Fast Black-Box Testing of System Recovery Code

Radu Banabic George Candea
School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
{radu.banabic,george.candea}@epfl.ch

Abstract

Fault injection—a key technique for testing the robustness of software systems—ends up rarely being used in practice, because it is labor-intensive and one needs to choose between performing random injections (which leads to poor coverage and low representativeness) or systematic testing (which takes a long time to wade through large fault spaces). As a result, testers of systems with high reliability requirements, such as MySQL, perform fault injection in an ad-hoc manner, using explicitly-coded injection statements in the base source code and manual triggering of failures.

This paper introduces AFEX, a technique and tool for automating the entire fault injection process, from choosing the faults to inject, to setting up the environment, performing the injections, and finally characterizing the results of the tests (e.g., in terms of impact, coverage, and redundancy). The AFEX approach uses a metric-driven search algorithm that aims to maximize the number of bugs discovered in a fixed amount of time. We applied AFEX to real-world systems—MySQL, Apache httpd, UNIX utilities, and MongoDB—and it uncovered new bugs automatically in considerably less time than other black-box approaches.

1. Introduction

Fault injection is a form of testing that consists of introducing faults in a system under test, with the goal of exercising the system's error-handling code paths. Fault injection is crucial to system testing, especially as increasingly more general-purpose systems (e.g., databases, backup software, Web servers) are used in business-critical settings.

These systems are often "black boxes," in that the tester has no (or little) knowledge of the internals, either because the software is closed source (e.g., commercial application servers) or because the system is just too complicated to understand (e.g., MySQL has over 1 million lines of code). Even when the internals are reasonably well understood, every major upgrade renders parts of this knowledge obsolete.

Engineers who employ fault injection to test these systems often do so in an ad-hoc manner, such as pulling out network cables or unplugging hard drives. These actions are typically dictated by the intersection of what is believed might happen in production and what testers can actually control or directly simulate. Such ad-hoc approaches are typically based on a poor understanding of the space of possible faults and lead to poor-coverage testing. Recent tools, like LFI [16], improve the state of the art by offering the ability to simulate a wide variety of fine-grained faults that occur in a program's environment and become visible to applications through the application–library interface. Such tools offer developers better control over fault injection.

However, with fine-grain control comes the necessity to make hard choices, because these tools offer developers a vast universe of possible fault scenarios. There exist three degrees of freedom: *what* fault to inject (e.g., `read()` call fails with `EINTR`), *where* to inject it (e.g., in the logging subsystem), and *when* to do so (e.g., while the DBMS is flushing the log to disk) [17]. Poor choices can cause test suites to miss important aspects of the tested system's behavior. Ideally, a tool could automatically identify the "interesting" faults and test the system; the human would then only need to describe the fault space and provide one or more generic fault injectors like LFI. Such an automated tool would then find a small set of high-impact fault injection tests that are good at breaking or validating the target system.

One way to avoid making the hard choices, yet still have automated testing, is to perform brute-force exhaustive testing, such as using large clusters to try out all combinations of faults that the available injectors can simulate—this approach will certainly find the faults that cause most damage. Alas, the universe of possible faults is typically overwhelming: even in our small scale evaluation on MySQL, the fault space consisted of more than 2 million possibilities for injecting a single fault. Exploring such fault spaces exhaustively requires many CPU-years, followed by substantial amounts of human labor to sift through the results. A commonly employed alternative is to not inject all faults,

ACM EuroSys Conference on Computer Systems, April 2012, Bern, Switzerland.
Copyright © 2012 ACM 978-1-4503-1223-3/12/04. . . $10.00

but only a randomly selected subset: the fault space can be uniformly sampled, and tests can be stopped whenever time runs out. However, random injection achieves poor coverage, and the chances of finding the high impact faults is low.

We propose an approach in which still a subset of the fault space is sampled, but this sampling is guided. We observe that the universe of faults often has an inherent *structure*, which both random and exhaustive testing are oblivious to, and that exploiting this structure can improve the efficiency of finding high impact faults. Since this structure is hard to identify and specify a priori, it needs to be *inferred*.

We describe AFEX, a cluster-based parallel testing system that uses a fitness-guided feedback-based algorithm to search for high-impact faults in spaces with unknown structure. It uses the effect of previously injected faults to dynamically learn the space's structure and choose new faults for subsequent tests. The search process continues until a specific target is reached, such as a given level of code coverage, a threshold on the faults' impact level (e.g., "find 3 disk faults that hang the DBMS"), or a time limit. Upon completion, AFEX analyzes the injected faults for redundancy, clusters and categorizes them, and then ranks them by severity to make it easier for developers to analyze. AFEX is also capable of leveraging domain-specific knowledge, whenever humans have such knowledge and can suitably encode it; in §7 we show how such knowledge can be provided to AFEX and the influence it has on the speed and quality of the search.

Our paper advances the state of the art in two ways: an adaptive algorithm for *finding high value faults*, and a technique for automatic *categorization and ranking* of faults in a set based on their degree of redundancy and reproducibility in testing. We embody these ideas in an extensible system that can automatically test real-world software with fault injection scenarios of arbitrary complexity.

The rest of the paper presents background and definitions (§2), the fault exploration algorithm (§3), trade-offs developers can make when using AFEX (§4), techniques for measuring result quality (§5), the AFEX prototype (§6), and an evaluation of its scalability and effectiveness on MySQL, Apache httpd, UNIX utilities, and MongoDB (§7). We close with related work (§8) and conclusions (§9).

2. Definitions

We start by defining three concepts that are central to this paper: fault space, fault impact, and fault space structure.

Fault Space A fault space is a concise description of the failures that a fault injector can simulate in a target system's environment. A fault injection tool T defines implicitly a fault space by virtue of the possible values its parameters can take. For example, a library-level fault injector can inject a variety of faults at the application–library interface—the library call in which to inject, the error code to inject, and the call number at which to inject represent three axes describing the universe of library-level faults injectable by T. We think of a fault space as a hyperspace, in which a point represents a fault defined by a combination of parameters that, when passed to tool T, will cause that fault to be simulated in the target system's environment. This hyperspace may have holes, corresponding to invalid combinations of parameters to T.

We define the attributes of a fault ϕ to be the parameters to tool T that cause ϕ to be injected. If Φ is the space of possible faults, then fault $\phi \in \Phi$ is a vector of attributes $< \alpha_1, ..., \alpha_N >$ where α_i is the value of the fault's i-th attribute. For example, in the universe of failed calls made by a program to POSIX library functions, the return of error code -1 by the 5-th call to `close` would be represented as $\phi =< \mathsf{close}, 5, -1 >$. The values of fault attributes are taken from the finite sets $A_1, ..., A_N$, meaning that, for any fault $\phi =< \alpha_1, ..., \alpha_N > \in \Phi$, the attribute value $\alpha_i \in A_i$.

In order to lay out the values contained in each A_i along an axis, we assume the existence of a total order $\prec_i$ on each set A_i, so that we can refer to attribute values by their index in the corresponding order. In the case of $\phi =< \mathsf{close}, 5, -1 >$, we can assume A_1 is ordered as $(\mathsf{open}, \mathsf{close}, \mathsf{read}, \mathsf{write}, ...)$, A_2 as $(1, 2, 3, ...)$, and A_3 as $(-1, 0, ...)$. If there is no intrinsic total order, then we can pick a convenient one (e.g., group POSIX functions by functionality: file, networking, memory, etc.), or simply choose an arbitrary one.

We now define a *fault space* Φ to be spanned by axes $X_1, X_2, ...X_N$, meaning $\Phi = X_1 \times X_2 \times .. \times X_N$, where each axis X_i is a totally ordered set with elements from A_i and order $\prec_i$. A fault space represents all possible combinations of values from sets $A_1, ..., A_N$, along with the total order on each such set. Using the example shown above, the space of failed calls to POSIX functions is spanned by three axes, one for call types $X_1 : (\mathsf{open} \prec \mathsf{close} \prec ...)$, one for the index of the call made by the caller program to the failed function $X_2 : (1 \prec 2 \prec ...)$, and one for the return value of the POSIX function $X_3 : (-1 \prec 0 \prec ...)$. This enables us to represent fault $\phi =< \mathsf{close}, 5, -1 >$ as $\phi =< 2, 5, 1 >$, because the index of close on axis X_1 under order $\prec_1$ is 2, the index of 5 on X_2 under $\prec_2$ is 5, and the index of -1 on X_3 under $\prec_3$ is 1. A fault space can have holes corresponding to invalid faults, such as close returning 1.

