Reconstructing Core Dumps

Jeremias Rößler* · Andreas Zeller* · Gordon Fraser† · Cristian Zamfir‡ · George Candea‡
*Saarland University, Saarbrücken, Germany
Email: {roessler, zeller}@cs.uni-saarland.de
†University of Sheffield, Sheffield, United Kingdom
Email: gordon.fraser@sheffield.ac.uk
‡École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland
Email: {cristian.zamfir, george.candea}@epfl.ch

***Abstract*—When a software failure occurs in the field, it is often difficult to reproduce. Guided by a memory dump at the moment of failure (a "core dump"), our RECORE test case generator searches for a series of events that precisely reconstruct the failure from primitive data. Applied on seven non-trivial Java bugs, RECORE reconstructs the exact failure in five cases without any runtime overhead in production code.**

***Keywords*-debugging; failure reproduction; test case generation; memory dumps**

I. Introduction

When a program fails in the field, the developer who wants to debug the failure must first be able to *reproduce* it. Being able to reproduce a failure locally is important for diagnostic purposes, because developers can then inspect arbitrary aspects of the failing execution to locate the failure cause. If the problem cannot be reproduced, though, one can never know whether the fix is successful: Only if the failure goes away after the fix has been applied does the developer know that she effectively found the failure cause.

Reproducing a failure is easy—in theory. Since a program is but a mathematical mapping of inputs to outputs, all it takes is to record the inputs to reproduce the output. The problem in practice is that recording inputs incurs time and memory overhead, and neither developers nor users may be willing to spend resources on a diagnostic feature that (in the best of cases) will never be used.

Once a failure *has* occurred, though, such diagnostic overhead no longer matters. A common practice on modern operating systems is to produce a *core dump*—a persistent snapshot of the stack and heap memory at the moment of the failure. Such a core dump can be loaded into interactive debuggers, and programmers can then inspect it post-mortem to understand how the failure came to be.

As an example, consider Figure 1, showing a stack trace originating from a failure in the JAVA JODATIME date and time library. We see that the `toInterval()` method was called with some specific date and time zone, resulting in the method `localToUTC()` raising an `IllegalArgumentException`. In addition to this stack trace, a full core dump would also hold all values on the stack and on the heap at the moment of failure.

Such a core dump, however, just reports the final state, not the history. Assuming one finds an illegal value in some variable: How did that value come to be? The input that caused the failure may long have been overwritten by later input: What was it that caused the problem in the first place? With a core dump, you are a detective faced with a corpse—and you have to reconstruct all the events that lead to it.

For computer programs, however, we do have tools that can construct such events automatically. *Test case generation* addresses the problem of generating inputs that fulfill a number of constraints—typically, reaching a particular location in the program. *Constraint-based test generators* do so by solving the path conditions from input to these locations. Reproducing a full core dump, though, would impose far too many constraints on any practical constraint solver.

In practice, it may suffice to have a test case that produces a state *as similar as possible* to the core dump. This is the domain of *search-based test generators,* which systematically evolve the input according to a fitness function. If similarity to the core dump induces high fitness, we could then *automatically obtain a test case that reproduces the core dump*—without incurring overhead during execution.

This is what we explore in this paper. Given a stack and heap dump, our RECORE prototype uses search-based test generation to automatically produce a set of inputs that

```
java.lang.IllegalArgumentException:
    Illegal instant due to time zone offset transition:
    2009-10-18T03:00:00.000
at org.joda.time.chrono.ZonedChronology.localToUTC
    (ZonedChronology.java:143)
at org.joda.time.chrono.ZonedChronology.getDateTimeMillis
    (ZonedChronology.java:119)
at org.joda.time.....AssembledChronology.getDateTimeMillis
    (AssembledChronology.java:133)
at org.joda.time.base.BaseDateTime.<init>
    (BaseDateTime.java:254)
at org.joda.time.DateMidnight.<init>
    (DateMidnight.java:268)
at org.joda.time.LocalDate.toDateMidnight
    (LocalDate.java:740)
at org.joda.time.LocalDate.toInterval
    (LocalDate.java:847)
```

Figure 1. JODATIME bug 2487417: `toInterval()` fails when processing October 18, 2009, Brazil time.

```
void recore_test() {
    LocalDate localDate1 =
        new LocalDate(1255824000000L);
    FixedDateTimeZone fixedDateTimeZone2 =
        new FixedDateTimeZone
        ("America/Sao_Paulo", "", 7, 7);
    TimeZone timeZone0 =
        fixedDateTimeZone2.toTimeZone();
    DateTimeZone dateTimeZone0 =
        DateTimeZone.forTimeZone(timeZone0);
    Interval interval0 =
        localDate1.toInterval(dateTimeZone0);
}
```

Figure 2. The test case generated by RECORE reproduces exactly the stack trace in Figure 1.

cause a single-threaded program to throw the same exception and have the same stack trace at the time of throwing the exception; we refer to this as "reconstructing the failure". The generated test case comes as a sequence of method calls taking only primitives or generated objects as parameters; we can thus reconstruct the *entire history* of the failure-inducing state.

