AFRL-RI-RS-TR-2009-192
Final Technical Report
July 2009

THE OPEN SOURCE HARDENING PROJECT

Stanford University

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

NOTICE AND SIGNATURE PAGE

AFRL-RI-RS-TR-2009-192 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/
FRANCES A. ROSE
Work Unit Manager

/s/
WARREN H. DEBANY, Jr.
Technical Advisor, Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
JULY 2009	Final	April 2005 – January 2009

4. TITLE AND SUBTITLE
THE OPEN SOURCE HARDENING PROJECT

5a. CONTRACT NUMBER
N/A

5b. GRANT NUMBER
FA8750-05-2-0142

5c. PROGRAM ELEMENT NUMBER
N/A

6. AUTHOR(S)
Dawson Engler and David Dill

5d. PROJECT NUMBER
DHSD

5e. TASK NUMBER
04

5f. WORK UNIT NUMBER
03

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Stanford University
Computer Science and Electrical Engineering
353 Serra Mall, Gates Bld. 3A-314
Stanford, CA 94305-9030

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFRL/RIGE
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
N/A

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RI-RS-TR-2009-192

12. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
This effort has developed and deployed a broad range of tools for finding serious errors in code. They are designed to find large numbers of errors in large source bases quickly, and with few false reports. We validated these tools by suing them to find bugs in important open-source projects (e.g., Linux, BSD, and many other widely-used projects). As a crucial part of doing so, we built and roan an ongoing “open source hardening” project that automatically applied our tools to these projects as a nightly regression and published the bugs in a developer-available database of errors. The benefits of automated, regular regressions are fourfold. First, it gave an objective, highly-visible validation that our tools work well on real code. Second, it provided corrective guidance to development, forcing tools to focus on what matters. Third, it strengthened on our relationships with developers on these projects, leading to (among other things) valuable user feedback, checking ideas, and (from experience) customer leads. Finally, and in some ways most important, it led to immediate improvements in the vast open-source infrastructure that serves as a foundation to much of the Nation’s computing environments.

15. SUBJECT TERMS
Open Source, Static Checkers, Memory Corruption

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Frances A. Rose
U	U	U	UU	60	19b. TELEPHONE NUMBER (*Include area code*) N/A

Table of Contents

List of Figures

1. INTRODUCTION

Soon after the first “0” was concatenated to a “1” there has been a “software crisis.” Software pervades life and all software, with rare exception, is broken. Errors range from simple crashes, to medical devices harming rather than helping, to security holes that allow attackers to view sensitive data and spend large amounts of money that someone else owns. Finding, fixing, and preventing software errors are arguably one of the most important problems in computer science.

The work in this contract aimed to attack this problem in three ways. First, to commercially deploy static program checking techniques we developed in prior work that had proven their worth, hopefully achieving wide impact. Second, to run these commercial tools on widely used open source code both to improve such software’s quality and to demonstrate the tools’ effectiveness in a transparent way. Finally, to do new research that would develop more powerful methods able to find errors out of the reach of static techniques.

2. METHODS, ASSUMPTIONS, AND PROCEDURES

This effort has developed and commercialized static bug-finding and software model checking tools we have built at Stanford and Coverity, to identify and remediate vulnerabilities as specified in Technical Topic Area 2, Composable and Scalable Secure Systems.

Commercialization focused on static (i.e., compile time) checkers that had shown their ability to find large numbers of errors in large source bases quickly, and with few false reports. Commercialization was done through Coverity, a company previously founded on research done at Stanford. Coverity also applied these tools to widely used open source projects (as discussed below). The effort at Stanford focused on developing techniques based on symbolic execution and model checking that are deeper than static analysis (at least in its usual sense). A constant challenge was making the techniques work on real code: they are significantly more heavy weight than static analysis and in the past had not been particularly effective at dealing with non-toy programs. As the results in this report show, the tools we built can regularly handle large, complex code. We deployed these tools via open source releases.

We validated all tools built and commercialized by using them to find bugs in important open-source projects (e.g., Linux, BSD, and many other widely-used projects). As a crucial part of doing so, Coverity built and ran an ongoing “open source hardening” project that automatically applied our tools to these projects as a nightly regression and published the bugs in a developer-available database of errors. The benefits of automated, regular regressions are fourfold. First, it gave an objective, highly-visible validation that our tools work well on real code. Second, it provided corrective guidance to development, forcing tools to focus on what matters. Third, it strengthened on our relationships with developers on these projects, leading to (among other things) valuable user feedback, checking ideas, and (from experience) customer leads. Finally, and in some ways most important, it led to immediate improvements in the vast open-source infrastructure that serves as a foundation to much of the nation's computing environments.

Somewhat unusually, much of the contract went according to plan. There were two main deviations from the initial proposal. First, while the technical aspect of the race detection and security checkers were largely within the realm of what we understood how to do, the users were not: commercial users have a somewhat erratic grasp of static analysis which can have a surprising impact on what the tool can do. It’s not enough to find an error --- the tool must describe, clearly and in a way that is difficult to misinterpret, why the error is a true error. At a high level, what this means is that the analysis used by the tool is no longer invisible (as it is with optimizing compilers). In particular, each time your tool calls a complicated subroutine in order to detect an error, you will essentially have to explain to the user what that routine did. For example, why two pointers are aliased or why a variable can be equal to a given constant. Depending on the details of this calculation, this exposition can be surprisingly difficult. The main consequence is that commercialization of checkers did not proceed en masse but started with simple ones, refined them, added more complex checkers gradually and in some cases scaled back the types of errors reported --- not because the checkers did not find them, but because it was too challenging to find ways to describe them so that users would not mark them as false positives.

The second deviation was the result of a breakthrough in understanding. We had planned to take our FiSC model checker and make it more general and powerful. We succeeded in producing a more general and powerful model checker, but it was fundamentally different than FiSC. At a high level, FiSC worked by importing the code to check into the tool and running it in a fake environment. This fake environment allowed FiSC to easily control all the inputs and environmental actions the code saw (for example: when a machine crash happened, when memory allocation would fail). At the time, this approach seemed like the easiest way to go since we checked operating system code, which was otherwise hard to manipulate. The breakthrough came when we realized that we could instead interlace the tool into the checked, thereby completely eliminating the need to construct a fake environment or simulate anything. As a result we were then able to even check commercial software for which we lacked source. The interested reader is referred to the eXplode section in this document and the paper included in Appendix A.

3. RESEARCH RESULTS AND DISCUSSION

Coverity was tasked with commercializing three static analysis methods and developing a framework for doing nightly checks of open source projects. Stanford was tasked with developing open source tools for model checking and developing effective tools that exploited symbolic constraint analysis to find bugs out of the reach of static methods.

We first discuss the results at Coverity and then at Stanford.

3.1 Coverity: Commercializing Static Checking Tools

The specific static checking tools Coverity developed and productized were:

- RACE: a static tool that uses path-sensitive, inter-procedural analysis to detect both race conditions and deadlocks. It is based on our prior work[1] and explicitly designed to find errors in large, complex, un-annotated multi-threaded systems.

- SECURE: a suite of security checkers that flag improper stack/heap accessing, integer overflow, buffer overflow and user-controllable string management errors. These are a more powerful version of analysis we had done previously.[2] In addition it flagged potentially time-to-check-to-time-of-use bugs.

- EXTEND: a programming interface for writing custom static checkers. It allows companies to write their own custom checks that look for domain- or even program-specific errors.

As one might expect, despite being able to leverage earlier research efforts, commercialization had to overcome numerous obstacles that only show up when you go from a few people checking a few code bases to thousands of people checking hundreds.

We now give some examples of the specific checkers in SECURE and RACE. For SECURE:

- Unsafe Use of Returned Values – NEGATIVE_RETURNS and NULL_RETURNS

  In C/C++, when a program calls a function the results are normally passed back to the calling code in a return value. In many cases a function reports errors in its operation, or complains about the arguments it was called with, by returning a negative or a NULL value.

  When the analysis sees that a function can return a negative or NULL value, and a code path exists where that value is subsequently used unsafely, then a defect will be flagged.

---

[1] "Racer: Effective, Static Detection of Race Conditions and Deadlock," Dawson Engler and Ken Ashcraft, Proceedings of the 19th Symposium on Operating Systems Principles, 2003.

[2] "Using Programmer Written Compiler Extensions to Catch Security Holes," Ken Ashcraft and Dawson Engler, Proceedings of the Oakland Security Conference, May, 2002.

- The TAINTED_SCALAR checker finds many instances where scalars (for example: integers) are not properly bounds-checked (sanitized) before being used as array or pointer indexes, loop boundaries, or function arguments. Scalars that are not sanitized are considered tainted.

  Missing or inadequate scalar validation can cause buffer overflows, integer overflows, denials of service, memory corruption, and security vulnerabilities.

  Signed scalars must be upper- and lower-bounds checked. Unsigned integers need only an upper-bounds check. You can also sanitize scalars with an equality check since this effectively bounds the value to a single number.

- The TAINTED_STRING checker finds many instances of improper string validation. Incorrectly checked strings are the root cause of many security holes. A simple example is reading a string from an environment and writing it to an internal buffer without checking that it does not exceed a maximum length. A trickier example: forgetting to check if an externally supplied string used to do a database lookup has wildcards. In this case an attacker could provide “*” which would match everything, returning the entire database. Other possible attacks include access control violations, environment corruption, cross-site scripting, file corruption, format string vulnerabilities, command injection, and SQL injection.

  Because an array of characters must be validated as opposed to bounds checking a single value, string sanitation is inherently more difficult than scalar cleansing. Doing so, therefore, usually means passing the string to a sanitizing function before using it in a trusted sink.

  To fix tainted string defects, implementers can use a programmer-defined format-string, such as syslog(LOG_WARNING, "%s", error_msg). Or, they can check for format specifics before passing to syslog()code. In general, they should run tainted strings through a sanitizing routine before using in a potentially unsafe way.

- SECURE_CODING: is one of the simplest checkers in that it does not use any flow information at all but just looks for any use of unsafe functions.

  Certain unsafe functions should never be used (such as “gets”), while others have been identified a security threat (such as “strcpy”). Also, some functions that were designed to alleviate the problems associated with their predecessors (for example, “strncpy” instead of “strcpy”), can still cause issues when used incorrectly.

  This set of checkers serve as more of an auditing tool since they flag any call to a potentially dangerous function, without analyzing the behavior of the code or the context with which the function was called. As such, it serves to warn of uses of historically unsafe functions and possible alternatives.

A representative subset of the checkers in RACE:

- The ATOMICITY checker reports defects when it finds a variable definition inside a critical section, but the use of that variable outside of that critical section. A critical section is a block of code that accesses a shared resource and that must not be accessed concurrently by another thread, and so is protected by a lock. Both the definition and use of the variable must be protected by the same lock.

- The LOCK checker finds many instances of double locks (locks acquired twice) and missing locks (missing lock releases).

  Two types of locks are supported:
  - Exclusive. An exclusive lock cannot be acquired recursively and attempt to do so will deadlock.
  - Recursive. The same thread can recursively acquire a recursive lock.

  A lock can be either a global variable or local to a function. LOCK reports a defect when the following sequence occurs: (Note that the values in parenthesis, such as (+lock), are a documentation convention used to aid in illustrating the following examples.)
  a. A variable L is locked (+lock).
  b. L is not unlocked (-unlock).

  One of the following can now occur:
  a. The path's end is reached (-lock_returned) and L does not appear anywhere in the function's return value or its expression.
  b. L is locked again (+double_lock). (Only for exclusive locks.)

  No errors are reported for functions that intentionally lock a function argument.

  Defects are also reported when the following sequence occurs:

  a. L is unlocked (+unlock).
  b. L is passed to a function which asserts that lock L is held (+lockassert)

  Forgetting to release an acquired lock can result in the program hanging; subsequent attempts to acquire the lock fail as the program waits for a release that will never occur.

- The MISSING_LOCK checker finds many instances where global variables or fields of structs are updated without locks, causing potential race conditions. Race conditions can lead to unpredictable or incorrect program behavior.

  The MISSING_LOCK checker tracks when variables are updated with locks. If a variable update is found that does not have a lock, but usually does have a lock, a defect is reported.

  In the Coverity Defect Manager, events from this checker are displayed in the multi-event code browser, which shows the missing lock event, followed by example_lock and example_access events. You can click on the file names to see the events in-line with the code.

- The ORDER_REVERSAL checker finds many instances of acquiring locks in the wrong order which can potentially cause deadlocks.

  Acquiring pairs of locks in the incorrect order can result in the program hanging. Because of thread interleaving, it is possible for two threads to each be waiting on a lock that the other thread has acquired (deadlock). Other threads attempting to acquire either of the two locks will also deadlock.

- The SLEEP checker finds many instances where blocking functions can cause a lock to be held too long, preventing other threads trying to acquire the same lock from continuing until the lock is released.

  Corrective action for this defect includes acquiring the lock after, or releasing the lock before, the blocking function call.

  Incorrect derivations of blocking functions, such as a function which blocks occasionally but not in all cases, are the most common causes of false positives. You can correct this with a model correctly indicating the function's behavior or with an annotation to suppress the block model. The annotation should suppress the blocks property.

Both of these suites of checkers have seen wide deployment at Coverity customer sites.

3.2 Coverity: Open Source Scanning

In addition, Coverity made its suite of checking tools available to qualified open source software projects through the SCAN project. Through the SCAN website site, open source developers can retrieve the defects identified by Prevent analyses through a portal accessible only by qualified project developers. The SCAN site is located at: http://scan.coverity.com.

The site divides open source projects into rungs based on the progress each project makes in resolving defects. Projects at higher rungs receive access to additional analysis capabilities and configuration options. Projects are promoted as they resolve the majority of defects identified at their current rung. More information on the criteria for the SCAN ladder is available at: http://scan.coverity.com/ladder.

The hardware behind the project consists of 6 servers: 5 build machines, and one database/web server machine. The systems run Linux and NetBSD, have 4 processing cores, 4G of RAM, and 1 or more terabytes of mirrored (Raid 1) disk.

3.3 Stanford

Stanford had two main tasks:

- Do an open source release of the FiSC file system checker we had built in previous work, and extend it to a broader class of code than just file systems.

- Prototype static analysis and model checking tools with integrated logical constraint analysis. The goal here was to find bugs out of the reach of static analysis.

For the first, we obsoleted FiSC by developing a dramatically lighter weight yet more powerful approach that worked with a much broader set of applications. We built an open source version of a tool, eXplode, based on this approach instead. For the second, we developed a very fast constraint solver, STP, and two tools that used it to deeply check C code. The first, EXE, used it to find security holes and to automatically generate attacks. The second, KLEE, used constraints to automatically generate inputs that would execute most statements in real code. We discuss each below.

3.4 eXplode: Systematically Checking Storage Systems

Our main research paper on eXplode is:

- Junfeng Yang, Can Sar, and Dawson Engler, "eXplode: a Lightweight, General System for Finding Serious Errors in Storage Systems," 7th Symposium on Operating Systems Design And Implementation (OSDI), 2006.

Storage systems such as file systems, databases, and RAID systems have a simple, basic contract: you give them data, they do not lose or corrupt it. Often they store the only copy, making its irrevocable loss almost arbitrarily bad. Unfortunately, their code is exceptionally hard to get right, since it must correctly recover from any crash at any program point, no matter how their state was smeared across volatile and persistent memory.

In the paper above, we describe eXplode: a system that makes it easy to systematically check real storage systems for errors. It takes user-written, potentially system-specific checkers and uses them to drive a storage system into tricky corner cases, including crash recovery errors.

It uses a novel adaptation of ideas from model checking, a comprehensive, heavy-weight formal verification technique, that makes its checking more systematic (and hopefully more effective) than a pure testing approach while being just as lightweight.

eXplode is effective. It found serious bugs in a broad range of real storage systems (without requiring source code): three version control systems, BerkeleyDB, an NFS implementation, ten file systems, a RAID system, and the popular VMware GSX virtual machine. We found bugs in every system we checked, 36 bugs in total, typically with little effort.

Table 1 gives a rundown of which bugs were where and how much code we required to find them. More complex checkers can find more bugs, but even simple ones find serious errors in production code, where an inopportune crash will cause the unrecoverable loss of data.

Table 1: summary of storage systems checked by eXplode

System	Storage	Checker	Bugs
FS	744	5,477	18
CVS	27	68	1
Subversion	-	-	1
ExpENSIV	30	124	3
Berkeley DB	82	202	6
RAID	144	FS + 137	2
NFS	34	FS	4
VMware GSX/Linux	54	FS	1
Total	1,115	6,008	36

3.5 EXE: Using Constraint Solving to Automatically Generate Inputs of Death.

Systems code defines an error-prone execution state space built from deeply nested conditionals and function call chains, massive amounts of code, and enthusiastic use of casting and pointer operations. Such code is hard to test and difficult to inspect, yet a single error can crash a machine or form the basis of a security breach.

We developed EXE, a system designed to automatically find bugs in such code using symbolic execution. The central insight behind EXE is that code can be used to automatically generate its own (potentially highly complex) test cases. At a high level, we mark data from untrusted sources as unconstrained symbolic input, which we then run to produce constraints and test cases. Instead of running code on manually generated test cases, EXE runs it on symbolic input that is initially free to be any value. As the code executes, the data is “interrogated”; the results of conditional expressions and other operations incrementally inform EXE what constraints to place on the values of the input in order for execution to proceed on a given path. Each time the code performs a conditional check involving a symbolic value, EXE forks execution, adding on the true path a constraint that the branch condition held while on the false path a constraint that it did not. EXE generates test cases for the program by using a constraint solver to find concrete values that satisfy the constraints. These automatically generated inputs are then fed back into the code.

EXE has several novel features. First, it precisely models all operations on *symbolic* pointers -- pointers whose address values are not concrete but instead symbolically constrained. EXE correctly handles: (1) the constraints generated from pointer arithmetic expressions involving (concrete or symbolic) pointers and (concrete or symbolic) offsets, (2) reads and writes to memory by dereferencing a symbolic pointer, and (3) arrays of symbolic size. Implementing these features involves more subtleties than one may expect. For example, given a concrete pointer "a" and a symbolic variable "i" constrained to be between 0 and n, then the conditional expression "if(a[i] == 10)" is essentially equivalent to a big disjunction: “if(a[0] == 10 || ... || a[n] == 10)”.

Similarly, the assignment “a[i] = 42” can potentially assign to any element in the array that the index could name. By design, because EXE has a precise view of the concrete heap -- which it treats as a sequence of untyped bytes whose constraints are induced by observation -- it does not matter how data is manipulated or how pointers are manufactured or cast.

Second, EXE symbolically executes all of the C language with bit-level precision. EXE works in the presence of unions, bit-fields, casts, and aggressive bit-operations (such as shifting, masking, byte swapping, or check summing). Every bit of a program executing under test is either not symbolic and thus represented exactly in the program's memory, or has a corresponding symbolic constraint that is exactly accurate. Thus, if at any program point in a deterministic program we generate a concrete solution for the constraints at that point, and then re-execute the program on this solution, the execution is guaranteed to arrive exactly back at this point, with concrete values compatible with the current symbolic constraints.

Third, EXE amplifies the effect of running a single code path since the use of a constraint solver lets it reason about *all possible values* that the path could be run with, rather than a single set of concrete values from an individual test case. To illustrate, a dynamic memory checker such as the Purify tool will only catch an out-of-bounds array access if the index (or pointer) has a specific concrete value that is out-of-bounds. In contrast, EXE will identify this bug if there is any possible input value on the given path that can cause an out-of-bounds access to the array. In addition, for an arithmetic expression that uses symbolic data, EXE can solve the associated constraints for values that cause an overflow or a division by zero. Moreover, for an "assert" statement, EXE searches over all possible input values on the given path for values that cause the assert to fail. If the assert does not fail then either (1) no input on this path can cause it to fail or (2) there is a bug in EXE.

Finally, EXE works well on real code. It automatically found buffer overruns in the very mature and audited Berkeley Packet Filter (BPF) code, the Linux networking code, and a server for the DHCPD protocol. EXE can scale up to large, real (and sometimes overly complex) systems code.

We had two main papers on EXE, which we discuss below:

- Junfeng Yang, Can Sar, Paul Twohey, Cristian Cadar and Dawson Engler, “Automatically Generating Malicious Disks using Symbolic Execution,” IEEE Proceedings on Security and Privacy, 2006.

Many current systems allow data produced by potentially malicious sources to be mounted as a file system. File system code must check this data for dangerous values or invariant violations before using it. Because file system code typically runs inside the operating system kernel, even a single unchecked value can crash the machine or lead to an exploit. Unfortunately, validating an allegedly safe file system image is complex: they form directed acyclic graphs (DAGs) with complex dependency relationships across massive amounts of data bound together with intricate, undocumented assumptions. We have used EXE to automatically find bugs in such code using symbolic execution. The approach works well in practice: we checked the disk mounting code of three widely-used Linux file systems: ext2, ext3, and JFS and generated concrete disks that when mounted would either cause a kernel panic or form the basis of a buffer overflow attack.

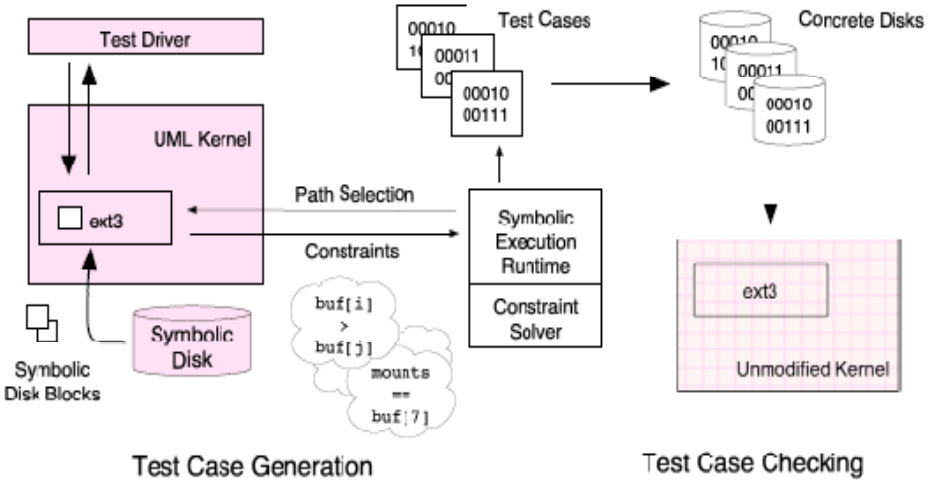


Figure 1: symbolic execution overview.