Impact Metric Our proposed approach uses the effect of past injected faults to guide the search for new, higher impact faults. Therefore, we need an *impact metric* that quantifies the change effected by an injected fault in the target system's behavior (e.g., the change in number of requests per second served by Apache when random TCP packets are dropped). Conceptually, an impact metric is a function $I_S : \Phi \rightarrow \mathbb{R}$ that maps a fault ϕ in the fault space Φ to a measure of the impact that fault ϕ has on system S. Since Φ is a hyperspace, I_S defines a hypersurface. The (adversarial) goal of a tester employing fault injection is to find peaks on this hypersurface, i.e., find those faults that have the largest impact on the behavior of the system under test. Since testing is often a race against time, the tester needs to find as many such peaks as possible in the allotted time budget.

Fault Space Structure We observe empirically that, often, there are some patterns in the distribution of fault impact over the fault space, as suggested by Fig. 1, which shows the impact of library-level faults on the `ls` utility in UNIX. The structure that emerges from these patterns is caused by structure and modularity in the underlying code of the system being tested. Engler et al. [9] made a similar observation of bug patterns, which emerge due to implementation-based correlations between bodies of code. When viewing the fault space as a search space, this structure can help make the search more efficient than random sampling; in the next section we draw an analogy between fault space exploration and the Battleship game.

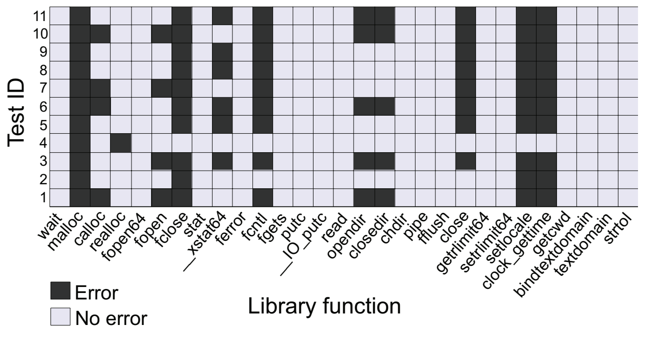

Figure 1. Part of the fault space created by LFI [16] for the `ls` utility. The horizontal axis represents functions in the C standard library that fail, and the vertical axis represents the tests in the default test suite for this utility. A point (x,y) in the plot is black if failing the first call to function x while running test y leads to a test failure, and is gray otherwise.

To characterize the structure of a fault space, we use a *relative linear density* metric ρ as follows: given a fault $\phi =< \alpha_1^0, ..., \alpha_k^0, ..., \alpha_N^0 >$, the relative linear density at ϕ along an axis X_k is the average impact of faults $\phi' =< \alpha_1^0, ..., \alpha_k, ..., \alpha_N^0 >$ with the same attributes as ϕ except along axis X_k, scaled by the average impact of all faults in the space. Specifically, $\rho_\phi^k = \frac{avg[\ I_S(<\alpha_1^0,...,\alpha_k,...,\alpha_N^0>),\ \alpha_k \in X_k\ ]}{avg[\ I_S(\phi_x),\ \phi_x \in \Phi\ ]}$. If $\rho_\phi^k > 1$, walking from ϕ along the X_k axis will encounter more high-impact faults than along a random direction. In practice, it is advantageous to compute ρ_ϕ over only a small vicinity of ϕ, instead of the entire fault space. This vicinity is a subspace containing all faults within distance D of ϕ, i.e., all faults ϕ'' s.t. $\delta(\phi, \phi'') \leq D$. Distance $\delta : \Phi \times \Phi \rightarrow \mathbb{N}$ is a Manhattan (or city-block) distance [6], i.e., the shortest distance between fault ϕ and ϕ'' when traveling along Φ's coordinate axes. $\delta(\phi, \phi'')$ gives the smallest number of increments/decrements of attribute indices that would turn ϕ into ϕ''. Thus, the D-vicinity of ϕ consists of all faults that can be obtained from ϕ with no more than D increments/decrements of ϕ's attributes $\alpha_j, 1 \leq j \leq N$.

To illustrate, consider fault $\phi =<$ fclose$, 7 >$ in Fig. 1, and its 4-vicinity (the faults within a distance $\delta \leq 4$). If the impact corresponding to a black square is 1, and to a gray one is 0, then the relative linear density at ϕ along the vertical axis is $\rho_\phi^2 = 2.27$. This means that walking in the vertical direction is more likely to encounter faults that cause test errors than walking in the horizontal direction or diagonally. In other words, exploration along the vertical axis is expected to be more rewarding than random exploration.

3. Fault Exploration

Fault exploration "navigates" a fault space in search of faults that have high impact on the system being tested. Visiting a point in the fault space incurs a certain cost corresponding to the generation of a test, its execution, and the subsequent measurement of the impact. Thus, ideally, one would aim to visit as few points as possible before finding a desired fault.

Exhaustive exploration (as used for instance by Gunawi et al. [11]) iterates through every point in the fault space by generating all combinations of attribute values, and then evaluates the impact of the corresponding faults. This method is complete, but inefficient and, thus, prohibitively slow for large fault spaces. Alternatively, random exploration [1] constructs random combinations of attribute values and evaluates the corresponding points in the fault space. When the fault space has no structure, random sampling of the space is no less efficient than any other form of sampling. However, if structure is present (i.e., the relative linear density is non-uniformly distributed over the fault space), then there exist more efficient search algorithms.

Fitness-guided Exploration We propose a fitness-guided algorithm for searching the fault space. This algorithm uses the impact metric I_S as a measure of a fault's "fitness," in essence steering the search process toward finding "ridges" on the hypersurface defined by (Φ, I_S) and then following these ridges to the peaks representing high-impact faults. Exploring a structured fault space like the one in Fig. 1 is analogous to playing Battleship, a board game involving two players, each with a grid. Before play begins, each player arranges a number of ships secretly on his/her own grid. After the ships have been positioned, one player announces a target square in the opponent's grid that is to be shot at; if a ship occupies that square, then it takes a hit. Players take turns and, when all of a ship's squares have been hit, the ship is sunk. A typical strategy is to start by shooting randomly until a target is hit, and then fire in the neighborhood of that target to guess the orientation of the ship and sink it.

Similarly, the AFEX algorithm exploits structure to avoid sampling faults that are unlikely to be interesting, and instead focus on those that are near other faults with high impact. For example, if a developer has a poor understanding of how some I/O library works, then there is perhaps a higher likelihood for bugs to lurk in the code that calls that library than in code that does not. Once AFEX finds a call to the

I/O library whose failure causes, say, data corruption, it will eventually focus on failing other calls to that same library, persisting for as long as it improves the achieved fault impact. Note that the AFEX algorithm focuses on related but distinct bugs, rather than on multiple instances of the same bug. If the fault space definition contains duplicates (i.e., different points in the fault space expose the same bug), then AFEX's clustering discovers and avoids the duplicates (§5).

Another perspective on AFEX is that impact-guided exploration merely generates a priority in which tests should be executed—if each fault in Φ corresponds to a test, the question is which tests to run first. AFEX both generates the tests and executes them; to construct new tests, it mutates previous high-impact tests in ways that AFEX believes will further increase their impact. It also avoids re-executing any tests that have already been executed in the past. Since it does not discard any tests, rather only prioritizes their execution, AFEX's coverage of the fault space increases proportionally to the allocated time budget.

Our proposed exploration algorithm is similar to the Battleship strategy mentioned earlier, except it is fully automated, so there are no human decisions or actions in the critical path. AFEX consists of the following steps:

1. Generate an initial batch of tests randomly, execute the tests, and evaluate their impact
2. Choose a previously executed high-impact test ϕ
3. Modify one of ϕ's attributes (injection parameters) to obtain a new test ϕ'
4. Execute ϕ' and evaluate its impact
5. Repeat step 2

In order to decide which test attribute to mutate, we could rely on the linear density metric to suggest the highest-reward axis. However, given that the fault space is not known a priori, we instead compute dynamically a sensitivity (described later) based on the historical benefit of choosing one dimension vs. another. This sensitivity calculation steers the search to align with the fault space structure observed thus far, in much the same way a Battleship player infers the orientation of her opponent's battleships.