RECORE extends the state of the art in several ways:

1) RECORE[1] is the first technique to use *evolutionary search-based test generation* to reconstruct failures from saved core dumps. The basic idea of RECORE is to use EVOSUITE [11] to produce a test case that is *as similar to the failure* as possible. Here, "similarity" means similarity of the *stack trace* (the active methods) as well as of the *stack dump* (local variables and method arguments) at the moment of the failure.
2) RECORE[2] goes beyond existing record/replay tools such as RECRASH [2] or BUGREDUX [15] in that the approach *requires zero runtime overhead.*[3] That is, production code can run unchanged, with same time and space requirements; RECORE kicks in only at the moment of failure to produce a core dump.
3) Being search-based, RECORE can also produce partial solutions. Furthermore, it turns out that a new *fitness function* and improved *argument seeding*, which both strive for similarity, are all it takes to turn a search-based test generator into a failure reconstructor; if more events were to be reconstructed, these parts can be easily extended. Both partial solutions and extensibility towards reproducing other recorded events are in contrast to constraint-based approaches such as execution synthesis [35].
4) Finally, rather than unserializing data and objects from the core dump, as RECRASH does, RECORE *reconstructs all objects from primitive values.* The resulting test case thus shows a history of how the failing data comes to be.

Our experiments are promising: On seven bugs, RECORE is able to reconstruct the exact failure in five cases, and partially in another case. The resulting test cases are self-contained, short, and comprehensible. At the same time, the RECORE approach incurs no overhead in production code.

The remainder of the paper is organized as follows: Section II demonstrates RECORE on a motivating example. Section III discusses the state of the art in reproducing failures, from recording and replaying values to execution synthesis. Section IV details how we collect core dumps from JAVA programs. Section V describes how RECORE leverages core dumps to systematically generate and evolve test cases. With this, RECORE is able to reconstruct five out of seven JAVA failures completely, and one more partially (Section VI). We close with conclusion and future work (Section VII).

II. AN EXAMPLE

As an example of how RECORE works, reconsider the core dump that corresponds to the stack trace in Figure 1. Given this core dump as input, RECORE automatically produces the test shown in Figure 2. This test case fails like the original, reproducing exactly the stack trace from Figure 1. The test has all the necessary input:

- `localDate1` holds a specific instant in JODATIME format—namely, 1,255,824,000,000 milliseconds after January 1, 1970. This corresponds to October 18, 2009.
- `fixedDateTimeZone2` holds the time zone for this instant—namely, "America/Sao_Paulo".
- Feeding this date and time zone into `toInterval()` raises an exception.

Developers can now load this test case into the debugger and reproduce the failure at will; as it is self-contained, no additional context (such as serialized objects from the original core dump) is required. Better yet, one can feed the test case as is into an automated debugger. The BUGEX tool [29], for instance, would pinpoint the failure-inducing branch, showing that October 18, 2009 is the last day of daylight savings time (DST) in Brazil, a condition that is improperly handled by JODATIME due to an inconsistent mapping between UTC time (the internal intermediate format) and local time. Applying the official fix for the original failure also fixes the RECORE test case, indicating that it indeed triggered the exact defect.

Figure 3 shows the basic steps of RECORE. It starts with a program and a given core dump, consisting of stack and heap data. RECORE then uses EVOSUITE as a test generator, leveraging the core dump as fitness guidance and source for primitive values. When the original stack trace is reconstructed, RECORE issues the appropriate test case.

[1] "RECORE" stands for "Reconstructing core dumps".

[2] "RECORE" is also one bit beyond "RECORD" (as in record/replay).

[3] This assumes directly executable binaries. For JAVA programs and other interpreted languages, we assume support of the virtual machine to produce core dumps, as discussed in Section IV.

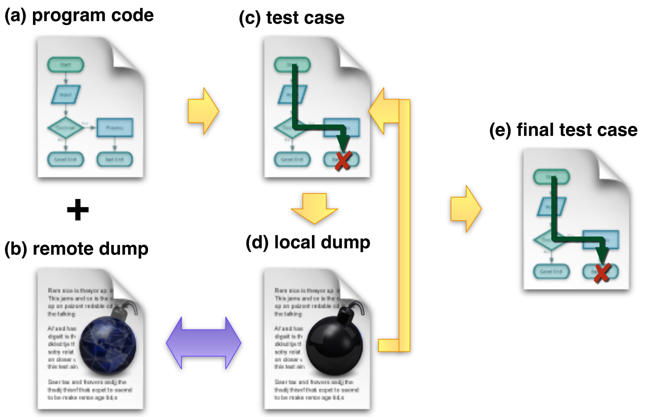

Figure 3. How RECORE works. Given a executable program (a) and a remote core dump (b), RECORE generates tests (c) whose final state (d) is then compared to the original core dump (b). The test generator systematically evolves the test case to increase similarity. The final test case (e) makes the program fail with the exact same stack trace.

III. Background

A. Reproducing Failures

Most approaches to reproducing failures rely on *record/replay:* The idea is to *record* events first, and to *replay* them later. This works well for low-bandwidth event sources; recording a user's input into a GUI, for instance, is unlikely to have a perceptible impact on performance.

As larger amounts of data are recorded, overhead becomes significant. At the system level, record-replay systems such as ReVirt [9], PRES [27] and ReSpec [21]) and hardware and/or compiler assisted systems (e.g., Capo [25], CoreDet [3]) support recording of multi-core, shared-memory intensive programs. However, these systems either incur high runtime overhead due to recording, or require specialized hardware that is not widely available. For data centers, for instance, making up for a 50% throughput drop requires provisioning twice as many machines. The additional storage required for recording also matters: record-replay systems [1] for datacenter applications report significant log sizes even after applying aggressive logging optimizations.

For general-purpose software, record/replay has been mainly explored in the context of *automated debugging.* ADDA [6] records events at the level of C standard library and file operation functions. JINSI [4] records the interaction between individual objects in terms of methods called and objects passed. The large overhead of such systems, however, limits data recording to the lab.

If one is willing to suffer execution overhead, one can record crucial information required to reproduce the error. The BUGREDUX framework [15] can record execution data in the field and (like RECORE) use test case generation to reconstruct the failure. In its evaluation, the authors found stack traces collected during execution to provide the greatest benefit compared to the overhead in collecting the data. Likewise, SherLog [33] uses clues from existing application logs to reconstruct a partial path to a failure using static analysis and [34] improves SherLog's accuracy using runtime recording, thus introducing runtime overhead.