Figure 1 gives an overview of how the system works. We run Linux at user-level using the user-mode Linux kernel. We have a simple test driver that calls the "mount()" system call to mount a symbolic disk using one of the three Linux file systems we test (ext2, ext3, or JFS). EXE runs the given file system and whenever it hits an error will emit a raw disk image that triggers it. These disk images can be mounted on a live running file system to demonstrate the security hole.

Offset	*Hex Values*
00000	0000 0000 0000 0000 0000 0000 0000 0000
...	...
08000	464a 3153 0000 0000 0000 0000 0000 0000
08010	1000 0000 0000 0000 0000 0000 0000 0000
08020	0000 0000 0100 0000 0000 0000 0000 0000
08030	e004 000f 0000 0000 0002 0000 0000 0000
08040	0000 0000 0000 0000 0000 0000 0000 0000
...	...
10000	

Figure 2: hex dump of a disk generated by EXE that will cause JFS to crash.

Figure 2 gives a hex dump of a 64KB disk generated by EXE that will cause JFS on the Linux 2.4.27 kernel to dereference a null pointer. If you save this to a file and mount it with JFS it will cause a kernel crash. To reproduce the null dereference simply create an empty 64K file and set the 64th sector to the values in the figure (… indicates repeat of the previous row).

- "EXE: a System for Automatically Generating Inputs of death" Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, Dawson R. Engler. Association for Computing Machinery (ACM) Conference on Computer and Communication Security, 2006.

This paper gives an operational view of EXE and applies it to networking code where it found numerous security holes including invalid memory reads and writes in a Dynamic Host Configuration Protocol (DHCPD server implementation to finding buffer overflow attacks in the BSD and Linux packet filter implementations.

In addition, it describes the optimizations done in EXE's custom constraint solver, STP, which gains significant speed and simplicity by directly translating constraints to a Boolean satisfiability solver.

3.6 KLEE: Automatically Running Most Statements in Real Code

Our main paper on KLEE:

- "Klee: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," Cristian Cadar, Daniel Dunbar and Dawson Engler, *Operating System Design and Implementation, 2008(Won Best paper)*

Many classes of errors, such as functional correctness bugs, are difficult to find without executing a piece of code. The importance of such testing, combined with the difficulty and poor performance of random and manual approaches has led us to develop a set of tools based on symbolic execution designed to deeply check real code.

At a high-level, these tools use variations on the following idea: Instead of running code on manually- or randomly-constructed input, they run it on symbolic input initially allowed to be "anything." They substitute program inputs with symbolic values and replace corresponding concrete program operations with ones that manipulate symbolic values. When program execution branches based on a symbolic value, the system (conceptually) follows both branches, on each path maintaining a set of constraints called the path condition which must hold on execution of that path. When a path terminates or hits a bug, a test case can be generated by solving the current path condition for concrete values. Assuming deterministic code, feeding this concrete input to an uninstrumented version of the checked code makes it follow the same path and hit the same bug.

Our most recent tool, KLEE, is capable of automatically generating tests that achieve high coverage on a diverse set of complex and environmentally-intensive programs. We used it to thoroughly check (1) all 89 stand-alone programs in the CoreUtils utility suite, which form the core user-level environment installed on almost all Unix systems, and, as such, represent some of the most heavily used and tested open-source programs in existence, (2) 72 application in the BusyBox utilities suite for embedded systems, and (3) the HiStar operating system kernel. KLEE- generated tests achieved high statement coverage --- on average over 90% per tool in CoreUtils and BusyBox (median: over 94%) --- and in aggregate significantly beat the coverage of the developers' own hand-written test suites.

We also used KLEE as a bug finding tool, applying it to 448 applications (over 433K total lines of code), where it found 56 serious bugs, including three in CoreUtils that had been missed for over 15 years. In addition, we also used KLEE to cross-check purportedly identical BusyBox and CoreUtils utilities, finding functional correctness errors and a myriad of inconsistencies.

To support this analysis we developed a new constraint solver, STP, which can efficiently solve the types of constraints generated by real code (arrays, bit operations, etc). Using STP we can support all of C with no imprecision, with the exception of floating point, which we have elided.

We now give a more detailed breakdown of these experiments.

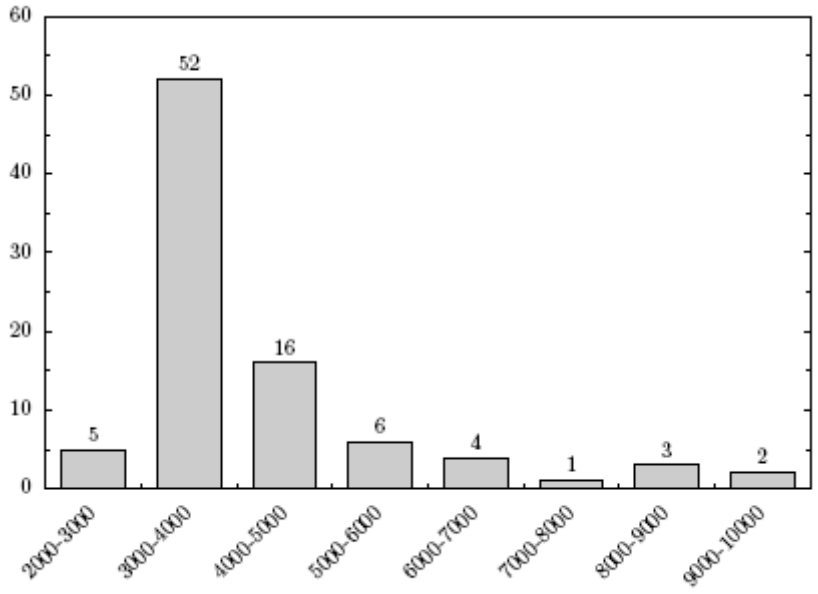


Figure 3 Histogram showing number of programs with the given number of executable lines of code

Figure 3 gives a breakdown of the CoreUtils program by size. While they are not millions of lines of code, they are not toys, ranging from 2K to 10K executable lines of code: this count is roughly a factor of three smaller than a simple line count, and excludes blank lines, variable declarations, structure definitions, etc.

Figure 4: Coverage with and without failing system calls

Our experiments consisted of running these 89 applications unaltered for an hour apiece with KLEE and then taking the tests it generated and rerunning these on uninstrumented versions of the programs to get line coverage. Figure 4 shows the distribution of the line coverage achieved on a per application basis. It sorts applications by their line coverage from least to greatest. 16 utilities have 100% line coverage, and the average is 91% (median 94.7%!).

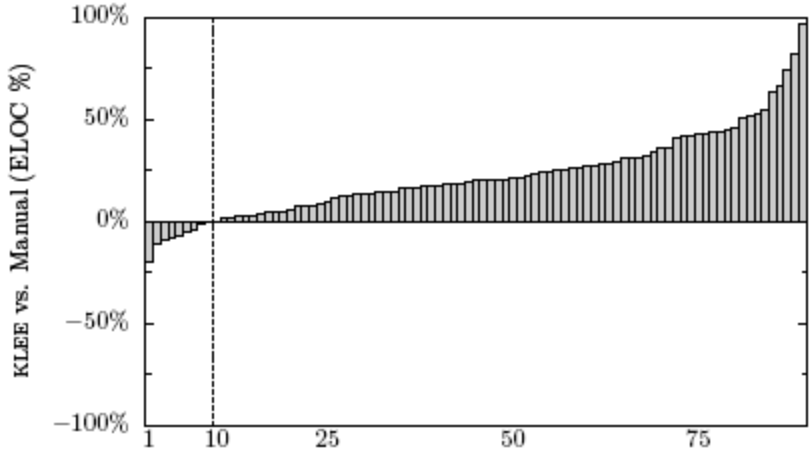


Figure 5: KLEE vs. the developers' manual test suite for CoreUtils. A bar above 0 means KLEE beat the developers and by how much.

Figure 5 compares the KLEE generated coverage to that of the developers' own manual written test suite, constructed over a period of 15 years. It subtracts the developers' coverage from KLEE's coverage and plots the result. A bar at 100% means KLEE got 100% coverage and the developers got nothing. A bar below the 0% line means the developers beat KLEE. As can be seen from the graph, this rarely happened (in 9 cases out of 89 utilities).

```
paste -d\\ abcdefghijklmnopqrstuvwxyz
pr -e t2.txt
tac -r t3.txt t3.txt
mkdir -Z a b
mkfifo -Z a b
mknod -Z a b p
md5sum -c t1.txt
ptx -F\\ abcdefghijklmnopqrstuvwxyz
ptx x t4.txt
seq -f %0 1
```

t1.txt: "\t \tMD5("
t2.txt: "\b\b\b\b\b\b\b\t"
t3.txt: "\n"
t4.txt: "a"

Figure 6: KLEE generated inputs (modified for readability) that will cause crashes on version 6.10.

Figure 6 gives KLEE generated command lines of death. When KLEE finds an error, it solves the constraints for the given path to get a concrete input that will trigger that error. It then provides these constraints to developers so that they can replicate the error. All of these produce crashes in CoreUtils version 6.10.

4 TRANSITION EFFORTS

4.1 Coverity

Commercialization has gone exceptionally well. Coverity has met all goals, and shipped the results to many customers. Department of Homeland Security sponsorship has been extremely helpful in terms of getting publicity and (as a result) into more companies.

Coverity has grown over the course of the contract from around 50 customers to over 500+ customers, with a combined source code base of over a *billion* lines of code. It has achieved good penetration into The Fortune 500: 57% of software companies, 54% of networking companies, 50% of computer companies, and 44% of aerospace companies. Many of these sales have included the checkers developed for this effort to find concurrency errors (RACE) and security holes (SECURE).

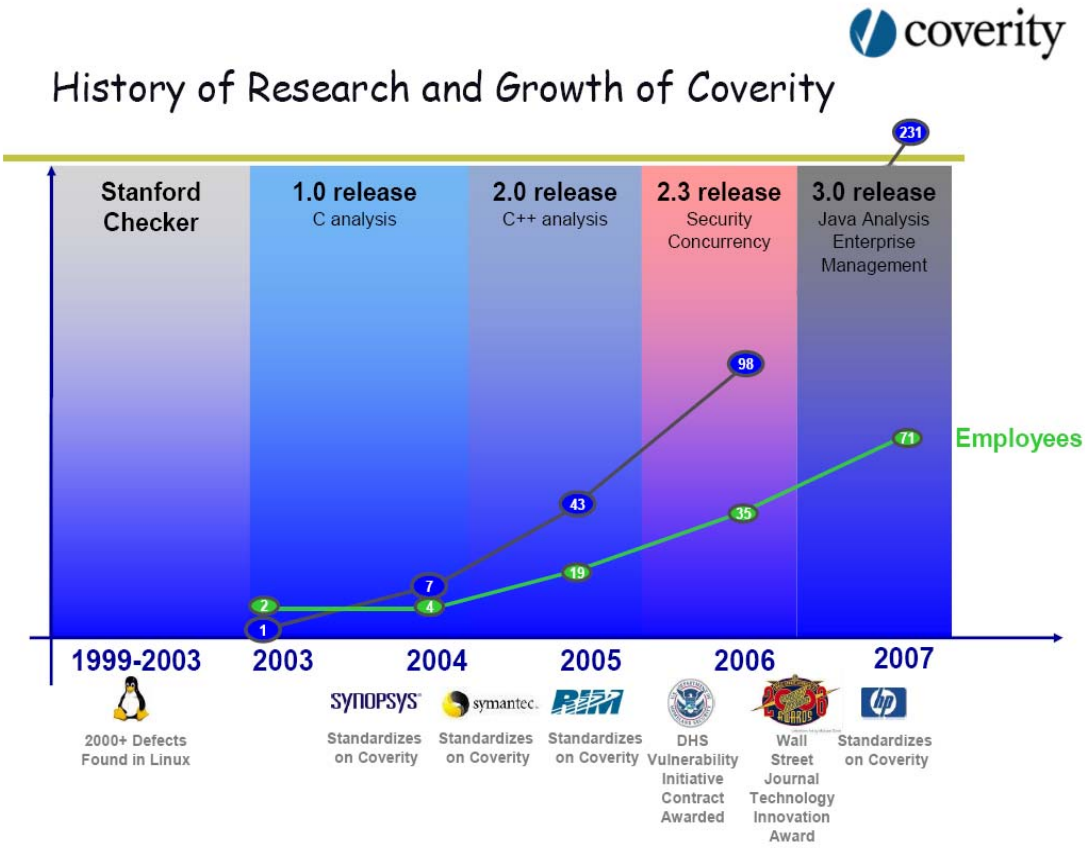


Figure 7: Coverity growth in terms of customers and employees up to 2007

Figure 7 is a somewhat outdated graphic showing the early history of growth: the initial research was started at Stanford in 1999, commercialized in 2003, and after an initial point when employees outnumbered customers, switched the other way. As mentioned above, the trend lines have seen even sharper growth on the right, hitting over 500 customers and 120+ employees. Despite the current recession, the last quarter set a record in terms of most revenue brought in.

Since 2006, the SCAN site has analyzed over 55 million lines of code on a recurring basis from more than 250 open popular source projects such as Firefox, Linux, and PHP. This represents 14,238 individual project analysis runs for a total of nearly 10 billion lines of code analyzed. Of these 250 projects, over 120 have developers actively working to reduce the number of reported defects in their code. The efforts of these developers have resulted in the elimination of more than 8,500 defects in open source programs over a period of 24 months.

The effort has generated significant press, good will, and as can be seen from the bug counts, helped remove significant amounts of defects from key open source code. Additionally, numerous sales have come about because of developers have had a good initial impression of Coverity, because they either have read about it helping open source or have seen its effects first hand on open source they are involved with. Coverity has significant interest in finding a way to continue this effort after the current contract has expired.

With that said, the use of the EXTEND framework to add new, company-specific checkers has lagged significantly in comparison to RACE and SECURE. The main reason is that most companies have never used any automatic checker at all. Thus, usually the bugs found by the default Coverity checker suite are all they can handle. After the initial wave of sales we hope to see the more advanced customers pick it up more widely.

There have been literally hundreds of press articles on Coverity and the SCAN effort.

4.2 Stanford

Both the eXplode storage system checker and KLEE tool for automatically executing most statements in code automatically have been made available as open source. KLEE is the cornerstone of the Stanford research group and remains under active development (likely for the next several years).

The STP constraint solver developed at Stanford partially funded by DHS has seen wide-spread use in a variety of areas, detailed below. Tools that use STP:

- MINESWEEPER, Jim Newsome, David Brimley, Prof. Dawn Song and others at Berkeley.

- CATCHCONV, David Molnar and Prof. David Wagner at University of California, Berkeley

- A backward path-sensitive analysis of C programs to find bugs by Tim Leek from MIT Lincoln Labs

- JPF-SE, a symbolic execution extension to the Java Pathfinder model checker by SiSwati An and, Corona Pasadena and Willem Visor from NASA Ames Research Center

- REPLAYER, a tool that replays an application dialog between two hosts in order to analyze security exploits by Jim Newsome, David Brimley, Prof. Dawn Song and others at Carnegie Mellon University (CMU)

5 CONCLUSIONS

The effort funded by DHS has met all deliverable goals.

Coverity has commercialized the static checking tools it was tasked with, and has seen great commercial success, helped by DHS sponsorship. During the course of the effort it has grown from 50 customers to more than 500, with over a billion lines of code between them.

Stanford has developed several new, powerful tools for checking real code. The most recent, KLEE, can automatically execute most statements in a diverse set of widely used programs.

6 RECOMMENDATIONS

The field of effective bug finding has seen a revolution in the past decade. Real tools have emerged that work well on real code. We expect the next decade to see just as many advances. We urge the agency to aggressively fund such efforts.

In addition, we strongly urge the agency to provide a way to have the many tools being developed be continuously applied to the open source code crucial to the Nation's infrastructure. Doing so both provides effective hardening of key attacker exposed surfaces and serves as a rigorous test as to which tools work and which are a waste of effort.

7 BIBLIOGRAPHY

"Racer: Effective, Static Detection of Race Conditions and Deadlock," Dawson Engler and Ken Ashcraft, Proceedings of the 19th Symposium on Operating Systems Principles, 2003.

"Using Programmer Written Compiler Extensions to Catch Security Holes," Ken Ashcraft and Dawson Engler, Proceedings of the Oakland Security Conference, May, 2002.

Junfeng Yang, Can Sar, and Dawson Engler, "eXplode: a Lightweight, General System for Finding Serious Errors in Storage Systems," 7th Symposium on Operating Systems Design and Implementation (OSDI), 2006.

"EXE: a System for Automatically Generating Inputs of death" Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, Dawson R. Engler. Association for Computing Machinery (ACM) Conference on Computer and Communication Security, 2006.

"Klee: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," Cristian Cadar, Daniel Dunbar and Dawson Engler, *Operating System Design and Implementation, 2008(Won Best paper)*

8 LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

ACM: ASSOCIATION FOR COMPUTING MACHINERY.

BPF: BERKELEY PACKET FILTER.

BSD: BERKELEY UNIX

DHCPD: COMMONLY USED SERVER THAT IMPLEMENTS THE DYNAMIC HOST CONFIGURATION PROTOCOL.

EXE: A TOOL DEVELOPED AT STANFORD THAT USES CONSTRAINT-BASED EXECUTION TO AUTOMATICALLY GENERATE INPUTS OF DEATH THAT CRASH REAL PROGRAMS

EXPLODE: A TOOL DEVELOPED AT STANFORD THAT USED A VARIATION ON MODEL CHECKING TO DO LIGHTWEIGHT CHECKING OF STORAGE SYSTEMS TO FIND WHEN THEY WILL LOSE OR CORRUPT DATA.

FISC: THE INITIAL STORAGE CHECKING TOOL THAT EXPLODE (ABOVE) WAS BASED ON.

KLEE: A TOOL DEVELOPED AT STANFORD THAT USES CONSTRAINT-BASED EXECUTION TO AUTOMATICALLY GENERATE INPUTS THAT EXECUTE MOST STATEMENTS IN REAL PROGRAMS

JFS: FILE SYSTEM FOR LINUX, DEVELOPED BY IBM.

PREVENT: GENERIC NAME FOR THE COMMERCIAL TOOL DEVELOPED BY COVERITY.

SQL: WIDELY USED DATABASE QUERY LANGUAGE

STP: AN EFFICIENT CONSTRAINT SOLVER DEVELOPED AT STANFORD USED BY BOTH EXE AND KLEE.

APPENDIX: A

EXPLODE: a Lightweight, General System for Finding Serious Storage System Errors

Junfeng Yang, Can Sar, and Dawson Engler
Computer Systems Laboratory
Stanford University

Abstract

Storage systems such as file systems, databases, and RAID systems have a simple, basic contract: you give them data, they do not lose or corrupt it. Often they store the only copy, making its irrevocable loss almost arbitrarily bad. Unfortunately, their code is exceptionally hard to get right, since it must correctly recover from any crash at any program point, no matter how their state was smeared across volatile and persistent memory.

This paper describes EXPLODE*, a system that makes it easy to systematically check real storage systems for errors. It takes user-written, potentially system-specific checkers and uses them to drive a storage system into tricky corner cases, including crash recovery errors.* EXPLODE *uses a novel adaptation of ideas from model checking, a comprehensive, heavyweight formal verification technique, that makes its checking more systematic (and hopefully more effective) than a pure testing approach while being just as lightweight.*

EXPLODE *is effective. It found serious bugs in a broad range of real storage systems (without requiring source code): three version control systems, Berkeley DB, an NFS implementation, ten file systems, a RAID system, and the popular VMware GSX virtual machine. We found bugs in every system we checked, 36 bugs in total, typically with little effort.*

1 Introduction

Storage system errors are some of the most destructive errors possible. They can destroy persistent data, with almost arbitrarily bad consequences if the system had the only copy. Unfortunately, storage code is simultaneously both difficult to reason about and difficult to test. It must always correctly recover to a valid state if the system crashes at *any* program point, no matter what data is being mutated, flushed (or not flushed) to disk, and what invariants have been violated. Further, despite the severity of storage system bugs, deployed testing methods remain primitive, typically a combination of manual inspection (with the usual downsides), fixes in reaction to bug reports (from angry users) and, at advanced sites, the alleged use of manual extraction of power cords from sockets (a harsh test indeed, but not comprehensive).

This paper presents EXPLODE, a system that makes it easy to thoroughly check real systems for such crash recovery bugs. It gives clients a clean framework to build and plug together powerful, potentially system-specific dynamic storage checkers. EXPLODE makes it easy for checkers to find bugs in crash recovery code: as they run on a live system they tell EXPLODE when to generate the disk images that could occur if the system crashed at the current execution point, which they then check for errors.

We explicitly designed EXPLODE so that clients can check complex storage stacks built from many different subsystems. For example, Figure 1 shows a version control system on top of NFS on top of the JFS file system on top of RAID. EXPLODE makes it quick to assemble checkers for such deep stacks by providing interfaces that let users write small checker components and then plug them together to build many different checkers.

Checking entire storage stacks has several benefits. First, clients can often quickly check a new layer (sometimes in minutes) by reusing consistency checks for one layer to check all the layers below it. For example, given an existing file system checker, if we can slip a RAID layer below the file system we can immediately use the file system checker to detect if the RAID causes errors. (Section 9 uses this approach to check NFS, RAID, and a virtual machine.) Second, it enables strong end-to-end checks, impossible if we could only check isolated subsystems: correctness in isolation cannot guarantee correctness in composition [22]. Finally, users can localize errors by cross-checking different implementations of a layer. If NFS works incorrectly on seven out of eight file systems, it probably has a bug, but if it only breaks on one, that single file system probably does (§9.2).

