Heap Fuzzing: Automatic Garbage Collection Testing with Expert-Guided Random Events

Guillermo Polito, Pablo Tesone, Jean Privat, Nahuel Palumbo, Stéphane Ducasse

Heap Fuzzing: Automatic Garbage Collection Testing with Expert-Guided Random Events

1st Guillermo Polito
2nd Pablo Tesone
4th Nahuel Palumbo
5th Stéphane Ducasse
RMoD
Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL
F-59000 Lille, France
{name}.{surname}@inria.fr

3rd Jean Privat
Université du Québec à Montréal
Montreal, Quebec, Canada
privat.jean@uqam.ca

***Abstract*—Producing robust memory manager implementations is a challenging task. Defects in garbage collection algorithms produce subtle effects that are revealed later in program execution as memory corruptions. This problem is exacerbated by the fact that garbage collection algorithms deal with low-level implementation details to be efficient. Finding, reproducing, and debugging such bugs is complex and time-consuming.**

In this article, we propose to fuzz heaps by generating large sequences of random heap events guided by virtual machine experts. Randomly generated events exercise the garbage collection algorithm with the objective of crashing the virtual machine and finding bugs. Once a bug is found, we use a test case reduction algorithm to find the smaller subset of events that reproduces the issue.

We implemented our approach on top of the virtual machine simulator of the Pharo Virtual Machine, to test its sequential stop-the-world generational scavenger. Experts guided our fuzzing toward the ephemeron finalization mechanism, corner allocation cases, and the heap compaction algorithm. Our prototype found 6 bugs: 3 in Pharo's ephemeron implementation which is not yet in production, 2 bugs in the default compactor which has been in production for 8 years, and 1 bug in the VM simulator used daily by VM developers. We show how such test cases were automatically reduced to trivial sequences that were easy to debug.

***Index Terms*—garbage collection, testing, fuzzing, virtual machines**

I. Introduction

Producing robust implementations of memory managers is still nowadays a challenging task. Defects in garbage collection algorithms produce subtle effects that are revealed later in program execution as memory corruptions. This problem is exacerbated by the fact that garbage collection algorithms deal with low-level aspects to be efficient. The Chromium Project finds that since 2015 70% of high-severity bugs are due to memory unsafety, and most of them are use-after-free errors [26]. Finding, reproducing, and debugging such bugs is complex and time-consuming (See Section II).

Existing work proposes to perform static verifications of garbage collection (GC) algorithms using theorem provers [13], [27] and model checking [5], [29], [34]. Such approaches are heavy-weight to implement and execute, even at the expense of requiring specific techniques to optimize them [1] (See Section VI).

Existing fuzzing techniques applied to virtual machines (VM) are aimed at testing (just in time) JIT compiler engines and propose generally the use of template programs that are mutated [6], [7], [17], [21], [23], [24], [36]. We argue however that such approaches, although shown suitable to find compilation errors, are not the most efficient at finding garbage collection bugs. We show that in a set of 111 programs, a large majority have stable allocation patterns and most of them present similar allocation patterns. Moreover, such patterns fail to cover memory manager corner cases (See Section II-D).

In this article, we propose *heap fuzzing*: a lightweight test generation approach that automatically generates test scenarios targeting memory managers and garbage collectors. We propose to short-cut VMs and JIT execution engines and fuzz directly VM memory managers. Directly fuzzing a memory manager allows us to control aspects such as the location where objects are allocated, and low-level events such as GC invocations and their parameters. Our solution generates large sequences of random heap events that exercise the garbage collection algorithms to generate VM crashes and find bugs. We combine fuzzing with a test reduction algorithm that finds the smaller subset of events reproducing an issue (See Section III).

We implemented our approach on top of the Pharo Virtual Machine [9], which presents a mature implementation in stable production usage for over a decade. We built three heap fuzzers *tailored* by VM experts which found six different real-world bugs, drastically outperforming a fully random heap fuzzer that found nothing in the same number of runs. We show how combined with test reduction techniques, the bugs we found can be reproduced by trivial sequences of events that were easy to debug (See Section IV).

Paper contributions:

- The first implementation, to the best of our knowledge, of a heap fuzzer that tests memory managers and garbage

This work was funded by Inria - AlaMVic Action Exploratoire

collection algorithms;
- Empirical evidence that a lightweight heap fuzzer implementation using simple random GC events in combination with VM expert heuristics is capable of finding real-world bugs, both in the alpha-state ephemeron implementation but also in the stable and under-production compactor GC;
- An analysis showing that hitting failures with such an approach happens with high probability with the correct heuristics: expert-guided fuzzers find bugs more than 20% of the runs with low timeouts, outperforming a fully random fuzzer that finds nothing;
- Empirical evidence showing that such bugs can be reduced to trivial sequences that are easy to debug and synthesize as normal tests.

II. Garbage Collection Bugs

High-level object-oriented programming languages use automatic memory managers with garbage collection (GC) algorithms that are in charge of reclaiming memory [19]. GC algorithms reclaim unused memory by observing the allocated memory chunks, and *proving* at runtime which ones are unreachable by the application graph. Depending on the underlying allocation mechanism, unreachable objects are *e.g.,* freed either by returning memory to the operating system or marking their memory as reusable.

Defects in garbage collection algorithms produce subtle effects that are revealed later in program execution as memory corruptions. The temporal distance between the cause and the failure makes these issues difficult to debug and understand [5]. These problems are exacerbated by memory model extensions such as ephemeron finalization [14] and weak references, and by the fact that garbage collection algorithms deal with low-level aspects to be efficient. Finding, reproducing, and debugging such bugs is complex and time-consuming.

A. Use After Free Bugs

Use-after-free bugs [11] are one of the most prominent bugs in today's applications. For example, they appear as one of the top 25 vulnerabilities in the Common Weaknesses Enumeration [10]. Although automatic memory managers reduce the risk of such errors by not letting developers explicitly free objects, such defects can still be present in a garbage collection implementation.

B. Expired Pointer Dereference in Moving Collectors

Expired pointer dereference bugs [12] happen when the object referenced by a pointer is moved or invalidated, typically when an object is moved but not all of its incoming references are updated. This kind of defect appears in moving collectors such as semi-space collectors, or compacting collectors. Moving collectors relocate objects in memory with the purpose of, for example, de-fragmenting/compacting the memory, improving locality, and putting together objects that are treated similarly (pinned objects, large objects, young objects...). After an object is relocated, all of its incoming references must be updated to the new location.