A little collected information can go a long way. The RECRASH tool [2] records executions by recording parts of the program state at each method entry—namely those objects that are reachable via direct references. When the program crashes, it thus allows the developer to observe a run in several states before the actual crash. Even just recording these references results in a 13%–64% performance overhead. However, it allows the programmer to examine the last few steps before the failure. If the failure-inducing state was created outside of the last stack trace, however, RECRASH will only show that feeding this state caused the crash, but give no indication on how the state came to be.

For the special domain of concurrency bugs, the work of Weeratunge at al. [31] analyzes multi-core dumps to reconstruct the thread schedule that caused a crash. Again, this assumes the program inputs are recorded, inducing overhead; however, RECORE does not handle concurrent programs at this point.

What sets RECORE apart from all these approaches is that it aims at *zero overhead.* Furthermore, we not only *reproduce* the failure, but *reconstruct* it—by devising a self-contained sequence of events that reproduce the stack trace and failure from the core dump.

B. Execution Synthesis

The development of RECORE was mainly inspired by recent work on *execution synthesis.* Execution synthesis [35] is a technique that automatically reproduces bugs starting from a bug report, *without doing any recording while the software is running in production.* The ability to reproduce failures without runtime overhead in production sets execution synthesis and RECORE apart from record/replay systems.

Execution synthesis uses the core dump to extract the program failure condition and the stack trace (but not the heap). It combines static analysis, symbolic execution [19], [5] and thread schedule synthesis to find an execution that matches the stack trace and the failure in the core dump. This combination of analyses reduces the search space of possible executions. Execution synthesis uses context-sensitive proximity heuristic to select the executions that are more likely to reproduce the failure. Once it finds a path that reproduces the failure, it uses Klee [5] to compute the concrete program inputs required to reproduce the failure.

The main challenge of execution synthesis is reducing the search space. This is especially problematic for deep execution paths that generate hard-to-solve constraints. In such cases, the technique will take longer to reproduce a failure or it will timeout.

RECORE goes beyond execution synthesis in a number of ways. First, it is more general: While execution synthesis only uses the final stack trace, RECORE also relies on method arguments and targets—and can be easily extended to arbitrary mid-execution events or states. Second, by using a *search-based* approach to reconstructing the core dump, it can produce partial solutions—that is, test cases that reconstruct as much of the failure as possible, but which need not satisfy *all* constraints imposed by a constraint-based approach such as execution synthesis. On the other hand, execution synthesis is applicable to multi-threaded programs and can deduce inputs that would be hard to find through evolutionary search. In the long run, we expect test case generation (and thus failure reconstruction) to integrate evolutionary algorithms with constraint solvers to form a greater whole.

C. Test Generation

The recently most successful test generation approaches can roughly be divided into variants of *random testing* (e.g., Randoop [26]), *dynamic symbolic execution* (e.g., DART [13], Klee [5], Pex [30]), and *search-based testing* (e.g., EVOSUITE [11]). Although random testing can be effective in triggering faults, systematic approaches are required to explore more than just "shallow" parts of a program. Dynamic symbolic execution (DSE) derives path conditions for concrete runs using symbolic execution, and then drives exploration by negating individual constraints, for which solutions represent new inputs that follow different program paths. DSE can effectively explore all program paths, but one main advantage of search-based testing [24] (SBST) is that it is very easy to adapt it to new testing objectives (such as reconstructing core dumps), and tests can be optimized towards different functional and non-functional properties.

One of the most popular search algorithms applied in SBST is a Genetic Algorithm, where a population of candidate solutions is evolved using operators that intend to imitate natural evolutionary processes. A fitness function determines how close a candidate solution is to the optimization target, and the fitter an individual is, the more likely it is used during reproduction, which is based on selection, crossover and mutation.

RECORE is based on the EVOSUITE framework [11]. EVOSUITE applies a Genetic Algorithm to produce unit test suites for object-oriented software. EVOSUITE has two modes of operation: In whole test suite generation, individuals of the search are test suites, and the fitness guides these test suites towards satisfying all coverage goals. The other mode of operation, which is used in RECORE, optimizes individual test cases towards satisfying an individual goal. A test case is a sequence of statements (e.g., constructor and method calls, primitive value assignments, field assignments, array assignments), and mutation modifies a sequence by deleting, adding, and replacing statements. For more details about the search operators we refer to [11].

While RECORE uses search-based test generation, BUGREDUX and execution synthesis (see above) rely on symbolic execution to generate tests. In principle, search-based and symbolic approaches can also be combined to achieve better results (e.g., [22]).

IV. COLLECTING CRASH DATA

Operating systems such as Unix or Windows can be set up to produce a core dump upon abnormal termination of the program—a file (traditionally named "core") which would contain the program counter, stack pointer, memory contents, and other processor and operating system information. Typically, the core contains a partial heap image, however, full core dumps can be collected on demand [12]. Such a core file can be loaded into a debugger to explore the state at the moment of failure just as if the program were still running. As core dumps are only produced in case of failures, they induce zero overhead in regular runs.

RECORE is set up to work on JAVA programs. In JAVA programs, the equivalent of abnormal termination is an *unchecked exception*—that is, conditions that cannot be reasonably recovered from, which generally cause abnormal termination, and which indicate bugs to be fixed. In analogy to the execution of directly executable binaries, one would normally set up the JAVA virtual machine to produce a core dump whenever an unchecked exception is raised—again only incurring overhead in case of fatal failures.[4]

Our current RECORE prototype does *not* extend the JAVA virtual machine; instead, it instruments the code such that a raised exception captures the heap, all local variables, and all method arguments at the time of failure (by employing std. JAVA functionality). To do this for every method on the call stack, RECORE wraps each method in a try-catch block, inducing a potential overhead on most JAVA virtual machines. This potential overhead, however, is only due to the current implementation; if RECORE were written for C programs, or if RECORE would plug directly into the JAVA virtual machine, any program would be executed without any change to execution time or memory—until a fatal failure occurs, which is when dumping the current state sets in.

