Automatically Generating Malicious Disks using Symbolic Execution

Junfeng Yang, Can Sar, Paul Twohey, Cristian Cadar and Dawson Engler
Stanford University
Computer Systems Laboratory
{junfeng,csar,twohey,cristic,engler}@cs.stanford.edu

Abstract

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 file system images is complex: they form DAGs with complex dependency relationships across massive amounts of data bound together with intricate, undocumented assumptions. This paper shows how to automatically find bugs in such code using symbolic execution. Rather than running the code on manually-constructed concrete input, we instead run it on symbolic input that is initially allowed to be "anything." As the code runs, it observes (tests) this input and thus constrains its possible values. We generate test cases by solving these constraints for concrete values. The approach works well in practice: we checked the disk mounting code of three widely-used Linux file systems: ext2, ext3, and JFS and found bugs in all of them where malicious data could either cause a kernel panic or form the basis of a buffer overflow attack.

1 Introduction

Many current systems allow data produced by potentially malicious sources to be mounted as a file system. Just as network code must sanitize all packet data and system calls must check all parameters before use, file system code must vet the data it mounts to ensure that all explicit (and implicit) invariants are obeyed by the candidate disk. For example, inode indexes and block numbers should always be within prescribed bounds and any counts employed in a division operation should not be zero.

Since file systems typically run as privileged code inside the kernel, a single unchecked value can, at the least, crash the machine or at worst lead to an exploit. Bitter experience has shown the difficulty of validating network data. Unfortunately, validating an allegedly safe file system image is more complex: network packets have a simple, linear structure whereas file system data structures bind massive amounts of data into complex DAGs full of intricate, undocumented assumptions. Further, culturally file system designers lack the same level of paranoia as network implementors.

All of these factors make disk-focused attacks relatively easy. We discuss three possible attacks. The first, most general attack is for a bad person to generate a malevolent disk image and give it to a good, trusting person (e.g., on a USB memory stick or CD-ROM), who mounts it and then suffers. This ability could also be used by a virus that infects a user's account, perhaps through a buggy mail client or web browser, and writes a few choice malicious blocks to the user's removable media, thus crashing the next machine the user inserts the media into.

Second, on a system that allows unprivileged users to mount devices (the common case), a bad person with physical access to the machine can crash it without an intermediary by just mounting the malevolent media themselves. Even though physical access is often equated with root access, disk mounting attacks lower the barrier to entry, especially on public machines. No one thinks twice about a lab user inserting a CD, but someone unscrewing a computer and removing the hard disk is much more suspicious.

Finally, systems increasingly let unprivileged users mount arbitrary data as a file system. This ability enables the worst attack: using the file system to crash the machine or escalate privileges via a buffer overflow without physical access to the machine. Mac OS X lets unprivileged users mount normal files as file systems as part of the preferred software distribution mechanism [1]. Linux provides similar functionality with loop back mounts, which user-friendly distributions like Linspire [2] enable for regular users.

While file system validation bugs can have serious consequences, they are difficult to eliminate. The structure of a typical file system makes manual inspection even more erratic than usual: deeply nested conditionals and function call chains, the sheer mass of code, prolific use of casting and pointer operations, and difficult to follow dynamic dispatch calls through function pointers (such as in the VFS interface). In addition, the checks that must be done can be quite tricky, especially in the presence of arithmetic overflow, which programmers reason about poorly. Random testing faces its own difficulties. Bugs from arithmetic overflow often occur only for a narrow input range, making finding them with random test cases unlikely. Further, much of the file system code resides behind a thicket of deeply nested conditionals that vet the initial disk: reaching this code means that random testing must correctly guess all values that the conditionals depend on. For example, the Linux ext2 "read super block" routine has over forty if-statements checking the data associated with the super block. Any randomly generated super block must satisfy these tests before it can reach even the next level of vetting, much less triggering the execution of "real code" that performs actual file system operations. Worse, many conditionals are equalities on 32-bit values: hitting the exact value that will satisfy even one such conditional will probably require billions of attempts. More than a few is completely hopeless.

This paper shows how to automatically find bugs in file system code using the symbolic execution system EXE ("EXecution generated Executions") we developed in prior work [10]. 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 the disk as symbolic input to the kernel, which we then run to produce constraints and test cases.

Instead of running code on manually generated test cases, EXE instruments a program and runs it on symbolic input that is initially free to have any value. As the code executes, the data is "interrogated;" the results of conditional expressions and other operations incrementally inform EXE of 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.

This approach has several nice features. First, unlike most checking approaches it is constructive: when it finds an error, it gives an actual, concrete input to run through the code that will trigger the error. From this point of view EXE can be viewed as an automatic way to generate disk images that enable exploits. Furthermore, this constructiveness means that EXE has no false positives. Any input it claims causes an error can be (automatically) fed back into an uninstrumented version of the checked code to verify that the error does indeed occur. As a result, users do not have to trust that EXE worked correctly — they can verify themselves that the input causes the code to crash before inspecting it. This test case can be saved for later regressions.

Second, it exponentially amplifies the effect of running a single code path since it simultaneously reasons about many possible values that the path could be run with (all those that satisfy the current path's constraints), rather than a single set of concrete values from an individual test case. In fact, if EXE has a solvable, accurate, complete set of path constraints — no constraints were missed because symbolic values were used in uninstrumented code, no premature concretization occurred (see § 3.4) — then EXE reasons about *all possible values* that the path could execute with. To illustrate, a dynamic memory checker such as Purify[22] will only catch an out-of-bounds array access if the index (or base pointer) has a bad value at the time of memory access for the specific set of input values the code was run with. 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 index to the array (modulo the caveats above). In addition, for an arithmetic expression that uses symbolic data, EXE can solve the associated constraints for values to cause an overflow or a division by zero. Furthermore, the system does not just check values on a single path, but will forcibly construct input values to (ideally) go down all paths, getting coverage out of practical reach from random or manual testing.

This amplification effect has the potential to channel traditional quality assurance efforts towards finding security vulnerabilities. Security exploits are difficult to find with standard testing techniques because they usually arise from uncommon interactions and corner cases for which test generation is hard. Symbolic execution specializes in generating new and different inputs that drive a system into novel states. Thus, the same effort needed to create a testing framework can also be leveraged to create an environment that searches for security problems.