When deciding by how much to mutate an attribute (i.e., the magnitude of the increment), we choose a value according to a Gaussian distribution, rather than a static value, in order to keep the algorithm robust. This distribution favors ϕ's closest neighbors without completely dismissing points that are further away. By changing the standard deviation of the Gaussian distribution, we can control the amount of bias the algorithm has in favor of nearby points.

Finally, we use an "aging" mechanism among the previously executed tests: the fitness of a test is initially equal to its impact, but then decreases over time. Once the fitness of old tests drops below a threshold, they are retired and can never have offspring. The purpose of this aging mechanism is to encourage improvements in test coverage concomitantly with improvements in impact—without aging, the exploration algorithm may get stuck in a high-impact vicinity, despite not finding any new high-impact faults. In the extreme, discovering a massive-impact "outlier" fault with no serious faults in its vicinity would cause an AFEX with no aging to waste time exploring exhaustively that vicinity.

```
Data: $Q_{priority}$: priority queue of high-fitness fault
                  injection tests (already executed)
      $Q_{pending}$: queue of tests awaiting execution
      $History$: set of all previously executed tests
      $Sensitivity$: vector of $N$ sensitivity values, one for
                  each test attribute $\alpha_1, ... \alpha_N$
1  foreach fault injection test $\phi_x \in Q_{priority}$ do
2      $testProbs[\phi_x] := assignProbability(\phi_x.fitness)$
3  end
4  $\phi := sample(Q_{priority}, testProbs)$
5  $attributeProbs := normalize(Sensitivity)$
6  $\alpha_i := sample(\{\alpha_1, ..., \alpha_N\}, attributeProbs)$
7  $oldValue := \phi.\alpha_i$          // remember that oldValue $\in A_i$
8  $\sigma := chooseStdDev(\phi, A_i)$
9  $newValue := sample(A_i, Gaussian(oldValue, \sigma))$
10 $\phi' := clone(\phi)$
11 $\phi'.\alpha_i := newValue$
12 if $\phi' \notin History \wedge \phi' \notin Q_{priority}$ then
13     $Q_{pending} := Q_{pending} \cup \phi'$
14 end
```

Algorithm 1: Fitness-guided generation of the next test. Execution of tests, computation of fitness and sensitivity, and aging occur outside this algorithm.

Algorithm 1 embodies steps 2–3 of the AFEX algorithm. We now describe Algorithm 1 in more detail.

Choosing Which Test to Mutate The algorithm uses three queues: a priority queue $Q_{priority}$ of already executed high-impact tests, a queue $Q_{pending}$ of test cases that have been generated but not yet executed, and a set *History* containing all previously executed tests. Once a test in $Q_{pending}$ is executed and its impact is evaluated, it gets moved to $Q_{priority}$. $Q_{priority}$ has a limited size; whenever the limit is reached, a test case is dropped from the queue, sampled with a probability inversely proportional to its fitness (tests with low fitness have a higher probability of being dropped). As a result, the average fitness of tests in $Q_{priority}$ increases over time. When old test cases are retired from $Q_{priority}$, they go into *History*. This history set helps AFEX avoid redundant re-execution of already evaluated tests.

On lines 1–4, AFEX picks a parent test ϕ from $Q_{priority}$, to mutate into offspring ϕ'. Instead of always picking the highest fitness test, AFEX samples $Q_{priority}$ with a probability proportional to fitness—highest fitness tests are favored, but others still have a non-zero chance to be picked.

Mutating the Test The new test ϕ' is obtained from parent test ϕ by modifying one of ϕ's attributes.

On lines 5–6, we choose the fault/test attribute α_i with a probability proportional to axis X_i's normalized sensitivity. We use *sensitivity* to capture the history of fitness gain: the sensitivity of each axis X_i of the fault space reflects the historical benefit of modifying attribute α_i when generating a new test. This sensitivity is directly related to relative linear density (from §2): the inherent structure of the system under test makes mutations along one axis to be more likely to produce high-impact faults than along others. In other words, if there is structure in the fault space, the sensitivity biases future mutations to occur along high-density axes. Given a value n, the sensitivity of X_i is computed by summing the fitness value of the previous n test cases in which attribute α_i was mutated. This sum helps detect "impact ridges" present in the currently sampled vicinity: if X_i's density is high, then we expect this sum of previous fitness values—the sensitivity to mutations along X_i—to be high as well, otherwise not. Our use of sensitivity is similar to the fitness-gain computation in Fitnex [27], since it essentially corresponds to betting on choices that have proven to be good in the past.

Sensitivity guides the choice of *which* fault attribute to mutate; next we describe *how* to mutate the chosen attribute in order to obtain a new fault injection test.

On lines 7–9, we use a discrete approximation of a Gaussian probability distribution to choose a new value for the test attribute to be mutated. This distribution is centered at *oldValue* and has standard deviation σ. The chosen standard deviation is proportional to the number of values the α_i attribute can take, i.e., to the cardinality of set A_i. For the evaluation in this paper, we chose $\sigma = \frac{1}{5} \cdot |A_i|$. σ can also be computed dynamically, based on the evolution of tests in the currently explored vicinity within the fault space—we leave the pursuit of this alternative to future work.

Our use of a Gaussian distribution implicitly assumes that there is some similarity between neighboring values of a test attribute. This similarity of course depends on the meaning of attributes (i.e., parameters to a fault injector) and on the way the human tester describes them in the fault space. In our experience, many parameters to fault injectors do have such similarity and, by using the Gaussian distribution, we can make use of this particularity to further improve on the naive method of randomly chosing a new attribute value. Revisiting the example from §2, it is not surprising that there is correlation between library functions (e.g., close is related to open), call numbers (e.g., successive calls from a program to a given function are likely to do similar things), or even tests from a suite (e.g., they are often grouped by functionality). Profiling tools, like LibTrac [5], can be used to discover such correlation when defining the fault space.

Finally, Algorithm 1 produces ϕ' by cloning ϕ and replacing attribute α_i with the new value (lines 10–11). If ϕ' has not been executed before, it goes on $Q_{pending}$ (lines 12–14).

Alternative Algorithms The goal of building AFEX was to have an automated general-purpose fault exploration system that is not tied to any particular fault injection tool. AFEX is tool-independent—we evaluate it using library-level fault injection tools, but believe it to be equally suitable to other kinds of fault injection, such as flipping bits in data structures [26] or injecting human errors [13].

In an earlier version of our system, we employed a genetic algorithm [24], but abandoned it, because we found it inefficient. AFEX aims to optimize for "ridges" on the fault-impact hypersurface, and this makes global optimization algorithms (such as genetic algorithms) difficult to apply. The algorithm we present here is, in essence, a variation of stochastic beam search [24]—parallel hill-climbing with a common pool of candidate states—enhanced with sensitivity analysis and Gaussian value selection.

4. Developer Trade-Offs in AFEX

Leveraging Domain Knowledge By default, AFEX works in pure black-box mode, in that it has no a priori knowledge of the specifics of the system under test or its environment. This makes AFEX a good fit for generic testing, such as that done in a certification service [8].

However, developers often have significant amounts of domain knowledge about the system or its environment, and this could enable them to reduce the fault space explored by AFEX, thus speeding up exploration. For example, when using a library-level fault injector and an application known to use only blocking I/O with no timeouts, it makes sense to exclude EAGAIN from the set of possible values of the errno attribute of faults injected in read. AFEX can accept domain knowledge about the system under test, the fault space, and/or the tested system's environment; we evaluate in §7.5 the benefit of doing so.

AFEX can also benefit from static analysis tools, which provide a complementary method for detecting vulnerable injection points (LFI's callsite analyzer [17] is such a tool). For example, AFEX can use the results of the static analysis in the initial generation phase of test candidates. By starting off with highly relevant tests from the beginning, AFEX can quickly learn the structure of the fault space, which is likely to boost its efficiency. This increase in efficiency can manifest as finding high impact faults sooner, as well as finding additional faults that were not suggested by the static analysis. For example, we show in §7.1 how AFEX finds bugs in the Apache HTTP server and MySQL, both of which have already been analyzed by the Reasoning service [2].