V. RECONSTRUCTING FAILURES

A. Fitness Function

The key idea of RECORE is to set up a *fitness function* that checks similarity to guide the evolutionary algorithm. Generally speaking, the better the fitness of a particular test case, the likelier it is that it will influence the generation of future test cases. Hence, defining an appropriate fitness function is the central part of RECORE.

[4] Generally speaking, unchecked exceptions simply should not occur. Even if an unchecked exception would be caught and handled during execution, such that execution resumes, one would still be interested in reproducing the failure—and also in the resulting core dump.

1) Statement Distance: The first factor that determines the fitness of a test case is its ability to exactly reproduce the given *stack trace.* For this, we need a measure of how close we are in reaching a stack trace, for which we first need a measure of how close we are in reaching a specific location in the code.

For a given *test* and a location l in the code (in our case: class, method, and line), we assume a function $\mathit{StatementDistance}(\mathit{test}, l)$ whose value is higher the further away the *test* is from reaching l. A value of zero means zero distance (i.e. l was reached); a value of one means maximal distance.

StatementDistance can be based directly on a fitness function as used in search-based test generation [24]. In our case, *StatementDistance* is based on the unchanged fitness function from EvoSuite, the platform which RECORE uses to generate tests. The EvoSuite fitness function uses two measures as guidance:

- The **approach level** [32] determines how close the generated test is in reaching the target code (typically, yet uncovered code). It denotes the distance between the target point in the control dependence graph and the point at which execution diverged: The higher the approach level, the more control dependent branches between test and target.
- The **branch distance** [20] determines how close the branch predicate at the point of diversion is to evaluating to the desired value. The branch distance can be calculated using a simple set of rules [24]: For instance, given a branch condition `if (x < 15)`, the value `x = 20` implies a higher branch distance than `x = 16`.

These two measures are combined as follows:

$$\begin{aligned}&\mathit{StatementDistance}(\mathit{test}, l) = \\ &\mathit{approach\ level}(\mathit{test}, l) + \mathit{norm}\big(\mathit{branch\ distance}(\mathit{test}, l)\big)\end{aligned}$$

To normalize the branch distance in the range $[0, 1)$, we use $\mathit{norm}(x) = x/(x + 1)$.

2) Stack Trace Distance: Based on *StatementDistance*, we can now define stack trace distance. We denote a stack trace $S = \langle l_1, \ldots, l_n \rangle$ as a sequence of source code locations l_i, where l_1 is the outermost frame, l_n the innermost frame, and each frame l_i is invoked from the frame l_{i-1}.

Let $R = \langle r_1, \ldots, r_n \rangle$ be the reference stack trace from the core dump, and $S = \langle s_1, \ldots, s_m \rangle$ be the stack trace from the generated test. Let

$$\mathit{lcp} = \max\big\{j \cdot (\forall i \in \{1, \ldots, \min(j, n, m)\} \cdot r_i = s_i)\big\}$$

be the longest common prefix between R and S, starting from the outermost caller. We define the function $\mathit{StackTraceDistance}(R, S)$ as follows. If $\mathit{lcp} = |R| = |S|$ holds, then the stack traces are identical, and $\mathit{StackTraceDistance}(R, S) = 0$ holds. Otherwise, we define it to depend on the number of stack frames yet to reach (expressed by the value of $|R| - \mathit{lcp}$) and the distance between the test and the first diverging stack frame r_{lcp}:

$$\begin{aligned}&\mathit{StackTraceDistance}(R, S) = \\ &|R| - \mathit{lcp} - \big(1 - \mathit{norm}(\mathit{StatementDistance}(\mathit{test}, r_{\mathit{lcp}}))\big)\end{aligned}$$

By using *StackDistance* as guidance in the fitness function, test case generation will thus strive to maximize stack trace similarity, starting from the outermost frames.

3) Stack Dump Distance: The second factor that determines the fitness of a stack trace is its ability to reconstruct variable values as found in the core dump. In principle, we could guide RECORE to reconstruct *all* variables from the heap and from the stack. However, such an attempt is likely to be misleading. First, we would have to reconstruct thousands of individual values, which is unlikely to be feasible. Second, few of these values actually matter for reproducing and fixing the failure. We thus decided to focus on those values which we also require to reconstruct a given stack trace, namely *those values found in the stack.*

We assume a function *ObjectDistance(x, y)* $\in [0, 1)$ which compares two objects x and y and again returns a normalized value between zero and one. We obtain this value by relying on the EvoSuite object distance, which is based on the comparison of primitive values:

- If x and y contain **numbers**, the normalized absolute difference is added to the total difference.
- If x and y contain **strings**, the normalized Levenshtein distance is added to the total difference.
- If x and y contain **complex objects**, these are compared recursively, adding the result to the total difference.
- If either field value is **null**, 1 is added to the total difference.

The total difference is then divided by the number of fields.

Based on *ObjectDistance*, we can now define the *stack dump distance* across all objects as found on the stack. Let $\mathit{obj}(s)$ denote the set of object identifiers in local scope at a location s (i.e., the currently active object, all method parameters and all local variables). Let $s(o)$ denote the object identified by o. We then define the *stack dump distance* as the sum over all object distances as found on the common stack frames:

$$\mathit{StackDumpDistance}(R, S) = \sum_{i=0}^{\mathit{lcp}} \sum_{o \in \mathit{obj}(r_i)} \mathit{ObjectDistance}\big(r_i(o), s_i(o)\big)$$

Again, the larger the distance, the more object differences.