Finally, the approach works. We checked the disk mounting code of three widely-used Linux file systems — ext2, ext3[15] and JFS[24] — and found bugs in all of them.

The paper is organized as follows: Section 2 gives an overview of our approach, Section 3 provides more details about symbolic execution (including its limitations), Section 4 details the changes we needed in order to make EXE work with Linux, Section 5 describes the bugs we found, Section 6 discusses related work, and Section 7 concludes.

2 System Overview

This section gives an overview of our approach, graphically sketched in Figure 1. At a high level the system consists of four pieces:

1. A trivial test driver that issues a `mount` system call to cause the kernel to mount a symbolic disk as a file system.
2. A modified version of the User Mode Linux kernel [3] containing the file systems we are testing. These modifications were mostly done in prior work [37] and consist of simplifications such as removing threading and changing kernel memory allocators to call into the EXE runtime. Section 4 provides more details.
3. A (virtual) disk driver that manages the symbolic disk.
4. The EXE system consisting of the EXE compiler (`exe-cc`), which instruments code for symbolic execution and the constraint management runtime which interfaces with the STP [1] constraint solver.

The first three pieces are compiled with `exe-cc`, and the resultant executable is then run with the EXE runtime, causing the following to happen:

1. The test driver tries to mount a file system, causing the file system to request disk blocks from the virtual disk driver.
2. At each block request, the virtual driver checks if the file system has requested the block before. If not, it creates a new, unconstrained symbolic block whose contents are initially "anything," by (1) allocating a block of memory as large as the disk read, (2) calling into the EXE runtime system to mark the memory as symbolic, and (3) returning a pointer to the memory back to the file system. Otherwise it returns a pointer to the previously read copy, implicitly preserving any existing constraints on the block.
3. As the file system uses and observes symbolic blocks, constraints are generated and tracked by the EXE runtime.
4. When EXE detects an error or the `mount` system call finishes and returns to the test driver, EXE generates a concrete disk image by asking the constraint solver for a solution to the current set of constraints — literally the actual 0 and 1 values of bits that will satisfy the disk constraints on the current path.
5. Each disk image generated by EXE in response to an error is then mounted on an *uninstrumented* version of the kernel to verify that it does indeed cause an error. All errors found this way will be true errors since they are caused by a real input that does not depend on EXE in any way. Note that even disk images that cause no obvious bugs are useful test cases since they provide a way to run many paths in the code, aiding general correctness testing.

EXE is dynamic: it literally runs the checked program. Thus, EXE has access to all the information that a dynamic analysis has (and a static analysis typically does not). All non-symbolic operations happen exactly as they would in uninstrumented code, and produce exactly the same values. Thus, when these values appear in constraints they are correct, not approximations. Symbolic expressions are also exactly accurate. EXE models all of the

[1]Written by David Dill and Vijay Ganesh, STP departs from the decades old standard approach of using Nelson and Oppen's cooperating decision procedure framework [30] to solve constraints, and instead just preprocesses and then bitblasts constraints to SAT, which it solves using MiniSat [14]. The STP approach is much simpler and preliminary results suggest it is significantly faster than the traditional method.

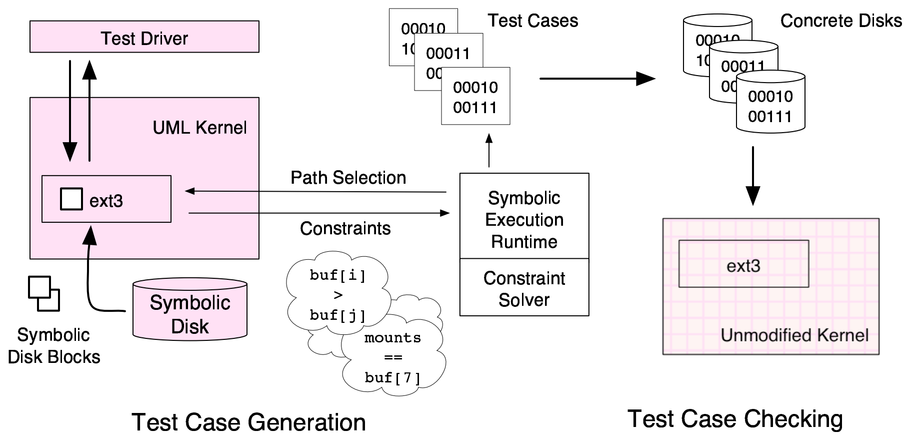

Figure 1. Symbolic execution overview. Instrumented file system code (shaded, on left) gets symbolic blocks from our symbolic block device. When the code branches on the result of a symbolic operation, the symbolic execution runtime tries to run the code through both the true and the false branches and generate concrete test cases. Generated tests are then checked by running them on an unmodified kernel (thatched, on right).

C language, and works in the presence of pointers, unions, bit-fields, casts (even between integers and pointers), and aggressive bit-operations such as shifting, masking, byte swapping, and checksumming. The only way that EXE loses even a single bit of precision (i.e., it lets a bit be either 0 or 1 when it could not be) on a given path is if (1) EXE is missing constraints, e.g., because the checked system called uninstrumented code (such as assembly) or (2) EXE has a bug. Section 3.4 discusses causes of lost constraints in more detail.

In our context, what bit-level precision means is that if (1) EXE has the full set of constraints for a given path, (2) the constraint solver can produce a concrete solution, and (3) the code is deterministic then, (4) rerunning the checked system on these concrete values will cause the file system code to follow the same exact path to the error or termination that generated the image.

Handling exponential branching. EXE aims to get path coverage. While in general the number of paths grows (roughly) exponential with the total lines of code, our domain is more manageable: only branches on symbolic expressions cause forked executions. In the code we check, most branches involve non-symbolic conditions, which means they execute concretely (as in uninstrumented code) with linear cost. The caveat to this is loops. A simple loop that compares a counter to a symbolic bound can run until the counter reaches the (potentially enormous) maximum value the symbolic could possibly contain. And if this was not already expensive enough, more complex symbolic loop conditions can, of course, run longer.