We believe EXPLODE as described so far is a worthwhile engineering contribution. A second conceptual contribution is its adaptation of ideas from model checking [6, 15, 17], a typically heavyweight formal verification technique, to make its checking more systematic (and thus hopefully more effective) than a pure testing approach while remaining as lightweight as testing.

Traditional model checking takes a specification of a system (a "model") which it checks by starting from an initial state and repeatedly performing all possible actions to this state and its successors. A variety of techniques exist to make this exponential search less inefficient. Model checking has shown promise in finding

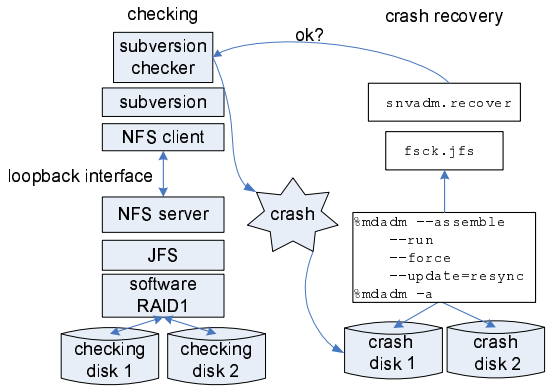


Figure 1: A snapshot of EXPLODE with a stack of storage systems being checked on the left and the recovery tools being run on the right after EXPLODE "crashes" the system to generate possible crash disks. This example checks Subversion running on top of NFS exporting a JFS file system running on RAID.

corner-case errors. However, requiring implementors to rewrite their system in an artificial modeling language makes it extremely expensive for typical storage systems (read: almost always impractical).

Recent work on *implementation-level model checking* [3, 13, 18] eliminates the need to write a model by using code itself as its own (high-fidelity) model. We used this approach in prior work to find serious errors in Linux file systems [30]. However, while more practical than a traditional approach, it required running the checked Linux system inside the model checker itself as a user-space process, which demanded enormously invasive modifications. The nature of the changes made it hard to check anything besides file systems and, even in the best case, checking a new file system took a week's work. Porting to a new Linux kernel, much less a different operating system, could take months.

This paper shows how to get essentially all the model checking benefits of our prior work with little effort by turning the checking process inside out. Instead of shoehorning the checked system inside the model checker (or worse, cutting parts of the checked system out, or worse still, creating models of the checked code) it interlaces the control needed for systematic state exploration *in situ*, throughout the checked system, reducing the modifications needed down to a single device driver, which can run inside of a lightly-instrumented, stock kernel running on real hardware. As a result, EXPLODE can thoroughly check large amounts of storage system code with little effort.

Running checks on a live, rather than emulated, system has several nice fallouts. Because storage systems already provide many management and configuration utilities, EXPLODE checkers can simply use this pre-built machinery rather than re-implementing or emulating it. It also becomes trivial to check new storage systems: just mount and run them. Finally, any check that can be run on the base system can also be run with EXPLODE.

The final contribution of the paper is an experimental evaluation of EXPLODE that shows the following:

1. EXPLODE checkers are effective (§7—§9). We found bugs in every system we checked, 36 bugs in total, typically with little effort, and often without source code (§8.1, §9.3). Checking without source code is valuable, since many robust systems rely on third-party software that must be vetted in the context of the integrated system.
2. EXPLODE checkers have enough power to do thorough checks, demonstrated by using it to comprehensively check ten Linux file systems (§7).
3. Even simple checkers find bugs (§8). Tiny checkers found bugs in three version control systems (§8.1) and a widely-used database (§8.2).
4. EXPLODE makes it easy to check subsystems designed to transparently slip into storage stacks (§9). We reused file system checkers to quickly find errors in RAID (§9.1), NFS (§9.2), and VMware (§9.3), which should not (but do) break the behavior of storage systems layered above or below them.

The paper is organized as follows. We first state our principles (§2) and then show how to use EXPLODE to check an example storage system stack (§3). We then give an overview of EXPLODE (§4) and focus on how it: (1) explores alternative actions in checked code (§5) and (2) checks crashes (§6). After the experimental evaluation (§7—§9), we discuss our experiences porting EXPLODE to FreeBSD (§ 10), contrast with related work (§11), and then conclude (§12).

2 Principles

In a sense, this entire paper boils down to the repeated application of a single principle:

Explore all choices: When a program point can legally do one of `N` different actions, fork execution `N` times and do each. For example, the kernel memory allocator can return `NULL`, but rarely does so in practice. For each call to this allocator we want to fork and do both actions. The next principle feeds off of this one:

Exhaust states: Do every possible action to a state before exploring another state. In our context, a state is defined as a snapshot of the system we check.

We distilled these two principles after several years of using model checking to find bugs. Model checking has a variety of tricks, some exceptionally complex. In retrospect, these capture the one feature of a model checking approach that we would take over all others: systemat-

ically do every legal action to a state, missing nothing, then pick another state, and repeat. This approach reliably finds interesting errors, even in well-tested code. We are surprised when it does not work. The key feature of this principle over traditional testing is that it makes low-probability events (such as crashes) as probable as high-probability events, thereby quickly driving the checked system into tricky corner-cases. The final two principles come in reaction to much of the pain we had with naive application of model checking to large, real systems.

Touch nothing. Almost invariably, changing the behavior of a large checked system has been a direct path to experiences that we never want to repeat. The internal interfaces of such systems are often poorly defined. Attempting to emulate or modify them produces corner-case mistakes that model checking is highly optimized to detect. Instead we try to do everything possible to run the checked system as-is and parasitically gather the information we need for checking as it runs.

Report only true errors, deterministically. The errors our system flags should be real errors, reduced to deterministic, replayable traces. All checking systems share this motherhood proclamation, but, in our context it has more teeth than usual: diagnosing even deterministic, replayable storage errors can take us over a day. The cost of a false one is enormous, as is the time needed to fight with any non-determinism.

3 How to Check a Storage System

This section shows how clients use EXPLODE interfaces to check a storage system, using a running example of a simple file system checker. Clients use EXPLODE to do two main things to a storage system. First, systematically exhaust all possibilities when the checked system can do one of several actions. Second, check that it correctly recovers from a crash. Clients can also check non-crash properties by simply inserting code to do so in either their checker or checked code itself without requiring EXPLODE support (for an example see §7.2).

Below, we explain how clients expose decision points in the checked code (§ 3.1). We then explain the three system-specific components that clients provide (written in C++). One, a *checker* that performs storage system operations and checks that they worked correctly (§3.2). Two, a *storage component* that sets up the checked system (§3.3). Finally, a *checking stack* that combines the first two into a checking harness (§3.4).

3.1 How checked code exposes choice: `choose`

Like prior model checkers [13, 30], EXPLODE provides a function, `choose`, that clients use to select among possible choices in checked code. Given a program point that has `N` possible actions clients insert a call "`choose(N)`," which will appear to fork execution `N` times, returning the values $0, 1, ..., N-1$ in each child execution respectively. They then write code that uses this return value to pick one unique action out of the `N` possibilities. EXPLODE can exhaust all possible actions at this `choose` call by running all forked children. We define a code location that can pick one of several different legal actions to be a *choice point* and the act of doing so a *choice*.

An example: in low memory situations the Linux `kmalloc` function can return `NULL` when called without the `__GFP_NOFAIL` flag. But it rarely does so in practice, making it difficult to comprehensively check that callers correctly handle this case. We can use `choose` to systematically explore both success and failure cases of each `kmalloc` call as follows:

```
void * kmalloc(size_t size, int flags) {
    if((flags & __GFP_NOFAIL) == 0)
        if(choose(2) == 0)
            return NULL;
    ...
```

Typically clients add a small number of such calls. On Linux, we used `choose` to fail six kernel functions: `kmalloc` (as above), `page_alloc` (page allocator), `access_ok` (verify user-provided pointers), `bread` (read a block), `read_cache_page` (read a page), and `end_request` (indicate that a disk request completed). The inserted code mirrors that in `kmalloc`: a call `choose(2)` and an if-statement to pick whether to either (0) return an error or (1) run normally.

3.2 Driving checked code: The checker

The client provides a checker that EXPLODE uses to drive and check a given storage system. The checker implements five methods:

1. `mutate`: performs system-specific operations and calls into EXPLODE to explore choices and to do crash checking.
2. `check`: called after each EXPLODE-simulated crash to check for storage system errors.
3. `get_sig`: an optional method which returns a byte-array signature representing the current state of the checked system. It uses domain-specific knowledge to discard irrelevant details so that EXPLODE knows when two superficially different states are equivalent and avoids repeatedly checking them. The default `get_sig` simply records all choices made to produce the current state.
4. `init` and `finish`: optional methods to set up and clear the checker's internal state, called when EXPLODE mounts and unmounts the checked system.

```
1 : const char *dir = "/mnt/sbd0/test-dir";
2 : const char *file = "/mnt/sbd0/test-file";
3 : static void do_fsync(const char *fn) {
4 :     int fd = open(fn, O_RDONLY);
5 :     fsync(fd);
6 :     close(fd);
7 : }
8 : void FsChecker::mutate(void) {
9 :     switch(choose(4)) {
10:     case 0: systemf("mkdir %s%d", dir, choose(5)); break;
11:     case 1: systemf("rmdir %s%d", dir, choose(5)); break;
12:     case 2: systemf("rm %s", file); break;
13:     case 3: systemf("echo \"test\" > %s", file);
14:       if(choose(2) == 0)
15:         sync();
16:       else {
17:         do_fsync(file);
18:         // fsync parent to commit the new directory entry
19:         do_fsync("/mnt/sbd0");
20:       }
21:       check_crash_now(); // invokes check() for each crash
22:       break;
23:     }
24: }
25: void FsChecker::check(void) {
26:     ifstream in(file);
27:     if(!in)
28:       error("fs", "file gone!");
29:     char buf[1024];
30:     in.read(buf, sizeof buf);
31:     in.close();
32:     if(strncmp(buf, "test", 4) != 0)
33:       error("fs", "wrong file contents!");
34: }
```

Figure 2: Example file system checker. We omit the class initialization code and some sanity checks.

Checkers range from aggressively system-specific (or even code-version specific) to the fairly generic. Their size scales with the complexity of the invariants checked, from a few tens to many thousands of lines.

Figure 2 shows a file system checker that checks a simple correctness property: a file that has been synchronously written to disk (using either the `fsync` or `sync` system calls) should persist after a crash. Mail servers, databases and other application storage systems depend on this behavior to prevent crash-caused data obliteration. While simple, the checker illustrates common features of many checkers, including the fact that it catches some interesting bugs.

The `mutate` method calls `choose(4)` (line 9) to fork and do each of four possible actions: (1) create a directory, (2) delete it, (3) create a test file, or (4) delete it. The first two actions then call `choose(5)` and create or delete one of five directories (the directory name is based on `choose`'s return value). The file creation action calls `choose(2)` (line 14) and forces the test file to disk using `sync` in one child and `fsync` in the other. As Figure 3 shows, one `mutate` call creates thirteen children.

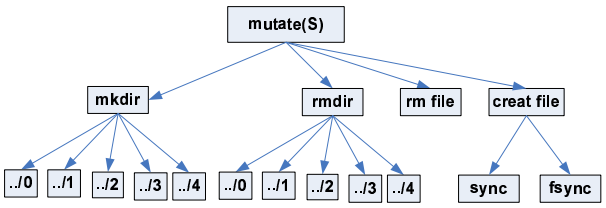


Figure 3: Choices made by one invocation of the `mutate` method in Figure 2's checker. It creates thirteen children.

The checker calls EXPLODE to check crashes. While other code in the system can also initiate such checking, typically it is the `mutate` method's responsibility: it issues operations that change the storage system, so it knows the correct system state and when this state changes. In our example, after `mutate` forces the file to disk it calls the EXPLODE routine `check_crash_now()`. EXPLODE then generates all crash disks at the exact moment of the call and invokes the `check` method on each after repairing and mounting it using the underlying storage component (see § 3.3). The `check` method checks if the test file exists (line 27) and has the right contents (line 32). While simple, this exact checker catches an interesting bug in JFS where upon crash, an `fsync`'d file loses all its contents triggered by the corner-case reuse of a directory inode as a file inode (§7.3 discusses a more sophisticated version of this checker).

So far we have described how a single `mutate` call works. The next section shows how it fits in the checking process. In addition, checking crashes at only a single code point is crude; Section 6 describes the routines EXPLODE provides for more comprehensive checking.

3.3 Setting up checked code: Storage components

Since EXPLODE checks live storage systems, these systems must be up and running. For each storage subsystem involved in checking, clients provide a storage component that implements five methods:

1. `init`: one-time initialization, such as formatting a file system partition or creating a fresh database.
2. `mount`: set up the storage system so that operations can be performed on it.
3. `unmount`: tear down the storage system; used by EXPLODE to clear the storage system's state so it can explore a different one (§5.2).
4. `recover`: repair the storage system after an EXPLODE-simulated crash.
5. `threads`: return the thread IDs for the storage system's kernel threads. EXPLODE reduces nondeterminism by only running these threads when it wants to (§5.2).

```
void Ext3::init(void) {
    // create an empty ext3 FS with user-specified block size
    systemf("mkfs.ext3 -F -j -b %d %s",
      get_option(blk_size), children[0]->path());
}
void Ext3::recover() {
  systemf("fsck.ext3 -y %s", children[0]->path());
}
void Ext3::mount(void) {
    int ret = systemf("sudo mount -t ext3 %s %s",
        children[0]->path(), path());
    if(ret < 0) error("Corrupt FS: Can't mount!");
}
void Ext3::umount(void) {
  systemf("sudo umount %s", path());
}
void Ext3::threads(threads_t &thids) {
    int thid;
    if((thid=get_pid("kjournald")) != -1)
        thids.push_back(thid);
    else
        explode_panic("can't get kjournald pid!");
}
```

Figure 4: Example storage component for the `ext3` file system. The C++ class member `children` chains all storage components that a component is based on; `ext3` has only one child.

Clients write a component once for a given storage system and then reuse it in different checkers. Storage systems tend to be easy to set up, otherwise they will not get used. Thus, components tend to be simple and small since they can merely wrap up already-present system commands (e.g., shell script invocations).

Figure 4 shows a storage component for the `ext3` file system that illustrates these points. Its first four methods call standard `ext3` commands. The one possibly non-obvious method is `threads`, which returns the thread ID of `ext3`'s kernel thread (`kjournald`) using the expedient hack of calling the built-in EXPLODE routine `get_pid` which automatically extracts this ID from the output of the `ps` command.

3.4 Putting it all together: The checking stack

The checking stack builds a checker by glueing storage system components together and then attaching a single checker on top of them. The lowest component of a checking stack typically is a custom RAM disk (downloaded from [24] and slightly modified). While EXPLODE runs on real disks, using a RAM disk avoids non-deterministic interrupts and gives EXPLODE precise, fast control over the contents of a checked system's "persistent" storage. The simplest storage stack attaches a checker to one EXPLODE RAM disk. Such a stack does no useful crash checking, so clients typically glue one or more storage subsystems between these two. Currently a stack can only have one checker. However, there can be a fan-out of storage components, such as setting up multiple RAM disks to make a RAID array. Given a stack, EXPLODE initializes the checked storage stack by calling each `init` bottom up, and then `mount` bottom up. After a crash, it calls the `recover` methods bottom up as well. To unmount, EXPLODE applies `unmount` top down. Figure 5 shows a three-layer storage stack.

```
// Assemble FS + RAID storage stack step by step.
void assemble(Component *&top, TestDriver *&driver) {
  // 1. load two RAM disks with size specified by user
  ekm_load_rdd(2, get_option(rdd, sectors));
  Disk *d1 = new Disk("/dev/rdd0");
  Disk *d2 = new Disk("/dev/rdd1");

  // 2. plug a mirrored RAID array onto the two RAM disks.
  Raid *raid = new Raid("/dev/md0", "raid1");
  raid->plug_child(d1);
  raid->plug_child(d2);

  // 3. plug an ext3 system onto RAID
  Ext3 *ext3 = new Ext3("/mnt/sbd0");
  ext3->plug_child(raid);
  top = ext3; // let eXplode know the top of storage stack

  // 4. attach a file system test driver onto ext3 layer
  driver = new FsChecker(ext3);
}
```

Figure 5: Checking stack: file system checker (Figure 2) on an ext3 file system (Figure 4) on a mirrored RAID array on two EXPLODE RAM disks. We elide the trivial class definitions `Raid` and `Disk`.

4 Implementation Overview

This section gives an overview of EXPLODE. The next two sections discuss the implementation of its most important features: choice and crash checking.

The reader should keep in mind that conceptually what EXPLODE does is very simple. If we assume infinite resources and ignore some details, the following would approximate its implementation:

1. Create a clean initial state (§3.3) and invoke the client's `mutate` on it.
2. At every `choose(N)` call, fork `N` children.
3. On client request, generate all crash disks and run the client `check` method on them.
4. When `mutate` returns, re-invoke it.

This is it. The bulk of EXPLODE is code for approximating this loop with finite resources, mainly the machinery to save and restore the checked system so it can run one child at a time rather than an exponentially increasing number all-at-once. As a result, EXPLODE unsurprisingly looks like a primitive operating system: it has a queue of saved processes, a scheduler that picks which of these jobs to run, and time slices (that start when `mutate` is invoked and end when it returns). EXPLODE's scheduling algorithm: exhaust all possible combinations of choices within a single `mutate` call be-

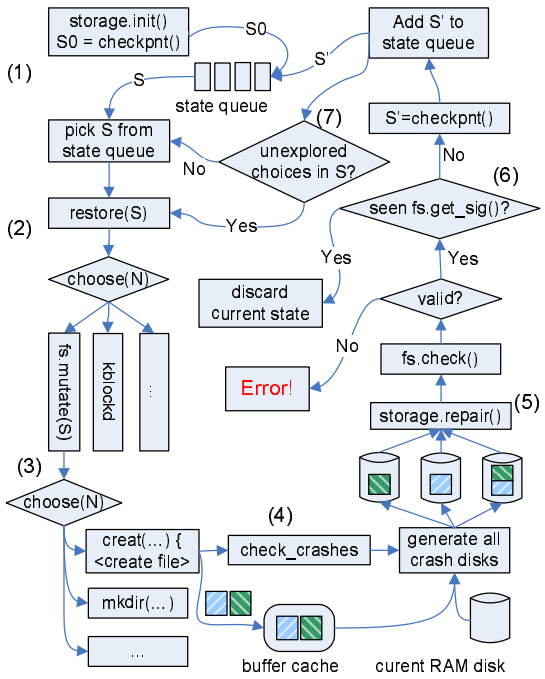


Figure 6: Simplified view of EXPLODE's state exploration loop for the file system checker in Figure 2; some `choose` transitions and method calls elided for space.

fore doing another (§ 2). (Note that turning EXPLODE into a random testing framework is easy: never save and restore states and make each `choose(N)` call return a random integer $[0, N)$ rather than forking, recording each choice for error replay.) The above sketch glosses over some important details; we give a more accurate description below, but the reader should keep this helpful, simplistic one in mind.

From a formal method's perspective, the core of EXPLODE is a simple, standard model checking loop based on exhausting state choices. Figure 6 shows this view of EXPLODE as applied to the file system checker of the previous section; the numbered labels in the figure correspond to the numbers in the list below:

1. EXPLODE initializes the checked system using client-provided `init` methods. It seeds the checking process by saving this state and putting it on the state queue, which holds all states (jobs) to explore. It separately saves the created disk image for use as a pristine initial disk.
2. The EXPLODE "scheduler" selects a state S from its state queue, restores it to produce a running storage system, and invokes `choose` to run either the `mutate` method or one of the checked systems' kernel threads. In the figure, `mutate` is selected.
3. `mutate` invokes `choose` to pick an action. In our example it picks `creat` and calls it, transferring control to the running Linux kernel. The `creat` system call writes two dirty blocks to the buffer cache and returns back to `mutate`.
4. `mutate` calls EXPLODE to check that the file system correctly recovers from any crash at this point.
5. EXPLODE generates combinations of disks that could be seen after a crash. It then runs the client code to: `mount` the crash disk, `recover` it, and `check` it. If these methods flag an error or they crash, EXPLODE records enough information to recreate this error, and stops exploring this state.
6. Otherwise EXPLODE returns back into `mutate` which in turn returns. EXPLODE checks if it has already seen the current state using the abstracted representation returned by `get_sig`. If it has, it discards the state to avoid redundant work, otherwise it checkpoints it and puts it on the state queue.
7. EXPLODE then continues exploring any remaining choices in the original state S. If it has exhausted all choice combinations on S it picks a previously saved state off the state queue and repeats this process on it. This loop terminates when the state queue is empty or the user loses patience. (The number of possible states means the former never happens.)

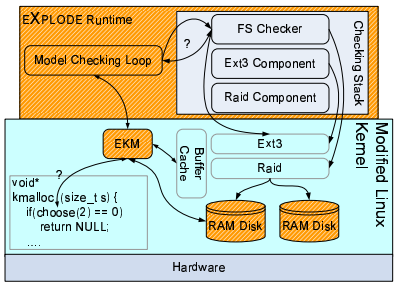


Figure 7: Snapshot: EXPLODE with Figure 5's checking stack

After crash checking, the checked system may have a butchered internal state. Thus, before continuing, EXPLODE restores a clean copy of the current state without doing crash checking (not pictured). In addition, since checking all possible crash disks can take too long, users can set a deterministic threshold: if the number of crash disks is bigger than this threshold, EXPLODE checks a configurable number of random combinations.