Injection Point Precision An injection point is the location in the execution of a program where a fault is to be injected. Even though AFEX has no knowledge of how injection points are defined, this does affect its accuracy and speed.

In our evaluation we use LFI [16], a library fault injection tool that allows the developer to fine-tune the definition of an injection point according to their own needs. Since we aim to maximize the accuracy of our evaluation, we define an injection point as the tuple ⟨ *testID, functionName, callNumber* ⟩. *testID* identifies a test from the test suite of the target system (in essence specifying one execution path, modulo non-determinism), *functionName* identifies the called library function in which to inject an error, and *callNumber* identifies the cardinality of the call to that library function that should fail. This level of precision ensures that injection points are unique on each tested execution path, and all possible library-level faults can be explored. However, this produces a large fault space, and introduces the possibility of test redundancy (we will show how AFEX explores this space efficiently in §7.1 and mitigates redundancy in §7.4).

A simple way to define an injection point is via the callsite, i.e., file and line number where the program is to encounter a fault (e.g., failed call to a library or system call, memory operation). Since the same callsite may be reached on different paths by the system under test, this definition is relatively broad. A more accurate definition is the "failure ID" described by Gunawi et al. [11], which associates a stack trace and domain-specific data (e.g., function arguments or state of system under test) to the definition of the injection point. This definition is more tightly related to an individual execution path and offers more precision. However, unlike the 3-tuple definition we use in our evaluation, failure IDs ignore loops and can lead to missed bugs.

The trade-off in choosing injection points is one between precision of testing and size of the resulting fault space: on the one hand, a fine-grain definition requires many injection parameters (fault attributes) and leads to a large fault space (many possible combinations of attributes), which takes longer to explore. On the other hand, more general injection points reduce the fault space, but may miss important fault scenarios (i.e., incur false negatives).

5. Quantifying Result Quality

In addition to automatically finding high-impact faults, AFEX also quantifies the level of confidence users can have in its results. We consider three aspects of interest to practitioners: cutting through redundant tests (i.e., identifying equivalence classes of redundant faults within the result set), assessing the precision of our impact assessment, and identifying which faults are representative and practically relevant.

Redundancy Clusters One measure of result quality is whether the different faults generated by AFEX exercise diverse system behaviors—if two faults exercise the same code path in the target system, it is sufficient to test with only one of the faults. Being a black-box testing system, AFEX cannot rely on source code to identify redundant faults.

AFEX computes clusters (equivalence classes) of closely related faults as follows: While executing a test that injects fault ϕ, AFEX captures the stack trace corresponding to ϕ's injection point. Subsequently, it compares the stack traces of all injected faults by computing the edit distance between every pair of stack traces (specifically, we use the Levenshtein distance [14]). Any two faults for which the distance is below a threshold end up in the same cluster. In §7.4 we evaluate the efficiency of this technique in avoiding test redundancy. In choosing this approach, we were inspired by the work of Liblit and Aiken on identifying which parts of a program led to a particular bug manifestation [15].

Besides helping developers to analyze fault exploration results, redundancy clusters are also used online by AFEX itself, in a feedback loop, to steer fault exploration away from test scenarios that trigger manifestations of the same underlying bug. This improves the efficiency of exploration.

Impact Precision Impact precision indicates, for a given fault ϕ, how likely it is to consistently have the same impact on the system under test S. To obtain it, AFEX runs the same test n times (with n configured by the developer) and computes the variance $Var(I_S(\phi))$ of ϕ's impact across the n trials. The impact precision is $\frac{1}{Var(I_S(\phi))}$, and AFEX reports it with each fault in the result set. The higher the precision, the more likely it is that re-injecting ϕ will result in the same impact that AFEX measured. In other words, a high value for impact precision suggest that the system's response to that fault in that environment is likely to be deterministic. Developers may find it easier, for example, to focus on debugging high-precision (thus reproducible) failure scenarios.

Practical Relevance The final quality metric employed by AFEX is a measure of each fault's representativeness and, thus, practical relevance. Using published studies [4, 25] or proprietary studies of the particular environments where a system will be deployed, developers can associate with each class of faults a probability of it occurring in practice. Using such statistical models of faults, AFEX can automatically associate with each generated fault a probability of it occurring in the target environment. This enables developers to better choose which failure scenarios to debug first.

6. AFEX Prototype

We built a prototype that embodies the techniques presented in this paper. The user provides AFEX with a description of the explorable fault space Φ, dictated by the available fault injectors, along with scripts that start/stop/measure the system under test S. AFEX automatically generates tests that inject faults from Φ and evaluates their quality. Our prototype is a parallel system that runs on clusters of computers, thus taking advantage of the parallelism inherent in AFEX.

The core of AFEX consists of an explorer and a set of node managers, as shown in Fig. 2. The explorer receives as input a fault space description and an exploration target (e.g., find the top-10 highest impact faults), and produces a set of fault injection tests that satisfy this target. AFEX synthesizes configuration files for each injection tool and instructs the various node managers to proceed with the injection of the corresponding faults. The managers then report to the explorer the results of the injections, and, based on this, the explorer decides which faults from Φ to inject next.

Each of the managers is associated with several fault injectors and sensors. One manager is in charge of all tests on one physical machine. When the manager receives a fault scenario from the explorer (e.g., "inject an EINTR error in the third read socket call, and an ENOMEM error in the seventh malloc call"), it breaks the scenario down into atomic faults and instructs the corresponding injectors to perform the injection. The sensors are instructed to run the developer-provided workload scripts (e.g., a benchmark) and perform measurements, which are then reported back to the manager. The manager aggregates these measurements into a single impact value and returns it to the explorer.

The goal of a sequence of such injections—a fault exploration session—is to produce a set of faults that satisfy a given criterion. For example, AFEX can be used to find combinations of faults that cause a database system to lose or corrupt data. As another example, one could obtain the top-50 worst faults performance-wise (i.e., faults that affect system performance the most). Prior to AFEX, this kind of test generation involved significant amounts of human labor.

To further help developers, AFEX presents the faults it found as a map, clustered by the degree of redundancy with respect to code these faults exercise in the target S. For each fault in the result set, AFEX provides a generated script that can run the test and replay the injection. Representatives of each redundancy cluster can thus be directly assembled into (or inserted into existing) regression test suites.

6.1 Architecture

Since tests are independent of each other, AFEX enjoys "embarrassing parallelism." Node managers need not talk to each other, only the explorer communicates with node managers. Given that the explorer's workload (i.e., selecting the next test) is significantly less than that of the managers (i.e., actually executing and evaluating the test), the system has no problematic bottleneck for clusters of dozens of nodes, maybe even larger. In this section, we provide more details on our prototype's two main components, the explorer and the node manager.

Explorer The AFEX explorer is the main control point in the exploration process. It receives the fault space description and the search target, and then searches the fault space for tests to put in the result set. The explorer can navigate the fault space in three ways: using the fitness-guided Algorithm 1, exhaustive search, or random search.

Node Manager The node manager coordinates all tasks on a physical machine. It contains a set of plugins that convert fault descriptions from the AFEX-internal representation to concrete configuration files and parameters for the injectors and sensors. Each plugin, in essence, adapts a subspace of the fault space to the particulars of its associated injector. The actual execution of tests on the system S is done via three user-provided scripts: A startup script prepares the environment (setting up workload generators, necessary environment variables, etc.). A test script starts up S and signals the injectors and sensors to proceed; they in turn will report results to the manager. A cleanup script shuts S down after the test and removes all side effects of the test.

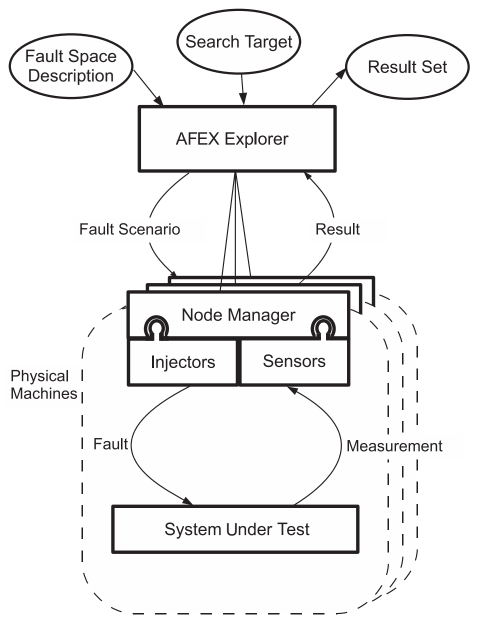

Figure 2. AFEX prototype architecture: an explorer coordinates multiple managers, which in turn coordinate the injection of faults and measurement of the injections' impact on the system under test.