EXE currently handles loops using search heuristics. When EXE forks execution it can chose which branch to follow (or whether to resume the child of an unexplored prior branch). By default EXE uses depth-first search (DFS) to keep the number of processes small (linear in the depth of the process chain). In addition, when forking on loop conditions it will (by default) explore the false branch first, which means that loops run zero times, then all paths in the loop are run once (and each exits after), then all combinations run twice, etc.

Unfortunately, simple DFS works poorly in some cases since it cannot backtrack. For example, if checked code has two consecutive loops, L1 and L2, DFS will get "stuck" on L2 and be unable to backtrack to L1 until it has executed L2 as many times as possible. To counter this, EXE provides a set of heuristics to guide search. Our current favorite uses a mixture of "best-first" and DFS search. It picks the process that will execute the line of code run the fewest number of times. It then runs this process (and its children) in a DFS manner for a while. It then picks another best-first candidate and iterates.

Despite these challenges, our experiments needed less than an hour to generate tests that trigger the errors we discuss. While the `mount` system call implementations checked in this paper have complex control flow, they do little symbolic looping.

However, in some sense we are happy to let the system run for weeks. As long as the tests EXE generates explore paths difficult to reach randomly (as most are), then the only real alternative is manual test generation, which (generally speaking) has not had impressive results. Finally, once the tests are generated, they can be run on the uninstrumented, checked program at full speed and saved for later regression runs.

We describe EXE in more detail in the next section, including limitations that require a bit more technical background, and then discuss issues in applying EXE to Linux.

3 Symbolic Execution

EXE has one main goal: at any point on a program path to have an accurate, complete set of all constraints on symbolic input for that path. When EXE can both see and solve the constraints on a given path (it cannot always), it can do two useful things: (1) drive execution down all paths and (2) use a path's constraints to check if any possible input value exists that could cause an error such as division by zero or an invalid dereference at any point on that path. Our entire motivation for working on EXE is the hope of achieving all path coverage plus all value checking for large amounts of code.

This section gives a high-level overview of the key features of EXE needed to check the file systems code in this paper: (1) the mechanics of supporting symbolic execution (accurately tracking path constraints and executing all paths), (2) its universal checks, and (3) how it models memory (so that an expression's constraints reflect all possible memory locations that the expression could refer to). We close by discussing EXE's limitations, including the cases when it cannot track all constraints. For more operational detail, the interested reader can refer to [10], which introduced EXE.

3.1 Instrumentation

The first step in using EXE is to compile the code to check using `exe-cc`, which uses the CIL front-end [29] to instrument the checked program for symbolic execution. The inserted instrumentation has two primary tasks: (1) supporting mixed concrete and symbolic execution and (2) exploring all program choices by forking program execution when a symbolic value could cause several different actions. We discuss each below.

EXE supports mixed concrete and symbolic execution by inserting dynamic checks around every expression, such as assignments, dereferences, and conditionals. If an expression's operands are concrete, then the expression is performed concretely (i.e., as in the original, uninstrumented program). If any of its operands are symbolic, the expression is not performed, but instead the EXE runtime system adds it as a constraint. For example, given the expression `x = y + z`, EXE checks if `y` and `z` are concrete and, if so, lets the expression execute and records that `x` holds a concrete value. If `y` or `z` (or both) are symbolic, EXE instead just adds the constraint $x = y + z$ and records that `x` corresponds to a symbolic value.

EXE is designed to explore everything "interesting" that can happen because of an input value. It explores both branches of a symbolic conditional (if they are possible) by (1) literally using the `fork` system call to clone execution and (2) adding on the true path the symbolic constraint that the branch condition is true, and on the false path that it is not. As an example, consider the if-statement `if(x*x + y*y == z*z)`. If `x`, `y`, and `z` are concrete, then execution happens as normal: the expression is evaluated and if true, the true branch is taken, otherwise the false branch is. However, if `x`, `y` or `w` is symbolic, then EXE forks execution and on the true path asserts that $y \times y + x \times x = z \times z$ is true, and on the false path that it is not. (Note: a non-symbolic variable involved in an expression will be concretely evaluated, and its value encoded as a constant in the constraint.) Figure 2 gives the rewrite transformation for conditional statements.

EXE uses forking in two other situations as a way to drive execution into often-buggy corner cases: (1) arithmetic overflow and (2) casting surprises. EXE attempts to force overflow in each symbolic arithmetic operation as follows. It builds two symbolic expressions. The first encodes the operation at the precision specified by the program being tested, while the second encodes the operation at an essentially infinite precision. EXE then queries its constraint solver to see if the constraints on the current path ever allow these expressions to differ. If so, an

```
IF-TRANSFORMATION(Expr $e$, Stmt $s_1$, Stmt $s_2$)
  if is-symbolic($e$) $= \langle false \rangle$
      if $e$
          $s_1$
      else
          $s_2$
  else
      if fork() $= child$
          add-symbolic-constraint($e = true$)
          $s_1$
      else
          add-symbolic-constraint($e = false$)
          $s_2$
```

Figure 2. Rewrite transformation for conditional expressions `if(e)` s_1 `else` s_2**.**

arithmetic overflow is possible, and EXE generates a concrete test case that triggers the overflow.

Narrowing casts lose information and thus may add bugs. EXE checks if a truncation cast from an n-bit symbolic expression e to an m-bit symbolic expression ($m < n$) can lose bits in a manner similar to overflow checking. It builds a symbolic expression that extracts m bits from e and sign- or zero-extends it (as appropriate) back to n bits and then queries the constraint solver to see if the constructed expression could differ from e. If so, it forks execution into two different execution paths and adds constraints so that one path loses information and the other does not.

Casts from a signed to an unsigned value are a common source of security holes – if the signed variable is negative, its unsigned representation will be a very large value. Given such a cast on a symbolic value, EXE queries the constraint solver to check if the high bit of the symbolic value can be both 0 and 1. If so, EXE forks execution to create two paths: one where the sign bit is constrained to be 0, and another where it is constrained to be 1 (and thus will generate a large unsigned value).