Figure 7 gives a snapshot of EXPLODE; Table 1 breaks down the lines of code for each of the components. It consists of two user-level pieces: a client-provided checking stack and the EXPLODE runtime, which implements most of the model checking loop described above. EXPLODE also has three kernel-level pieces: (1) one or more RAM disks, (2) a custom kernel module, EKM, and (3) a modified Linux kernel (either version 2.6.11 or 2.6.15). EXPLODE uses EKM to monitor and determinis-

	Name	Line Count
Linux	EKM	1,261
	RAM disk Driver	326
	Kernel Patch	328
	EKM-generated	2,194
BSD	EKM	729
	RAM disk Driver	357
	Kernel Patch	116
User-mode	EXPLODE	5,802
	RPC Library	521

Table 1: EXPLODE lines of code (ignoring comments and blank lines), broken down by modules. The EKM driver contains 2,194 lines of automatically generated code (**EKM-generated**). The EXPLODE runtime and the RPC library run at user-level, the rest is in the kernel. The RPC library is used to check virtual machines (§ 9.3). **BSD** counts are smaller because this port does not yet provide all EXPLODE features.

tically control checking-relevant actions done by kernel code and record system events needed for crashes. The modified kernel calls EKM to log system events and when it reaches a choice point. These modifications add 328 lines of mostly read-only instrumentation code, typically at function entry or exit. We expect them to generally be done by EXPLODE users. Unlike EXPLODE's user-space code, its RAM disk driver and EKM are kernel-specific, but are fairly small and easily ported to a new OS. We recently ported EXPLODE's core to FreeBSD, which Section 10 describes in more detail.

Given all of these pieces, checking works as follows. First, the user compiles and links their code against the EXPLODE runtime, and runs the resultant executable. Second, the EXPLODE runtime dynamically loads its kernel-level components and then initializes the storage system. Finally, EXPLODE explores the checked system's states using its model checking loop.

While checking a live kernel simplifies many things, the downside is that many bugs we find with EXPLODE cause kernel crashes. Thus, we run the checked system inside a virtual machine monitor (VMware Workstation), where it can blow itself up without hurting anyone. This approach also makes checking a non-super-user operation, with the usual benefits.

5 Exploring Choices

EXPLODE exhausts a choice point by checkpointing the current state `S`, exploring one choice, restoring `S`, and then exploring the other choices. Below we discuss how EXPLODE implements checkpoint and restore by replaying choices (§ 5.1) deterministically (§ 5.2).

5.1 Checkpointing and restoring states.

A standard checkpoint implementation would copy the current system state to a temporary buffer, which restore would then copy back. Our previous storage checking system, FiSC, did just this [30]. Unfortunately, one cannot simply save and restore a kernel running on raw hardware, so we had to instead run a heavily-hacked Linux kernel inside FiSC at user level, turning FiSC into a primitive virtual machine. Doing so was the single largest source of FiSC complexity, overhead to check new systems, and limitation on what we could check.

EXPLODE uses computation rather than copying to recreate states. It checkpoints a state `S` by recording the set of choices the checked code took to reach `S`. It restores `S` by starting from a clean initial state and replaying these choices. Thus, assuming deterministic actions, this method regenerates `S`. Mechanically, checkpoint records the sequence of n choices that produced `S` in an array; during replay the ith `choose` call simply returns the ith entry in this array.

This one change led to orders of magnitude reduction in complexity and effort in using EXPLODE as opposed to FiSC, to the degree that EXPLODE completely subsumes our prior work in almost every aspect by a large amount. It also has the secondary benefit that states have a tiny representation: a sequence of integers, one for each choice point, where the integer specifies which of N choices were made. Note that some model checkers (and systems in other contexts [10]) already use replay-recreation of states, but for error reporting and state size reduction, rather than for reducing invasiveness. One problem with the approach is that the restored state's timestamps will not match the original, making it harder to check some time properties.

Naively, it might seem that to reset the checked systems' state we have to reboot the machine, re-initialize the storage system, mount it, and only then replay choices. This expensive approach works, but fortunately, storage systems have the observed, nice property that simply unmounting them clears their in-memory state, removing their buffer cache entries, freeing up their kernel data structures, etc. Thus, EXPLODE uses a faster method: call the client-supplied `unmount` to clear the current state, then load a pristine initial state (saved after initialization) using the client-supplied `mount`.

It gets more costly to restore states as the length of their choice sequence grows. Users can configure EXPLODE to periodically chop off the prefix of choice sequences. It does so by (1) calling `unmount` to force the checked system state to disk and (2) using the resultant disk image as a new initial state that duplicates the effect of the choices before the `unmount` call. The downside is that it can no longer reorder buffer cache entries from before this point during crash checking.

5.2 Re-executing code deterministically

EXPLODE's restore method only works if it can deterministically replay checked code. We discuss how EXPLODE does so below, including the restrictions imposed on the checked system.

Doing the same choices. Kernel code containing a `choose` call can be invoked by non-checking code, such as interrupt handlers or system calls run by other processes. Including such calls makes it impossible to replay traces. EXPLODE filters them by discarding any calls from an interrupt context or calls from any process whose ID is not associated with the checked system.

Controlling threads. EXPLODE uses priorities to control when storage system threads run (§ 4, bullet 2). It quiesces storage system threads by giving them the lowest priority possible using an EKM `ioctl`. It runs a thread by giving it a high priority (others still have the lowest) and calling the kernel scheduler, letting it schedule the right thread. It might seem more sensible for EXPLODE to orchestrate thread schedules via semaphores. However, doing so requires intrusive changes and, in our experience [30], backfires with unexpected deadlock since semaphores prevent a given thread from running even if it absolutely must. Unfortunately, using priorities is not perfect either, and still allows non-deterministic thread interleaving. We detect pathological cases where a chosen thread does not run, or other "disabled" threads do run using the "last-run" timestamps in the Linux process data structure. These sanity checks let us catch when we generate an error trace that would not be replayable or when replaying it takes a different path. Neither happens much in practice.

Requirements on the checked system. The checked system must issue the same `choose` calls across replay runs. However, many environmental features can change across runs, providing many sources of potential non-deterministic input: thread stacks in different locations, memory allocations that return different blocks, data structures that have different sizes, etc. None of these perturbations should cause the checked code to behave differently. Fortunately, the systems we checked satisfy this requirement "out of the box" — in part because they are isolated during checking, and nothing besides the checker and their kernel threads call into them to modify their RAM disk(s). Non-deterministic systems require modification before EXPLODE can reliably check them. However, we expect such cases to rarely occur. If nothing else, usability forces systems to ensure that re-executing the same user commands produces the same system state. As a side-effect, they largely run the same code paths (and thus would hit the same `choose` calls).

While checked code must do the same `choose` calls for deterministic error replay, it does not have to allocate the same physical blocks. EXPLODE replays choices, but then regenerates all different crash combinations after the last choice point until it (re)finds one that fails checking. Thus, the checked code can put logical contents in different physical blocks (e.g., an inode resides in disk block 10 on one run and in block 20 on another) as long as the logical blocks needed to cause the error are still marked as dirty in the buffer cache.

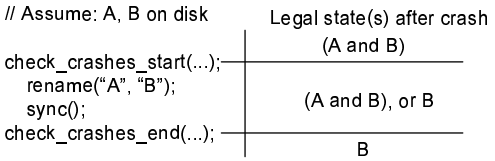
6 Checking Crashes

This section discusses crash checking issues: EXPLODE's checking interface (§ 6.1), how it generates crash disks (§ 6.2), how it checks crashes during recovery (§ 6.3), how it checks for errors caused by application crashes (§ 6.4), and some refinements (§ 6.5).

6.1 The full crash check interface

The `check_crashes_now()` routine is the simplest way to check crashes. EXPLODE also provides a more powerful (but complex) interface clients can use to directly inspect the log EXPLODE extracts from EKM. They can also add custom log records. Clients use the log to determine what state the checked system should recover to. They can initiate crash checking at any time while examining the log. For space reasons we do not discuss this interface further, though many of our checkers use it. Instead we focus on two simpler routines `check_crashes_start` and `check_crashes_end` that give most of the power of the logging approach.

Clients call `check_crashes_start` before invoking the storage system operations they want to check and `check_crashes_end` after. For example, assume we want to check if we can atomically rename a file `A` to `B` by calling `rename` and then `sync()`. We could write the following code in `mutate`:



EXPLODE generates all crash disks that can occur (inclusively) between these calls, invoking the client's `check` method on each. Note how the state the system should recover to changes. At the `check_crashes_start` call, the recovered file system should contain both `A` and `B`. During the process of renaming, the recovered file system can contain either (1) the original `A` and `B` or (2) `B` with `A`'s original contents. After `sync` completes, only `B` with `A`'s original contents should exist.

This pattern of having an initial state, a set of legal intermediate states, and a final state is a common one for checking. Thus, EXPLODE makes it easy for `check` to distinguish between these epochs by passing a flag that tells `check` if the crash disk could occur at the first call (`EXP_BEGIN`), the last call (`EXP_END`), or in between (`EXP_INBETWEEN`). We could write a check method to use these flags as follows:

```
check(int epoch, ...) {
    if(epoch == EXP_BEGIN)
        // check (A and B)
    else if(epoch == EXP_INBETWEEN)
        // check (A and B) or B
    else //     EXP_END
        // check B
}
```

EXPLODE uses C++ tricks so that clients can pass an arbitrary number of arguments to these two routines (up to a user-specified limit) that in turn get passed to their `check` method.

6.2 Generating crash disks

EXPLODE generates crash disks by first constructing the current *write set*: the set of disk blocks that currently *could be* written to disk. Linux has over ten functions that affect whether a block can be written or not. The following two representative examples cause EXPLODE to add blocks to the write set:

1. `mark_buffer_dirty(b)` sets the dirty flag of a block `b` in the buffer cache, making it eligible for asynchronous write back.
2. `generic_make_request(req)` submits a list of sectors to the disk queue. EXPLODE adds these sectors to the write set, even if they are clean, which can happen for storage systems maintaining their own private buffer caches (as in the Linux port of XFS).

The following three representative examples cause EXPLODE to remove blocks from the write set:

1. `clear_buffer_dirty(b)` clears `b`'s dirty flag. The buffer cache does not write clean buffers to disk.
2. `end_request()`, called when a disk request completes. EXPLODE removes all versions of the request's sectors from the write set since they are guaranteed to be on disk.
3. `lock_buffer(b)`, locks `b` in memory, preventing it from being written to disk. A subsequent `clear_buffer_locked(b)` will add `b` to the write set if `b` is dirty.

Writing any subset of the current write set onto the current disk contents generates a disk that could be seen if the system crashed at this moment. Figure 8 shows how EXPLODE generates crash disks; its numbered labels correspond to those below:

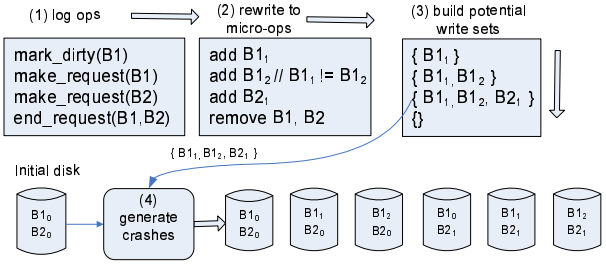


Figure 8: Generating all potential crash disks.

1. As the storage system executes, EKM logs operations that affect which blocks could be written to disk.
2. EXPLODE extracts this log using an EKM `ioctl` and reduces the logged operations to micro-operations that add or remove blocks from the write set.
3. It then applies these add and remove operations, in order, to the initial write set.
4. Whenever the write set shrinks, it generates all possible crash disks by applying all subsets of the write set to the current disk. (Doing so when the write set shrinks rather than grows makes it trivial to avoid duplicate work.)

Note that the write set tracks a block's contents in addition to the block itself. Naively it may appear that when EXPLODE adds a block b to the write set it should replace any previous copy of b with this more recent one. (Our previous work [30] did exactly this.) However, doing so misses errors. For example, in the figure, doing so misses one crash disk $(B1_1, B2_1)$ since the second insertion of block $B1$ replaces the previous version $B1_1$ with $B1_2$.

6.3 Checking crashes during recovery

Clients can also use EXPLODE to check that storage systems correctly handle crashes during recovery. Since environmental failures are correlated, once one crash happens, another is not uncommon: power may flicker repeatedly in a storm or a machine may keep rebooting because of a bad memory board. EXPLODE generates the disks that could occur if recovery crashes, by tracking the write set produced while running `recover`, and then applying all its subsets to the initial crash disk. It checks these "crash-crash" disks as it would a crash disk. Note this assumes recovery is idempotent in that if a correct recovery with no crash produces state S_{valid} then so should a prematurely crashed repair followed by a successful one. We do not (but could) check for further crashes during recovery since implementors seem uninterested in such errors [30].

6.4 Checking "soft" application crashes

In addition to "hard" machine crashes that wipe volatile state, EXPLODE can also check that applications cor-

rectly recover from "soft" crashes where they crashed, but the operating system did not. Such soft crashes are usually more frequent than hard crashes with causes ranging from application bugs to impatient users pressing "ctrl-C." Even applications that ignore hard crashes should not corrupt user data because of a soft crash.

EXPLODE checks soft crashes in two steps. First, it runs the checker's `mutate` method and logs all mutating file system operations it performs. Second, for each log prefix EXPLODE mounts the initial disk and replays the operations in the prefix in the order they are issued. If the log has n operations EXPLODE generates n storage states, and passes each to the `check` method.

6.5 Refinements

In some cases we remove blocks from the write set too eagerly. For example, we always remove the sectors associated with `end_request`, but doing so can miss permutations since subsequent writes may not in fact have waited for (depended on) the write to complete. Consider the events: (1) a file system writes sector S1, (2) the write completes, (3) it then writes sector S2. If the file system wrote S2 without explicitly waiting for the S1 write to complete then these writes could have been reordered (i.e., there is no happens-before dependency between them). However, we do not want to grovel around inside storage systems rooting out these false dependencies, and conservatively treat all writes that complete as waited for. A real storage system implementor could obviously do a better job.

To prevent the kernel from removing buffers from the write set, we completely disable the dirty buffer flushing threads `pdflush`, and only schedule the kernel thread `kblockd` that periodically flushes the disk queue between calls to the client `mutate` method.

If a checked system uses a private buffer cache, EXPLODE cannot see all dirty blocks. We partially counter this problem by doing an unmount before generating crash disks, which will flush all private dirty buffers to disk (when EXPLODE can add them to its write set). Unfortunately, this approach is not a complete solution since these unmount-driven flushes can introduce spurious dependencies (as we discussed above).

7 In-Depth Checking: File Systems

This section demonstrates that EXPLODE's lightweight approach does not compromise its power by replicating (and sometimes superseding) the results we obtained with our previous, more strenuous approach [30]. It also shows EXPLODE's breadth by using it to check ten Linux file systems with little incremental effort.

We applied EXPLODE to all but one of the disk based file systems on Linux 2.6.11: ext2, ext3, JFS, ReiserFS, Reiser4, XFS, MSDOS, VFAT, HFS, and HFS+. We skipped NTFS because repairing a crashed NTFS disk requires mounting it in Windows. For most file systems, we used the most up-to-date utilities in the Debian "etch" Linux distribution. For HFS and HFS+, we had to download the source of their utilities from OpenDarwin [14] and compile it ourselves. The storage components for these file systems mirror `ext3`'s component (§ 3.3). Four file systems use kernel threads: JFS, ReiserFS, Reiser4 and XFS. We extracted these thread IDs using the same trick as with `ext3`.

While these file systems vary widely in terms of implementation, they are identical in one way: none give clean, precise guarantees of the state they recover to after a crash. As a result, we wrote three checkers that focused on different special cases where what they did was somewhat well-defined. We built these checkers by extending a common core, which we describe below. We then describe the checkers and the bugs they found.

7.1 The generic checker core

The basic checker starts from an empty file system and systematically generates file system topologies up to a user-specified number of files and directories. Its `mutate` exhaustively applies each of the following eight system calls to each node (file, link, directory) in the current topology before exploring the next: `ftruncate`, `pwrite` (which writes to a given offset within a file), `creat`, `mkdir`, `unlink`, `rmdir`, `link` and `rename`. For example, if there are two leaf directories, the checker will delete both, create files in both, etc. Thus, the number of possible choices for a given tree grows (deterministically) with its size. For file systems that support holes, the checker writes at large offsets to exercise indirect blocks. Other operations can easily be added.

For each operation it invokes, `mutate` duplicates its effect on a fake "abstract" file system it maintains privately. For example, if it performs three operations `mkdir(/a)`, `mkdir(/a/b)`, and `sync()` then the abstract file system will be the tree `/a/b`, which the real file system must match exactly. The checker's `get_sig` method returns a canonical version of this abstract file system. This canonicalization mirrors that in [30], and uses relabeling to make topologies differing only in naming equivalent and discards less interesting properties such as timestamps, actual disk blocks used, etc.

7.2 Check: Failed system calls have no effect

This check does not involve crash-recovery. It checks that if a file system operation (except `pwrite`) returns an error, the operation has no user-visible effect. It uses

EXPLODE to systematically fail calls to the six kernel functions discussed in Section 3.1. The actual check uses the abstract file system described in the previous subsection. If a system call succeeds, the checker updates the abstract file system, but otherwise does not. It then checks that the real file system matches the abstract one.

Bugs found. We found 2 bugs in total. One of them was an unfixed Linux VFS bug we already reported in [30]. The other one was a minor bug in ReiserFS `ftruncate` which can fail with its job half-done if memory allocation fails. We also found that Reiser4 calls `panic` on memory allocation failures, and ReiserFS calls `panic` on disk read failures. (We did not include these two undesired behaviors in our bug counts.)

7.3 Check: "sync" operations work

Applications such as databases and mail servers use operating system-provided methods to force their data to disk in order to prevent crashes from destroying or corrupting it. Unfortunately, they are completely at these routines' mercy — there is no way to check they do what they claim, yet their bugs can be almost arbitrarily bad.

Fortunately, EXPLODE makes it easy to check these operations. We built a checker (similar to the one in Figure 2) to check four methods that force data to disk:

1. `sync` forces all dirty buffers to disk.
2. `fsync(fd)` forces `fd`'s dirty buffers to disk.
3. Synchronously mounted file system: a system call's modifications are on disk when the call returns.
4. Files opened with `O_SYNC`: all modifications done by a system call through the returned file descriptor are on disk when the call returns.

After each operation completes and its modifications have been forced to disk, the sync-checker tells EXPLODE to do crash checking and verifies that the modifications persist.

Note, neither `fsync` nor `O_SYNC` guarantee that directory entries pointing to the sync'd file are on disk, doing so requires calling `fsync` on any directory containing the file (a legal operation in Linux). Thus, the checker does an `fsync` on each directory along the path to the sync'd file, ensuring there is a valid path to it in the recovered file system.

Bugs found. Table 2 summarizes the 13 bugs found with this checker. Three bugs show up in multiple ways (but are only counted three times): a VFS limitation caused all file systems to fail the `O_SYNC` check, and both HFS and HFS+ mangled file and directory permissions after crashing, therefore failing all four sync checks. We describe a few of the more interesting bugs below.

Besides HFS/HFS+, both MSDOS and VFAT mishandled `sync`. Simple crashes after `sync` can introduce directory loops. The maintainers confirmed they knew of these bugs, though they had not been publicly disclosed. These bugs have subsequently been fixed. Eight file systems had synchronous mount bugs. For example, `ext2` gives no consistency guarantees by default, but mounting it synchronously still allows data loss.

FS	`sync`	`mount sync`	`fsync`	`O_SYNC`
ext2		✘	✘	✘
ext3				✘
ReiserFS		✘		✘
Reiser4				✘
JFS		✘	✘	✘
XFS		✘		✘
MSDOS	✘	✘		✘
VFAT	✘	✘		✘
HFS	✘	✘	✘	✘
HFS+	✘	✘	✘	✘

Table 2: Sync checking results: ✘ indicates the file system failed the check. There were 13 bugs, three of which show up more than once, causing more ✘ marks than errors.

There were two interesting `fsync` errors, one in JFS (§3.2) and one in `ext2`. The `ext2` bug is a case where an implementation error points out a deeper design problem. The bug occurs when we: (1) shrink a file "A" with `truncate` and (2) subsequently `creat`, `write`, and `fsync` a second file "B." If file B reuses the indirect blocks of A freed via `truncate`, then following a crash `e2fsck` notices that A's indirect blocks are corrupt and clears them, destroying the contents of B. (For good measure it then notices that A and B share blocks and "repairs" B by duplicating blocks from A.) Because ext2 makes no guarantees about what is written to disk, fundamentally one cannot use `fsync` to *safely* force a file to disk, since the file can still have implicit dependencies on other file system state (in our case if it reuses an indirect blocks for a file whose inode has been cleared in memory but not on disk).

7.4 Check: a recovered FS is "reasonable"

Our final check is the most stringent: after a crash a file system recovers to a "reasonable" state. No files, directories, or links flushed to disk are corrupted or disappear (unless explicitly deleted). Nor do they spontaneously appear without being created. For example, if we crash after performing two operations `mkdir(/A)` and `mkdir(/A/B)` on an empty file system, then there are exactly three correct recovered file systems: (1) `/` (no data), (2) `/A`, or (3) `/A/B`. We should not see directories or files we never created. Similarly, if `/A` was forced to disk before the crash, it should still exist.

For space reasons we only give a cursory implementation overview. As `mutate` issues operations, it builds two sets: (1) the stable set, which contains the operations it knows are on the disk, (2) the volatile set, which

contains the operations that may or may not be on disk. The `check` method verifies that the recovered file system can be constructed using some sequence of volatile operations legally combined with all the stable ones. The implementation makes heavy use of caching to prune the search and "desugars" operations such as `mkdir` into smaller atomic operations (in this case it creates an inode and then forms a link to it) to ensure it can describe their intermediate effects.

Bugs found. We applied this check to ext2, ext3, JFS, ReiserFS and Reiser4. Unsurprisingly, since `ext2` gives no crash guarantees, files can point to uninitialized blocks, and sync'd files and directories can be removed by its `fsck`. Since JFS journals metadata but not data, its files can also point to garbage. These behaviors are design decisions so we did not include them in our bug counts. We found two bugs (one in JFS, one in Reiser4) where crashed disks cannot be recovered by `fsck`. We could not check many topologies for ReiserFS and Reiser4 because they appear to leak large amounts of memory on every `mount` and `unmount` (Our bug counts do not include these leaks.)