C. Weak References and Ephemeron Finalization

Weak references are references that do not prevent a garbage collector to collect the referred object. If an object is only referenced by weak references it will be a candidate for reclamation. Related to weak references are ephemerons [14], which are key-value pairs where the key is held strongly (in contrast to weakly) and the value is not traced unless the key is held strongly by some other object. Ephemerons are used as a finalization mechanism to notify the runtime when their key is a candidate for collection because they are the only object referencing it.

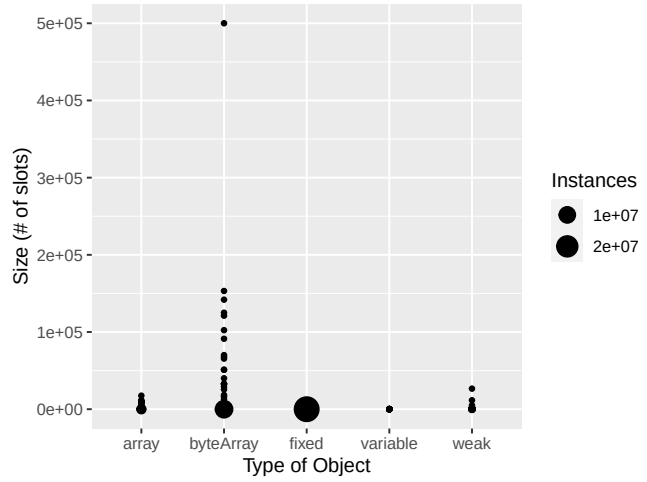
Without getting into the implementation of ephemeron finalization and weak references, which is outside of the scope of this paper, it is important to notice that ephemerons and weak references introduce many complexities in garbage collection algorithms. They need to be partially traversed or traversed separately from normal objects. Weak references may need to be replaced by tombstones marking the collection of an object. Object references in both ephemerons and weak references need to be properly updated in moving collectors. All these subtleties make testing such implementations even more challenging.

D. Stable Allocation Patterns

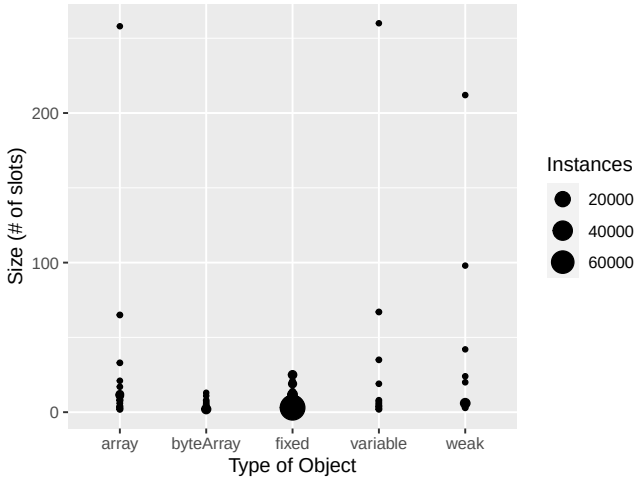
In mature applications and language implementations, chances are that memory corruption bugs will appear in seemingly random situations when rare conditions arise. We argue that this happens because common situations happen often and have been previously tested and debugged, and applications present stable allocation patterns that do not stress continuously memory manager corner cases. In other words, memory corruptions appear in extraneous cases, difficult to reproduce and debug.

To understand application allocation patterns, we instrumented all object allocations in 111 example programs. Our 111 programs included synthetic benchmarks, and the entire test suite executions of Pharo's standard library, its IDE, and high-level libraries such as its bytecode compiler and its Roassal3 visualization engine[1]. Such programs represent daily Pharo executions on continuous integration systems. Figure 1 illustrates such allocation patterns on three examples and an aggregated plot of all 111 program results. Each plot shows for each type of object (x-axis) of different object sizes in bytes (y-axis) how many allocations happened during the execution, illustrated by the size of the dots. The x-axis segregates allocations by their type: the `array` category includes all variable-sized objects with strong pointers, the `byteArray` category groups together all variable-sized opaque objects (byte arrays, word arrays), the `fixed` category groups together all objects with only instance variables/attributes, the `variable` category groups together all pointer objects that contain both instance variable/attributes and an array-like variable part with only strong pointers, the `weak` category groups together all objects containing weak pointers.

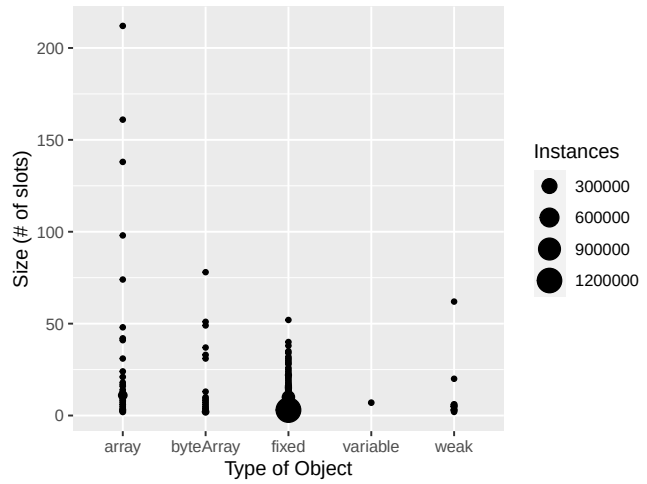
[1] https://github.com/ObjectProfile/Roassal3



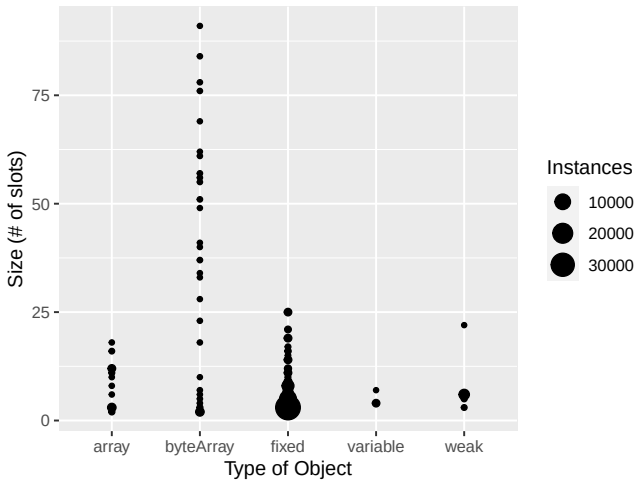
(a) Aggregated allocation profiles



(b) Bytecode Compiler



(c) Roassal3 Visualisation Engine



(d) String Standard Library

Fig. 1. **Four Exemplar Allocation Profiles in the Pharo Ecosystem.** Each plot shows how many objects are instantiated of each type and size. Plot 1(a) at the top left shows the accumulated profile of running all the 111 benchmark programs, including tests and synthetic benchmarks. Plot 1(b) at the top right shows the profile of the bytecode compiler tests. Plot 1(c) at the bottom left shows the profile of the Roassal3 visualization engine tests. Plot 1(d) at the bottom right shows the profile of Pharo's String base library tests.