3.2 Symbolic Checks = Power

A key advantage of symbolic execution over concrete is that concrete execution operates only on a single possible set of concrete values, whereas symbolic execution operates on all values that the current path constraints allow (modulo the power of the constraint solver). EXE uses this ability to provide several "universal" checks. When an execution reaches a program operation that gives an error for certain values (0 for division, null or out-of-bounds pointers for dereferences) EXE checks if any possible input value exists that (1) satisfies the current path constraints and (2) causes the operation to blow up. Such "all value" checking is a dramatic amplification over concretely checking a single value.

EXE does three universal checks: (1) that an integer divisor or modulus is never zero, (2) that a dereferenced symbolic pointer is never null, and (3) that a dereferenced pointer lies within a valid object. All checks follow the same general pattern: the front-end inserts the check at each relevant point in the checked program and calls the constraint solver to determine if the condition could occur. If so, EXE forks execution and (1) on one branch asserts that the condition does occur, emits a test case, and terminates; (2) on the false path asserts that the condition does not occur (e.g., that an index is in bounds, that a divisor is non-zero) and continues execution (to hunt for more bugs).

The most complex check is determining if a symbolic pointer dereference could be out of bounds. EXE tracks the size of each memory block and the block that each symbolic pointer should point within using techniques similar to CRED [32] or Purify [22]. Given this information, then for any dereference `*p`, EXE asks the constraint solver whether `p` could be outside its base object. If so, EXE produces a concrete assignment for the initial symbolic inputs that makes the concrete execution of the program perform an out-of-bounds memory access. It then adds the constraint that the pointer is within its object and continues execution. Using these checks, we found a complex buffer overflow error in the ext2 file system (replicated in ext3), which we discuss in Section 5.

Finally, note that EXE's goal of path coverage implicitly turns programmer `assert`s on a symbolic expression into universal checks of the asserted condition. When EXE hits an `assert` it will systematically search the set of constraints to try to reach the false path of the assert check (as with any conditional). If the `assert` passes, it was because EXE could not find any input that would violate it. If there is some input that lies within EXE's constraint solver's ability to solve, then it will find it. This exponentially amplifies the domain of a given assertion in code over just checking it for a single concrete value. (Note, more generally, any correctness check the programmer puts in their code will receive the same amplification since EXE will try to drive execution down all paths in the checking code, including the paths that catch errors.)

3.3 Modeling Memory

Two memory stores coexist during a run of an EXE program: (1) the *concrete store*, which is just the memory of the underlying machine (i.e., a flat byte array addressed by 32-bit pointers) and (2) the *symbolic store*, which resides inside of the constraint solver. The concrete store is what concrete operations act upon and includes the heap, stack, and data segments. The symbolic store includes both (1) the set of symbolic variables used in the current set of constraints (in addition to constants) and, less obviously, (2) the set of constraints themselves. Solving the symbolic store's constraints gives a concrete store. Thus, a symbolic store describes zero to many concrete stores: zero if its constraints have no solution, many if there are many different solutions. If the symbolic store is described by an accurate, complete set of constraints, then any solution is guaranteed to be a valid concrete store.

Concrete bytes holding concrete values have no corresponding storage in the symbolic store. All values start out as concrete. When the user marks a set of bytes as symbolic, EXE creates a corresponding, identically-sized range of bytes in the symbolic store, and records this correspondence in a hash table that maps byte addresses to their corresponding symbolic bytes. As the program executes, this table grows as more bytes become symbolic, either by assigning a symbolic expression to a concrete variable (parameter passing can be viewed as a form of assignment) or by indexing a data block by a symbolic index.

Accurately tracking constraints that only involve strongly-typed scalar variables is relatively simple: just name the variables uniquely (e.g., as in the original code) and use these names consistently in the constraints. There are two problems we had to handle for real C code: (1) that it treats memory as untyped bytes, and (2) it uses pointers. We give more detail below.

Untyped memory A simple, natural way to build the symbolic store would be to map each symbolic object in the textual program (such as a variable, a structure, an array) to a corresponding object (variable, structure, array) in the symbolic store, essentially associating a single type with each memory location. However, systems code often observes a single memory location in multiple ways. For example, by casting signed variables to unsigned, or (in the code we checked) treating an array of bytes as an inode or superblock structure.

Since C treats memory as untyped bytes, EXE does as well. It uses two STP primitives — bitvectors and arrays — to encode the memory associated with a symbolic object as an (untyped) STP array of 8-bit bitvector elements. Using bitvectors let us treat memory as untyped, using arrays let us handle pointers (discussed below). Each read of memory generates constraints based on the static type of the read (such as `int`, `unsigned`) but these types do not persist (other than in the single constraint generated by the symbolic expression that used the given read). Observing bits using an `unsigned` access does not impede a subsequent `signed` access. Both accesses are performed as expected, and their respective constraints are conjoined.

Symbolic pointers. Unlike scalars, symbolic pointers can refer to many different symbolic variables. For example, given an array `a` of size `n` and an in-bounds symbolic index `i`, then a simple boolean expression `(a[i] != 0)` essentially becomes a big disjunction:

```
  (i == 0 && a[0] != 0)
  || (i == 1 && a[1] != 0)
  || ...
  || (i == n-1 && a[n-1] != 0)
```

Similarly, the simple array assignment `a[i] = 42` could update any value in `a`.

While encoding such array expressions using a raw SAT-solver interface is tricky, by using STP we can just let it worry about such encoding complexity. Our main challenge is taking a given pointer dereference `*(p+i)` (where `p` or `i` could be symbolic) and mapping it to the correct STP array and index within that array. This mapping proceeds as follows. We map the pointer to its corresponding STP array in two steps. First, we (re)use EXE's machinery for checking out-of-bounds memory references (from § 3.2), which given the address of a concrete memory location `p` that holds a pointer will return the starting address of the memory block `b` that it (should) point into (`b = base(&p)`). Second, we then lookup this base address `b` in an auxiliary hash table to get its corresponding STP array name b_{sym}. If it has no symbolic counterpart we allocate one with initial values set to those in the current concrete memory location.

Given the symbolic array b_{sym} associated with `p` we then build a symbolic expression that gives the

```
1 : #include <assert.h>
2 : int main() {
3 :     unsigned char i, j, k, a[4] = {11, 13, 17, 19};
4 :     make_symbolic(&i); // these macros make
5 :     make_symbolic(&j); // i, j, and k
6 :     make_symbolic(&k); // symbolic
7 :     if(i >= 4 || j >= 4 || k >= 4) // force in-bounds
8 :             exit(0);
9 :     a[i] = 1;
10:     if ( (a[j] + a[k] == 14) )
11:             assert((i != 1));
12: }
```

Figure 3. A simple example using pointers.