In addition, we used the crash-during-recovery check (§6.3) on Reiser4. It found a bug where Reiser4 becomes so corrupted that mounting it causes a kernel panic. (Since our prior work explored this check in detail we did not apply it to more systems.)

Finally, we did a crude benchmark run by running the checker (without crash-during-recovery checking) to `ext3` inside a virtual machine with 1G memory on a Intel P4 3.2GHZ with 2G memory. After about 20 hours, EXPLODE checked 230,744 crashes for 327 different FS topologies and 1582 different FS operations. The run died because Linux leaks memory on each `mount` and `unmount` and runs out of memory. Although we fixed two leaks, more remain (we did not count these obliquely-detected errors in our bug counts but were tempted to). We intend to have EXPLODE periodically checkpoint itself so we can reboot the machine and let EXPLODE resume from the checkpoints.

8 Even Simple Checkers Find Bugs

This section shows that even simple checkers find interesting bugs by applying it to three version control systems and the Berkeley DB database.

The next two sections demonstrate that EXPLODE works on many different storage systems by applying it to many different ones. The algorithm for this process: write a quick checker, use it to find a few errors, declare success, and then go after another storage system. In all cases we could check many more invariants. Table 3 summarizes all results.

System	Storage	Checker	Bugs
FS	744	5,477	18
CVS	27	68	1
Subversion	-	-	1
EXPENSIV	30	124	3
Berkeley DB	82	202	6
RAID	144	FS + 137	2
NFS	34	FS	4
VMware GSX/Linux	54	FS	1
Total	1,115	6,008	36

Table 3: Summary of all storage systems checked. All line counts ignore comments and whitespace. **Storage** gives the line count for each system's storage component, which for **FS** includes the components for all ten file systems. **Checker** gives the checker line counts, which for EXPENSIV includes two checkers. We reused the FS checker to check RAID, NFS and VMware. We wrote an additional checker for RAID. We checked Subversion using an early version of EXPLODE; we have not yet ported its component and checker.

8.1 Version control software

This section checks three version control systems: CVS, Subversion [27], and an expensive commercial system we did not have source code for, denoted as EXPENSIV (its license precludes naming it directly). We check that these systems meet their fundamental goal: do not lose or corrupt a committed file. We found errors in all three.

The storage component for each wraps up the commands needed to set up a new repository on top of one of the file systems we check. The checker's `mutate` method checks out a copy of the repository, modifies it, and commits the changes back to the main repository. After this commit completes, these changes should persist after any crash. To test this, `mutate` immediately calls `check_crashes_now()` after the commit completes. The `check` method flags an error if: (1) the version control systems' crash recovery tool (if any) gives an error or (2) committed files are missing.

Bugs found. All three systems made the same mistake. To update a repository file `A` without corrupting it, they first update a temporary file `B`, which they then atomically rename to `A`. However, they forget to force `B`'s contents to disk before the rename, which means a crash can destroy it.

In addition EXPENSIV purports to atomically merge two repositories into one, where any interruption (such as crash) will either leave the two original repositories or one entirely (correctly) merged one. EXPLODE found a bug where a crash during merge corrupts the repository, which EXPENSIV's recovery tool (`EXPENSIV -r check -f`) cannot fix. This error seems to be caused by the same renaming mistake as above.

Finally, we found that even a soft crash during a merge corrupts EXPENSIV's repository. It appears EXPENSIV renames multiple files at the end of the merge. Although

each individual rename is atomic against a soft crash, their aggregation is not. The repository is corrupted if not all files are renamed.

8.2 Berkeley DB

The database checker in this section checks that after a crash no committed transaction records are corrupted or disappear, and no uncommitted ones appear. It found six bugs in Berkeley DB 4.3 [2].

Berkeley DB's storage component only defines the `init` method, which calls Berkeley DB utilities to create a database. It does not require `mount` or `unmount`, and has no threads. It performs recovery when the database is opened with the `DB_RECOVER` flag (in the `check` method). We stack this component on top of a file system one.

The checker's `mutate` method is a simple loop that starts a transaction, adds several records to it, and then commits this transaction. It records committed transactions. It calls `check_crashes_start` before each commit and `check_crashes_end` (§ 6.1) after to verify that there is a one-to-one mapping between the transactions it committed and those in the database.

Bugs found. We checked Berkeley DB on top of ext2, ext3, and JFS. On ext2 creating a database inside a transaction, while supposedly atomic, can lead to a corrupted database if the system crashes before the database is closed or `sync` is manually called. Furthermore, even with an existing database, committed records can disappear during a crash. On ext3 an unfortunate crash while adding a record to an existing database can again leave the database in an unrecoverable state. Finally, on all three file systems, a record that was added but never committed can appear after a crash. We initially suspected these errors came from Berkeley DB incorrectly assuming that file system blocks were written atomically. However, setting Berkeley DB to use sector-aligned writes did not fix the problem. While the errors we find differ depending on the file system and configuration settings, some are probably due to the same underlying problem.

9 Checking "Transparent" Subsystems

Many subsystems transparently slip into a storage stack. Given a checker for the original system, we can easily check the new stack: run the same checker on top of it and make sure it gives the same results.

9.1 Software RAID

We ran two checkers on RAID. The first checks that a RAID transparently extends a storage stack by running the file system sync-checker (§ 7.3) on top of it. A file system's crash and non-crash behavior on top of RAID should be the same as without it: any (new) errors the checker flags are RAID bugs. The second checks that losing any single sector in a RAID1 or RAID5 stripe does not cause data loss [20]. I.e., the disk's contents were always correctly reconstructed from the non-failed disks.

We applied these checks to Linux's software RAID [26] levels 1 and 5. Linux RAID groups a set of disks and presents them as a single block device to the rest of the system. When a block write request is received by the software RAID block device driver, it recomputes the parity block and passes the requests to the underlying disks in the RAID array. Linux RAID repairs a disk using a very simple approach: overwrite all of the disk's contents, rather than just those sectors that need to be fixed. This approach is extremely slow, but also hard to mess up. Still, we found two bugs.

The RAID storage component methods map directly to different options for its administration utility `mdadm`. The `init` method uses `mdadm --create` to assemble either two or four RAM disks into a RAID1 or RAID5 array respectively. The `mount` method calls `mdadm --assemble` on these disks and the `unmount` method tears down the RAID array by invoking `mdadm --stop`. The `recover` method reassembles and recovers the RAID array. We used the `mdadm --add` command to replace failed disks after a disk failure. The checking stack is similar to that in Figure 5.

Bugs found. The checker found that Linux RAID does not reconstruct the contents of an unreadable sector (as it easily could) but instead marks the *entire* disk that contains the bad sector as faulty and removes it from the RAID array. Such a fault-handling policy is not so good: (1) it makes a trivial error enough to prevent the RAID from recovering from *any* additional failure, and (2) as disk capacity increases, the probability that another sector goes bad goes to one.

Given this fault-handling policy, it is unsurprising our checker found that after two sector read errors happen on different disks, requiring manual maintenance, almost all maintenance operations (such as `mdadm --stop` or `mdadm --add`) fail with a "Device or resource busy" error. Disk write requests also fail in this case, rendering the RAID array unusable until the machine is rebooted. One of the main developers confirmed that these behaviors were bad and should be fixed with high priority [4].

9.2 NFS

NFS synchronously forces modifications to disk before requests return [23]. Thus, with only a single client modifying an NFS file system, after a crash NFS must recover to the same file system tree as a local file system mounted synchronously. We check this property by running the

sync-checker (§7.3) on NFS and having it treat NFS as a synchronously mounted file system. This check found four bugs when run on the Linux kernel's NFS (NFSv3) implementation [19].

The NFS storage component is a trivial 15-lines of code (plus a hand-edit of "`/etc/exports`" to define an NFS mount point). It provides two methods: (1) `mount`, which sets up an NFS partition by exporting a local FS over the NFS loop-back interface and (2) `unmount`, which tears down an NFS partition by unmounting it. It does not provide a `recover` method since the `recover` of the underlying local file system must be sufficient to repair crashed NFS partitions. We did not model network failures, neither did we control the scheduling of NFS threads, which could make error replay non-deterministic (but did not for ours).

Bugs found. The checker found a bug where a client that writes to a file and then reads the same file through a hard link in a different directory will not see the values of the first write. We elide the detailed cause of this error for space, other than noting that diagnosing this bug as NFS's fault was easy, because it shows up regardless of the underlying file system (we tried ext2, ext3, and JFS).

We found additional bugs specific to individual file systems exported by NFS. When JFS is exported over NFS, the `link` and `unlink` operations are not committed synchronously. When an ext2 file system is exported over NFS, our checker found that many operations were not committed synchronously. If the NFS server crashes these bugs can lose data and cause data values to go backwards for remote clients.

9.3 VMware GSX server

In theory, a virtual machine slipped beneath a guest OS should not change the crash behavior of a correctly-written guest storage system. Roughly speaking, correctness devolves to not lying about when a disk block actually hits a physical disk. In practice, speed concerns make lying tempting. We check that a file system on top of a virtual machine provided "disk" has the same synchronous behavior as running without it (again) using the sync-checker (§7.3). We applied this check to VMware GSX 3.2.0 [29] running on Linux. GSX is an interesting case for EXPLODE: a large, complex commercial system (for which we lack source code) that, from the point of view of a storage system checker, implements a block device interface in a strange way.

The VMware GSX scripting API makes the storage component easy to build. The `init` method copies a precreated empty virtual disk image onto the file system on top of EXPLODE RAM disk. The `mount` method starts the virtual machine using the command

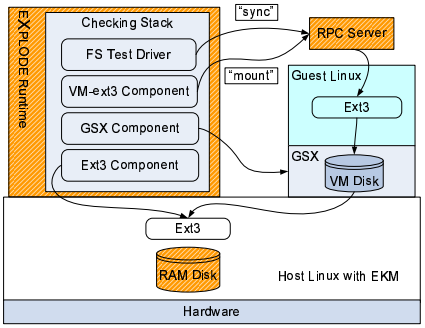


Figure 9: The VMware checking stack.

`vmware-cmd start` and `unmount` stops it using `vmware-cmd stop hard`. The `recover` method calls `vmware-cmd start`, which repairs a crashed virtual machine, and then removes a dangling lock (created by the "crashed" virtual machine to prevent races on the virtual disk file).

As shown in Figure 9 the checking stack was the most intricate of this paper. It has five layers, starting from bottom to top: (1) a RAM disk, (2) the ext3 file system in the host, storing the GSX virtual disk file, (3) GSX, (4) the ext3 file system in the guest, (5) the sync-checker. The main complication in building this stack was the need to split EXPLODE into two pieces, one running in the host, the other in the guest. Since the virtual machine will frequently "crash" we decided to keep the part running inside it simple and make it a stateless RPC server. The entire storage stack and the sync-checker reside in the host. When the sync-checker wants to run an operation in the guest, or a storage method wants to run a utility, they do RPC calls to the server in the guest, which then performs the operation.

Bugs found. Calling `sync` in the guest OS does not correctly flush dirty buffers to disk, but only to the host's buffer cache. According to VMware documents, setting the "disable write caching" configuration flag forces all writes to disk. However, we hit the same bug even with this flag on. This bug makes it impossible to reliably run a storage system on top of this VMM on Linux. We confirmed this problem with one of the main developers who stated that it should not show up in the latest version [28].

10 Checking on a new system: FreeBSD

We ported EXPLODE to FreeBSD 6.0 to ensure porting was easy and to shake out Linux-specific design assumptions. We spent most of our time writing a new RAM disk and EKM module; we only needed to change a few lines in the user-level runtime to run on FreeBSD.

The FreeBSD version of EXPLODE supports crash checking, but currently does not provide a kernel-level `choose` nor logging of system calls. Neither should

present a challenge here or in general. Even without these features, we reproduced the errors in CVS and EXPENSIV we saw on Linux as well as finding new errors in FreeBSD UFS2. Below, we discuss issues in writing EKM and the RAM disk.

EKM. Crash checking requires adding calls to EKM in functions that mark buffers as clean, dirty, or write them to disk. While a FreeBSD developer could presumably enumerate all such functions easily, our complete lack of experience with FreeBSD meant it took us about a week to find all corner-cases. For example, FreeBSD's UFS2 file system sometimes bypasses the buffer cache and writes directly to the underlying disk.

There were also minor system-differences we had to correct for. As an example, while Linux and FreeBSD have similar structures for buffers, they differ in how they store bookkeeping information (e.g., representing offsets in sectors on Linux, and in bytes on FreeBSD). We adjusted for such differences inside EKM so that EXPLODE's user-level runtime sees a consistent interface. We believe porting should generally be easy since EKM only logs the offset, size, and data of buffer modifications, as well as the ID of the modifying thread. All of these should be readily available in any OS.

RAM disk. We built our FreeBSD RAM disk by modifying the `/dev/md` memory-based disk device. We expect developers can generally use this approach: take an existing storage device driver and add trivial `ioctl` commands to read and write its disk state by copying between user- and kernel-space.

Bug-Finding Results. In addition to our quick tests to replicate the EXPENSIV and CVS bugs, we also ran our sync-checker (§7.3) on FreeBSD's UFS2 with soft updates disabled. It found errors where `fsck` with the `-p` option could not recover from crashes. While `fsck` without `-p` could repair the disk, the documentation for `fsck` claims `-p` can recover from all errors unless unexpected inconsistencies are introduced by hardware or software failures. Developers confirmed that this is a problem and should be further investigated.

11 Related Work

Below we compare EXPLODE to file system testing, software model checking, and static bug finding.

File system testing tools. There are many file system testing frameworks that use application interfaces to stress a "live" file system with an adversarial environment. These testing frameworks are less comprehensive than our approach, but they work "out of the box." Thus, there is no reason not to both test a file system and then test with EXPLODE (or vice versa).

Recently, Prabhakaran *et al* [21] studied how file systems handle disk failures and corruption. They developed a testing framework that uses techniques from [25] to infer disk block types and then inject "type-aware" block failure and corruption into file systems. Their results provide motivation for using existing checksum-based file systems (such as Sun's ZFS [32]). While their technique is more precise than random testing, it does not find the crash errors that EXPLODE does, nor is it as systematic. Extending EXPLODE to similarly return garbage on disk reads is trivial.

Software Model Checking. Model checkers have been previously used to find errors in both the design and the implementation of software systems [1, 3, 7, 13, 15, 16, 18, 30]. Two notable examples are Verisoft [13], which systematically explores the interleavings of a concurrent C program, and Java PathFinder [3] which used a specialized virtual machine to check concurrent Java programs by checkpointing states.

The model checking ideas EXPLODE uses — exhausting states, systematic exploration, and choice — are not novel. This paper's conceptual contribution is dramatically reducing the large work factor that plagues traditional model checking. It does so by turning the checking process inside out. It interlaces the control it needs for systematic state exploration *in situ*, throughout the checked system. As far as we know, EXPLODE is the first example of *in situ* model checking. The paper's engineering contribution is building a system that exploits this technique to effectively check large amounts of storage system code with relatively little effort.

Static bug finding. There has been much recent work on static bug finding (e.g., [1, 5, 8, 9, 11, 12]). Roughly speaking, because dynamic checking runs code, it is limited to just executed paths, but can more effectively check deeper properties implied by the code (e.g., `sync()` actually commits data to stable storage or crash recovery works). The errors we found would be difficult to get statically. However, we view static analysis as complementary: easy enough to apply that there is no reason not to use it and then use EXPLODE.

12 Conclusion and Future Work

EXPLODE comprehensively checks storage systems by adapting key ideas from model checking in a way that retains their power but discards their intrusiveness. Its interface lets implementors quickly write storage checkers, or simply compose them from existing components. These checkers run on live systems, which means they do not have to emulate either the environment or pieces of the system. As a result, we often have been able to check a new system in minutes. We used EXPLODE to

find serious bugs in a broad range of real, widely-used storage systems, even when we did not have their source code. Every system we checked had bugs. Our gut belief has become that an unchecked system *must* have bugs — if we do not find any we immediately look to see what is wrong with our checker (a similar dynamic arose in our prior static checking work).

The work in this paper can be extended in numerous ways. First, we only checked systems we did not build. While this shows EXPLODE gets good results without a deep understanding of checked code, it also means we barely scratched the surface of what could be checked. In the future we hope to collaborate with system builders to see just how deep EXPLODE can push a valued system.

Second, we only used EXPLODE for bug-finding, but it is equally useful as an end-to-end validation tool (with no bug fixing intended). A storage subsystem implementor can use it to double-check that the environment the subsystem runs in meets its interface contracts and that the implementor did not misunderstand these contracts. Similarly, a user can use it to check that slipping a subsystem into a system breaks nothing. Or use it to pick a working mechanism from a set of alternatives (e.g., if `fsync` does not work use `sync` instead).

Finally, we can do many things to improve EXPLODE. Our biggest missed opportunity is that we do nothing clever with states. A big benefit of model checking is perspective: it makes state a first-class concept. Thus it becomes natural to think about checking as a state space search; to focus on hitting states that are most "different" from those already seen; to infer what actions cause "interesting" states to be hit; and to extract the essence of states so that two superficially different ones can be treated as equivalent. We have a long list of such things to add to EXPLODE in the future.

Acknowledgements

We thank Xiaowei Yang, Philip Guo, Daniel Dunbar, Silas Boyd-Wickize, Ben Pfaff, Peter Pawlowski, Mike Houston, Phil Levis for proof-reading. We thank Jane-Ellen Long and Jeff Mogul for help with time management. We especially thank Ken Ashcraft and Cristian Cadar for detailed comments, Jeremy Sugerman for his help in reasoning about the GSX error, and Paul Twohey and Ben Pfaff for help in the initial stages of this project (described in [31]). We thank Martin Abadi (our shepherd) and the anonymous reviewers for their struggles with our opaque submission. This research was supported by National Science Foundation (NSF) CAREER award CNS-0238570-001 and Department of Homeland Security grant FA8750-05-2-0142.

References

[1] T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001 Workshop on Model Checking of Software*, May 2001.

[2] Berkeley DB. `http://www.sleepycat.com`.

[3] G. Brat, K. Havelund, S. Park, and W. Visser. Model checking programs. In *IEEE International Conference on Automated Software Engineering*, 2000.

[4] N. Brown. Private communication., Mar. 2005.

[5] W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30(7):775–802, 2000.

[6] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[7] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *ICSE 2000*, 2000.

[8] The Coverity software analysis toolset. `http://coverity.com`.

[9] M. Das, S. Lerner, and M. Seigle. Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[10] G. Dunlap, S. T. King, S. Cinar, M. Basrat, and P. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, Dec. 2002.

[11] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of Operating Systems Design and Implementation*, Sept. 2000.

[12] C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 234–245. ACM Press, 2002.

[13] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, 1997.

[14] HFS and HFS+ utilities. `http://darwinsource.opendarwin.org/10.2.6/diskdev_cmds-208.11`.

[15] G. J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[16] G. J. Holzmann. From code to models. In *Proc. 2nd Int. Conf. on Applications of Concurrency to System Design*, pages 3–10, Newcastle upon Tyne, U.K., 2001.

[17] M. K. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[18] M. Musuvathi, D. Y. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, 2002.

[19] Linux NFS. `http://nfs.sourceforge.net/`.

[20] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks. *ACM SIGMOD Conference*, pages 109–116, June 1988.

[21] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Iron file systems. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, pages 206–220, New York, NY, USA, 2005. ACM Press.

[22] J. Saltzer, D. Reed, and D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, Nov. 1984.

[23] Sandberg, Goldberg, Kleiman, Walsh, and Lyon. Design and implementation of the Sun network file system, 1985.

[24] A simple block driver. `http://lwn.net/Articles/58719/`.

[25] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-smart disk systems. In *Second USENIX Conference on File and Storage Technologies*, 2003.

[26] Linux software RAID. `http://cgi.cse.unsw.edu.au/~neilb/SoftRaid`.

[27] Subversion. `http://subversion.tigris.org`.

[28] J. Sugerman. Private communication., Dec. 2005.

[29] VMware GSX server. `http://www.vmware.com/products/server/`.

[30] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation*, Dec. 2004.

[31] J. Yang, P. Twohey, B. Pfaff, C. Sar, and D. Engler. eXplode: A lightweight, general approach for finding serious errors in storage systems. In *Workshop on the Evaluation of Software Defect Detection Tools*, June 2005.

[32] Zfs: the last word in file systems. `http://www.sun.com/2004-0914/feature/`.

APPENDIX: B

KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs

Cristian Cadar, Daniel Dunbar, Dawson Engler *
Stanford University

Abstract

We present a new symbolic execution tool, KLEE, capable of automatically generating tests that achieve high coverage on a diverse set of complex and environmentally-intensive programs. We used KLEE to thoroughly check all 89 stand-alone programs in the GNU COREUTILS utility suite, which form the core user-level environment installed on millions of Unix systems, and arguably are the single most heavily tested set of open-source programs in existence. KLEE-generated tests achieve high line coverage — on average over 90% per tool (median: over 94%) — and significantly beat the coverage of the developers' own hand-written test suite. When we did the same for 75 equivalent tools in the BUSYBOX embedded system suite, results were even better, including 100% coverage on 31 of them.

We also used KLEE as a bug finding tool, applying it to 452 applications (over 430K total lines of code), where it found 56 serious bugs, including three in COREUTILS that had been missed for over 15 years. Finally, we used KLEE to crosscheck purportedly identical BUSYBOX and COREUTILS utilities, finding functional correctness errors and a myriad of inconsistencies.

1 Introduction

Many classes of errors, such as functional correctness bugs, are difficult to find without executing a piece of code. The importance of such testing — combined with the difficulty and poor performance of random and manual approaches — has led to much recent work in using *symbolic execution* to automatically generate test inputs [11, 14–16, 20–22, 24, 26, 27, 36]. At a high-level, these tools use variations on the following idea: Instead of running code on manually- or randomly-constructed input, they run it on symbolic input initially allowed to be "anything." They substitute program inputs with symbolic values and replace corresponding concrete program operations with ones that manipulate symbolic values. When program execution branches based on a symbolic value, the system (conceptually) follows both branches, on each path maintaining a set of constraints called the *path condition* which must hold on execution of that path. When a path terminates or hits a bug, a test case can be generated by solving the current path condition for concrete values. Assuming deterministic code, feeding this concrete input to a raw, unmodified version of the checked code will make it follow the same path and hit the same bug.