6.2 Input

AFEX takes as input descriptions of the fault spaces to be explored, sensor plugins to measure impact metrics (which AFEX then uses to guide fault exploration), and search targets describing what the user wants to search for, in the form of thresholds on the impact metrics.

Fault Description Language The language for describing fault spaces must be expressive enough to allow the definition of complex fault spaces, but succinct and easy to use and understand. Fig. 3 shows the grammar of the fault space description language used in AFEX.

Fault spaces are described as a Cartesian product of sets, intervals, and unions of subspaces (subtypes). Subspaces are separated by ";". Sets are defined with "{ }". Intervals are defined using "[ ]" or "< >". The difference between these two is that, during fault selection, intervals marked with "[ ]" are sampled for a single number, while intervals marked with "< >" are sampled for entire sub-intervals (we say $< 5,10 >$ is a sub-interval of $< 1,50 >$).

```
syntax       = {space};
space        = (subtype | parameter )+";";
subtype      = identifier;
parameter    = identifier ":"
( "{" identifier ( "," identifier )+ "}" |
  "[" number "," number "]"              |
  "<" number "," number ">"
);
identifier   =  letter ( letter | digit | "_" )*;
number       = (digit)+;
```

Figure 3. AFEX fault space description language.

```
function   : { malloc, calloc, realloc }
errno      : { ENOMEM }
retval     : { 0 }
callNumber : [ 1 , 100 ] ;

function   : { read }
errno      : { EINTR }
retVal     : { -1 }
callNumber : [ 1 , 50 ] ;
```

Figure 4. Example of a fault space description.

```
 function malloc errno ENOMEM retval 0
 callNumber 23
```

Figure 5. Example of a fault scenario description.

Fig. 4 shows an example representing the fault space for a library fault injection tool. This fault space is a union of two hyperspaces, separated by ";". The injection can take place in any of the first 100 calls to memory allocation, or in any of the first 50 calls to read. One possible fault injection scenario the explorer may sample from this fault space is shown in Fig. 5. This scenario would be sent by the explorer to a node manager for execution and evaluation.

6.3 Output

AFEX's output consists of a set of faults that satisfy the search target, a characterization of the quality of this fault set, and generated test cases that inject the faults of the result set into the system under test *S* and measure their impact on *S*. In addition to these, AFEX also reports operational aspects, such as a synopsis of the search algorithms used, injectors used, CPU/memory/network resources used, exploration time, number of explored faults, etc.

Quality Characterization: The quality characterization provides developers with additional information about the tests, in order to help guide their attention toward the most relevant ones. The two main quality metrics are redundancy and repeatability/precision, described earlier in §5. AFEX aims to provide a confidence characterization, similar to what a Web search engine user would like to get with her search results: which of the returned pages have the same (or similar) content. A practical relevance evaluation of the fault set is optionally available if the developer can provide the corresponding statistical model (§5).

Test Suites: AFEX automatically generates test cases that can directly reproduce each fault injection and the observed impact on the system. Each test case consists of a set of configuration files for the system under test, configuration of the fault injector(s), a script that starts the system and launches the fault injector(s), and finally a script that generates workload on the system and measures the faults' impact. We find these generated test scripts to save considerable human time in constructing regression test suites.

6.4 Extensibility and Control

The AFEX system was designed to be flexible and extensible. It can be augmented with new fault injectors, new sensors and impact metrics, custom search algorithms, and new result-quality metrics.

We present below the steps needed to use AFEX on a new system under test *S*. In our evaluation, adapting AFEX for use on a new target *S* took on the order of hours.

1. *Write fault injection plugins*. These are small Java code snippets required to wrap each fault injector tool to be used (~150 lines of code).
2. *Choose fault space*. The developer must write a fault space descriptor file (using the language specified in Fig. 3). We found that the simplest way to come up with this description is to analyze the target system *S* with a tracer like *ltrace*, or to use a static analysis tool, such as the profiler that ships with LFI [17].
3. *Design impact metric*. The impact metric guides the exploration algorithm. The easiest way to design the metric is to allocate scores to each event of interest, such as 1 point for each newly covered basic block, 10 points for each hang bug found, 20 points for each crash, etc.
4. *Provide domain knowledge*. Optionally, the developer can give AFEX domain knowledge in various ways. For example, if the system under test is to be deployed in a production environment with highly reliable storage, it may make sense to provide a fault relevance model in which I/O faults are deemed less relevant (since they are less likely to occur in practice), unless they have catastrophic impact on *S*. This will discouraging AFEX from exploring faults of little interest.
5. *Write test scripts*. Developers must provide three scripts: startup, test, and cleanup. These are simple scripts and can be written in any scripting language supported on the worker nodes.
6. *Select search target*. The tester can choose to stop the tests after some specified amount of time, after a number of tests executed, or after a given threshold is met in terms of code coverage, bugs found, etc.

7. *Run AFEX*. AFEX is now ready to start. It provides progress metrics in a log, so that developers can follow its execution, if they wish to do so.
8. *Analyze results*. AFEX produces tables with measurements for each test (fitness, quality characterization, etc.), and it identifies a representative test for each redundancy cluster (as described in §5). AFEX also creates a folder for each test, containing logs, core dumps, or any other output produced during the test.

7. Evaluation

In this section, we address the following questions about the AFEX prototype: Does it find bugs in real-world systems (§7.1)? How efficient is AFEX exploration compared to random and exhaustive search (§7.2)? To what extent does fault space structure improve AFEX's efficiency (§7.3)? Can AFEX leverage result-quality metrics to improve its search efficiency (§7.4)? To what extent can system-specific knowledge aid AFEX (§7.5)? How does AFEX's usefulness vary across different stages of system development (§7.6)? How well does the AFEX prototype scale (§7.7)?

Evaluation Targets Most of our experiments focus on three real-world code bases: the MySQL 5.1.44 database management system, the Apache httpd 2.3.8 Web server, and the coreutils 8.1 suite of UNIX utilities. These systems range from large ($> 10^6$ lines of code in MySQL) to small (~10^3 lines of code per UNIX utility). We used AFEX to find new bugs in MySQL and Apache httpd, both mature systems considered to be highly reliable. The UNIX utilities, being small yet still real-world, allow us to closer examine various details of AFEX and show how the various ideas described in the paper come together into a fitness-guided exploration framework that is significantly more efficient than random exploration, while still requiring no source code access. We also report measurements on the MongoDB NoSQL database, versions 0.8 and 2.0.

Fault Space Definition Methodology AFEX can explore both single-fault and multi-fault scenarios, but we limit our evaluation to only single-fault scenarios, which offer sufficient opportunity to examine all aspects of our system. Even this seemingly simple setup produces fault spaces with more than 2 million faults, which are infeasible to explore with brute-force approaches.

Our fault space is defined by the fault injection tools we use, along with profiling tools and, for some specific experiments, general knowledge of the system under test. We use the LFI library-level fault injector [16] and focus on injecting error returns into calls made to functions in the standard C library, libc.so. This library is the principal way for UNIX programs to interact with their environment, so we can use LFI to simulate a wide variety of faults in the file system, network, and memory. To define the fault space, we first run the default test suites that ship with our test targets, and use the ltrace library-call tracer to identify the calls that our target makes to libc and count how many times each libc function is called. We then use LFI's callsite analyzer, applied to the libc.so binary, to obtain a fault profile for each libc function, indicating its possible error return values and associated errno codes.