4) Total Fitness: During the execution of a generated test, we obtain several stack traces, each with its own stack dump distance. To measure the fitness of a test, we go for the *minimum of all traces obtained.* Let $S_1, \ldots, S_n$ denote the stack traces obtained during the execution of a test *test*. We can then define test *trace and dump distances* as follows:

$$\mathit{TestStackTraceDistance}(R, \mathit{test}) = \\ \min\big\{S \in \{S_1, \ldots S_n\} \cdot \mathit{StackTraceDistance}(R, S)\big\}$$

as well as

$$\mathit{TestStackDumpDistance}(R, \mathit{test}) = \\ \min\big\{S \in \{S_1, \ldots S_n\} \cdot \mathit{StackDumpDistance}(R, S)\big\}$$

Again, the higher the similarity, the lower the fitness value.

We give the highest priority to reconstruct the stack trace, whereas reconstructing the local variables is a lesser goal. Hence, the stack trace distance determines the overall fitness, whereas the stack dump distance is in $[0, 1)$. This is also helpful in guiding test generation while no progress is made reconstructing the stack.

Finally, we impose a penalty for runs were the target exception is not thrown and thus install some lock-in effect: once the exception was thrown by a test, other factors (i.e. stack dump distance) cannot guide test case generation away from the exception again.

$$\mathit{ExceptionPenalty}(\mathit{test}) = \begin{cases} 0 & \text{if same exception is thrown} \\ 1 & \text{otherwise} \end{cases}$$

The sum of these three gives the fitness function whose value RECORE strives to minimize:

$$\begin{aligned} \mathit{Fitness}(R, \mathit{test}) = {} & \mathit{TestStackTraceDistance}(R, \mathit{test}) \\ & + \mathit{TestStackDumpDistance}(R, \mathit{test}) \\ & + \mathit{ExceptionPenalty}(\mathit{test}) \end{aligned}$$

RECORE extends EVOSUITE with the fitness function above, thus guiding test case generation towards reconstructing the given core dump. The genetic algorithm stops once the best known fitness remains unchanged for 1,000 generations.

B. Seeding Primitive Values

A core dump is not only helpful for guiding the search. It also contains all the primitive values found on the heap at the point of failure. These values can be used to *seed* the search—that is, provide useful starting points rather than values taken at random.

In evolutionary search, *seeding* refers to any technique that exploits previous related knowledge to help solve the problem at hand. Although in general seeding should not be a requirement to solve the problem at hand, it can boost the search such that a solution is found with a limited search budget, where this would be impossible without. Seeding is also an important component of EVOSUITE [10]; for example, EVOSUITE re-uses primitive values and strings found in the bytecode.

RECORE sets up EVOSUITE seeding to take advantage of the core dump as follows:

1) **Seeding values.** To seed values, RECORE *only* uses values from the heap dump (as the dump also contains all constants from the source code).
   - With a probability of $p = \frac{1}{3}$, RECORE uses the *feedback* it receives from the distance function (Section V-A3): If a primitive P in the reference stack dump is different from the current best test case, then P is directly fed back to be used.
   - With $p = \frac{2}{3}$, RECORE uses *a value from the heap dump:* If the value is to be used in a method or constructor that occurs in the stack trace, then it reuses the parameters for that method or constructor from the stack dump. If the method or constructor stems from a class that is found somewhere on the stack dump, RECORE reuses the primitive values it finds as the field variables of that class. Otherwise, it uses an arbitrary value from the heap dump.
2) **Creating method calls.** When inserting or replacing method calls or constructors in a test case, EVOSUITE choses randomly from the set of known calls.
   - With $p = \frac{3}{7}$, RECORE makes EVOSUITE use one of the calls on the stack trace. This way, we give preference to the methods that need to be on the stack trace in the end.
   - With $p = \frac{2}{7}$, RECORE uses any method or constructor of the classes on the stack trace.
   - With $p = \frac{1}{7}$, RECORE uses the *feedback* it receives from the distance function (Section V-A3): If an object in the reference stack dump is different from the current best test case (i.e. different class or non-null), then a method or constructor from the class of that object is used.
   - Eventually, with $p = \frac{1}{7}$, an arbitrary method from any class (of the target project) that can be found on the heap is used.

Both the above measures dramatically speed up failure reconstruction; RECORE thus makes the same use of core dumps as a human debugger doing a post-mortem analysis.[5]

C. Limitations

As a test generator, RECORE suffers from the general limitations of such tools. To start with, there is no general constructive way of reaching a specific program state—this is an instance of the halting problem. Test generation tools such as RECORE will thus fail to reconstruct a failure if the search space is ill-formed or the conditions are very

[5] Rather than just seeding values, the core dump would of course also allow us to simply bypass most of test case generation by *taking objects from the core dump* and feeding these into the failing methods. As discussed in Section III-A, this can result in very precise reconstruction of the failure per se; however, the *history* of how the objects got into their final state is lost. As we are interested in reconstructing the failure history from the beginning, RECORE only uses primitive values from the core dump.

Table I
RECORE EVALUATION SUBJECTS

ID	Section	Subject	Lines of code
JOD1	VI-A	Brazilian Date bug	62,326
VM1	VI-B	Vending Machine bug	68
MAT1	VI-C	Sparse Iterator bug	53,496
JOD2	VI-D	Western Hemisphere bug	62,326
JOD3	VI-E	Parse French Date bug	53,845
COD1	VI-F	Base64 Decoder bug	8,147
COD2	VI-G	Base64 Lookup bug	6,154

complex. In a situation in which only few specific inputs lead to the desired state (such as passwords in cryptographic checks, for instance), RECORE is unlikely to construct helpful inputs.