Figure 1 shows on the top left the aggregated allocations of all executions, on the top right the allocations of the bytecode compiler, on the bottom right the allocations of the Roassal3 visualization engine, and on the bottom left the allocations of the String standard library. We can make several observations in Figure 1. First, most programs as well as the overall allocation profiles have allocations skewed to the lower part of the graph, showing they allocate mostly objects smaller than 50 bytes. Second, large objects are mostly rare. We can observe some instances of byte arrays with at most 100 slots, with most byte array instances coming from the String test case. Third, the allocations of large objects in both the Roassal3 programs and the String test case are sparse and concentrated in some sizes.

From these benchmarks, we can conclude that:

- Common executions do not thoroughly explore the allocation search space, potentially leaving much of the memory manager implementation uncovered;
- Even when allocation seems distributed across several values, chances are that corner cases are not covered. For example, in our benchmarks, we do not often observe any allocations of objects with 255 slots, a special case in Pharo's memory manager.

This simple analysis shows that there is a need for a testing solution that validates corner allocation cases and uncommon allocation patterns.

III. Heap Fuzzing

We propose *heap fuzzing*: a lightweight test generation approach illustrated in Figure 2. Heap fuzzing automatically generates scenarios directly targeting memory managers and garbage collectors, short-cutting JIT execution engines. Directly fuzzing a memory manager allows us to have fine-grained control on low-level details such as where objects are allocated, when GC invocations happen, and when memory grows. This is more precise than classical test programs that state only the nature or the size of allocated objects (instantiated classes) or the mutation (instance variable assignments) but is limited to the semantics of the programming language and not its implementation.

The key idea is to generate scenarios as sequences of (guided) random events affecting the heap, to generate VM crashes and find bugs. Events are guided by event builders (or fuzzers) written by Virtual Machine experts. Event fuzzers specify the kind of events generated and their frequency of appearance. Following such specifications, scenarios are generated randomly (with chosen probability) and executed on top of a VM with assertions enabled. The assertions present in the VM validate functions pre/post conditions and play the role of test oracle: we consider a scenario as failing if it triggers an error or assertion failure.

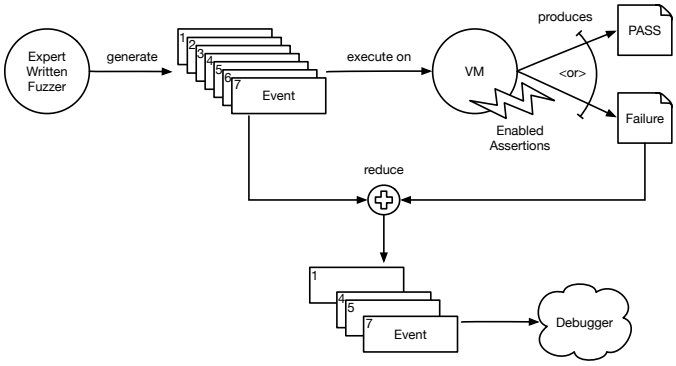


Fig. 2. **Solution overview**. A fuzzer randomly generates events that are executed on a VM with assertions enabled. The execution either passes or produces a failure. Once a failure is discovered, the scenario is reduced and used to debug.

The scenarios can either be generated ahead of time (thus fixed once generated) and then executed or generated while executed. In both cases, the observed issue is reproducible by re-executing a scenario.

Once a bug is found, we use a test case reduction algorithm to find a smaller subset of events that reproduces the issue. Reduction helps in understanding the issue and finding its cause. It eliminates events that do not participate in the issue and that only add noise. Moreover, it produces small object graphs that are easy to follow and reason about.

A. Fuzzing Model

Each new event is initialized with random values and fixed afterwards to guarantee reproducibility. The generation of each event has access to previously generated events. This allows mutation events to be configured with valid previously created allocations. Our fuzzing model currently supports the following kind of events:

a) Object Allocation: The allocation of an object is parameterized by the number of memory slots of the object, where the object should be allocated (*e.g.,* in the new or the old space as we have GC with two generations), whether the object must be a GC root, whether the object is an ephemeron or not. To make the memory model consistent, allocated objects need to have a class. When an allocation event is executed, a class for the object is created on a by-need basis. All objects that have the same shape/format share the same class. All slots are initialized with the nil object. Both objects and classes are allocated programmatically using the allocator API. Classes are created containing no methods, only defining object structure.

b) Object Mutation: The mutation of an object's slot will make an object reference another object (or itself). It is parameterized by a referrer and referee object, and the assigned slot index. When a mutation event is created, it randomly takes two allocation events from the previously generated events. One allocation is used as a referrer and the other is used as a referee. The slot index is chosen at random among the existing slots. Mutations are performed programmatically using the allocator API.

c) Garbage Collection: The execution of a garbage collection is parameterized with garbage collection arguments. For example, in our current implementation, a garbage collection event is parametrized with its level being either (a) an incremental GC in the new space (a.k.a., a scavenge), (b) a full GC implying a scavenge followed by a mark-and-compact of the old space with a single pass of compaction, or (c) a full GC followed with aggressive compaction. Garbage collections are triggered programmatically using the allocator API.

d) Pinning Event: Pinning makes objects unmovable. On the one hand, pinned objects stress the GC compaction. On the other hand, pinning forces objects to be moved to a non-movable area, and the addition of forwarding pointers, stresses object traversal. Objects are pinned programmatically using the allocator API.

e) Nop Event: Does nothing. It is used when a specific kind of event cannot be generated.