Results are promising. However, while researchers have shown such tools can sometimes get good coverage and find bugs on a small number of programs, it has been an open question whether the approach has any hope of consistently achieving high coverage on real applications. Two common concerns are (1) the exponential number of paths through code and (2) the challenges in handling code that interacts with its surrounding environment, such as the operating system, the network, or the user (colloquially: "the environment problem"). Neither concern has been much helped by the fact that most past work, including ours, has usually reported results on a limited set of hand-picked benchmarks and typically has not included any coverage numbers.

This paper makes two contributions. First, we present a new symbolic execution tool, KLEE, which we designed for robust, deep checking of a broad range of applications, leveraging several years of lessons from our previous tool, EXE [16]. KLEE employs a variety of constraint solving optimizations, represents program states compactly, and uses search heuristics to get high code coverage. Additionally, it uses a simple and straightforward approach to dealing with the external environment. These features improve KLEE's performance by over an order of magnitude and let it check a broad range of system-intensive programs "out of the box."

* Author names are in alphabetical order. Daniel Dunbar is the main author of the KLEE system.

Second, we show that KLEE's automatically-generated tests get high coverage on a diverse set of real, complicated, and environmentally-intensive programs. Our most in-depth evaluation applies KLEE to all 89 programs [1] in the latest stable version of GNU COREUTILS (version 6.10), which contains roughly 80,000 lines of library code and 61,000 lines in the actual utilities [2]. These programs interact extensively with their environment to provide a variety of functions, including managing the file system (e.g., `ls`, `dd`, `chmod`), displaying and configuring system properties (e.g., `logname`, `printenv`, `hostname`), controlling command invocation (e.g., `nohup`, `nice`, `env`), processing text files (e.g., `sort`, `od`, `patch`), and so on. They form the core user-level environment installed on many Unix systems. They are used daily by millions of people, bug fixes are handled promptly, and new releases are pushed regularly. Moreover, their extensive interaction with the environment stress-tests symbolic execution where it has historically been weakest.

Further, finding bugs in COREUTILS is hard. They are arguably the single most well-tested suite of open-source applications available (e.g., is there a program the reader has used more under Unix than "`ls`"?). In 1995, random testing of a subset of COREUTILS utilities found markedly fewer failures as compared to seven commercial Unix systems [35]. The last COREUTILS vulnerability reported on the SecurityFocus or US National Vulnerability databases was three years ago [5, 7].

In addition, we checked two other UNIX utility suites: BUSYBOX, a widely-used distribution for embedded systems [1], and the latest release for MINIX [4]. Finally, we checked the HISTAR operating system kernel as a contrast to application code [39].

Our experiments fall into three categories: (1) those where we do intensive runs to both find bugs and get high coverage (COREUTILS, HISTAR, and 75 BUSYBOX utilities), (2) those where we quickly run over many applications to find bugs (an additional 204 BUSYBOX utilities and 77 MINIX utilities), and (3) those where we crosscheck equivalent programs to find deeper correctness bugs (67 BUSYBOX utilities vs. the equivalent 67 in COREUTILS).

In total, we ran KLEE on more than 452 programs, containing over 430K total lines of code. To the best of our knowledge, this represents an order of magnitude more code and distinct programs than checked by prior symbolic test generation work. Our experiments show:

1 KLEE gets high coverage on a broad set of complex programs. Its automatically generated tests covered 84.5% of the total lines in COREUTILS and 90.5% in BUSYBOX (ignoring library code). On average these tests hit over 90% of the lines in each tool (median: over 94%), achieving perfect 100% coverage in 16 COREUTILS tools and 31 BUSYBOX tools.

2 KLEE can get significantly more code coverage than a concentrated, sustained manual effort. The roughly 89-hour run used to generate COREUTILS line coverage beat the developers' own test suite — built incrementally over fifteen years — by 16.8%!

3 With one exception, KLEE achieved these high-coverage results on unaltered applications. The sole exception, `sort` in COREUTILS, required a single edit to shrink a large buffer that caused problems for the constraint solver.

4 KLEE finds important errors in heavily-tested code. It found ten fatal errors in COREUTILS (including three that had escaped detection for 15 years), which account for more crashing bugs than were reported in 2006, 2007 and 2008 combined. It further found 24 bugs in BUSYBOX, 21 bugs in MINIX, and a security vulnerability in HISTAR– a total of 56 serious bugs.

5 The fact that KLEE test cases can be run on the raw version of the code (e.g., compiled with `gcc`) greatly simplifies debugging and error reporting. For example, all COREUTILS bugs were confirmed and fixed within two days and versions of the tests KLEE generated were included in the standard regression suite.

6 KLEE is not limited to low-level programming errors: when used to crosscheck purportedly identical BUSYBOX and GNU COREUTILS tools, it automatically found functional correctness errors and a myriad of inconsistencies.

7 KLEE can also be applied to non-application code. When applied to the core of the HISTAR kernel, it achieved an average line coverage of 76.4% (with disk) and 67.1% (without disk) and found a serious security bug.

The next section gives an overview of our approach. Section 3 describes KLEE, focusing on its key optimizations. Section 4 discusses how to model the environment. The heart of the paper is Section 5, which presents our experimental results. Finally, Section 6 describes related work and Section 7 concludes.

2 Overview

This section explains how KLEE works by walking the reader through the testing of MINIX's `tr` tool. Despite its small size — 169 lines, 83 of which are executable — it illustrates two problems common to the programs we check:

1 *Complexity.* The code aims to translate and delete characters from its input. It hides this intent well beneath non-obvious input parsing code, tricky boundary conditions, and hard-to-follow control flow. Figure 1 gives a representative snippet.

[1] We ignored utilities that are simply wrapper calls to others, such as `arch` ("`uname -m`") and `vdir` ("`ls -l -b`").

2 *Environmental Dependencies.* Most of the code is controlled by values derived from environmental input. Command line arguments determine what procedures execute, input values determine which way if-statements trigger, and the program depends on the ability to read from the file system. Since inputs can be invalid (or even malicious), the code must handle these cases gracefully. It is not trivial to test all important values and boundary cases.

The code illustrates two additional common features. First, it has bugs, which KLEE finds and generates test cases for. Second, KLEE quickly achieves good code coverage: in two minutes it generates 37 tests that cover all executable statements. [2]

KLEE has two goals: (1) hit every line of executable code in the program and (2) detect at each dangerous operation (e.g., dereference, assertion) if *any* input value exists that could cause an error. KLEE does so by running programs *symbolically*: unlike normal execution, where operations produce concrete values from their operands, here they generate constraints that exactly describe the set of values possible on a given path. When KLEE detects an error or when a path reaches an `exit` call, KLEE solves the current path's constraints (called its *path condition*) to produce a test case that will follow the same path when rerun on an unmodified version of the checked program (e.g, compiled with `gcc`).

KLEE is designed so that the paths followed by the unmodified program will always follow the same path KLEE took (i.e., there are no false positives). However, non-determinism in checked code and bugs in KLEE or its models have produced false positives in practice. The ability to rerun tests outside of KLEE, in conjunction with standard tools such as `gdb` and `gcov` is invaluable for diagnosing such errors and for validating our results.

We next show how to use KLEE, then give an overview of how it works.

2.1 Usage

A user can start checking many real programs with KLEE in seconds: KLEE typically requires no source modifications or manual work. Users first compile their code to bytecode using the publicly-available LLVM compiler [33] for GNU C. We compiled `tr` using:

```
llvm-gcc --emit-llvm -c tr.c -o tr.bc
```

Users then run KLEE on the generated bytecode, optionally stating the number, size, and type of symbolic inputs to test the code on. For `tr` we used the command:

```
klee --max-time 2 --sym-args 1 10 10
   --sym-files 2 2000 --max-fail 1 tr.bc
```

[2] The program has one line of dead code, an unreachable return statement, which, reassuringly, KLEE cannot run.

```
1 : void expand(char *arg, unsigned char *buffer) {          8
2 :   int i, ac;                                              9
3 :   while (*arg) {                                          10*
4 :     if (*arg == '\\') {                                   11*
5 :       arg++;
6 :       i = ac = 0;
7 :       if (*arg >= '0' && *arg <= '7') {
8 :         do {
9 :           ac = (ac << 3) + *arg++ - '0';
10:           i++;
11:         } while (i<4 && *arg>='0' && *arg<='7');
12:         *buffer++ = ac;
13:       } else if (*arg != '\0')
14:         *buffer++ = *arg++;
15:     } else if (*arg == '[') {                             12*
16:       arg++;                                              13
17:       i = *arg++;                                         14
18:       if (*arg++ != '-') {                                15!
19:         *buffer++ = '[';
20:         arg -= 2;
21:         continue;
22:       }
23:       ac = *arg++;
24:       while (i <= ac) *buffer++ = i++;
25:       arg++;      /* Skip ']' */
26:     } else
27:       *buffer++ = *arg++;
28:   }
29: }
30: ...
31: int main(int argc, char* argv[]) {                        1
32:   int index = 1;                                          2
33:   if (argc > 1 && argv[index][0] == '-') {               3*
34:     ...                                                   4
35:   }                                                       5
36:   ...                                                     6
37:   expand(argv[index++], index);                           7
38:   ...
39: }
```

Figure 1: Code snippet from MINIX's `tr`, representative of the programs checked in this paper: tricky, non-obvious, difficult to verify by inspection or testing. The order of the statements on the path to the error at line 18 are numbered on the right hand side.

The first option, `--max-time`, tells KLEE to check `tr.bc` for at most two minutes. The rest describe the symbolic inputs. The option `--sym-args 1 10 10` says to use zero to three command line arguments, the first 1 character long, the others 10 characters long. [3] The option `--sym-files 2 2000` says to use standard input and one file, each holding 2000 bytes of symbolic data. The option `--max-fail 1` says to fail at most one system call along each program path (see § 4.2).

2.2 Symbolic execution with KLEE

When KLEE runs on `tr`, it finds a buffer overflow error at line 18 in Figure 1 and then produces a concrete test

[3] Since strings in C are zero terminated, this essentially generates arguments of *up to* that size.

case (`tr [ "" ""`) that hits it. Assuming the options of the previous subsection, KLEE runs `tr` as follows:

1. KLEE constructs symbolic command line string arguments whose contents have no constraints other than zero-termination. It then constrains the number of arguments to be between 0 and 3, and their sizes to be 1, 10 and 10 respectively. It then calls `main` with these initial path constraints.
2. When KLEE hits the branch `argc > 1` at line 33, it uses its constraint solver STP [23] to see which directions can execute given the current path condition. For this branch, both directions are possible; KLEE forks execution and follows both paths, adding the constraint `argc` > 1 on the false path and `argc` ≤ 1 on the true path.
3. Given more than one active path, KLEE must pick which one to execute first. We describe its algorithm in Section 3.4. For now assume it follows the path that reaches the bug. As it does so, KLEE adds further constraints to the contents of `arg`, and forks for a total of five times (lines denoted with a "`*`"): twice on line 33, and then on lines 3, 4, and 15 in `expand`.
4. At each dangerous operation (e.g., pointer dereference), KLEE checks if any possible value allowed by the current path condition would cause an error. On the annotated path, KLEE detects no errors before line 18. At that point, however, it determines that input values exist that allow the read of `arg` to go out of bounds: after taking the true branch at line 15, the code increments `arg` twice without checking if the string has ended. If it has, this increment skips the terminating `'\0'` and points to invalid memory.
5. KLEE generates concrete values for `argc` and `argv` (i.e., `tr [ "" ""`) that when rerun on a raw version of `tr` will hit this bug. It then continues following the current path, adding the constraint that the error does not occur (in order to find other errors).

3 The KLEE Architecture

KLEE is a complete redesign of our previous system EXE [16]. At a high level, KLEE functions as a hybrid between an operating system for symbolic processes and an interpreter. Each symbolic process has a register file, stack, heap, program counter, and path condition. To avoid confusion with a Unix process, we refer to KLEE's representation of a symbolic process as a *state*. Programs are compiled to the LLVM [33] assembly language, a RISC-like virtual instruction set. KLEE directly interprets this instruction set, and maps instructions to constraints without approximation (i.e. bit-level accuracy). [4]

[4] KLEE does not currently support: symbolic floating point, `longjmp`, threads, and assembly code. Additionally, memory objects are required to have concrete sizes.

3.1 Basic architecture

At any one time, KLEE may be executing a large number of states. The core of KLEE is an interpreter loop which selects a state to run and then symbolically executes a single instruction in the context of that state. This loop continues until there are no states remaining, or a user-defined timeout is reached.

Unlike a normal process, storage locations for a state — registers, stack and heap objects — refer to expressions (trees) instead of raw data values. The leaves of an expression are symbolic variables or constants, and the interior nodes come from LLVM assembly language operations (e.g., arithmetic operations, bitwise manipulation, comparisons, and memory accesses). Storage locations which hold a constant expression are said to be *concrete*.

Symbolic execution of the majority of instructions is straightforward. For example, to symbolically execute an LLVM add instruction:

```
%dst = add i32 %src0, %src1
```

KLEE retrieves the addends from the `%src0` and `%src1` registers and writes a new expression `Add(%src0, %src1)` to the `%dst` register. For efficiency, the code that builds expressions checks if all given operands are concrete (i.e., constants) and, if so, performs the operation natively, returning a constant expression.

Conditional branches take a boolean expression (branch condition) and alter the instruction pointer of the state based on whether the condition is true or false. KLEE queries the constraint solver to determine if the branch condition is either provably true or provably false along the current path; if so, the instruction pointer is updated to the appropriate location. Otherwise, both branches are possible: KLEE clones the state so that it can explore both paths, updating the instruction pointer and path condition on each path appropriately.

Potentially dangerous operations implicitly generate branches that check if any input value exists that could cause an error. For example, a division instruction generates a branch that checks for a zero divisor. Such branches work identically to normal branches. Thus, even when the check succeeds (i.e., an error is detected), execution continues on the false path, which adds the negation of the check as a constraint (e.g., making the divisor not zero). If an error is detected, KLEE generates a test case to trigger the error and terminates the state.

As with other dangerous operations, load and store instructions generate checks: in this case to check that the address is in-bounds of a valid memory object. However, load and store operations present an additional complication. The most straightforward representation of the memory used by checked code would be a flat byte array. In this case, loads and stores would simply map to

array read and write expressions respectively. Unfortunately, our constraint solver STP would almost never be able to solve the resultant constraints (and neither would the other constraint solvers we know of). Thus, as in EXE, KLEE maps every memory object in the checked code to a distinct STP array (in a sense, mapping a flat address space to a segmented one). This representation dramatically improves performance since it lets STP ignore all arrays not referenced by a given expression.

Many operations (such as bound checks or object-level copy-on-write) require object-specific information. If a pointer can refer to many objects, these operations become difficult to perform. For simplicity, KLEE sidesteps this problem as follows. When a dereferenced pointer `p` can refer to N objects, KLEE clones the current state N times. In each state it constrains `p` to be within bounds of its respective object and then performs the appropriate read or write operation. Although this method can be expensive for pointers with large points-to sets, most programs we have tested only use symbolic pointers that refer to a single object, and KLEE is well-optimized for this case.

3.2 Compact state representation

The number of states grows quite quickly in practice: often even small programs generate tens or even hundreds of thousands of concurrent states during the first few minutes of interpretation. When we ran COREUTILS with a 1GB memory cap, the maximum number of concurrent states recorded was 95,982 (for `hostid`), and the average of this maximum for each tool was 51,385. This explosion makes state size critical.

Since KLEE tracks all memory objects, it can implement copy-on-write at the object level (rather than page granularity), dramatically reducing per-state memory requirements. By implementing the heap as an immutable map, portions of the heap structure itself can also be shared amongst multiple states (similar to sharing portions of page tables across `fork()`). Additionally, this heap structure can be cloned in constant time, which is important given the frequency of this operation.

This approach is in marked contrast to EXE, which used one native OS process per state. Internalizing the state representation dramatically increased the number of states which can be concurrently explored, both by decreasing the per-state cost and allowing states to share memory at the object (rather than page) level. Additionally, this greatly simplified the implementation of caches and search heuristics which operate across all states.

3.3 Query optimization

Almost always, the cost of constraint solving dominates everything else — unsurprising, given that KLEE generates complicated queries for an NP-complete logic. Thus, we spent a lot of effort on tricks to simplify expressions and ideally eliminate queries (no query is the fastest query) before they reach STP. Simplified queries make solving faster, reduce memory consumption, and increase the query cache's hit rate (see below). The main query optimizations are:

Expression Rewriting. The most basic optimizations mirror those in a compiler: e.g., simple arithmetic simplifications (`x + 0 = x`), strength reduction (`x * 2`^{n} `= x << n`), linear simplification (`2*x - x = x`).

Constraint Set Simplification. Symbolic execution typically involves the addition of a large number of constraints to the path condition. The natural structure of programs means that constraints on same variables tend to become more specific. For example, commonly an inexact constraint such as $x < 10$ gets added, followed some time later by the constraint $x = 5$. KLEE actively simplifies the constraint set by rewriting previous constraints when new equality constraints are added to the constraint set. In this example, substituting the value for x into the first constraint simplifies it to `true`, which KLEE eliminates.

Implied Value Concretization. When a constraint such as $x + 1 = 10$ is added to the path condition, then the value of `x` has effectively become concrete along that path. KLEE determines this fact (in this case that $x = 9$) and writes the concrete value back to memory. This ensures that subsequent accesses of that memory location can return a cheap constant expression.

Constraint Independence. Many constraints do not overlap in terms of the memory they reference. Constraint independence (taken from EXE) divides constraint sets into disjoint independent subsets based on the symbolic variables they reference. By explicitly tracking these subsets, KLEE can frequently eliminate irrelevant constraints prior to sending a query to the constraint solver. For example, given the constraint set $\{i < j, j < 20, k > 0\}$, a query of whether $i = 20$ just requires the first two constraints.

Counter-example Cache. Redundant queries are frequent, and a simple cache is effective at eliminating a large number of them. However, it is possible to build a more sophisticated cache due to the particular structure of constraint sets. The counter-example cache maps sets of constraints to counter-examples (i.e., variable assignments), along with a special sentinel used when a set of constraints has no solution. This mapping is stored in a custom data structure — derived from the UBTree structure of Hoffmann and Hoehler [28] — which allows efficient searching for cache entries for both subsets and supersets of a constraint set. By storing the cache in this fashion, the counter-example cache gains three additional ways to eliminate queries. In the example below, we assume that the counter-example cache

Optimizations	Queries	Time (s)	STP Time (s)
None	13717	300	281
Independence	13717	166	148
Cex. Cache	8174	177	156
All	699	20	10

Table 1: Performance comparison of KLEE's solver optimizations on COREUTILS. Each tool is run for 5 minutes without optimization, and rerun on the same workload with the given optimizations. The results are averaged across all applications.

currently has entries for $\{i < 10, i = 10\}$ (no solution) and $\{i < 10, j = 8\}$ (satisfiable, with variable assignments $i \rightarrow 5, j \rightarrow 8$).

1 When a subset of a constraint set has no solution, then neither does the original constraint set. Adding constraints to an unsatisfiable constraint set cannot make it satisfiable. For example, given the cache above, $\{i < 10, i = 10, j = 12\}$ is quickly determined to be unsatisfiable.

2 When a superset of a constraint set has a solution, that solution also satisfies the original constraint set. Dropping constraints from a constraint set does not invalidate a solution to that set. The assignment $i \rightarrow 5, j \rightarrow 8$, for example, satisfies either $i < 10$ or $j = 8$ individually.

3 When a subset of a constraint set has a solution, it is likely that this is also a solution for the original set. This is because the extra constraints often do not invalidate the solution to the subset. Because checking a potential solution is cheap, KLEE tries substituting in all solutions for subsets of the constraint set and returns a satisfying solution, if found. For example, the constraint set $\{i < 10, j = 8, i \neq 3\}$ can still be satisfied by $i \rightarrow 5, j \rightarrow 8$.

To demonstrate the effectiveness of these optimizations, we performed an experiment where COREUTILS applications were run for 5 minutes with both of these optimizations turned off. We then deterministically reran the exact same workload with constraint independence and the counter-example cache enabled separately and together for the same number of instructions. This experiment was done on a large sample of COREUTILS utilities. The results in Table 1 show the averaged results.

As expected, the independence optimization by itself does not eliminate any queries, but the simplifications it performs reduce the overall running time by almost half (45%). The counter-example cache reduces both the running time and the number of STP queries by 40%. However, the real win comes when both optimizations are enabled; in this case the hit rate for the counter-example cache greatly increases due to the queries first being simplified via independence. For the sample runs, the average number of STP queries are reduced to 5% of the original number and the average runtime decreases by more than an order of magnitude.

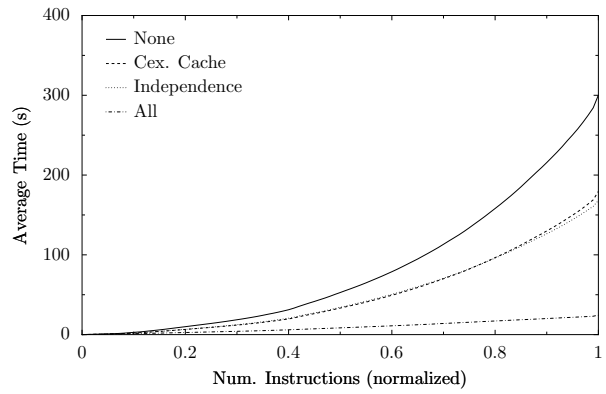


Figure 2: The effect of KLEE's solver optimizations over time, showing they become more effective over time, as the caches fill and queries become more complicated. The number of executed instructions is normalized so that data can be aggregated across all applications.

It is also worth noting the degree to which STP time (time spent solving queries) dominates runtime. For the original runs, STP accounts for 92% of overall execution time on average (the combined optimizations reduce this by almost 300%). With both optimizations enabled this percentage drops to 41%. Finally, Figure 2 shows the efficacy of KLEE's optimizations increases with time — as the counter-example cache is filled and query sizes increase, the speed-up from the optimizations also increases.