These general concerns are offset by the fact that test case generation achieves high coverage in practical settings. In our specific setting of reconstructing core dumps, we even *know* that the specific state we search is reachable. In principle, we can therefore reconstruct any state simply by having RECORE search long enough—but this is only a theoretical option. Additionally, the core dump may contain precisely those specific inputs we are looking for—but they may just as well have been overwritten by later computations before the failure occurs.

For all these reasons, it is *unrealistic to assume that RECORE will always be successful.* It may be useful and effective in practice in a number of situations, though. Whether this is the case on real-life programs with real-life failures will be explored in the next section.

VI. Case Studies

To evaluate the potential of RECORE, we applied it to a set of seven bugs used previously for evaluating the BUGEX automated debugger [29], summarized in Table I. Each of these bugs is produced by a single test case which we set up such that a core dump would be produced upon failure (Section IV). We then fed RECORE with the original program as well as the core dump. What we wanted to know were the answers to three questions:

- **Can RECORE produce test cases that produce the exact same stack trace?** Note that this is one of RECORE's success criteria, so this question may also be posed as "Does RECORE produce a result?"
- **Does the generated test case pass after applying the original fix?** This question asks "Is the test case useful in debugging the original failure?" It should actually run the other way round: If we fix the error based on the produced test case, would we also fix the original failure? For the original failure, however, we have the single official fix, so we use this one as ground truth.
- **Are the test cases readable and easy to understand?** The answer to this question is best left to the reader. For this purpose, we provide a detailed analysis of each failure and test case; this is also the reason why we prefer seven in-depth case studies to statistical summaries over bug collections.

Table II provides the running time of RECORE for the seven subjects.[6] Note that the great variance of the runtime is due to the randomness of the underlying test case generation. In the remaining sections, we present the seven reconstructed failures.

A. JODATIME *Brazilian Date Bug*

Our first bug is the Brazilian Date Bug discussed in Section II, namely bug report 2487417 for JODATIME [17]. It takes RECORE 40 minutes to reconstruct this failure. The stack trace matches exactly, and the official fix also makes the test case pass. This is a poster example for the capabilities of RECORE—reconstructing the exact time and time zone in which the error occurs.

B. *Vending Machine Bug*

Vending Machine is a small artificial example used for earlier studies on automated debugging [4], [29]. A vending operation that can cause the credit to fall below zero is erroneously enabled, raising an exception when invoked. RECORE reproduces the precise failure:

```
VendingMachine vendingMachine0 =
   new VendingMachine();
String string0 = "SILVERDOLLAR";
Coin coin0 = Coin.create(string0);
vendingMachine0.insert(coin0);
vendingMachine0.vend();
vendingMachine0.vend();
```

Again, the stack trace matches exactly, and applying the official fix makes the test case pass.

C. *Commons Math Sparse Iterator Bug*

Apache Commons Math is a library of lightweight, self-contained mathematics and statistics components. Defect number 367 [23] is a `NullPointerException` raised by a *sparse iterator,* which should iterate over the non-zero values in a vector. The test case generated by RECORE reconstructs the failure:

Table II
SUMMARY OF RESULTS

ID	Section	Duration	Reproduces stack?	Passes after fix?
JOD1	VI-A	35 m ± 17 m	yes	yes
VM1	VI-B	9 m ± 2 m	yes	yes
MAT1	VI-C	17 m ± 18 m	yes	yes
JOD2	VI-D	22 m ± 14 m	**no**	**no**
JOD3	VI-E	18 m ± 12 m	yes	**no**
COD1	VI-F	149 m ± 108 m	yes	yes
COD2	VI-G	6 m ± 1 m	yes	yes

[6] All times were measured on a non-dedicated quad-core 2.67 GHz Intel x86 CPU with 8 GB RAM; RECORE and EVOSUITE are single-threaded.

```
ArrayRealVector arrayRealVector0 =
  new ArrayRealVector();
double double0 =
  -1832.3093176108437;
ArrayRealVector arrayRealVector1 =
  (ArrayRealVector)
  arrayRealVector0.append(double0);
ComposableFunction composableFunction0 =
  ComposableFunction.SIGNUM;
RealVector realVector0 =
  arrayRealVector1.map(composableFunction0);
```

Interestingly, this test case produces a stack trace that *extends* the original—that is, it has more callers on the outside. Rather than calling the failing iterator functions directly, RECORE reproduces the failure indirectly as it finds that the method `map()` by itself invokes the iterator and triggers the failure. Again, the generated test case passes as the official fix is applied.

D. JODATIME *Western Hemisphere Bug*

In JODATIME bug report 2889499 [18], selecting a time zone at the border of the Western hemisphere and calling `toDateTimeZone()` can cause an arithmetic exception after a long chain of subsequent time/date calculations. Due to the complex dependencies, RECORE fails to reach the statement at which the error was raised, and thus cannot reconstruct the issue.

With this test case, we have met the limitations discussed in Section V-C; a better test case generator may be able to address this issue. Note, though, that this failure comes at virtually no cost: The attempt to reproduce the error is fully automatic, and the production code has no runtime overhead.