(possibly symbolic) offset of `p` from the base of the concrete memory block it points to (i.e., $o = p - b$). We then add this to the original (possibly symbolic) offset i. The final symbolic expression $b_{sym}[i{+}o]$ can then be used in constraints to accurately refer to all possible set of symbolic locations that the original expression could point to.

The end result of these gyrations is that EXE can handle both reads and writes of pointer expressions where the pointer or the offset expression or both are symbolic.

To get a feel for what this ability means consider the code in Figure 3. When compiled with `exe-cc`, the program will execute correctly and produce no assertion violations. This code presents two main challenges. First, the assignment `a[i] = 1` uses the symbolic index `i`, which could refer to four different values in `a`. Thus, it creates a symbolic store that can generate four different concrete memory stores, depending on which element of `a` was overwritten. Second, both `a[j]` and `a[k]` read using symbolic indexes that could refer to any value in `a`. However, when EXE hits the `assert(a[j] + a[k] == 14)` statement on the true branch of the `if` statement on line 10, the number of possible concrete stores is reduced to three, because the only way in which this condition can be true is when the value 13 is still present in the array, i.e. when the second element is not overwritten. The program checks this fact using `assert`, which EXE proves true.

3.4 Limitations

EXE has several limitations, some ephemeral, some more fundamental. In the ephemeral category, EXE is still research quality so handling Linux is not always a smooth ride. The fact that a concrete test case is produced helps a lot in eliminating false positives. As a further check, EXE optionally tracks the basic blocks visited when generating a give case and will verify that the same path is executed when the concrete value is rerun on the checked code. This check found many bugs inside EXE.

There are two places where EXE replaces a symbolic value with a concrete, constant value ("concretization") or places additional constraints on it in order to make progress but discarding certain execution paths or values:

1. Because STP does not handle division or modulo by a symbolic value, when EXE encounters either operation it constrains the operand to be a power of two and replaces the division or modulo operation by a shift or bitwise mask, respectively.
2. Given a double-dereferences of a symbolic pointer such as `**p` (where `p` is symbolic) EXE will currently concretize the first dereference (`*p`), thereby fixing it to one of the possibly many storage locations it could refer to. (However, the result of `**p` can still be a symbolic expression.) This concretization is an artifact of the way we name arrays in STP; we are currently working on removing it.

There are several places where EXE will miss constraints:

1. If STP cannot solve the constraints on a path, EXE terminates that path. This termination has not happened when checking `mount` code, but could in general since constraint solving is an NP-hard problem.
2. If uninstrumented code (e.g. inline assembly or functions in files that cannot be correctly compiled by `exe-cc`) attempts to use symbolic values. The next section discusses where this occurs in the code we check.
3. Because of exponential branching (§ 2).

Currently the system is missing several features, none of which mattered for this paper:

1. STP does not correctly handle all operations on 64-bit primitives, so we are not able to generate proper constraints for code that makes extensive use of `long long` values. Fortunately, disk mounting code only uses such values in simple ways that STP can handle.
2. STP does not support floating-point operations.
3. EXE does not handle call through symbolic function pointers (which could be added by concretization) and may not correctly track symbolics passed to variable-length argument functions.

4 Applying EXE to Linux

Ideally, file systems would provide a unit testing framework that would let us automatically extract them from the host operating system and check them at user level with EXE. Unfortunately, in practice file systems are so tightly entwined with their host OS that *cutting* file system code from the rest of the kernel in such a manner is hopeless because it interacts with virtually every part of the kernel, from timers to the virtual memory layer and all its associated buffers. Even approximating a small portion of the kernel in order to check device drivers is very labor intensive and prone to inaccuracies, requiring many careful adjustments to the modeled functions [6, 28, 37].

Thus, we run most of the Linux kernel through `exe-cc`. By pushing the entire Linux kernel into our symbolic execution system we can check unanticipated interactions between file system and other kernel code. If we had attempted to model the Linux VFS implementation while checking JFS, we most likely would have missed the JFS bug we found because it involves rather complicated interactions with the Linux inode manipulation routines.

Rather than run Linux on the bare hardware, we instead use the CMC framework [28, 37] an adaptation of User Mode Linux (UML) to run the 2.4.19 kernel as an unprivileged user-level process. There are two reasons for this. First, the current EXE requires the ability to clone and wait for processes, operations that we cannot do to a Linux kernel running on hardware. Second, checking code running on the bare hardware makes many things unnecessarily difficult: debuggers run poorly, if at all; pointer errors reset the machine rather than causing a catchable segmentation fault; etc.

Our virtual disk driver mostly does what one would expect. Its main trick (as mentioned in § 2) is to lazily make blocks symbolic when they are read, as opposed to simply making the entire disk symbolic upfront. This laziness can make a big difference in speed. For example, the minimum size of a JFS disk is 16MB, and making the entire disk symbolic would generate a prohibitively expensive number of constraints right from the beginning. Lazily making individual disk blocks symbolic the first time they are inspected drastically reduces the number of constraints, which in turn allows us to check JFS.

Linux is a large piece of software that does exciting things. As a result, we had adapt EXE in a number of ways to work with Linux and Linux to work with EXE.

We made two modifications to EXE in response to checking Linux. First, given an assignment `v = e` of a symbolic expression `e` to a concrete variable `v`, EXE initially would always make `v` symbolic, even if `e` was constrained to be a single value. Checking for this special case and just assigning `v` the value that `e` was constrained to hold dramatically reduced the number of symbolics. Second, freeing a heap object and then reusing it causes problems in the current implementation of EXE because of the way we determine the base object of a pointer involved in a double-dereference, after we concretize it. We work-around this problem by not freeing heap objects if they are symbolic.

We modified Linux in several ways. First, EXE does not support threads. Fortunately, CMC gives us enough control over threading that we can easily disable it for the purposes of running the `mount` system call.