3.4 State scheduling

KLEE selects the state to run at each instruction by interleaving the following two search heuristics.

Random Path Selection maintains a binary tree recording the program path followed for all active states, i.e. the leaves of the tree are the current states and the internal nodes are places where execution forked. States are selected by traversing this tree from the root and randomly selecting the path to follow at branch points. Therefore, when a branch point is reached, the set of states in each subtree has equal probability of being selected, regardless of the size of their subtrees. This strategy has two important properties. First, it favors states high in the branch tree. These states have less constraints on their symbolic inputs and so have greater freedom to reach uncovered code. Second, and most importantly, this strategy avoids starvation when some part of the program is rapidly creating new states ("fork bombing") as it happens when a tight loop contains a symbolic condition. Note that the simply selecting a state at random has neither property.

Coverage-Optimized Search tries to select states likely to cover new code in the immediate future. It uses heuristics to compute a weight for each state and then randomly selects a state according to these weights. Currently these heuristics take into account the minimum distance to an uncovered instruction, the call stack of the state, and whether the state recently covered new code.

KLEE uses each strategy in a round robin fashion. While this can increase the time for a particularly effective strategy to achieve high coverage, it protects against cases where an individual strategy gets stuck. Furthermore, since strategies pick from the same state pool, interleaving them can improve overall effectiveness.

The time to execute an individual instruction can vary widely between simple instructions (e.g., addition) and instructions which may use the constraint solver or fork execution (branches, memory accesses). KLEE ensures that a state which frequently executes expensive instructions will not dominate execution time by running each state for a "time slice" defined by both a maximum number of instructions and a maximum amount of time.

4 Environment Modeling

When code reads values from its environment — command-line arguments, environment variables, file data and metadata, network packets, etc — we conceptually want to return all values that the read could legally produce, rather than just a single concrete value. When it writes to its environment, the effects of these alterations should be reflected in subsequent reads. The combination of these features allows the checked program to explore all potential actions and still have no false positives.

Mechanically, we handle the environment by redirecting calls that access it to *models* that understand the semantics of the desired action well enough to generate the required constraints. Crucially, these models are written in normal C code which the user can readily customize, extend, or even replace without having to understand the internals of KLEE. We have about 2,500 lines of code to define simple models for roughly 40 system calls (e.g., `open`, `read`, `write`, `stat`, `lseek`, `ftruncate`, `ioctl`).

4.1 Example: modeling the file system

For each file system operation we check if the action is for an actual concrete file on disk or a symbolic file. For concrete files, we simply invoke the corresponding system call in the running operating system. For symbolic files we emulate the operation's effect on a simple symbolic file system, private to each state.

Figure 3 gives a rough sketch of the model for `read()`, eliding details for dealing with linking, reads on standard input, and failures. The code maintains a set of file descriptors, created at file `open()`, and records

```
1 : ssize_t read(int fd, void *buf, size_t count) {
2 :   if (is_invalid(fd)) {
3 :     errno = EBADF;
4 :     return -1;
5 :   }
6 :   struct klee_fd *f = &fds[fd];
7 :   if (is_concrete_file(f)) {
8 :     int r = pread(f->real_fd, buf, count, f->off);
9 :     if (r != -1)
10:       f->off += r;
11:     return r;
12:   } else {
13:     /* sym files are fixed size: don't read beyond the end. */
14:     if (f->off >= f->size)
15:       return 0;
16:     count = min(count, f->size - f->off);
17:     memcpy(buf, f->file_data + f->off, count);
18:     f->off += count;
19:     return count;
20:   }
21: }
```

Figure 3: Sketch of KLEE's model for `read()`.

for each whether the associated file is symbolic or concrete. If `fd` refers to a concrete file, we use the operating system to read its contents by calling `pread()` (lines 7-11). We use `pread` to multiplex access from KLEE's many states onto the one actual underlying file descriptor.[5] If `fd` refers to a symbolic file, `read()` copies from the underlying symbolic buffer holding the file contents into the user supplied buffer (lines 13-19). This ensures that multiple `read()` calls that access the same file use consistent symbolic values.

Our symbolic file system is crude, containing only a single directory with N symbolic files in it. KLEE users specify both the number N and the size of these files. This symbolic file system coexists with the real file system, so applications can use both symbolic and concrete files. When the program calls `open` with a concrete name, we (attempt to) open the actual file. Thus, the call:

```
int fd = open("/etc/fstab", O_RDNLY);
```

sets `fd` to point to the actual configuration file `/etc/fstab`.

On the other hand, calling `open()` with an unconstrained symbolic name matches each of the N symbolic files in turn, and will also fail once. For example, given $N = 1$, calling `open()` with a symbolic command-line argument `argv[1]`:

```
int fd = open(argv[1], O_RDNLY);
```

will result in two paths: one in which `fd` points to the single symbolic file in the environment, and one in which `fd` is set to `-1` indicating an error.

[5] Since KLEE's states execute within a single Unix process (the one used to run KLEE), then unless we duplicated file descriptors for each (which seemed expensive), a `read` by one would affect all the others.

Unsurprisingly, the choice of what interface to model has a big impact on model complexity. Rather than having our models at the system call level, we could have instead built them at the C standard library level (`fopen`, `fread`, etc.). Doing so has the potential performance advantage that, for concrete code, we could run these operations natively. The major downside, however, is that the standard library contains a huge number of functions — writing models for each would be tedious and error-prone. By only modeling the much simpler, low-level system call API, we can get the richer functionality by just compiling one of the many implementations of the C standard library (we use uClibc [6]) and let it worry about correctness. As a side-effect, we simultaneously check the library for errors as well.

4.2 Failing system calls

The real environment can fail in unexpected ways (e.g., `write()` fails because of a full disk). Such failures can often lead to unexpected and hard to diagnose bugs. Even when applications do try to handle them, this code is rarely fully exercised by the regression suite. To help catch such errors, KLEE will optionally simulate environmental failures by failing system calls in a controlled manner (similar to [38]). We made this mode optional since not all applications care about failures — a simple application may ignore disk crashes, while a mail server expends a lot of code to handle them.

4.3 Rerunning test cases

KLEE-generated test cases are rerun on the unmodified native binaries by supplying them to a replay driver we provide. The individual test cases describe an instance of the symbolic environment. The driver uses this description to create actual operating system objects (files, pipes, ttys, directories, links, etc.) containing the concrete values used in the test case. It then executes the unmodified program using the concrete command-line arguments from the test case. Our biggest challenge was making system calls fail outside of KLEE — we built a simple utility that uses the `ptrace` debugging interface to skip the system calls that were supposed to fail and instead return an error.

5 Evaluation

This section describes our in-depth coverage experiments for COREUTILS (§ 5.2) and BUSYBOX (§ 5.3) as well as errors found during quick bug-finding runs (§ 5.4). We use KLEE to find deep correctness errors by crosschecking purportedly equivalent tool implementations (§ 5.5) and close with results for HISTAR (§5.6).

5.1 Coverage methodology

We use line coverage as a conservative measure of KLEE-produced test case effectiveness. We chose executable line coverage as reported by `gcov`, because it is widely-understood and uncontroversial. Of course, it grossly underestimates KLEE's thoroughness, since it ignores the fact that KLEE explores many different unique paths with all possible values. We expect a path-based metric would show even more dramatic wins.

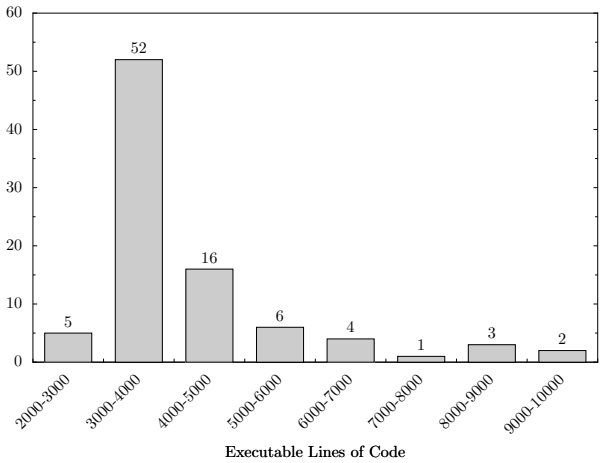


Figure 4: Histogram showing the number of COREUTILS tools that have a given number of executable lines of code (ELOC).

We measure coverage by running KLEE-generated test cases on a stand-alone version of each utility and using `gcov` to measure coverage. Running tests independently of KLEE eliminates the effect of bugs in KLEE and verifies that the produced test case runs the code it claims.

Note, our coverage results only consider code in the tool itself. They do not count library code since doing so makes the results harder to interpret:

1. It double-counts many lines, since often the same library function is called by many applications.
2. It unfairly under-counts coverage. Often, the bulk of a library function called by an application is effectively dead code since the library code is general but call sites are not. For example, `printf` is exceptionally complex, but the call `printf("hello")` can only hit a small a fraction (missing the code to print integers, floating point, formatting, etc.).

However, we do include library code when measuring the raw size of the application: KLEE must successfully handle this library code (and gets no credit for doing so) in order to exercise the code in the tool itself. We measure size in terms of executable lines of code (ELOC) by counting the total number of executable lines in the final executable after global optimization, which eliminates uncalled functions and other dead code. This measure is usually a factor of three smaller than a simple line count (using `wc -l`).

In our experiments KLEE minimizes the test cases it

Coverage (w/o lib)	COREUTILS KLEE tests	COREUTILS Devel. tests	BUSYBOX KLEE tests	BUSYBOX Devel. tests
100%	16	1	31	4
90-100%	40	6	24	3
80-90%	21	20	10	15
70-80%	7	23	5	6
60-70%	5	15	2	7
50-60%	-	10	-	4
40-50%	-	6	-	-
30-40%	-	3	-	2
20-30%	-	1	-	1
10-20%	-	3	-	-
0-10%	-	1	-	30
Overall cov.	84.5%	67.7%	90.5%	44.8%
Med cov/App	94.7%	72.5%	97.5%	58.9%
Ave cov/App	90.9%	68.4%	93.5%	43.7%

Table 2: Number of COREUTILS tools which achieve line coverage in the given ranges for KLEE and developers' tests (library code not included). The last rows shows the aggregate coverage achieved by each method and the average and median coverage per application.

generates by only emitting tests cases for paths that hit a new statement or branch in the main utility code. A user that wants high library coverage can change this setting.

5.2 GNU COREUTILS

We now give KLEE coverage results for all 89 GNU COREUTILS utilities.

Figure 4 breaks down the tools by executable lines of code (ELOC), including library code the tool calls. While relatively small, the tools are not toys — the smallest five have between 2K and 3K ELOC, over half (52) have between 3K and 4K, and ten have over 6K.

Previous work, ours included, has evaluated constraint-based execution on a small number of hand-selected benchmarks. Reporting results for the entire COREUTILS suite, the worst along with the best, prevents us from hand-picking results or unintentionally cheating through the use of fragile optimizations.

Almost all tools were tested using the same command (command arguments explained in § 2.1):

```
./run <tool-name>  --max-time 60
                   --sym-args 10 2 2
                   --sym-files 2 8
                   [--max-fail 1]
```

As specified by the `--max-time` option, we ran each tool for about 60 minutes (some finished before this limit, a few up to three minutes after). For eight tools where the coverage results of these values were unsatisfactory, we consulted the `man` page and increased the number and size of arguments and files. We found this easy to do,

Figure 5: Line coverage for each application with and without failing system calls.

so presumably a tool implementer or user would as well. After these runs completed, we improved them by failing system calls (see § 4.2).

5.2.1 Line coverage results

The first two columns in Table 2 give aggregate line coverage results. On average our tests cover 90.9% of the lines in each tool (median: 94.7%), with an overall (aggregate) coverage across all tools of 84.5%. We get 100% line coverage on 16 tools, over 90% on 56 tools, and over 80% on 77 tools (86.5% of all tools). The minimum coverage achieved on any tool is 62.6%.

We believe such high coverage on a broad swath of applications "out of the box" convincingly shows the power of the approach, especially since it is across the entire tool suite rather than focusing on a few particular applications.

Importantly, KLEE generates high coverage with few test cases: for our non-failing runs, it needs a total of 3,321 tests, with a per-tool average of 37 (median: 33). The maximum number needed was 129 (for the "`[`" tool) and six needed 5. As a crude measure of path complexity, we counted the number of static branches run by each test case using `gcov`[6] (i.e., an executed branch counts once no matter how many times the branch ran dynamically). The average path length was 76 (median: 53), the maximum was 512 and (to pick a random number) 160 were at least 250 branches long.

Figure 5 shows the coverage KLEE achieved on each tool, with and without failing system call invocations. Hitting system call failure paths is useful for getting the last few lines of high-coverage tools, rather than significantly improving the overall results (which it improves from 79.9% to 84.5%). The one exception is `pwd` which requires system call failures to go from a dismal 21.2% to 72.6%. The second best improvement for a single tool is a more modest 13.1% extra coverage on the `df` tool.

[6] In `gcov` terminology, a branch is a possible branch direction, i.e. a simple if statement has two branches.

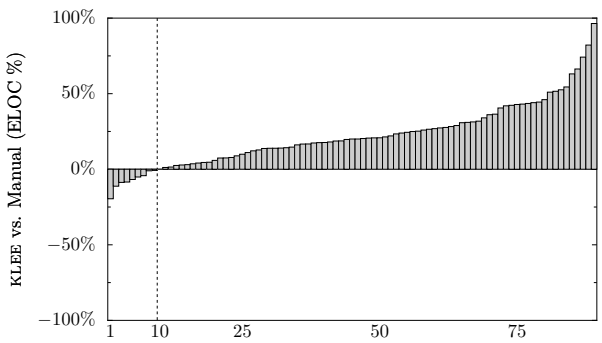


Figure 6: Relative coverage difference between KLEE and the COREUTILS manual test suite, computed by subtracting the executable lines of code covered by manual tests (L_{man}) from KLEE tests (L_{klee}) and dividing by the total possible: $(L_{klee} - L_{man})/L_{total}$. Higher bars are better for KLEE, which beats manual testing on all but 9 applications, often significantly.

```
paste -d\\ abcdefghijklmnopqrstuvwxyz
pr -e t2.txt
tac -r t3.txt t3.txt
mkdir -Z a b
mkfifo -Z a b
mknod -Z a b p
md5sum -c t1.txt
ptx -F\\ abcdefghijklmnopqrstuvwxyz
ptx x t4.txt
seq -f %0 1
```

t1.txt: `"\t \tMD5("`
t2.txt: `"\b\b\b\b\b\b\b\t"`
t3.txt: `"\n"`
t4.txt: `"a"`

Figure 7: KLEE-generated command lines and inputs (modified for readability) that cause program crashes in COREUTILS version 6.10 when run on Fedora Core 7 with SELinux on a Pentium machine.

5.2.2 Comparison against developer test suites

Each utility in COREUTILS comes with an extensive manually-written test suite extended each time a new bug fix or extra feature is added. [7] As Table 2 shows, KLEE beats developer tests handily on all aggregate measures: overall total line coverage (84.5% versus 67.7%), average coverage per tool (90.9% versus 68.4%) and median coverage per tool (94.7% versus 72.5%). At a more detailed level, KLEE gets 100% coverage on 16 tools and over 90% coverage on 56 while the developer tests get 100% on a single utility (`true`) and reach over 90% on only 7. Finally, the developers tests get below 60% coverage on 24 tools while KLEE always achieves over 60%. In total, an 89 hour run of KLEE (about one hour per application) exceeds the coverage of a test suite built over a period of fifteen years by 16.8%!

Figure 6 gives a relative view of KLEE versus developer tests by subtracting the lines hit by manual testing from those hit by KLEE and dividing this by the total possible. A bar above zero indicates that KLEE beat the manual test (and by how much); a bar below shows the opposite. KLEE beats manual testing, often significantly, on the vast majority of the applications.

To guard against hidden bias in line coverage, we also compared the taken branch coverage (as reported by `gcov`) of the manual and KLEE test suites. While the absolute coverage for both test suites decreases, KLEE's relative improvement over the developers' tests remains: KLEE achieves 76.9% overall branch coverage, while the developers' tests get only 56.5%.

Finally, it is important to note that although KLEE's runs significantly beat the developers' tests in terms of coverage, KLEE only checks for low-level errors and violations of user-level asserts. In contrast, developer tests typically validate that the application output matches the expected one. We partially address this limitation by validating the output of these utilities against the output produces by a different implementation (see § 5.5).

5.2.3 Bugs found

KLEE found ten unique bugs in COREUTILS (usually memory error crashes). Figure 7 gives the command lines used to trigger them. The first three errors existed since at least 1992, so should theoretically crash any COREUTILS distribution up to 6.10. The others are more recent, and do not crash older COREUTILS distributions. While one bug (in `seq`) had been fixed in the developers' unreleased version, the other bugs were confirmed and fixed within two days of our report. In addition, versions of the KLEE-generated test cases for the new bugs were added to the official COREUTILS test suite.

As an illustrative example, we discuss the bug in `pr` (used to paginate files before printing) hit by the invocation "`pr -e t2.txt`" in Figure 7. The code containing the bug is shown in Figure 8. On the path that hits the bug, both `chars_per_input_tab` and `chars_per_c` equal tab width (let's call it T). Line 2665 computes `width` $= (T -$ `input_position` $\bmod T)$ using the macro on line 602. The root cause of the bug is the incorrect assumption that $0 \leq x \mod y < y$, which only holds for positive integers. When `input_position` is positive, `width` will be less than T since $0 \leq$ `input_position` $\bmod T < T$. However, in the pres-

[7] We ran the test suite using the commands: `env RUN_EXPENSIVE_TESTS=YES RUN_VERY_EXPENSIVE_TESTS=YES make check` and `make check-root` (as root). A small number of tests (14 out of 393) which require special configuration were not run; from manual inspection we do not expect these to have a significant impact on our results.

```
 602: #define TAB_WIDTH(c_, h_) ((c_) − ((h_) % (c_)))
...
1322: clump_buff = xmalloc(MAX(8,chars_per_input_tab));
...     // (set s to clump_buff)
2665: width = TAB_WIDTH(chars_per_c, input_position);
2666:
2667: if (untabify_input)
2668: {
2669:   for (i = width; i; −−i)
2670:     *s++ = ' ';
2671:   chars = width;
2672: }
```

Figure 8: Code snippet from `pr` where a memory overflow of `clump_buff` via pointer `s` is possible if `chars_per_input_tab == chars_per_c` and `input_position < 0`.

ence of backspaces, `input_position` can become negative, so $(-T < \texttt{input\_position} \mod T < T)$. Consequently, `width` can be as large as $2T - 1$.

The bug arises when the code allocates a buffer `clump_buff` of size T (line 1322) and then writes `width` characters into this buffer (lines 2669–2670) via the pointer `s` (initially set to `clump_buff`). Because `width` can be as large as $2T - 1$, a memory overflow is possible.

This is a prime example of the power of symbolic execution in finding complex errors in code which is hard to reason about manually — this bug has existed in `pr` since at least 1992, when COREUTILS was first added to a CVS repository.

5.2.4 Comparison with random tests

In our opinion, the COREUTILS manual tests are unusually comprehensive. However, we compare to random testing both to guard against deficiencies, and to get a feel for how constraint-based reasoning compares to blind random guessing. We tried to make the comparison apples-to-apples by building a tool that takes the same command line as KLEE, and generates random values for the specified type, number, and size range of inputs. It then runs the checked program on these values using the same replay infrastructure as KLEE. For time reasons, we randomly chose 15 benchmarks (shown in Figure 9) and ran them for 65 minutes (to always exceed the time given to KLEE) with the same command lines used when run with KLEE.

Figure 9 shows the coverage for these programs achieved by random, manual, and KLEE tests. Unsurprisingly, given the complexity of COREUTILS programs and the concerted effort of the COREUTILS maintainers, the manual tests get significantly more coverage than random. KLEE handily beats both.

Because `gcov` introduces some overhead, we also performed a second experiment in which we ran each

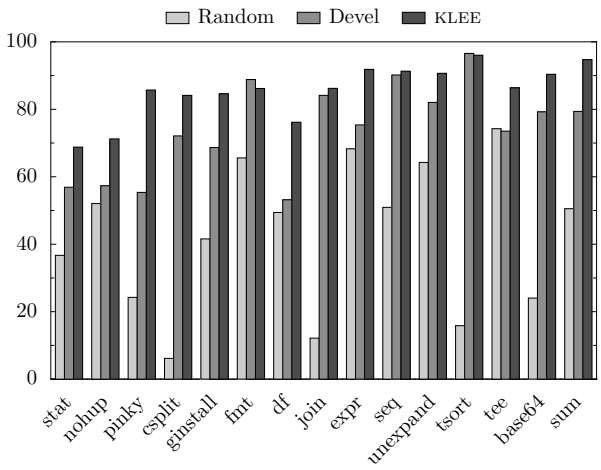


Figure 9: Coverage of random vs. manual vs. KLEE testing for 15 randomly-chosen COREUTILS utilities. Manual testing beats random on average, while KLEE beats both by a significant margin.

tool natively without `gcov` for 65 minutes (using the same random seed as the first run), recorded the number of test cases generated, and then reran using `gcov` for that number. This run completely eliminates the `gcov` overhead, and overall it generates 44% more tests than during the initial run.

However, these 44% extra tests increase the average coverage per tool by only 1%, with 11 out of 15 utilities not seeing any improvement — showing that random gets stuck for most applications. We have seen this pattern repeatedly in previous work: random quickly gets the cases it can, and then revisits them over and over. Intuitively, satisfying even a single 32-bit equality requires correctly guessing one value out of four billion. Correctly getting a sequence of such conditionals is hopeless. Utilities such as `csplit` (the worst performer), illustrate this dynamic well: their input has structure, and the difficulty of blindly guessing values that satisfy its rules causes most inputs to be rejected.