E. JODATIME Parse French Date Bug

An `IllegalArgumentException` is raised when parsing a legal french date in the JODATIME bug report 1788282 [16]. The test generated by RECORE reproduces the exact stack trace:

```
DateTimeFormatter dateTimeFormatter0 =
   ISODateTimeFormat.time();
Locale locale0 = Locale.FRENCH;
DateTimeFormatter dateTimeFormatter1 =
   dateTimeFormatter0.withLocale(locale0);
String string0 = "11.sept..2007";
DateTime dateTime0 =
   dateTimeFormatter1.parseDateTime(string0);
```

With this, however, RECORE has not recreated the original failure. The input `"11.sept..2007"` is correctly reconstructed, and the stack trace is the same. However, the failure is different:

- In the original run, `"dd.MMM.yyyy"` is the date format. Here, `"MMM"` stands for the possibly abbreviated month, which in French can include a dot (i.e., the `"sept."` in `"11.sept..2007"`).
- The test case generated by RECORE uses an ISO date format, which is `"yyyy-MM-dd"`. The provided input `"11.sept..2007"` obviously does not match the ISO format; hence parsing fails.

The difference in failure manifests itself in a different state; the original run fails to parse the month, whereas the generated run fails at the first dot. This also results in different exception messages, which instructs the programmer to proceed with caution.

Why can't RECORE reproduce the original failure? The actual `DateTimeFormatter` object in the core dump takes a string in its constructor. To recreate the object, RECORE must pass the original string `"dd.MMM.yyyy"`. However, the fact that this string was used in the constructor is lost in history, and reverse-executing the parser to reconstruct it is beyond RECORE's capabilities (and actually, beyond the capabilities of any of today's test generators).

With this test case, we thus again have met the limitations discussed in Section V-C. It should be noted, though, that half of the input (namely, the french date) is correctly reconstructed, which may speed up manual debugging.

F. Commons Codec Base64 Decoder Bug

The *Apache Commons Codec library* provides implementations of common encoders and decoders such as Base64, Hex, Phonetic and URLs. Issue 98 [8] reports that "certain (malformed?) input to `Base64InputStream` causes a `NullPointerException` in `Base64.decode()`."

The test case generated by RECORE reconstructs this failure as follows:

```
String string1 = CharEncoding.US_ASCII;
byte[] byteArray0 =
   Base64.decodeBase64(string1);
ByteArrayInputStream byteArrayInputStream0 =
   new ByteArrayInputStream(byteArray0);
Base64InputStream base64InputStream0 =
   new Base64InputStream(
        byteArrayInputStream0);
byte[] byteArray1 =
   Base64TestData.streamToBytes(
        base64InputStream0, byteArray0);
```

The input in this test case (`US_ASCII`) differs from the test case which triggered the original failure; in particular, a character encoding (`"US-ASCII"`) would normally not be used as a string to encode. However, this generated input (8 characters) is much smaller than the reported input (1190 characters). The test case produces the exact same stack trace; and again, applying the original fix also makes this test case pass.

G. Commons Codec Base64 Lookup Bug

In issue number 22 in the Apache Commons Codec library [7], an `ArrayIndexOutOfBoundsException` is thrown. To reconstruct the failure, RECORE produces an array with one negative element as input, which is also the case in the original failure:

```
byte byte0 = -125;
byte[] byteArray0 = new byte[3];
byteArray0[0] = byte0;
byte byte1 = 64;
byteArray0[2] = byte1;
boolean boolean0 =
    Base64.isArrayByteBase64(byteArray0);
```

Again, the stack trace is identical and the original fix also fixes the generated test case.

H. Summary

Our results are summarized in Table II. In five out of seven issues, RECORE is able to recreate the precise failure: The generated tests reproduce the stack trace, and the original fix also fixes the test. Even when failure reconstruction is not complete (Section VI-E), the generated test still partially reconstructs the input.

All generated test cases reconstruct the failure from scratch, using primitive values only, and are all short enough to not only ease comprehension, but also debugging in itself. All this is achieved with zero overhead at runtime, and in an all-automatic run after the failure. If integrated in a bug reporting system, RECORE could try to construct a test case before any developer even looks at the report.

I. Threats to Validity

As any empirical study, our evaluation is subject to threats to validity.

Threats to *construct validity* have to do with how we measured the effectiveness of our technique. We assume a failure to be reconstructed if the stack trace is the same, and if the original fix makes the generated test case pass. For a full evaluation of effectiveness, we would have developers design fixes based on the RECORE results, and see how effective these fixes are in addressing the original issues. This is part of our future work (Section VII); for now, assessing the usefulness and readability of the generated tests is left to the reader.

Threats to *internal validity* might come from how the study was performed. We carefully tested RECORE to reduce the likelihood of defects. To counter the issue of randomized algorithms being affected by chance, we ran each experiment at least three times; all reported results are representative. The running time is the average over all runs.

Threats to *external validity* concern the generalization to software and faults other than the ones we studied, which is a common issue in empirical analysis. Our sample size is small; only seven different programs and bugs were used in the study. The reason for this is that it is time consuming to find and reproduce real bugs by manually analyzing bug reports. This produces a bias towards well documented and easy to reproduce issues; given the general limitations of test case generation (Section V-C), there will be a number of failures which RECORE will not be able to reproduce.

RECORE and its underlying EVOSUITE test case generator have several parameters such as weights, timeouts, and thresholds which all may influence the result. Wherever possible, we picked default values as used in other studies. Given the apparent effectiveness of search-based test generation in reconstructing failures, we believe that changing these parameters may affect the time it takes to search for these results, but not necessarily the result quality.

VII. CONCLUSION AND FUTURE WORK

Search-based test case generation is useful not only for exploring unknown behavior, but also for reproducing *known* behavior. Provided with only the data contained in a core dump, modern search-based frameworks can effectively reconstruct executions that reproduce the failure. Applied on seven JAVA bugs, RECORE was able to reconstruct the failure exactly in five cases, and partially in another case. The RECORE approach incurs zero overhead in production code; the resulting test cases capture the essence of the failure, are easy to understand, and easy to run through a debugger. RECORE could even be deployed at user's sites, limiting the information sent to developers to the exact amount required to reproduce the failure.