Second, EXE instruments C code, but the kernel uses a fair amount of hand-optimized assembly for common memory manipulation routines such as `memcpy` and `strlen`. While these improve runtime performance in a real kernel, they cause EXE to lose symbolic constraints when the input to one of these routines contains symbolic values. We thus replaced the optimized kernel routines with slower, but instrumented, versions to ensure we track constraints across all kernel calls.

Third, the UML kernel maps itself into a fixed virtual address range to simplify porting from the real Linux kernel. The large number of temporary variables introduced by `exe-cc` resulted in code and data segments so large that they caused this mapping to fail. We elided this problem by only linking the necessary kernel modules into the UML kernel, and by reworking the `exe-cc` transforms to reduce the number of generated temporaries.

Fourth, the front-end transformations we use for instrumentation do not always handle the GNU C in which Linux is written; they failed to compile eight files. We compile these problem files with `gcc` and treat the functions inside them as uninstrumented library code. EXE is designed to halt execution with an error whenever symbolic data is passed to uninstrumented code to flag such losses of precision. Fortunately, in the mount code we check, no symbolic data is ever passed to these uninstrumented

File System	*Kernel Panic*	*Read/Write Arbitrary Memory*	**Total**
ext2	3	1	4
JFS	1	0	1
Total	4	1	5

Figure 4. Summary of unique vulnerabilities found.

routines.

Finally, to help debugging we manually simplified a hash function (`_hashfn`), used to hash the block and device number during buffer cache lookups, whose extensive uses of shifts generated a lot of constraints.

5 Results

We applied our technique to three Linux file systems, ext2, ext3 and JFS. For the rest of this section we treat ext2 and ext3 as one file system because the code (and its bugs) implementing the mount operation in ext3 is almost identical to that in ext2. We found four vulnerabilities in ext2 that were replicated in ext3, and one vulnerability in JFS. The instrumented EXE code typically finds the bugs in less than a few minutes and in no case in more than hour on a modern desktop.

For technical reasons, we run the file system code on top of an older kernel, Linux 2.4.19, and many vulnerabilities we found have already been fixed in the latest Linux 2.6 kernels. However, we were previously unaware of any of these, and two of the vulnerabilities we found are still present in the latest Linux kernel.

Figure 4 summarizes the errors we found. Four of these errors cause the kernel to panic, while one error makes the kernel read and write arbitrary memory. We discuss these errors in detail in the following sections. The inlined comments in the code snippets in this section are all ours and are provided for clarity.

5.1 Ext2 and Ext3

Of the four errors we found in ext2 and ext3, one causes the kernel to read and write arbitrary memory, while the other three make the kernel crash.

The most dangerous exploit we found allows a carefully crafted malicious disk to bypass an upper-bound check of an offset through an arithmetic overflow, and then use this unconstrained offset to read and write from arbitrary regions in memory.

We step through this bug below in some detail below in order to give a feel for the type of bugs EXE can find.

The source code that leads to this exploit is shown in Figure 5. The function `ext2_free_blocks` frees `count` disk blocks belonging to inode `inode`, starting at block number `block`. The parameter `block` is read from disk and therefore is treated as a symbolic variable by EXE. On lines 8-9, ext2 checks that `block` is within the valid range. However, if `block` is very large, `block+count` can overflow and pass the check `block+count` > `le32_to_cpu(es->s_blocks_count)` (Note that `le32_to_cpu` is a macro that expands to the identity function on little endian machines). Later, ext2 computes `block_group` (lines 16-17) using `block`, and then calls `load_block_bitmap` (line 25). The `load_block_bitmap` procedure uses `block_group` as an array index (lines 53-55), which allows an attacker to read from arbitrary memory addresses.

Moreover, if the condition on lines 50 to 55 is true, the code assigns the value of `block_group` to `slot` on line 57, after which it returns this value back to the caller of the `load_block_bitmap` function. Then, this value is used to make an assignment to variable `bh` (line 31), after which `bh->data` can be used to write at arbitrary locations in memory (line 37). This bug was fixed in later versions of the 2.4 kernel series.

While this bug was immediately found by EXE, it is hard to find by manual inspection because it involves a combination of two events (an arithmetic overflow followed by a buffer overflow), which can easily be overlooked.

The bug highlights the benefit of EXE's memory model. Without EXE's bit-level precision and support for symbolic pointers and casting, it is extremely difficult to detect such exploits. In addition, EXE's multiple symbolic checks made it easy to diagnose the error. EXE flagged an arithmetic overflow on lines 9 and 10, and buffer overflows on lines 31, 37, 53 and 55, which we used to identify the root cause of the bug.

Finally, because EXE produced concrete values that trigger these overflows we could easily verify them by simply mounting the EXE-generated disk image in an uninstrumented Linux 2.4.19 kernel, which generated a kernel panic when `block_group` was used to index past the bounds of an array.

```
1 : /* fs/ext2/balloc.c */
2 : void ext2_free_blocks (struct inode * inode,
3 :                       unsigned long block,
4 :                       unsigned long count) {
5 :     /* ... */
6 :     // EXE: block is symbolic. block + count can overflow
7 :     //        and be smaller than s_blocks_count
8 :     if (block < le32_to_cpu(es−>s_first_data_block)
9 :         || (block + count) >
10:             le32_to_cpu(es−>s_blocks_count)) {
11:             ext2_error (...);
12:             goto error_return;
13:     }
14:     /* ... */
15:     // EXE: block_group becomes symbolic
16:     block_group =
17:             (block − le32_to_cpu(es−>s_first_data_block))
18:             / EXT2_BLOCKS_PER_GROUP(sb);
19:
20:     /* ... */
21:     // EXE: call load_block_bitmap with symbolic
22:     //        argument block_group. load_block_bitmap
23:     //        can return block_group, so bitmap_nr
24:     //        becomes symbolic
25:     bitmap_nr = load_block_bitmap (sb, block_group);
26:     if (bitmap_nr < 0)
27:             goto error_return;
28:
29:     // ERROR! read out of bounds.
30:     //           sb−>u.ext2_sb.s_block_bitmap has size 8
31:     bh = sb−>u.ext2_sb.s_block_bitmap[bitmap_nr];
32:     /* ... */
33:     for (i = 0; i < count; i++) {
34:             // ERROR! bh and bh−>b_data can point to
35:             //           anywhere. read and write to
36:             //           arbitrary kernel memory.
37:             if (!ext2_clear_bit (bit + i, bh−>b_data))
38:                     /* ... */
39:     }
40:     /* ... */
41: }
42:
43: static inline int
44: load_block_bitmap (struct super_block * sb,
45:                    unsigned int block_group) {
46:     /* ... */
47:     if (...) {/* ... */}
48:     // ERROR! read out of bounds.
49:     //           EXT2_MAX_GROUP_LOADED is 8
50:     else if (sb−>u.ext2_sb.s_groups_count <=
51:              EXT2_MAX_GROUP_LOADED &&
52:              sb−>u.ext2_sb.
53:                s_block_bitmap_number[block_group]
54:                == block_group &&
55:              sb−>u.ext2_sb.s_block_bitmap[block_group]) {
56:             // EXE: slot becomes symbolic
57:             slot = block_group;
58:     } else {
59:             slot = __load_block_bitmap (sb, block_group);
60:     }
61:     /* ... */
62:     return slot; // EXE: potential symbolic return
63: }
```