B. Assertions as Test Oracles

One of the biggest challenges in test generation is the generation of test oracles [3]. In our solution, we aim at exposing memory corruption. Thus, we expect that generated scenarios do not fail with run-time exceptions, nor do they produce memory corruptions.

Our model uses for this purpose two heuristics as oracles:

a) Assertions of the virtual machine: They correspond to existing assertions inside the VM used as code checks, such as pre and post-conditions of methods. They are disabled in the production VM but we enable them when fuzzing. We rely on

such autovalidating assertions to check the health of the GC algorithm.

b) Heap Invariant Checks: The Pharo VM includes an extensive (and expensive) memory consistency checker that iterates all heap entities (objects, free memory chunks) and validates invariants such as ensuring that their classes are correct, that their slots point to valid objects, and that old objects refer to young objects are remembered. These checks, normally disabled, are activated by the fuzzer to run on each GC pass.

C. Scenario Execution

Scenario execution is orchestrated by a Fuzzer object that generates events in a stream until a timeout or an error is found. The execution of a scenario relies on three main steps, as illustrated in Figure 3. First, the fuzzer sets up the fuzzing by initializing a heap object, containing an underlying VM simulation. Second, we generate and execute heap events in a loop until timeout. Such an execution is stateful and produces side effects such as allocating objects and moving them in memory. For simplicity, our implementation puts into events transient data such as the address of allocated objects, to be remapped if moved (see Section III-E). Finally, we perform cleanups such as resetting all events to make the fuzzing reusable.

$prepareHeap()$
while $notTimeout$ **do**
  $e \leftarrow buildEvent()$
  $executeEvent(e)$
end while
$cleanup()$

Fig. 3. Orchestration of Test Execution. First, we initialize the heap. Then we execute all events on the current heap. Finally, we reset all events.

D. Heap Preparation

The fuzzer executes events on top of the Pharo VM simulation environment that has been previously adapted to execute unit tests [25]. A VM simulation represents the heap byte per byte equally to the production VM. The GC algorithms that run on the simulation are the same as found in the production VM; the production VM is obtained through transpilation of the simulation VM [18]. The simulation has, for fuzzing purposes, a complete runtime environment with a correctly configured and initialized VM simulator and memory regions. This means allocating the memory area, segmenting it, and initializing mandatory objects and classes.

E. Remapping Objects and Invalid Events

Given the model above, mutation events can become invalid in two cases. First, at construction time, if a mutation is created before any allocation event happens or if the referee object has no slots, then no mutation is possible. The event builder has then the possibility to react by either creating one allocation or producing an event without effect (*nop*). Our current prototype implements the latter.

Second, when building the stream of events ahead of time a mutation may have been statically configured to perform an assignment from/to an object that was collected by a previous collection event. This happens because the event builder does not statically predict if an allocation remains valid during construction. Predicting object collections at build time would require implementing exact GC semantics in the builder.

Instead, our solution detects after each GC event: which objects are collected (and invalidates their associated address) and which objects remain, remapping their associated address. For each allocation, we keep track of its address and its identity hash. After a GC, we iterate the heap and build a relocation map $(identityhash \rightarrow address)$ that we use to remap the addresses above and mark collected objects. This approach proved to be practical because the generated heaps present objects in the order of the tens.

F. Root Management

The Pharo GC determines which objects are no longer being used by starting from the root objects. These roots include, for instance, some special static object location (such as the class table) known by the GC and the temporary variables of each execution frame in the stack of each thread.

Our prototype does not simulate frames, methods, or threads. Instead, we store root objects using a linked list whose head is referenced by a static variable known by the GC to contain a root.

The execution of an allocating event which must be a GC root triggers the allocation of a small 2-slot object used as a node in the linked list. The execution of a GC event will trigger the execution of the GC pass, the GC will therefore consider the objects stored in the linked list to be used.

G. Test Case Reduction

Once a bug is found we use the *ddmin* delta debugging algorithm to reduce the event sequence of a scenario [37]. The objective of this reduction is to simplify further debugging, speed up the test execution and provide a base to develop a regression test. This algorithm reduces the input successively by removing events while the issue is reproduced.

We adapted ddmin to work on a list of events: we successively remove sublists of the original list while the same issue is found. When the removal of events makes the issue disappear or makes a different issue appear, the algorithm restores the removed events, backtracks, and continues with the removal of other events.

Our implementation pays attention to event dependencies: events in the future may depend on events in the past, *e.g.* an object mutation requires two allocated objects in previous events. Thus, removing an allocation from a list of events may break subsequent mutations. When this happens, we ignore the execution of events whose pre-conditions are not met.

Fuzzer	New space collection	Old space collection	Old space compaction	Allocation	Mutation	Pinning
Random	freq=1	freq=1	–	freq=20, types=#all sizes={ 0 .. 1000 }	freq=20	freq=10
Corner allocations	freq=1	freq=1	–	freq=10, types=#array, sizes={0. 20. 255. 100000}	–	–
Ephemerons	freq=1	freq=1	–	freq=10, types=#ephemeron, sizes={0. 1. 2}	freq=10	–
Compaction	–	–	freq=3	freq=10, types=#array, sizes={0. 20. 255. 100000}	–	freq=2

TABLE I

FUZZER CHARACTERISATION. EACH FUZZER (ROWS) IS CONFIGURED TO BUILD DIFFERENT KINDS OF HEAP EVENTS (COLUMNS) WITH DIFFERENT FREQUENCIES DENOTED `FREQ` IN THE TABLE. FOR READABILITY, WE REPRESENT WITH A DOUBLE DASH ALL EVENTS WITH NO PROBABILITY OF HAPPENING (`FREQ=0`). THE ALLOCATION EVENTS PRESENT AS PARAMETERS THE TYPES AND SIZES OF OBJECTS CREATED. IN OUR CURRENT FUZZERS, ALL TYPES AND SIZES CONFIGURED ARE EQUIPROBABLE. NOTE THAT THE RANDOM FUZZER ALLOCATION SIZES RANGE BETWEEN 0 AND 1000.