Despite these advances, there is still much to do. Besides general improvements such as scalability, stability, and speed, our future work will focus on the following topics:

- **User study.** To fully evaluate the effectiveness of RECORE, we are currently setting up a large-scale study with hundreds of users [28]. This study will tell how useful RECORE is in reproducing, and how well it integrates with other debugging tools.
- **System test generation.** A new generation of test generators [14] applies search-based testing at the system level, synthesizing GUI events as inputs. Such generation techniques could also be integrated into RECORE, resulting in real inputs leading to real failures.
- **Capture more events.** As illustrated in Section VI-E, the lack of history can effectively prevent failure reconstruction. We are following the BUGREDUX way by investigating which events can be captured during execution to maximize the effectiveness of reconstructing failures, while minimizing the overhead on executions.
- **More language features.** We plan to extend RECORE such that it supports features of C/C++ programs such as indirect calls through function pointers or corrupt memory. We are adapting techniques from execution synthesis [35] to control and reconstruct failure-inducing thread schedules.
- **Symbolic execution.** BUGREDUX and execution synthesis use symbolic execution and constraint solving to generate runs that recreate the failure. We are working on integrating search-based test generation (as in RECORE) with dynamic symbolic execution to combine the best of both worlds.

RECORE is part of the BUGEX framework. For details on RECORE, see

http://www.st.cs.uni-saarland.de/bugex/

Acknowledgments. Juan Pablo Galeotti, Alessandra Gorla, Kim Herzig, Kevin Streit, and the anonymous reviewers gave helpful comments on earlier revisions of this paper. This work was supported by DFG grant Ze509/4-1.

References

[1] G. Altekar, C. Zamfir, G. Candea, and I. Stoica. Automating the debugging of datacenter applications with ADDA. Technical Report UCB/EECS-2011-22, EECS Department, University of California, Berkeley, Apr 2011.

[2] S. Artzi, S. Kim, and M. D. Ernst. ReCrash: Making software failures reproducible by preserving object states. In *ECOOP '08*, pages 542–565, Berlin, Heidelberg, 2008. Springer-Verlag.

[3] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: A compiler and runtime system for deterministic multithreaded execution. In *ASPLOS*, 2010.

[4] M. Burger and A. Zeller. Minimizing reproduction of software failures. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 221–231, New York, NY, USA, July 2011. ACM.

[5] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symp. on Operating Systems Design and Implementation*, 2008.

[6] J. Clause and A. Orso. A technique for enabling and supporting debugging of field failures. In *ICSE '07*, pages 261–270, Washington, DC, USA, 2007. IEEE Computer Society.

[7] Codec-22. https://issues.apache.org/jira/browse/CODEC-22.

[8] Codec-98. https://issues.apache.org/jira/browse/CODEC-98.

[9] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Symp. on Operating Systems Design and Implementation*, 2002.

[10] G. Fraser and A. Arcuri. The seed is strong: Seeding strategies in search-based software testing. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST 2012)*, pages 121–130, 2012.

[11] G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 2012. To appear.

[12] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: ten years of implementation and experience. In *Symp. on Operating Systems Principles*, 2009.

[13] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI 2005)*, pages 213–223, June 2005.

[14] F. Gross, G. Fraser, and A. Zeller. Search-based system testing: High coverage, no false alarms. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 67–77, New York, NY, USA, 2012. ACM.

[15] W. Jin and A. Orso. BugRedux: Reproducing field failures for in-house debugging. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 474–484, Piscataway, NJ, USA, 2012. IEEE Press.

[16] JodaTime-1788282. http://sourceforge.net/tracker/?func=detail&aid=1788282&group_id=97367&atid=617889.

[17] JodaTime-2487417. http://sourceforge.net/tracker/?func=detail&aid=2487417&group_id=97367&atid=617889.

[18] JodaTime-2889499. http://sourceforge.net/tracker/?func=detail&aid=2889499&group_id=97367&atid=617889.

[19] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 1976.

[20] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, pages 870–879, 1990.

[21] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Online multiprocessor replay via speculation and external determinism. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2010.

[22] J. Malburg and G. Fraser. Combining search-based and constraint-based testing. In *ASE*, 2011.

[23] Math-367. https://issues.apache.org/jira/browse/MATH-367.

[24] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.

[25] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: A software-hardware interface for practical deterministic multiprocessor replay. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2009.

[26] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 75–84, 2007.

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

[28] J. Rößler. How helpful are automated debugging tools? In *USER '12: Proceedings of the first Workshop on User evaluation for Software Engineering Researchers*, June 2012.

[29] J. Rößler, G. Fraser, A. Zeller, and A. Orso. Isolating failure causes through test case generation. In *International Symposium on Software Testing and Analysis*, Jul 2012.

[30] N. Tillmann and N. J. de Halleux. Pex — white box test generation for .NET. In *International Conference on Tests And Proofs (TAP)*, pages 134–253, 2008.

[31] D. Weeratunge, X. Zhang, and S. Jagannathan. Analyzing multicore dumps to facilitate concurrency bug reproduction. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2010.

[32] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, 2001.

[33] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. SherLog: Error diagnosis by connecting clues from runtime logs. In *Proceedings of the International Conference on Architecture Support for Programming Languages and Operating Systems*, March 2010.

[34] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. Improving software diagnosability via log enhancement. In *Proceedings of the International Conference on Architecture Support for Programming Languages and Operating Systems*, March 2011.

[35] C. Zamfir and G. Candea. Execution synthesis: A technique for automated debugging. In *ACM SIGOPS/EuroSys European Conf. on Computer Systems*, 2010.