Figure 5. Ext2 buffer overflow.

We found another bug in ext2 that allows a malicious disk to panic the kernel at mount time. (This bug was live in our version of the kernel but fixed in 2.6.) Linux file systems such as ext2 may respond in three different ways when faced with an error: they may try to continue to execute normally, they may continue as read-only, or they may panic the kernel. The actual behavior can be specified either through global mount options (usually in a file called "`/etc/fstab`"), or by setting certain flags in the disk's super block. A correct implementation should always give priority to the global mount options, which are usually set by system administrators. The ext2 implementation incorrectly allows on-disk flags to override global mount options, which malicious disks can exploit to panic the kernel by setting the panic flag in the super block.

The other two exploits that we found in ext2 are caused by a division by zero bug and a modulo by zero bug, both taking advantage of file system code which uses values read from the disk as divisors, without first checking that they are not zero. One of the errors is fixed in Linux version 2.6.10, but the other still exists in the latest kernel version at the time of publication.

Figure 6 shows the unfixed bug. The `data` argument to `ext2_read_super` contains symbolic values read from our symbolic disk. The function first locates the symbolic super block `es` within `data` on line 9, then copies its fields to the in-memory super block `sb->u.ext2_sb` on line 14-15 and checks if the fields are valid. The field `s_inodes_per_group` specifies the number of inodes contained in one block group. Although the ext2 developers carefully check it against an upper bound on line 18, they fail to check it against zero. Subsequent use of this value as denominator on line 39 can cause a modulo by zero error.

The division by zero bug which was fixed in Linux 2.6.10 is very similar to the modulo bug presented here, except that it only spans a single function. The fact that the modulo by zero bug spans two different files, one where checks are performed (fs/etc/super.c) and one where the modulo operation is done (ft/ext2/inode.c), probably explains why it has not been fixed.

5.2 JFS

The bug we found in JFS is a NULL pointer dereference that exists in the latest 2.4 series kernels and in a slightly modified form in the latest

```
1 : // fs/ext2/super.c
2 : // EXE: parameter data contain symbolic values
3 : //          read from the symbolic disk
4 : struct super_block * ext2_read_super (
5 :   struct super_block * sb, void * data, int silent) {
6 :   /* . . . */
7 :   // EXE: cast data to es. Now es points to the
8 :   //          symbolic super block
9 :   es = (struct ext2_super_block *)
10:        (((char *)bh->b_data) + offset);
11:   // EXE: copy values from es to sb
12:   //          sb->u.ext2_sb.s_inodes_per_group
13:   //          becomes symbolic
14:   sb->u.ext2_sb.s_inodes_per_group =
15:        le32_to_cpu(es->s_inodes_per_group);
16:   /* . . . */
17:   // EXE: check uppper bound
18:   if(sb->u.ext2_sb.s_inodes_per_group >
19:       sb->s_blocksize * 8) {
20:     printk("EXT2-fs: #inodes per group too big: %lu\n",
21:         sb->u.ext2_sb.s_inodes_per_group);
22:     goto failed_mount;
23:   }
24:
25:   /* . . . */
26:   // EXE: read root inode. iget will call ext2_read_inode
27:   sb->s_root = d_alloc_root(iget(sb, EXT2_ROOT_INO));
28:   /* . . . */
29: // include/linux/ext2_fs.h
30: #define EXT2_INODES_PER_GROUP(s) \
31:      ((s)->u.ext2_sb.s_inodes_per_group)
32:
33: // fs/ext2/inode.c
34: void ext2_read_inode (struct inode * inode) {
35:   /* . . . */
36: //ERROR!: EXT2_INODES_PER_GROUP(inode->i_sb)
37: //            is symbolic and can be 0.
38:   offset = ((inode->i_ino − 1)
39:          % EXT2_INODES_PER_GROUP(inode->i_sb)) *
40:        EXT2_INODE_SIZE(inode->i_sb);
```

Figure 6. Ext2 modulo-by-zero bug.

```
1 : /* fs/jfs/jfs_umount.c */
2 : int jfs_umount(struct super_block *sb) {
3 :      struct jfs_sb_info *sbi = JFS_SBI(sb);
4 :      struct inode *ipaimap = sbi->ipaimap;
5 :      /* . . . */
6 :      sbi->ipimap = NULL;
7 :      /* . . . */
8 :      ipaimap = sbi->ipaimap;
9 :      // EXE: this will eventually call diFree on ipaimap
10:      diFreeSpecial(ipaimap);
11: }
12: /* fs/jfs/jfs_imap.c */
13: int diFree(struct inode *ip) {
14:      /* . . . */
15:      // EXE: jfs_umount sets ipimap = NULL
16:      //          so ipimap is also NULL
17:      struct inode *ipimap = JFS_SBI(ip->i_sb)->ipimap;
18:      // ERROR! expands to imap = ipimap->u.generic_ip
19:      //              where ipimap is NULL and will cause a
20:      //              NULL pointer dereference
21:      struct inomap *imap = JFS_IP(ipimap)->i_imap;
```

Figure 7. JFS NULL pointer dereference.