Fuzzer	A. bug	B. average events	C. reduction
Random	0%	–	–
Corner allocations	100%	87.77 ± 87.01	10.6 ± 8.52
Ephemerons	100%	25.9 ± 17.23	13.38 ± 12.29
Compaction	23.33%	142.14 ± 96.29	4.26 ± 2.4

TABLE II

BENCHMARK DATA. THE TABLE REPORTS THE DATA FROM OUR BENCHMARKS USING THE FOUR DIFFERENT FUZZERS, ONE PER ROW. FOR EACH FUZZER, WE REPORT (A) THE OBSERVED PROBABILITY TO HIT A BUG OVER 30 RUNS WITH A TIMEOUT OF 1024 EVENTS, (B) HOW MANY EVENTS ON AVERAGE WERE REQUIRED TO SPOT THE BUG AND (C) WHAT PERCENTAGE OF EVENTS REMAINED AFTER THE REDUCTION IN AVERAGE. ALL AVERAGES REPORT ALSO THE STANDARD DEVIATION.

IV. EVALUATION

In this section, we show the effectiveness of our solution by evaluating four different heap fuzzers and our ability to reduce fuzzed inputs. We evaluate our solution on top of the Pharo Virtual Machine (See Section IV-B) using four different heap fuzzers (See Section IV-A)

We answer the following research questions:

- *RQ1: How does expert-guided heap fuzzing improve over random fuzzing?* (See Section IV-C)
- *RQ2: How does the number of events increase the probability of finding bugs?* i.e., *What is a reasonable timeout for our fuzzers?* (See Section IV-D)
- *RQ3: How effective is test reduction in the found bugs?* (See Section IV-E)

Table II presents an overview of our results that we analyze in the sections below.

A. Fuzzers

For this evaluation, we implemented four different heap fuzzers: one random fuzzer and three guided fuzzers created by experts with knowledge of the internals of the Pharo VM, as described below. Table I characterizes more precisely each of the four fuzzers and how they are configured.

a) Random: A random fuzzer creates at random any event supported by our fuzzing model. To better represent real cases, allocations and mutations are more frequent than collection events. The objective of this random fuzzer is to work as a baseline comparison.

b) Corner case allocations: This fuzzer creates allocations with corner case sizes. This fuzzer has a high chance of allocating objects of 0, 255, and 10000 slots. Its objective is to stress specific allocation scenarios.

c) Ephemeron allocations: This fuzzer creates exclusively ephemeron objects. Its objective is to test the VM finalization mechanism, under development during the writing of this paper.

d) Compaction: This fuzzer pins objects in memory to forbid them to move, and uses a more aggressive compaction mechanism on the old space. Its objective is to stress the compaction mechanism, from which developers have observed intermittent crashes during continuous integration runs.

B. Experimental Platform

We implemented our approach on top of the Pharo Virtual Machine [9], an industrial-level Virtual Machine supported by an industrial consortium[2]. The VM implements a sequential stop-the-world generational scavenger garbage collector that uses a copying collector for young objects and a mark-compact collector for older objects [30].

We implemented our prototype to run on Pharo VM's simulator [22] and using the pre-existing Pharo VM testing infrastructure [25]. Our prototype implementation is hosted in https://github.com/Alamvic/heapFuzzer and is MIT licensed. We run all our benchmarks on the following configuration:

- Pharo Version: Pharo 11 alpha build #219[3]
- Heap Fuzzer commit[4] c18a831
- Pharo VM commit[5] 3172ed61

Unless stated otherwise, each benchmark is run 30 times (so 30 scenarios) with a limit of 1024 events per scenario.

C. Random vs. Expert-Guided Fuzzers

To answer *RQ1* (*How does expert-guided heap fuzzing improve over random fuzzing?*) we measured the effectiveness of three expert-built fuzzers against a fully random fuzzer. Each of the three expert-built fuzzers guided the execution toward a specific part of the allocation and garbage collection code.

Figure 4 shows the percentage of executions of each fuzzer that hit a bug. Notably, the random fuzzer found no bug in the execution. Two out of the three expert fuzzers (ephemerons and corner case allocations) found bugs in over 100% of their executions. The compaction fuzzer only hit a bug on 23.3% of the cases.

[2]https://consortium.pharo.org/

[3]Full Commitish ccb9b558742bf1d86df35c44ae5802e4a05a03ca

[4]Full Commitish c18a8314bdbddfafb2a4c14afb09451d2b0e506f

[5]Full Commitish 3172ed61ee1accbccdd11130e0e797a106ca4e69

These results indicate that:

1) random fuzzing does not visit the search space in a way useful to find bugs,
2) expert-guided fuzzers are useful to spot bugs in GC implementations, and
3) expert-guided fuzzers are not 100% accurate and may need to be refined to improve their efficiency, as shown in the case of the compaction fuzzer.

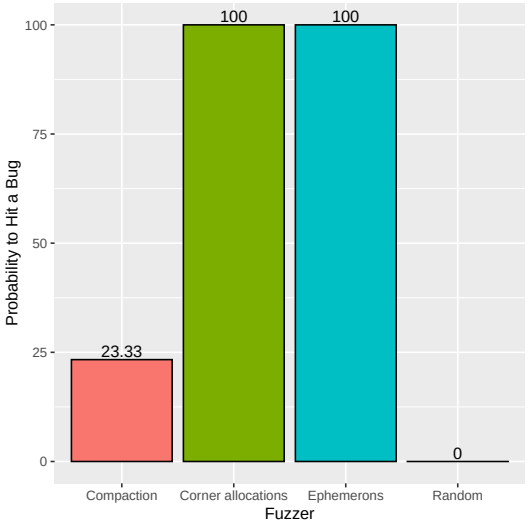


Fig. 4. Percentage of 30 executions of each fuzzer that hit a bug using a timeout of 1024 events.

D. Analyzing Fuzzing Timeout

To answer *RQ2* (*How does the number of events increase the probability of finding bugs? i.e.,* What is a reasonable timeout for our fuzzers?) we measured how many events needed to be executed in a failing fuzzed execution before hitting a bug. If a fuzzer executes all events (1024 by default) without failing we consider it a success. Otherwise, we consider it as an error, we record the failure and the number of events before producing the failure. Figure 5 shows a violin plot with the distribution of the number of events to fail each fuzzer. We can observe in the figure that all expert-guided fuzzers found a failure on average before 200 events in most cases, which is way below our timeout of 1024 events. The figure does not show the random fuzzer, as it did not find any failure in our runs.