We use this methodology to define a fault space for the UNIX coreutils that is defined by three axes: X_{test} corresponds to the tests in the coreutils test suite, with $X_{test} = (1, ..., 29)$. X_{func} corresponds to a subset of libc functions used during these tests, and their index values give us $X_{func} = (1, ..., 19)$. Finally, X_{call} corresponds to the call number at which we want to inject a fault (e.g., the *n*-th call to malloc). In order to keep the fault space small enough for exhaustive search to be feasible (which allows us to obtain baseline measurements for comparison), we restrict these values to $X_{call} = (0, 1, 2)$, where 0 means no injection, and 1 or 2 correspond to the first or second call, respectively. The size of the resulting fault space $\Phi_{coreutils}$ is $29 \times 19 \times 3 = 1,653$ faults. To measure the impact of an injected fault, we use a combination of code coverage and exit code of the test suite. This encourages AFEX to both inject faults that cause the default test suite to fail and to cover as much code as possible.

For the MySQL experiments, we use the same methodology to obtain a fault space with the same axes, except that $X_{test} = (1, ..., 1147)$ and $X_{call} = (1, ..., 100)$, which gives us a fault space Φ_{MySQL} with 2,179,300 faults. If we assume that, on average, a test takes 1 minute, exploring this fault space exhaustively would take on the order of 4 CPU-*years*. MySQL therefore is a good example of why leveraging fault space structure is important. We use a similar impact metric to that in coreutils, but we also factor in crashes, which we consider to be worth emphasizing in the case of MySQL.

For the Apache httpd experiments, we have the same fault space axes, but $X_{test} = (1, ..., 58)$ and $X_{call} = (1, ..., 10)$, for a fault space Φ_{Apache} of 11,020 possible faults.

Metrics To assess efficiency of exploration, we count the number of failing tests from the X_{test} axis and analyze the generated coredumps. While imperfect, this metric has the benefit of being objective. Once fault injection becomes more widely adopted in test suites, we expect developers to write fault injection-oriented assertions, such as "under no circumstances should a file transfer be only partially completed when the system stops," in which case one can count the number of failed assertions.

Experimental Platform The experiments reported in §7.1 and §7.3–§7.5 ran on an Intel quad-core 2.4GHz CPU with 4GB RAM and 7200rpm HDD. The experiments in §7.2 ran on 5 small Amazon EC2 instances [3], the ones reported in §7.6 on a quad-core Intel 2.3GHz CPU with 16GB RAM and Intel SSD, and the ones reported in §7.7 on a range of 1–14 small Amazon EC2 instances.

7.1 Effectiveness of Search

In this section, we show that AFEX can be used to successfully test the recovery code of large, production-grade software, such as MySQL and the Apache httpd server, with minimal human effort and no access to source code. After running for 24 hours on a small desktop computer, AFEX found 2 new bugs in MySQL and 1 new bug in Apache httpd.

MySQL After exploring the Φ_{MySQL} fault space for 24 hours, AFEX found 464 fault injection scenarios that cause MySQL to crash. By analyzing the generated core dumps, we found two new bugs in MySQL. The results are summarized in Table 1. Comparison to exhaustive search is impractical, as it would take multiple CPU-years to explore all of Φ_{MySQL}.

	MySQL test suite	Fitness-guided	Random
Coverage	54.10%	52.15%	53.14%
# failed tests	0	1,681	575
# crashes	0	464	51

Table 1. Comparison of the effectiveness of fitness-guided fault search vs. random search vs. MySQL's own test suite.

We find that AFEX's fitness-guided fault search is able to find almost 3× as many failed tests as random exploration, and cause more than 9× as many crashes. Of course, not all these crashes are indicative of bugs—many of them result from MySQL aborting the current operation due to the injected fault. The random approach produces slightly higher general code coverage, but spot checks suggest that the *recovery* code coverage obtained by fitness-guided search is better. Unfortunately, it is impractical to estimate recovery code coverage, because this entails manually identifying each block of recovery code in MySQL. We now describe briefly the two bugs found by AFEX.

The first one [20] is an example of buggy error recovery code—the irony of recovery code is that it is hard to test, yet, when it gets to run in production, it cannot afford to fail. Since MySQL places great emphasis on data integrity, it has a significant amount of recovery code that provides graceful handling of I/O faults. However, in the code snippet shown in Fig. 6, the recovery code itself has a bug that leads to an abort. It occurs in a function that performs a series of file operations, any of which could fail. There is a single block of error recovery code (starting at line mi_create.c:836) to which all file operations in this function jump whenever they fail. The recovery code performs some cleanup, including releasing the THR_LOCK_myisam lock (on line 837), and then returns an error return code up the stack. However, if it is the call to my_close (on line 831) that fails, say due to an I/O error, then the recovery code will end up unlocking THR_LOCK_myisam twice and crashing.

The second MySQL bug is a crash that happens when a read from errmsg.sys fails. This bug is a new manifestation

```
mi_create.c
...
830: pthread_mutex_unlock(&THR_LOCK_myisam);
831: if (my_close(file,MYF(0)))
       goto err;
...
836: err:
837:   pthread_mutex_unlock(&THR_LOCK_myisam);
```

Figure 6. Buggy recovery code in MySQL.

of a previously discovered bug [19] that was supposedly fixed. MySQL has recovery code that checks whether the read from errmsg.sys was successful or not, and it correctly logs any encountered error if the read fails. However, after completing this recovery, regardless of whether the read succeeded or not, MySQL proceeds to use a data structure that should have been initialized by that read. This leads to MySQL crashing.

It is worth noting that MySQL does use an (admittedly ad-hoc) method for fault injection testing. The code has macros that override return values and errno codes after calls to libc, in order to simulate faults. These macros are enabled by recompiling MySQL with debug options. However, doing fault injection this way is laborious, because the macros need to be manually placed in each location where a fault is to be injected, and the specifics of the injected fault need to be hardcoded. This explains, in part, why MySQL developers appear to have lacked the human resources to test the scenarios that AFEX uncovered. It also argues for why AFEX-style automated fault exploration is useful, in addition to the fact that AFEX allows even testers unfamiliar with the internal workings of MySQL—such as ourselves—to productively test the server in black-box mode.

Apache httpd We let AFEX explore the Φ_{Apache} fault space of 11,020 possible faults, and we stopped exploration after having executed 1,000 tests. Using fitness-guided exploration, AFEX found 246 scenarios that crash the server; upon closer inspection, we discovered one new bug.

Table 2 summarizes the results. When limited to 1,000 samplings of Φ_{Apache}, AFEX finds 3× more faults that fail Apache tests and almost 12× more tests that crash Apache, compared to random exploration. It finds 27 manifestations of the bug in Figure 7, while random exploration finds none.

	Fitness-guided	Random
# failed tests	736	238
# crashes	246	21

Table 2. Effectiveness of fitness-guided fault search vs. random search for 1,000 test iterations (Apache httpd).

An industrial strength Web server is expected to run under high load, when it becomes more susceptible to running out of memory, and so it aims to handle such errors gracefully. Thus, not surprisingly, Apache httpd has extensive checking code for error conditions like NULL returns from malloc throughout its code base. The recovery code for an out-of-memory error generally logs the error and shuts down the server. Nevertheless, AFEX found a malloc failure scenario that is incorrectly handled by Apache and causes it to crash with no information on why. Fig. 7 shows the code.

```
config.c
...
578: ap_module_short_names[m->module_index]
                           = strdup(sym_name);
579: ap_module_short_names[m->module_index][len]
                           = '\0';
```

Figure 7. Missing recovery code in Apache httpd.

What the Apache developer missed is that strdup itself uses malloc, and can thus incur an out-of-memory error that is propagated up to the Apache code. When this happens, the code on line config.c:579 dereferences the NULL pointer, triggers a segmentation fault, and the server crashes without invoking the recovery code that would log the cause of the error. As a result, operators and technical support will have a hard time understanding what happened.

This bug illustrates the need for black-box fault injection techniques: an error behavior in a third-party library causes the calling code to fail, because it does not check the return value. Such problems are hard to detect with source code analysis tools, since the fault occurs not in the test system's code but in the third-party library, and such libraries are often closed-source or even obfuscated.

The MySQL and Apache experiments in this section show that automatic fault injection can test real systems and find bugs, with minimal human intervention and minimal system-specific knowledge. We now evaluate the efficiency of this automated process.

7.2 Efficiency of Search

To evaluate efficiency, we compare fitness-based exploration not only to random search but also to exhaustive search. In order for this to be feasible, we let AFEX test a couple UNIX coreutils, i.e., explore the $\Phi_{coreutils}$ fault space (1,653 faults). This is small enough to provide us an exhaustive-search baseline, yet large enough to show meaningful results.