One unexpected result was that for 11 of these 15 programs, KLEE explores paths to termination (i.e., the checked code calls `exit()`) only a few times slower than random does! KLEE explored paths to termination in roughly the same time for three programs and, in fact, was actually faster for three others (`seq`, `tee`, and `nohup`). We were surprised by these numbers, because we had assumed a constraint-based tool would run orders of magnitude more slowly than raw testing on a per-path basis, but would have the advantage of exploring more unique paths over time (with all values) because it did not get stuck. While the overhead on four programs matched this expectation (where constraint solver overhead made paths ran 7x to 220x more slowly than

native execution), the performance tradeoff for the others is more nuanced. Assume we have a branch deep in the program. Covering both true and false directions using traditional testing requires running the program from start to finish twice: once for the true path and again for the false. In contrast, while KLEE runs each instruction more slowly than native execution, it only needs to run the instruction path before the branch once, since it forks execution at the branch point (a fast operation given its object-level copy-on-write implementation). As path length grows, this ability to avoid redundantly rerunning path prefixes gets increasingly important.

With that said, the reader should view the per-path costs of random and KLEE as very crude estimates. First, the KLEE infrastructure random uses to run tests adds about 13ms of per-test overhead, as compared to around 1ms for simply invoking a program from a script. This code runs each test case in a sandbox directory, makes a clean environment, and creates various system objects with random contents (e.g., files, pipes, tty's). It then runs the tested program with a watchdog to terminate infinite loops. While a dedicated testing tool must do roughly similar actions, presumably it could shave some milliseconds. However, this fixed cost matters only for short program runs, such as when the code exits with an error. In cases where random can actually make progress and explore deeper program paths, the inefficiency of rerunning path prefixes starts to dominate. Further, we conservatively compute the path completion rate for KLEE: when its time expires, roughly 30% of the states it has created are still alive, and we give it no credit for the work it did on them.

5.3 BUSYBOX utilities

BUSYBOX is a widely-used implementation of standard UNIX utilities for embedded systems that aims for small executable sizes [1]. Where there is overlap, it aims to replicate COREUTILS functionality, although often providing fewer features. We ran our experiments on a bug-patched version of BUSYBOX 1.10.2. We ran the 75 utilities [8] in the BUSYBOX "`coreutils`" subdirectory (14K lines of code, with another 16K of library code), using the same command lines as when checking COREUTILS, except we did not fail system calls.

As Table 2 shows, KLEE does even better than on COREUTILS: over 90.5% total line coverage, on average covering 93.5% per tool with a median of 97.5%. It got 100% coverage on 31 and over 90% on 55 utilities.

BUSYBOX has a less comprehensive manual test suite than COREUTILS (in fact, many applications don't seem to have any tests). Thus, KLEE beats the developers tests by roughly a factor of two: 90.5% total line coverage versus only 44.8% for the developers' suite. The developers do better on only one benchmark, `cp`.

[8] We are actually measuring coverage on 72 files because several utilities are implemented in the same file.

```
date -I
ls --co
chown a.a -
kill -l a
setuidgid a ""
printf "% *" B
od t1.txt
od t2.txt
printf %
printf %Lo
tr [
tr [=
tr [a-z
```

t1.txt: `a`
t2.txt: `A`
t3.txt: `\t\n`

```
cut -f t3.txt
install --m
nmeter -
envdir
setuidgid
envuidgid
envdir -
arp -Ainet
tar tf_ /
top d
setarch "" ""
<full-path>/linux32
<full-path>/linux64
hexdump -e ""
ping6 -
```

Figure 10: KLEE-generated command lines and inputs (modified for readability) that cause program crashes in BUSYBOX. When multiple applications crash because of the same shared (buggy) piece of code, we group them by shading.

5.4 Bug-finding: MINIX + all BUSYBOX tools

To demonstrate KLEE's applicability to bug finding, we used KLEE to check all 279 BUSYBOX tools and 84 MINIX tools [4] in a series of short runs. These 360+ applications cover a wide range of functionality, such as networking tools, text editors, login utilities, archiving tools, etc. While the tests generated by KLEE during these runs are not sufficient to achieve high coverage (due to incomplete modeling), we did find many bugs quickly: 21 bugs in BUSYBOX and another 21 in MINIX have been reported (many additional reports await inspection). Figure 10 gives the command lines for the BUSYBOX bugs. All bugs were memory errors and were fixed promptly, with the exception of `date` which had been fixed in an unreleased tree. We have not heard back from the MINIX developers.

5.5 Checking tool equivalence

Thus far, we have focused on finding generic errors that do not require knowledge of a program's intended behavior. We now show how to do much deeper checking, including verifying full functional correctness on a finite set of explored paths.

KLEE makes no approximations: its constraints have perfect accuracy down to the level of a single bit. If KLEE reaches an `assert` and its constraint solver states the false branch of the `assert` cannot execute given the current path constraints, then it has *proved* that no value exists on *the current path* that could violate the assertion,

```
1 : unsigned mod_opt(unsigned x, unsigned y) {
2 :   if((y & −y) == y) // power of two?
3 :     return x & (y−1);
4 :   else
5 :     return x % y;
6 : }
7 : unsigned mod(unsigned x, unsigned y) {
8 :   return x % y;
9 : }
10: int main() {
11:   unsigned x,y;
12:   make_symbolic(&x, sizeof(x));
13:   make_symbolic(&y, sizeof(y));
14:   assert(mod(x,y) == mod_opt(x,y));
15:   return 0;
16: }
```

Figure 11: Trivial program illustrating equivalence checking. KLEE proves total equivalence when $y \neq 0$.

modulo bugs in KLEE or non-determinism in the code. [9] Importantly, KLEE will do such proofs for any condition the programmer expresses as C code, from a simple non-null pointer check, to one verifying the correctness of a program's output.

This property can be leveraged to perform deeper checking as follows. Assume we have two procedures `f` and `f'` that take a single argument and purport to implement the same interface. We can verify functional equivalence on a per-path basis by simply feeding them the same symbolic argument and asserting they return the same value: `assert(f(x) == f'(x))`. Each time KLEE follows a path that reaches this assertion, it checks if any value exists on that path that violates it. If it finds none exists, then it has proven functional equivalence on that path. By implication, if one function is correct along the path, then equivalence proves the other one is as well. Conversely, if the functions compute different values along the path and the `assert` fires, then KLEE will produce a test case demonstrating this difference. These are both powerful results, completely beyond the reach of traditional testing. One way to look at KLEE is that it automatically translates a path through a C program into a form that a theorem prover can reason about. As a result, proving path equivalence just takes a few lines of C code (the assertion above), rather than an enormous manual exercise in theorem proving.

Note that equivalence results only hold on the finite set of paths that KLEE explores. Like traditional testing, it cannot make statements about paths it misses. However, if KLEE is able to exhaust all paths then it has shown total equivalence of the functions. Although not tractable in general, many isolated algorithms can be tested this way, at least up to some input size.

We help make these points concrete using the contrived example in Figure 11, which crosschecks a trivial modulo implementation (`mod`) against one that optimizes for modulo by powers of two (`mod_opt`). It first makes the inputs `x` and `y` symbolic and then uses the `assert` (line 14) to check for differences. Two code paths reach this `assert`, depending on whether the test for power-of-two (line 2) succeeds or fails. (Along the way, KLEE generates a division-by-zero test case for when $y = 0$.) The true path uses the solver to check that the constraint $(y\& - y) == y$ implies $(x\&(y-1)) == x\%y$ holds for all values. This query succeeds. The false path checks the vacuous tautology that the constraint $(y\& - y) \neq y$ implies that $x\%y == x\%y$ also holds. The KLEE checking run then terminates, which means that KLEE has proved equivalence for all non-zero values using only a few lines of code.

This methodology is useful in a broad range of contexts. Most standardized interfaces — such as libraries, networking servers, or compilers — have multiple implementations (a partial motivation for and consequence of standardization). In addition, there are other common cases where multiple implementations exist:

1. `f` is a simple reference implementation and `f'` a real-world optimized version.
2. `f'` is a patched version of `f` that purports only to remove bugs (so should have strictly fewer crashes) or refactor code without changing functionality.
3. `f` has an inverse, which means we can change our equivalence check to verify $f^{-1}(f(x)) \equiv x$, such as: `assert(uncompress(compress(x))==x)`.

Experimental results. We show that this technique can find deep correctness errors and scale to real programs by crosschecking 67 COREUTILS tools against their allegedly equivalent BUSYBOX implementations. For example, given the same input, the BUSYBOX and COREUTILS versions of `wc` should output the same number of lines, words and bytes. In fact, both the BUSYBOX and COREUTILS tools intend to conform to IEEE Standard 1003.1 [3] which specifies their behavior.

We built a simple infrastructure to make crosschecking automatic. Given two tools, it renames all their global symbols and then links them together. It then runs both with the same symbolic environment (same symbolic arguments, files, etc.) and compares the data printed to `stdout`. When it detects a mismatch, it generates a test case that can be run to natively to confirm the difference.

Table 3 shows a subset of the mismatches found by KLEE. The first three lines show hard correctness errors (which were promptly fixed by developers), while the others mostly reveal missing functionality. As an example of a serious correctness bug, the first line gives the inputs that when run on BUSYBOX's `comm` causes it to behave as if two non-identical files were identical.

[9] Code that depends on the values of memory addresses will not satisfy determinism since KLEE will almost certainly allocate memory objects at different addresses than native runs.

Input	BUSYBOX	COREUTILS
`comm t1.txt t2.txt`	[does not show difference]	[shows difference]
`tee -`	[does not copy twice to stdout]	[does]
`tee "" <t1.txt`	[infinite loop]	[terminates]
`cksum /`	`"4294967295 0 /"`	`"/: Is a directory"`
`split /`	`"/: Is a directory"`	
`tr`	[duplicates input on stdout]	`"missing operand"`
`[ 0 ''<'' 1 ]`		`"binary operator expected"`
`sum -s <t1.txt`	`"97 1 -"`	`"97 1"`
`tail -2l`	[rejects]	[accepts]
`unexpand -f`	[accepts]	[rejects]
`split -`	[rejects]	[accepts]
`ls --color-blah`	[accepts]	[rejects]
t1.txt: a *t2.txt:* b		

Table 3: Very small subset of the mismatches KLEE found between the BUSYBOX and COREUTILS versions of equivalent utilities. The first three are serious correctness errors; most of the others are revealing missing functionality.

Test	Random	KLEE	ELOC
With Disk	50.1%	67.1%	4617
No Disk	48.0%	76.4%	2662

Table 4: Coverage on the HISTAR kernel for runs with up to three system calls, configured with and without a disk. For comparison we did the same runs using random inputs for one million trials.

5.6 The HiStar OS kernel

We have also applied KLEE to checking non-application code by using it to check the HiStar [39] kernel. We used a simple test driver based on a user-mode HISTAR kernel. The driver creates the core kernel data structures and initializes a single process with access to a single page of user memory. It then calls the test function in Figure 12, which makes the user memory symbolic and executes a predefined number of system calls using entirely symbolic arguments. As the system call number is encoded in the first argument, this simple driver effectively tests all (sequences of) system calls in the kernel.

Although the setup is restrictive, in practice we have found that it can quickly generate test cases — sequences of system call vectors and memory contents — which cover a large portion of the kernel code and uncover interesting behaviors. Table 4 shows the coverage obtained for the core kernel for runs with and without a disk. When configured with a disk, a majority of the uncovered code can only be triggered when there are a large number of kernel objects. This currently does not happen in our testing environment; we are investigating ways to exercise this code adequately during testing. As a quick comparison, we ran one million random tests through the same driver (similar to § 5.2.4). As Table 4 shows, KLEE's tests achieve significantly more coverage than random testing both for runs with (+17.0%) and without (+28.4%) a disk.

```
static void test(void *upage, unsigned num_calls) {
  make_symbolic(upage, PGSIZE);
  for (int i=0; i<num_calls; i++) {
    uint64_t args[8];
    for (int j=0; j<8; j++)
      make_symbolic(&args[j], sizeof(args[j]));
    kern_syscall(args[0], args[1], args[2], args[3],
                 args[4], args[5], args[6], args[7]);
  }
  sys_self_halt();
}
```

Figure 12: Test driver for HISTAR: it makes a single page of user memory symbolic and executes a user-specified number of system calls with entirely symbolic arguments.

KLEE's constraint-based reasoning allowed it to find a tricky, critical security bug in the 32-bit version of HISTAR. Figure 13 shows the code for the function containing the bug. The function `safe_addptr` is supposed to set `*of` to true if the addition overflows. However, because the inputs are 64 bit long, the test used is insufficient (it should be `(r < a) || (r < b)`) and the function can fail to indicate overflow for large values of `b`.

The `safe_addptr` function validates user memory addresses prior to copying data to or from user space. A kernel routine takes a user address and a size and computes if the user is allowed to access the memory in that range; this routine uses the overflow to prevent access when a computation could overflow. This bug in computing overflow therefore allows a malicious process to gain access to memory regions outside its control.

6 Related Work

Many recent tools are based on symbolic execution [11, 14–16, 20–22, 24, 26, 27, 36]. We contrast how KLEE deals with the environment and path explosion problems.

To the best of our knowledge, traditional symbolic ex-

```
1 : uintptr_t safe_addptr(int *of, uint64_t a, uint64_t b) {
2 :     uintptr_t r = a + b;
3 :     if (r < a)
4 :         *of = 1;
5 :     return r;
6 : }
```

Figure 13: HISTAR function containing an important security vulnerability. The function is supposed to set `*of` to true if the addition overflows but can fail to do so in the 32-bit version for very large values of `b`.

ecution systems [17, 18, 32] are static in a strict sense and do not interact with the running environment at all. They either cannot handle programs that make use of the environment or require a complete working model. More recent work in test generation [16, 26, 36] does allow external interactions, but forces them to use entirely concrete procedure call arguments, which limits the behaviors they can explore: a concrete external call will do exactly what it did, rather than all things it could potentially do. In KLEE, we strive for a functional balance between these two alternatives; we allow both interaction with the outside environment and supply a model to simulate interaction with a symbolic one.

The path explosion problem has instead received more attention [11, 22, 24, 27, 34]. Similarly to the search heuristics presented in Section 3, search strategies proposed in the past include Best First Search [16], Generational Search [27], and Hybrid Concolic Testing [34]. Orthogonal to search heuristics, researchers have addressed the path explosion problem by testing paths compositionally [8, 24], and by tracking the values read and written by the program [11].

Like KLEE, other symbolic execution systems implement their own optimizations before sending the queries to the underlying constraint solver, such as the simple syntactic transformations presented in [36], and the *constraint subsumption* optimization discussed in [27].

Similar to symbolic execution systems, model checkers have been used to find bugs in both the design and the implementation of software [10, 12, 19, 25, 29, 30]. These approaches often require a lot of manual effort to build test harnesses. However, the approaches are somewhat complementary to KLEE: the tests KLEE generates can be used to drive the model checked code, similar to the approach embraced by Java PathFinder [31, 37].

Previously, we showed that symbolic execution can find correctness errors by crosschecking various implementations of the same library function [15]; this paper shows that the technique scales to real programs. Subsequent to our initial work, others applied similar ideas to finding correctness errors in applications such as network protocol implementations [13] and PHP scripts [9].

7 Conclusion

Our long-term goal is to take an arbitrary program and routinely get 90%+ code coverage, crushing it under test cases for all interesting inputs. While there is still a long way to go to reach this goal, our results show that the approach works well across a broad range of real code. Our system KLEE, automatically generated tests that, on average, covered over 90% of the lines (in aggregate over 80%) in roughly 160 complex, system-intensive applications "out of the box." This coverage significantly exceeded that of their corresponding hand-written test suites, including one built over a period of 15 years.

In total, we used KLEE to check 452 applications (with over 430K lines of code), where it found 56 serious bugs, including ten in COREUTILS, arguably the most heavily-tested collection of open-source applications. To the best of our knowledge, this represents an order of magnitude more code and distinct programs than checked by prior symbolic test generation work. Further, because KLEE's constraints have no approximations, its reasoning allow it to prove properties of paths (or find counter-examples without false positives). We used this ability both to prove path equivalence across many real, purportedly identical applications, and to find functional correctness errors in them.

The techniques we describe should work well with other tools and provide similar help in handling a broad class of applications.

8 Acknowledgements

We thank the GNU COREUTILS developers, particularly the COREUTILS maintainer Jim Meyering for promptly confirming our reported bugs and answering many questions. We similarly thank the developers of BUSYBOX, particularly the BUSYBOX maintainer, Denys Vlasenko. We also thank Nickolai Zeldovich, the designer of HISTAR, for his great help in checking HISTAR, including writing a user-level driver for us. We thank our shepard Terence Kelly, the helpful OSDI reviewers, and Philip Guo for valuable comments on the text. This research was supported by DHS grant FA8750-05-2-0142, NSF TRUST grant CCF-0424422, and NSF CAREER award CNS-0238570-001. A Junglee Graduate Fellowship partially supported Cristian Cadar.

References

[1] Busybox. `www.busybox.net`, August 2008.

[2] Coreutils. `www.gnu.org/software/coreutils`, August 2008.

[3] IEEE Std 1003.1, 2004 edition. `www.unix.org/version3/ieee_std.html`, May 2008.

[4] MINIX 3. `www.minix3.org`, August 2008.

[5] SecurityFocus, `www.securityfocus.com`, March 2008.

[6] uCLibc. `www.uclibc.org`, May 2008.

[7] United States National Vulnerability Database, `nvd.nist.gov`, March 2008.

[8] ANAND, S., GODEFROID, P., AND TILLMANN, N. Demand-driven compositional symbolic execution. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*.

[9] ARTZI, S., KIEŻUN, A., DOLBY, J., TIP, F., DIG, D., PARADKAR, A., AND ERNST, M. D. Finding bugs in dynamic web applications. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2008)*.

[10] BALL, T., AND RAJAMANI, S. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN 2001)*.

[11] BOONSTOPPEL, P., CADAR, C., AND ENGLER, D. RWset: Attacking path explosion in constraint-based test generation. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*.

[12] BRAT, G., HAVELUND, K., PARK, S., AND VISSER, W. Model checking programs. In *IEEE International Conference on Automated Software Engineering (ASE 2000)*.

[13] BRUMLEY, D., CABALLERO, J., LIANG, Z., NEWSOME, J., AND SONG, D. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *Proceedings of USENIX Security Symposium (USENIX Security 2007)*.

[14] BRUMLEY, D., NEWSOME, J., SONG, D., WANG, H., AND JHA, S. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy (IEEE S&P 2006)*.

[15] CADAR, C., AND ENGLER, D. Execution generated test cases: How to make systems code crash itself. In *Proceedings of the 12th International SPIN Workshop on Model Checking of Software (SPIN 2005)*.

[16] CADAR, C., GANESH, V., PAWLOWSKI, P., DILL, D., AND ENGLER, D. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS 2006)*.

[17] CLARKE, E., AND KROENING, D. Hardware verification using ANSI-C programs as a reference. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC 2003)*.

[18] CLARKE, E., KROENING, D., AND LERDA, F. A tool for checking ANSI-C programs. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*.

[19] CORBETT, J., DWYER, M., HATCLIFF, J., LAUBACH, S., PASAREANU, C., ROBBY, AND ZHENG, H. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the International Conference on Software Engineering (ICSE 2000)*.

[20] COSTA, M., CASTRO, M., ZHOU, L., ZHANG, L., AND PEINADO, M. Bouncer: Securing software by blocking bad input. In *Proceedings of the 21th ACM Symposium on Operating Systems Principles (SOSP 2007)*.

[21] COSTA, M., CROWCROFT, J., CASTRO, M., ROWSTRON, A., ZHOU, L., ZHANG, L., AND BARHAM, P. Vigilante: end-to-end containment of Internet worms. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP 2005)*.

[22] EMMI, M., MAJUMDAR, R., AND SEN, K. Dynamic test input generation for database applications. In *International Symposium on Software Testing and Analysis (ISSTA 2007)*.

[23] GANESH, V., AND DILL, D. L. A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV 2007)*.

[24] GODEFROID, P. Compositional dynamic test generation. In *Proceedings of the 34th Symposium on Principles of Programming Languages (POPL 2007)*.

[25] GODEFROID, P. Model Checking for Programming Languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL 1997)*.

[26] GODEFROID, P., KLARLUND, N., AND SEN, K. DART: Directed automated random testing. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI 2005)*.

[27] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. Automated whitebox fuzz testing. In *Proceedings of Network and Distributed Systems Security (NDSS 2008)*.

[28] HOFFMANN, J., AND KOEHLER, J. A new method to index and query sets. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI 1999)*.

[29] HOLZMANN, G. J. From code to models. In *Proceedings of 2nd International Conference on Applications of Concurrency to System Design (ACSD 2001)*.

[30] HOLZMANN, G. J. The model checker SPIN. *Software Engineering 23*, 5 (1997), 279–295.

[31] KHURSHID, S., PASAREANU, C. S., AND VISSER, W. Generalized symbolic execution for model checking and testing. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003)*.

[32] KROENING, D., CLARKE, E., AND YORAV, K. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of the 40th Design Automation Conference (DAC 2003)*.

[33] LATTNER, C., AND ADVE, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization (CGO 2004)*.

[34] MAJUMDAR, R., AND SEN, K. Hybrid concolic testing. In *Proceedings of the 29th International Conference on Software Engineering (ICSE 2007)*.

[35] MILLER, B., KOSKI, D., LEE, C. P., MAGANTY, V., MURTHY, R., NATARAJAN, A., AND STEIDL, J. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Tech. rep., University of Wisconsin - Madison, 1995.

[36] SEN, K., MARINOV, D., AND AGHA, G. CUTE: A concolic unit testing engine for C. In *In 5th joint meeting of the European Software Engineering Conference and ACM Symposium on the Foundations of Software Engineering (ESEC/FSE 2005)*.

[37] VISSER, W., PASAREANU, C. S., AND KHURSHID, S. Test input generation with Java PathFinder. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2004)*.

[38] YANG, J., SAR, C., AND ENGLER, D. eXplode: a lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006)*.

[39] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIÈRES, D. Making information flow explicit in HiStar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006)*.