These results indicate that expert-guided heap fuzzers are useful with low timeouts (less than 200 events) making them a fast and practical solution.

E. Analyzing Reduction Effectiveness

Our solution includes test reduction at its core because regardless of how effectively a fuzzer is to find a bug, GC test cases remain difficult to debug.

To answer *RQ3* (*How effective is test reduction in the found bugs?*), we run our implementation of the ddmin algorithm (see Section III-G) against each of our fuzzed test

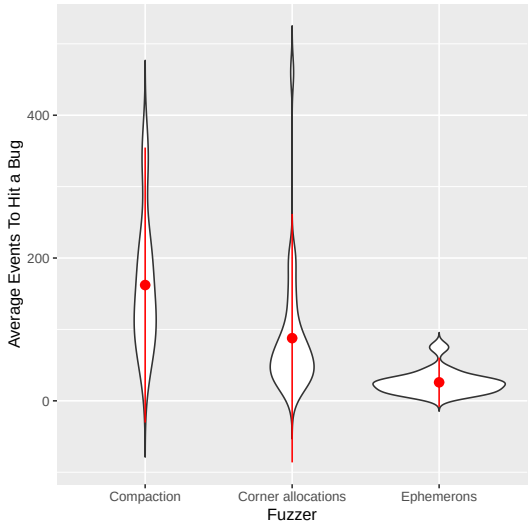


Fig. 5. Number of events that needed to be executed before hitting a bug in average per fuzzer. In red the plot shows the average and standard deviation.

cases and we compared the number of events of the reduced test case against the number before reduction. Figure 6 reports the results as an average reduction per fuzzer. The results show that all our fuzzers generated sequences that could be reduced to less than 14% of the original sequence. Ephemeron fuzzers reduced their average number of events to almost 14%, corner case allocations to 11%, and the compaction fuzzer reduced its events to below 5% of the original on average.

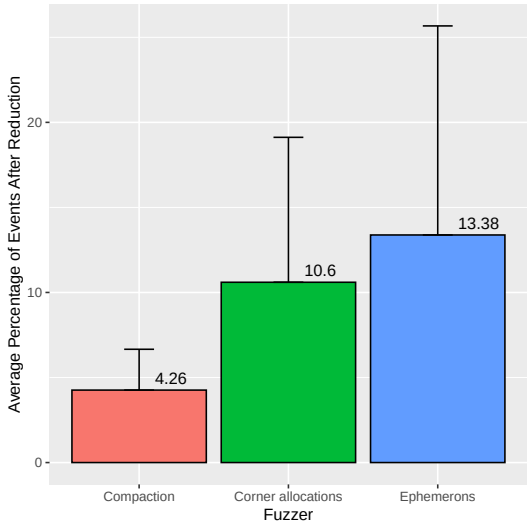


Fig. 6. Percentage of remaining events per fuzzer in average after test case reduction.

These results indicate that:

1) test reduction is useful to remove superfluous events in the case of heap fuzzing and improve debugging;
2) there is still room for improvement in expert-guided fuzzers to make them more efficient at finding bugs.

V. Identified Bugs

In this section, we present the bugs the fuzzers found. Applying our approach we found six bugs in total - three in the ephemeron implementation which is not in production at the time of writing, two in the compacting collector which is used in production for more than 8 years, and one in the simulation environment itself used during the VM development and in our approach.

All the identified bugs have been reported to the PharoVM developers with a test case to reproduce it and the proposition of a patch to fix it.

The workflow was the following:

1) Run the fuzzer until it produces one (or more) scenario that fails (as the previous section shows, expert-guided fuzzers were efficient). Note that the fuzzer can produce many failing scenarios but cannot describe or classify them. The analysis of a scenario is the job of the expert.
2) Reduce the scenario to something more manageable for a human.
3) Replay and debug the scenario. Since each scenario is replayable by construction, there is no heisenbug.
4) Fix the bug locally.
5) Write a unit test to avoid regression. The test can be inspired by the scenario.
6) Restart at step one to try to find another bug. If the fix is correct, the bug should not be found again.

A. Severe Bug 1: Marking Ephemerons

Our first discovered bug, presented only four events after reduction: two root allocations in the new space, a mutation, and a full GC. The test discovered with the invariant checker that the second ephemeron's class was mistakenly garbage-collected producing a memory corruption. This issue has been reported in the PharoVM repository, and it has already been fixed.[6]

B. Severe Bug 2: Scanning Ephemeron References

Our second discovered bug, presented also four events after reduction: one *root* allocation in the new space, one non-root allocation in the new space, a mutation, and a full GC. The test discovered with an assertion during pointer update that the GC was not scanning ephemeron value references, producing a memory corruption due to a *use-after free error*. This issue has been reported in the PharoVM repository, and it has been already fixed.[7]

C. Bug 3: Ephemeron Instances Without Instance Variables Produce Memory Access Error

This bug is identified by two events after reduction: one allocation of an ephemeron without instance variables, and a garbage collection event. The test discovered that when an ephemeron is marked, the GC code assumes that the ephemeron will have at least one instance variable that works as the key of the ephemeron, producing an *invalid read of uninitialized memory*. The instance creation code does not validate this assertion, and the GC code does not handle this invalid situation. This issue has been reported in the PharoVM repository.[8]

D. Bug 4: Compacting Unmarked Object at End of Memory

This bug is identified by five events after reduction: one *root* allocation in the new space, a full GC, one non-root allocation in the new space, a mutation, and another full GC. The test discovered with the invariant checker that an unmarked object remained in the old space after compaction, *i.e.,* the object was not compacted and returned to free memory. This issue has been reported in the PharoVM repository, and it has already been fixed.[9]

E. Bug 5: Freeing an Object While Iterating the Memory Might Invalidate the Iteration

This bug was identified after six events after reduction: one non-root allocation in new space, one aggressive compaction, two allocations of root objects in old space, one allocation of a non-root object in old space, and one aggressive compaction. The test discovered that when an object is freed while iterating the old space (because the GC has found an unmarked object), and this object is followed by a free chunk, the coalescence of both free chunks to form a bigger one renders invalid the pointer used to iterate the old space. As the iterating pointer is not correctly pointing to the next valid object start, the iteration of the memory access invalid memory or corrupts the heap.