First, we evaluate how efficiently can AFEX find interesting fault injection scenarios. We let AFEX run for 250 test iterations, in both fitness-guided and random mode, on the ln and mv coreutils. These are utilities that call a large number of library functions. We report in the first two columns of Table 3 how many of the tests in the test suite failed due to fault injection. These results show that, given a fixed time budget, AFEX is 2.3× more efficient at finding failed tests (i.e., high-impact fault injections) than random exploration. Exhaustive exploration finds 2.77× more failed tests than fitness-guided search, but takes 6.61× more time (each test takes roughly the same amount of time, and ~90% of the time in each test iteration is taken up by our coverage computation, which is independent of injected fault or workload).

	Fitness-guided	**Random**	**Exhaustive**
Code coverage	36.14%	35.84%	36.17%
# tests executed	250	250	1,653
# failed tests	74	32	205

Table 3. Coreutils: Efficiency of fitness-guided vs. random exploration for a fixed number (250) of faults sampled from the fault space. For comparison, we also show the results for exhaustive exploration (all 1,653 faults sampled).

Fitness-guided exploration is efficient at covering error recovery code. Consider the following: running the entire coreutils test suite without fault injection obtains 35.53% code coverage, while running additionally with exhaustive fault exploration (third column of Table 3) obtains 36.17% coverage; this leads us to conclude that roughly 0.64% of the code performs recovery. Fitness-guided exploration with 250 iterations (i.e., 15% of the fault space) covers 0.61% additional code, meaning that it covers 95% of the recovery code while sampling only 15% of the fault space.

Code coverage clearly is not a good metric for measuring the quality of reliability testing: even though all three searches achieve similar code coverage, the number of failed tests in the default test suite differs by up to 6×.

In Fig. 8 we show the number of failed tests induced by injecting faults found via fitness-guided vs. random exploration. As the number of iterations increases, the difference between the rates of finding high-impact faults increases as well: the fitness-guided algorithm becomes more efficient, as it starts inferring (and taking advantage of) the structure of the fault space. We now analyze this effect further.

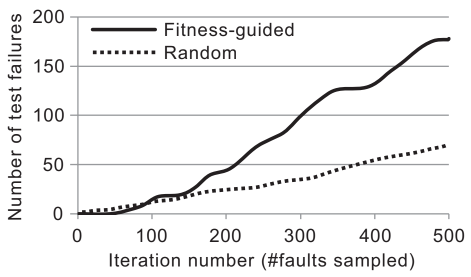

Figure 8. Number of test-failure-inducing fault injections for fitness-guided vs. random exploration.

7.3 Benefits of Fault Space Structure

To assess how much AFEX leverages the structure of the fault space, we evaluate its efficiency when one of the fault space dimensions is randomized, i.e., the values along that X_i are shuffled, thus eliminating any structure it had. If the efficiency of AFEX is hurt by such randomization, then it means that the structure along that dimension had been suitably exploited by AFEX. We perform this experiment on Apache httpd with Φ_{Apache}.

The results are summarized in Table 4: the randomization of each axis results in a reduction in overall impact. For example, 25% of the faults injected by AFEX with the original structure of Φ_{Apache} led to crashes; randomizing X_{test} causes this number to drop to 22%, randomizing X_{func} makes it drop to 13%, and randomizing X_{call} makes it drop to 17%. The last column, random search, is equivalent to randomizing all three dimensions. This is clear evidence that AFEX takes substantial advantage of whatever fault space structure it can find in each dimension in order to improve its efficiency.

	Original structure	**Rand. X_{test}**	**Rand. X_{func}**	**Rand. X_{call}**	**Random search**
# failed tests	73%	59%	43%	48%	23%
# crashes	25%	22%	13%	17%	2%

Table 4. Efficiency of AFEX in the face of structure loss, when shuffling the values of one dimension of the fault space (Apache httpd). Percentages represent the fraction of injected faults that cause a test in Apache's test suite to fail, respectively crash (thus, 25% crashes means that 25% of all injections led to Apache crashing).

Additionally, in the MySQL experiments of §7.1, we inspected the evolution of the sensitivity parameter (described in §3) and the choice of test scenarios, in order to see what structure AFEX infers in the Φ_{MySQL} fault space. The sensitivity of X_{func} converges to 0.1, while that of X_{test} and X_{call} both converge to 0.4 for MySQL. Table 4 suggests that Φ_{Apache} is different from Φ_{MySQL}: randomizing X_{func}, which was the least sensitive dimension in the case of MySQL, causes the largest drop in number of crashes, which means that for Φ_{Apache} it is actually the most sensitive dimension.

7.4 Benefits of Result-Quality Feedback

Another source of improvement in efficiency is the use of immediate feedback on the quality of a candidate fault relative to those obtained so far. AFEX aims to generate a result set that corresponds as closely as possible to the search target, and it continuously monitors the quality of this result set. One important dimension of this assessment is the degree of redundancy. In this section, we show how AFEX automatically derives redundancy clusters and uses these online, in a feedback loop, to increase its exploration efficiency.

As mentioned in §5, AFEX uses the Levenshtein edit distance for redundancy detection. In this experiment, we compare the stack traces at the injection points in the Apache tests, cluster them, and tie the outcome into a feedback loop: When evaluating the fitness of a candidate injection scenario, AFEX computes the edit distance between that scenario and all previous tests, and uses this value to weigh the fitness on a linear scale (100% similarity ends up zero-ing the fitness, while 0% similarity leaves the fitness unmodified).

The results are shown in Table 5. Even though the use of the feedback loop produces fewer failed tests overall, the search target is more closely reached: fitness-guided exploration with feedback produces about 40% more "unique" failures (i.e., the stack traces at the injection points are distinct) than fitness-guided exploration without feedback, and 75% more "unique" crashes. Of course, the method is not 100% accurate, since injecting a fault with the same call stack at the injection point can still trigger different behavior (e.g., depending on the inputs, the system under test may or may not use a NULL pointer generated by an out-of-memory error), but it still suggests improved efficiency.

	Fitness-guided	**Fitness-guided with feedback**	**Random search**
# failed tests	736	512	238
# unique failures	249	348	190
# unique crashes	4	7	2

Table 5. Number of unique failures/crashes (distinct stack traces at injection point) found by 1,000 tests (Apache).

Having assessed AFEX's properties when operating with no human assistance, we now turn our attention to evaluating the benefits of human-provided system-specific knowledge.

7.5 Benefits of System-Specific Knowledge

So far, we used AFEX purely in black-box mode, with no information about the system being tested. We now construct an experiment to evaluate how much benefit AFEX can obtain from knowledge of the tested system and environment.

We choose as search target finding all out-of-memory scenarios that cause the ln and mv coreutils to fail. Based on an exhaustive exploration of $\Phi_{coreutils}$, we know there are 28 such scenarios for these two utilities. Our goal is to count how many samplings of the fault space are required by the AFEX explorer to find these 28 faults.

	Fitness-guided	**Exhaustive**	**Random**
Black-box AFEX	417	1,653	836
Trimmed fault space	213	783	391
Trim + Env. model	103	783	391

Table 6. Number of samples (injection tests) needed to find all 28 malloc faults in $\Phi_{coreutils}$ that cause ln and mv to fail, for various levels of system-specific knowledge.

Table 6 shows the results for three different levels of system-specific knowledge. First, we run AFEX in its default black-box mode; this constitutes the baseline. Then we trim the fault space by reducing X_{func} to contain only the 9 libc functions that we know these two coreutils call—this reduces the search space. Next, we also add knowledge about the environment in the form of a statistical environment model, which specifies that malloc has a relative probability of failing of 40%, all file-related operations (fopen, read, etc.) have a combined weight of 50%, and opendir, chdir a combined weight of 10%. We use this model to weigh the measured impact of each test according to how likely it is to have occurred in the modeled environment.

The results show that trimming the fault space improves AFEX's efficiency by almost 2×, and adding the environment model further doubles efficiency—AFEX is able to reach the search target more than 4× faster than without this knowledge. Furthermore, compared to uninformed random search, leveraging this domain-specific knowledge helps AFEX be more than 8× faster, and compared to uninformed exhaustive search more than 16× faster.

