that operate later in execution can detect them. One method for mitigating this effect may be to annotate primitives for potential deadlock detection so that engineers can more easily determine whether false positives have occurred.

Furthermore, oracles located closer to faults are not always more effective than those located further from faults in terms of the fault types they can detect. For example, data race detection can be applied to both "mismatched lock pairs" and "missing lock pairs", while critical section violation detection can be applied only to "mismatched lock pairs". While data race detection may discover a larger number of faults than critical section violation detection, critical section violation detection can still cause deadlocks that can be detected by data race detection oracles. It may be useful to run critical section violation detection first to correct faults that can be discovered earlier, and then run data race detection to find faults involving "missing lock pairs".

The primary implication of these observations is that researchers should continue to study and compare various internal oracles to identify those that are more effective, to determine conditions under which effectiveness may vary, and to discover mechanisms for improving effectiveness.

C. Key Points in Utilizing Internal Oracles

As we have seen, internal oracles can report more false positives and negatives than output-based oracles. However, such incorrect reporting information can still be useful in helping developers identify intentional data races, limitations in oracle implementations, or faults that may exist under different system configurations. Next, we describe some key points that developers can use to leverage the seemingly wrong reporting information to identify more faults.

1) Internal oracles are fault-type specific: An internal oracle can detect only those faults that it has been designed to detect. Consider the faults labeled "(c)" (four in Tables II and V); in these cases, output-based oracles detected failures while at least one internal oracle did not. Because our oracles were designed to detect only low level data races, they failed to detect these *atomicity violations* – a different class of concurrency faults with characteristics similar to low-level data races at program output. One solution is to strengthen our oracles to allow them to detect atomicity violations.

2) Internal oracles are implementation specific: An internal oracle can detect only those faults that it has been implemented to detect. Consider the fault labeled "(b)" in Table III. This fault, a potential deadlock in the DiningPhilosopher program, was reported by oracle O2 in the original program. On inspection, we discovered that O2 was implemented to recognize only synchronization primitives provided by the kernel, not user-defined synchronization primitives that the program uses. As such, this fault can also be considered a *false positive*. To further enhance the effectiveness of our oracle, we can extend it to recognize these user-defined primitives.

3) Internal oracles can be predictive: An internal oracle may report faults that cannot occur in the current configuration of the system but can occur in different system configurations. This is true of the faults labeled "(d)" (several occurrences across several tables). Consider Fault 6 in UART (Table IV) in particular. In this case, the interrupt handler under test never interleaves code within the different regions in which the potential fault exists. This means that within this program, with this interrupt handler, the potential fault cannot propagate to output. However, this does not imply that the potential fault should be ignored, because there could be other interrupt handlers which, in other environments, might interact with the fault differently. Arguably, pointing out the presence of such faults to developers, even when they represent false-positives, may be useful.

4) Detected violations may be intentional: An internal oracle may detect a violation that does not lead to failure. Consider the fault labeled "(a)" in Table II. This fault, a potential data race fault in BoundedBuffer, was reported by oracle O1 in the original program. The fault occurs on an unsynchronized counter used for profiling. On inspection, we discovered that the developers intentionally left the variable unprotected to achieve better performance [27]. As such, each instance of this intentional fault, while being classified by oracle O1 as a race, can be considered a *false positive* because the developers were aware of its existence and the fault is not harmful to the system.

V. Related Work

One common approach used to verify programs is to apply static analysis techniques to discover paths and regions in code that might be susceptible to certain types of property violations (e.g., [23], [28]) including memory safety and interrupt usage [29], [24]. Drawbacks of static analysis involve scalability problems and the difficulties of annotating all operations on manipulated hardware bits. Testing, which is the focus of this work, can be a less expensive, effective way to find faults.

There has been a great deal of research on basic techniques for testing concurrent programs for data races [14], actual deadlock [19], [30], and potential deadlock [20], [31]. Some techniques have been applied to large systems (e.g., Apache, Mozilla) [32], [33], [34]. These techniques use algorithms to analyze synchronization operations (e.g., lock acquire/release) among multiple threads in the execution traces of successful program executions. Researchers have also extended these

algorithms to test for concurrency faults that are related to interrupts [8], [26], [35]. To increase the possibility of exposing faults, techniques such as "active testing" [25] have also been proposed to permute thread interleaving orders (we use this approach to control thread interleavings in test cases.)

There has been work on test case generation using internal test oracles to increase the chance of exposing faults [36], [37]. Godefroid et al. [36] use dynamic symbolic execution to generate test cases that can trigger property violations by following the same paths executed by prior test cases that do not exhibit such violations. Our internal oracles are instances of such oracles. However, our work does not aim to develop new techniques for detecting violations of specific properties; instead, we use these properties in internal oracles and apply these oracles to different fault types with the goal of increasing the effectiveness of testing.

There has been some research on how internal test oracles affect testing effectiveness. Memon et al. [5] show that internal oracles can reveal faults more effectively than output oracles when applied to GUIs. Staats et al. [6] provide a theoretical foundation describing the importance of oracles in software testing. Our own previous work [7] has also shown that internal oracles can detect faults beyond those detectable by simple output-based oracles. None of this work considers factors such as false positives and negatives, or the effectiveness of different oracles for certain types of faults.

VI. Conclusion

We have presented an empirical study of a set of internal oracles that monitor internal program state rather than output to detect runtime faults common to modern software systems. Our results show that these oracles can be significantly more effective than output-based oracles at detecting faults of the classes considered.