Note: this bug was revealed after we applied the fix of another bug. This issue has been reported in the PharoVM repository, and it has already been fixed.[10]

F. Bug 6: Extending Memory in Two Non-continuous Segments Breaks Memory Allocation Simulation

This bug was identified by five events after reduction: two large array allocations of root objects, one old space garbage collection, one large array allocation, and an old space garbage collection. The test discovered that when running in the VM simulator, our simulation harness to simulate the dynamic virtual memory allocation provided by the operating system (simulation of functions like `malloc()`, `calloc()`, and virtual page allocation) produces an *invalid read of uninitialized memory* issue. This is an interesting result as the test provides a validation also of the simulation environment and allows us to improve the quality of this and how it should behave in different scenarios. This issue has been reported in the PharoVM repository.[11]

[6]https://github.com/pharo-project/pharo-vm/issues/468
[7]https://github.com/pharo-project/pharo-vm/issues/469
[8]https://github.com/pharo-project/pharo-vm/issues/493
[9]https://github.com/pharo-project/pharo-vm/issues/470
[10]https://github.com/pharo-project/pharo-vm/issues/489
[11]https://github.com/pharo-project/pharo-vm/issues/494

VI. Related Work

A. Automated Garbage Collection Testing and Verification

The testing and verification of garbage collection algorithms have been mostly approached with formal verification. Existing work is divided into two families. One family of solutions performs static verifications of GC algorithms using theorem provers [13], [27]. A second family of solutions implements model checking approaches [5], [29], [34].

Because formal verification aims to give strong guarantees, it is usually heavy-weight to implement and deploy. It requires the specification of GC semantics as a separate formalism that requires maintenance to stay in sync with the actual implementation. Thus, techniques have been developed to optimize formal approaches [1]. In comparison, our solution trades-off precision for being lightweight to implement, replicate and execute.

Another difference between existing approaches and ours is that they mostly focus on the verification of concurrent GCs. In this work, we focused on the verification of a sequential stop-the-world GC. While in the future we plan to use our approach to test concurrent GCs, our approach applies to partial features of a GC implementation without the need for an entire memory model.

B. Automated Virtual Machine Testing

Besides work on validating GCs, a lot of work focuses on testing VMs in general, and more specifically JIT compilers. The teams of Maxine and Pharo reported recently QEMU-based unit testing infrastructures for cross-ISA testing and debugging [8], [20], [25]. They reported that these infrastructures helped them in porting their VMs to ARMv7 and ARMv8 64bits respectively. Although their approaches are based on unit testing, they rely on manually written tests, while our approach performs automatic test generation.

Lately, several works have explored the path of program fuzzing and differential testing. Recent work explores the generation of test cases for JIT compilers using available interpreters [24]. Other solutions make use of differential testing between different Virtual Machines for a single language. Several works explore bytecode fuzzing on the Java Virtual Machine (JVM) [6], [7] or test generation for JVM native extensions (*i.e.,* JNI) [17]. Recent work explores cross-version comparison among a single JVM [38]. Similar work appeared recently for JavaScript engines using test transplantation [21] and compiler fuzzing [23], [36]. Although we share with these approaches the goal of automatic test generation, these explore JIT compiler testing and treat VMs as black boxes. Instead, our fuzzing approach is grey-box: it has knowledge of the GC behaviour to generate pertinent events. Moreover, our approach produces random yet reproducible tests that exercise the GC, while the other approaches are subject to VM non-determinisms in the JIT compilers and their profiling mechanisms.

C. VM Simulation Environments and Meta-circular VMs.

Meta-circular VMs and VM frameworks offer simulation environments for a long time. Such simulation environments help in testing and debugging virtual machines. Such is the case of Self [31], Smalltalk [18], [22] and Maxine [33]. Our solution exploits simulation environments to simplify the execution of generated unit tests.

D. Allocator Testing and Exploit Generators

The hacking and security communities have used previously fuzzing and symbolic execution to device scenarios for automatic exploit generation [2], [4], [15], [28], [35]. However, these solutions focus on stack-related exploits such as stack over/underflows.

Recent work on security proposes to fuzz heap allocators (*i.e.,* `malloc` implementations) to reproduce heap layouts with exploit purposes [16]. This work is similar to ours in that it generates events directed to the allocator, as we do with events directed to the object allocator and garbage collector. A key difference is that they look to produce heap layouts that can be exploited due to memory over/underflows, while we test the correctness of GC algorithms instead.

Vanegue et al. [32] propose a calculus for memory allocators that can be used to formalize our directed fuzzers.

VII. Conclusion

In this article, we proposed to fuzz heaps by generating large sequences of randomly expert-guided heap events. Heap events directly manipulate the memory manager and circumvent the programming language semantics to get access to more precise fuzzing *e.g.,* the fuzzer can decide where or how to allocate an object. Such events exercise the GC algorithm to produce VM crashes and find bugs. Once a bug is found, we use a test case reduction algorithm to find the smaller subset of events that reproduces the issue.

We implemented our approach on top of the VM simulator of the Pharo VM and showed that our prototype can find real-world bugs quickly. We further described six bugs our solution found in Pharo's garbage collection: three severe ephemeron implementation bugs that produced memory corruptions, two bugs in the compaction algorithm that prevented the compaction of some objects at the end of the heap, and one bug in the simulation environment used during the VM development.

In the future, we plan to extend this work in four different research directions. First, we will study how fuzzing techniques can be used to evaluate the correctness of GC implementations besides memory corruptions, checking for example that objects that should be freed are effectively freed and not producing memory leaks. Second, we plan to extend our fuzzer to model interactions between the GC and the execution engine by fuzzing execution stacks. Third, besides a mechanical reduction, there is still a need to use program-comprehension techniques to understand the cause of an issue. Finally, expert-guided fuzzers rely on the ability of experts to propose probabilities of events and values of parameters

that should trigger corner cases. We will explore automatic techniques to guide the creation of more effective fuzzers.

References

[1] T. Abe, T. Ugawa, T. Maeda, and K. Matsumoto. Reducing state explosion for software model checking with relaxed memory consistency models. *CoRR*, abs/1608.05893, 2016.

[2] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley. Automatic exploit generation. *Commun. ACM*, 57(2):74–84, feb 2014.

[3] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015.

[4] D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 143–157, 2008.

[5] J. Caballero, G. Grieco, M. Marron, and A. Nappa. Undangle: Early Detection of Dangling Pointers in Use-after-Free and Double-Free Vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 133–143, New York, NY, USA, 2012. Association for Computing Machinery.