7.6 Efficiency in Different Development Stages

So far, we have evaluated AFEX only on mature software; we now look at whether AFEX's effectiveness is affected by the maturity of the code base or not. For this, we evaluate AFEX on the MongoDB DBMS, looking at two different stages of development that are roughly 3 years apart: version 0.8 (pre-production) and version 2.0 (industrial strength production release). We ask AFEX to find faults that cause the tests in MongoDB's test suite to fail, and we expose both versions to identical setup and workloads. We let AFEX sample the fault space 250 times and compare its efficiency to 250 random samplings. The results are shown in Figure 9.

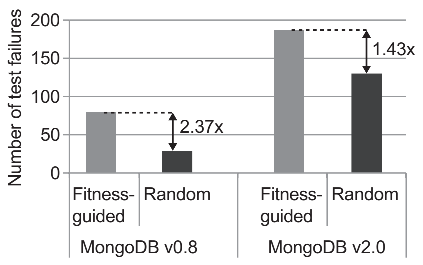

Figure 9. Changes in AFEX efficiency from pre-production MongoDB to industrial strength MongoDB.

For early versions of the software, AFEX is more efficient at discovering high-impact faults: compared to random search, AFEX finds 2.37× more faults that cause test failures; this efficiency drops in the industrial strength version to 1.43×. What may seem at first surprising is that AFEX causes more failures in v2.0 than in v0.8—this is due to increased complexity of the software and heavier interaction with the environment, which offers more opportunities for failure. Ironically, AFEX found an injection scenario that crashes v2.0, but did not find any way to crash v0.8. More features appear to indeed come at the cost of reliability.

7.7 Scalability

We have run AFEX on up to 14 nodes in Amazon EC2 [3], and verified that the number of tests performed scales linearly, with virtually no overhead. This is not surprising. We believe AFEX can scale much further, due to its embarrassing parallelism. In isolation, the AFEX explorer can generate 8,500 tests per second on a Xeon E5405 processor at 2GHz, which suggests that it could easily keep a cluster of several thousand node managers 100% busy.

8. Related Work

Pacheco [22] presents a technique that selects from a large set of test inputs a small subset likely to reveal faults in the software under test. This work focuses on finding software flaws in normal operation of software, different from our goal of finding weak points by injecting faults.

In our work we leverage the impact of previous injections to guide the exploration of the fault space. This impact is expressed using various metrics, and there is extensive prior work that has used metrics to characterize the effect of failures. Hariri et al. [12] present an agent-based framework that quantifies how attacks and faults impact network performance and services, discovers attack points, and examines how critical network components behave during an attack. Nagaraja et al. [21] propose a two-phase methodology for quantifying the performability of cluster-based Internet services, combining fault impact measurements with the expected fault load in a production environment.

Gunawi et al. [11] describe a framework for systematically injecting sequences of faults in a system under test. The authors present Failure IDs as a means of identifying injection points and describe a method for systematically generating injection sequences on the fly. The framework presented by Gunawi et al. takes an exhaustive exploration approach. An important merit of their method, however, is the means through which the fault space is automatically generated.

KLEE [7], a well-known symbolic execution tool, also has an optional fault injection operation mode. The approach in KLEE is again an example of exhaustive exploration. An important advantage of symbolic execution is that it allows fine-grain control on the execution paths through the system under test. We expect AFEX would benefit from the power of symbolic execution, allowing more control on the system under test, but at the cost of a significantly larger exploration space.

Godefroid et al. [10] and Pacheco et al. [23] introduce search algorithms for maximizing code coverage by generating input based on execution feedback. As suggested by our usage scenarios, we target finding errors at higher level (integration level).

McMinn surveyed various approaches towards the use of metaheuristic search techniques in software testing [18]. This survey covers various heuristics, such as Genetic Algorithms or Simulated Annealing, and various test techniques, from white box to black box testing. The survey relates to exploring the control flow graph of a program, rather than fault injection in particular.

9. Conclusion

We presented a set of techniques for enabling automated fault injection-based testing of software systems. Our main contribution is a fitness-driven fault exploration algorithm that is able to efficiently explore the fault space of a system under test, finding high-impact faults significantly faster than random exploration. We also show how to categorize the found faults into equivalence classes, such that the human effort required to analyze them is reduced. We implemented our ideas in the AFEX prototype, which found new bugs in real systems like MySQL and Apache httpd with minimal human intervention and no access to source code.

Acknowledgments

We thank the anonymous reviewers, our shepherd Dawson Engler, and the members of DSLAB for their valuable feedback on earlier versions of this paper. We are indebted to IBM and Google for their generous support of our work.

References

[1] Random fault injection in linux kernel. `http://lwn.net/Articles/209292/`.

[2] Reasoning software inspection services. `http://reasoning.com/index.html`.

[3] Amazon EC2. `http://aws.amazon.com/ec2`.

[4] L. N. Bairavasundaram, G. Goodson, B. Schroeder, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. An analysis of data corruption in the storage stack. In *USENIX Conf. on File and Storage Technologies*, 2008.

[5] E. Bisolfati, P. D. Marinescu, and G. Candea. Studying application–library interaction and behavior with LibTrac. In *Intl. Conf. on Dependable Systems and Networks*, 2010.

[6] P. E. Black, editor. *Dictionary of Algorithms and Data Structures*, chapter Manhattan distance. U.S. National Institute of Standards and Technology, 2006. `http://www.nist.gov/dads/HTML/manhattanDistance.html`.

[7] 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 Sys. Design and Implem.*, 2008.

[8] G. Candea, S. Bucur, and C. Zamfir. Automated software testing as a service. In *Symp. on Cloud Computing*, 2010.

[9] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Symp. on Operating Systems Principles*, 2001.

[10] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Network and Distributed System Security Symp.*, 2008.

[11] H. Gunawi, T. Do, P. Joshi, P. Alvaro, J. Hellerstein, A. Arpaci-Dusseau, R. Arpaci-Dusseau, K. Sen, and D. Borthakur. Fate and destini: A framework for cloud recovery testing. In *Symp. on Networked Systems Design and Implem.*, 2011.

[12] S. Hariri, G. Qu, T. Dharmagadda, M. Ramkishore, and C. S. Raghavendra. Impact analysis of faults and attacks in large-scale networks. *IEEE Security & Privacy*, 2003.

[13] L. Keller, P. Upadhyaya, and G. Candea. ConfErr: A tool for assessing resilience to human configuration errors. In *Intl. Conf. on Dependable Systems and Networks*, 2008.

[14] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics – Doklady*, 10, 1966.

[15] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Intl. Conf. on Programming Language Design and Implem.*, 2005.

[16] P. D. Marinescu and G. Candea. LFI: A practical and general library-level fault injector. In *Intl. Conf. on Dependable Systems and Networks*, 2009.

[17] P. D. Marinescu and G. Candea. Efficient testing of recovery code using fault injection. *ACM Transactions on Computer Systems*, 29(4), Dec. 2011.

[18] P. McMinn. Search-based software test data generation: A survey. *Intl. Journal on Software Testing, Verification and Reliability*, 14:105–156, 2004.

[19] Crash due to missing errmsg.sys. `http://bugs.mysql.com/bug.php?id=25097`, 2006.

[20] Crash due to double unlock. `http://bugs.mysql.com/bug.php?id=53268`, 2010.

[21] K. Nagaraja, X. Li, R. Bianchini, R. P. Martin, and T. D. Nguyen. Using Fault Injection and Modeling to Evaluate the Performability of Cluster-Based Services. *USENIX Symp. on Internet Technologies and Systems*, 2003.

[22] C. Pacheco. *Eclat: Automatic Generation and Classification of Test Inputs*. PhD thesis, Springer, 2005.

[23] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Intl. Conf. on Software Engineering*, 2007.

[24] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, 1995.

[25] M. Sullivan and R. Chillarege. Software defects and their impact on system availability – a study of field failures in operating systems. In *Intl. Symp. on Fault-Tolerant Computing*, 1991.

[26] T. K. Tsai and R. K. Iyer. Measuring fault tolerance with the FTAPE fault injection tool. In *Intl. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, 1995.

[27] T. Xie, N. Tillmann, P. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *Intl. Conf. on Dependable Systems and Networks*, 2009.