In future work, we intend to extend our study of internal oracles to consider other runtime errors such as atomicity violations and memory leaks. We also intend to investigate how the depth of monitoring affects oracle performance. Finally, we intend to perform more extensive empirical studies, including studies in which we examine the ability of engineers to make use of our oracles.

Acknowledgments

This work has been supported in part by the Air Force Office of Scientific Research through award FA9550-10-1-0406.

References

[1] I. van Lil, "Serial: UART_BUG_TXEN race conditions," Web page, http://lkml.org/lkml/2006/6/28/81.
[2] L. Morrell, "Theoretical insights into fault-based testing," in *TAV*, Jul. 1988.
[3] T. Goradia, "Dynamic impact analysis: A cost-effective technique to enforce error-propagation," in *ISSTA*, Jun. 1993.
[4] J. Voas, "PIE: A dynamic failure-based technique," *TSE*, Aug. 1992.
[5] A. Memon, I. Banerjee, and A. Nagarajan, "What test oracle should I use for effective GUI testing?" in *ASE*, 2003.
[6] M. Staats, M. W. Whalen, and M. P. Heimdahl, "Programs, tests, and oracles: The foundations of testing revisited," in *ICSE*, May 2011.
[7] T. Yu, A. Sung, W. Srisa-an, and G. Rothermel, "Using property-based oracles when testing embedded system applications," in *ICST*, Mar. 2011.
[8] T. Yu, W. Srisa-an, and G. Rothermel, "SimTester: A controllable and observable testing framework for embedded systems," in *VEE*, Mar. 2012.
[9] E. Farchi, Y. Nir, and S. Ur, "Concurrent bug patterns and how to test them," in *PDPS*, Apr. 2003.
[10] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics," in *ASPLOS*, Mar. 2008.
[11] Y. Yang, X. Chen, and G. Gopalakrishnan, "Inspect: A runtime model checker for multithreaded C programs," U. of Utah, Tech. Rep., 2008.
[12] Oracle, "Thread analyzer," http://bzip2smp.sourceforge.net/, 2005.
[13] C. Flanagan and S. N. Freund, "FastTrack: efficient and precise dynamic race detection," in *PLDI*, 2009.
[14] R. O'Callahan and J.-D. Choi, "Hybrid dynamic data race detection," in *PPoPP*, Jun. 2003.
[15] K. Serebryany and T. Iskhodzhanov, "ThreadSanitizer: Data race detection in practice," in *BIA*, 2009.
[16] 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 *PLDI*, 2005.
[17] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy, "Recovering device drivers," in *OSDI*, 2004.
[18] A. Silberschatz, P. Galvin, and G. Gagne, *Applied Operating System Concepts*, 1st ed. New York, NY: Wiley, 2001.
[19] R. Agarwal and S. D. Stoller, "Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables," in *PDSTD*, 2006.
[20] K. Havelund, "Using runtime analysis to guide model checking of Java programs," in *SPIN MCSV*, 2000.
[21] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers, 3rd Edition*. O'Reilly, 2005.
[22] Kernel Trap, "deadlock in 3c59x driver," Web page, http://git.kernel.org/.
[23] D. Engler, B. Chelf, A. Chou, and S. Hallem, "Checking system rules using system-specific, programmer-written compiler extensions," in *OSDI*, 2000.
[24] L. Tan, Y. Zhou, and Y. Padioleau, "aComment: Mining annotations from comments and code to detect interrupt related concurrency bugs," in *ICSE*, 2011.
[25] P. Joshi, M. Naik, C.-S. Park, and K. Sen, "CalFuzzer: An extensible active testing framework for concurrent programs," in *CAV*, 2009.
[26] M. Higashi, T. Yamamoto, Y. Hayase, T. Ishio, and K. Inoue, "An effective method to control interrupt handler for data race detection," in *AST*, 2010.
[27] S. Narayanasamy, Z. Wang, W. Tigani, A. Edwards, and B. Calder, "Automatically classifying benign and harmful data races using replay analysis," in *PLDI*, Jun. 2007.
[28] J. W. Voung, R. Jhala, and S. Lerner, "Relay: static race detection on millions of lines of code," in *FSE*, 2007.
[29] W. Le and M. L. Soffa, "Marple: A demand-driven path-sensitive buffer overflow detector," in *FSE*, 2008.
[30] P. Joshi, C.-S. Park, K. Sen, and M. Naik, "A randomized dynamic program analysis technique for detecting real deadlocks," *SN*, vol. 44, Jun. 2009.
[31] B.-C. Kim, S.-W. Jun, D. J. Hwang, and Y.-K. Jun, "Visualizing potential deadlocks in multithreaded programs," in *PCT*, 2009.
[32] W. Zhang, C. Sun, and S. Lu, "Conmem: detecting severe concurrency bugs through an effect-oriented approach," in *ASPLOS*, 2010.
[33] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou, "Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs," in *SOSP*, 2007.
[34] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea, "Deadlock immunity: enabling systems to defend against deadlocks," in *OSDI*, 2008.
[35] J. Regehr, "Random testing of interrupt-driven software," in *ES*, 2005.
[36] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Active property checking," in *ES*, 2008.
[37] D. Molnar, X. C. Li, and D. A. Wagner, "Dynamic test generation to find integer bugs in x86 binary Linux programs," in *USS*, 2009.