[6] Y. Chen, T. Su, and Z. Su. Deep Differential Testing of JVM Implementations. In *International Conference on Software Engineering (ICSE'19)*, pages 1257–1268. IEEE Press, 2019.

[7] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao. Coverage-directed differential testing of JVM implementations. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 85–99, June 2016.

[8] S. Ducasse, G. Polito, O. Zaitsev, M. Denker, and P. Tesone. Deprewriter: On the fly rewriting method deprecations. *Journal of Object Technologies (JOT)*, 21(1), 2022.

[9] S. Ducasse, D. Zagidulin, N. Hess, D. C. O. written by A. Black, S. Ducasse, O. Nierstrasz, D. P. with D. Cassou, and M. Denker. *Pharo by Example 5*. Square Bracket Associates, 2017.

[10] C. W. Enumeration. 2022 cwe top 25 most dangerous software weaknesses – consulted on 27 september 2022. https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html.

[11] C. W. Enumeration. Cwe-416: Use after free – consulted on 27 september 2022. https://cwe.mitre.org/data/definitions/416.html.

[12] C. W. Enumeration. Cwe-825: Expired pointer dereference – consulted on 27 september 2022. https://cwe.mitre.org/data/definitions/825.html.

[13] P. Gammie, A. Hosking, and K. Engelhardt. Relaxing Safely: Verified On-the-Fly Garbage Collection for x86-TSO. *ACM SIGPLAN Notices*, 50:99–109, jun 2015.

[14] B. Hayes. Ephemerons: A new finalization mechanism. In *International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'97)*, 1997.

[15] S. Heelan. *Automatic generation of control flow hijacking exploits for software vulnerabilities*. PhD thesis, University of Oxford, 2009.

[16] S. Heelan, T. Melham, and D. Kroening. Automatic heap layout manipulation for exploitation. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 763–779, Baltimore, MD, aug 2018. USENIX Association.

[17] S. Hwang, S. Lee, J. Kim, and S. Ryu. JUSTGen: Effective Test Generation for Unspecified JNI Behaviors on JVMs. In *International Conference on Software Engineering (ICSE'21)*, pages 1708–1718, 2021.

[18] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: The story of Squeak, a practical Smalltalk written in itself. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications conference (OOPSLA'97)*, pages 318–326. ACM Press, Nov. 1997.

[19] R. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, July 1996.

[20] C. Kotselidis, A. Nisbet, F. S. Zakkak, and N. Foutris. Cross-isa debugging in meta-circular VMs. In *Proceedings of International Workshop on Virtual Machines and Intermediate Languages (VMIL'17)*, pages 1–9, 2017.

[21] I. Lima, J. Silva, B. Miranda, G. Pinto, and M. d'Amorim. Exposing Bugs in JavaScript Engines through Test Transplantation and Differential Testing. *arXiv:2012.03759 [cs]*, 2020.

[22] E. Miranda, C. Béra, E. G. Boix, and D. Ingalls. Two decades of Smalltalk VM development: live VM development through simulation tools. In *Proceedings of International Workshop on Virtual Machines and Intermediate Languages (VMIL'18)*, pages 57–66. ACM, 2018.

[23] J. Park, S. An, D. Youn, G. Kim, and S. Ryu. JEST: N+1-Version Differential Testing of Both JavaScript Engines and Specification. In *International Conference on Software Engineering (ICSE'21)*, pages 13–24. IEEE Press, 2021.

[24] G. Polito, N. Palumbo, P. Tesone, S. Labsari, and S. Ducasse. Interpreter-guided Differential JIT Compiler Unit Testing. In *Programming Language Design and Implementation (PLDI'22)*, 2022.

[25] G. Polito, P. Tesone, S. Ducasse, L. Fabresse, T. Rogliano, P. Misse-Chanabier, and C. H. Phillips. Cross-ISA Testing of the Pharo VM: Lessons Learned While Porting to ARMv8. In *Proceedings of the 18th international conference on Managed Programming Languages and Runtimes (MPLR '21)*, Münster, Germany, Sept. 2021.

[26] T. C. Projects. Memory safety – consulted on 27 september 2022. https://www.chromium.org/Home/chromium-security/memory-safety/.

[27] A. Sandberg Ericsson, M. O. Myreen, and J. Åman Pohjola. A Verified Generational Garbage Collector for CakeML. *Journal of Automated Reasoning*, 63(2):463–488, 2019.

[28] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[29] T. Ugawa, T. Abe, and T. Maeda. Model Checking Copy Phases of Concurrent Copying Garbage Collection with Various Memory Models. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017.

[30] D. Ungar. Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, 1984.

[31] D. Ungar, A. Spitz, and A. Ausch. Constructing a metacircular Virtual machine in an exploratory programming environment. In *Companion to Object-Oriented Programming, Systems, Languages, and Applications conference (OOPSLA'05)*, pages 11–20, New York, NY, USA, 2005. ACM.

[32] J. Vanegue. Heap models for exploit systems. In *Proceedings of the IEEE Security and Privacy Workshop on Language-Theoretic Security (LangSec)*, 2015.

[33] C. Wimmer, M. Haupt, M. L. Van De Vanter, M. Jordan, L. Daynès, and D. Simon. Maxine: An approachable virtual machine for, and in, java. *ACM Transaction Architecture Code Optimization*, 9(4), Jan. 2013.

[34] B. Xu, E. Moss, and S. M. Blackburn. Towards a model checking framework for a new collector framework. In *Proc. MPLR*, pages 128–139. ACM, 2022.

[35] L. Xu, W. Jia, W. Dong, and Y. Li. Automatic exploit generation for buffer overflow vulnerabilities. In *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 463–468, 2018.

[36] G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, X. Sun, L. Bian, H. Wang, and Z. Wang. Automated conformance testing for javascript engines via deep compiler fuzzing. In *Conference on Programming Language Design and Implementation (PLDI'21)*, pages 435–450. ACM, 2021.

[37] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, SE-28(2):183–200, Feb. 2002.

[38] Y. Zhao, Z. Wang, J. Chen, M. Liu, M. Wu, Y. Zhang, and L. Zhang. History-Driven Test Program Synthesis for JVM Testing. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, pages 1133–1144, New York, NY, USA, 2022. Association for Computing Machinery.