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 8. The hex dump above is a 64KB disk that will cause JFS to dereference a NULL pointer in the Linux 2.4.27 kernel. (The · · · are repeats of the previous row.) To reproduce the NULL pointer dereference in Linux 2.4.27, simply create an empty 64K file and set the 64th sector to the above. The same disk causes JFS to dereference a garbage pointer in the 2.6.10 kernel.

2.6 kernels. The mounting code discovers an error in the provided disk and tries to undo the partially completed mount by calling `jfs_umount`. The code is shown in Figure 7. This function frees the fileset inode allocation map `sbi->ipimap` and then sets the pointer to NULL on line 6. Later on line 10 it tries to close the aggregate inode allocation map by calling `diFreeSpecial`. After several nested function calls `jfs_umount` calls `diFree` which retrieves the value of `ipimap` (which is NULL) from the superblock on line 17 and tries to dereference it on line 21, causing a NULL pointer exception. The same bug also exists in the 2.6.10 kernel where a fixed offset is subtracted from the NULL pointer before it is dereferenced, causing a page fault at an invalid virtual address. Figure 8 shows the actual disk image generated by EXE from the error path constraints; when mounted it will can cause the NULL or bogus pointer dereference.

6 Related Work

There are many file system testing frameworks that use application interfaces to stress a "live" file system with an adversarial environment; two good ones are [27,34]. However, these focus on errors that occur during the runtime operation of a file system and do not focus on corruption or data integrity issues. Recently there has been some work on characterizing how a file system responds to disk errors [31], but this work starts with an initial, good disk and then tries to cause problems. We instead aim to crash the kernel from the get-go.

In the remainder of this section we compare EXE to other symbolic execution work, static input generation, model checking, test generation, and finally generic bug finding methods.

Symbolic Execution. In prior work we developed the EGT [9] system whose approach is similar in spirit to EXE, but in practice has many limitations: no pointers, arrays, bit-fields, casting, overflow, sign extension and none of the extended checks we describe in Section 3.2. Simultaneously with EGT, the DART project [19] developed a similar approach of generating test cases from symbolic inputs. DART only handles constraints on integers of the form $a_1x_1 + \cdots + a_nx_n + c \bowtie 0$ where $\bowtie \;\in \{<, >, \leq, \geq, =, \neq\}$, and does not handle symbolic pointers. This restriction prevents DART from reasoning about the bit masking and pointer operations inherent in file system code.

The CUTE project [33], which splintered from DART, handles pointer constraints by using an approximate pointer theory that causes it to miss errors of the form: `buf[i] = 0; buf[j] = 1; if (buf[i] == 0) ERROR`, where `i` and `j` are symbolic.

EGT, DART, and CUTE focused on unit-testing small, user-level programs, rather than large, complex kernel code. Additionally, none looked for the types of errors focused on in this paper.

CBMC is a bounded model checker for ANSI-C programs [11] designed to cross-check an ANSI C implementation of a circuit against its Verilog counterpart. Unlike EXE, which uses a mixture of concrete and symbolic execution, CBMC runs code entirely symbolically. It takes (and requires) an entire, strictly-conforming ANSI C program, which it translates into constraints that are passed to a SAT solver. CBMC provides full support for C arithmetic and control operations and reads and writes of symbolic memory. However, it has several limitations that keep it from handling systems code. First, it has a strongly-typed view of memory, which prevents it from checking code that accesses memory through pointers of different types. From experimenting using CBMC, this limit means it cannot check a program that calls `memcpy`, much less a program that casts pointers to integers and back. Second, because CBMC must translate the entire program to SAT, it can only check stand-alone programs that do not interact with the environment (e.g., by using systems calls or even calling code for which there is no source). Both of these limits prevent CBMC from being used to check any of the file system code in which we found bugs.

Larson and Austin [26] present a system that dynamically tracks primitive constraints associated with "tainted" data (e.g., data that comes from untrusted sources such as network packets) and warns when the data could be used in a potentially dangerous way. At potentially dangerous uses of inputs, such as array references or calls to the string library, they check whether the index could be out of bounds, or if the string could violate the library function's contract. Thus, as EXE, this system can detect an error even if it did not actually occur during the program's concrete execution. However, their system lacks almost all of the symbolic power that EXE provides. Unlike EXE, they cannot generate inputs to cause paths to be executed; they require the user to provide test cases, and only check the paths covered by these test cases.

Dynamic input generation techniques. Past automatic input generation techniques appear to focus primarily on generating an input that will reach a given path, typically motivated by the problem of answering programmer queries as to whether control can reach a statement or not [16,21]. EXE differs from this work by focusing on the problem of comprehensively generating tests on all paths controlled by input, which makes it much more effective in exploring the state space of the programs being tested.

Model Checking. Model checkers have been previously used to find errors in both the design and the implementation of software systems [6,12,18,23,28]. These approaches tend to require significant manual effort to build test harnesses. However, to some degree, the approaches are complementary: the tests our approach generates could be used to drive the model checked code, similar to the approach embraced by the Java PathFinder (JPF) project [25]. JPF combines model checking and symbolic execution to check applications that manipulate complex data structures written in Java. JPF differs from EXE first of all in its application domain, but also in that it does not have support for untyped memory (not needed because Java is a strongly typed language), and symbolic pointers.

Static input generation. There has been a long stream of research that attempts to use static techniques to solve constraints to generate inputs that will cause execution to reach a specific program point or path [4, 5, 7, 20]. A nice feature of static techniques is that they do not require running code. However, in both theory and practice they are much weaker than a dynamic technique such as EXE, which has access to much useful information impossible to get without running the program.

Static checking. Much recent work has focused on static bug finding [6, 8, 13, 17, 35]. The insides of these tools look dramatically different than EXE. The Saturn tool [36] is one exception, and expresses program properties as boolean constraints, and which models pointers and heap data down to the bit level. Roughly speaking, because dynamic checking runs code, it is limited to just executed paths, but can more effectively check deeper properties. Examples include program executions that loop on bad inputs, or byzantine errors that occur when a formatting command (such as for `printf`) is not properly obeyed. Many of the errors in this paper would be difficult to discover statically. However, we view static analysis as complementary to EXE testing — it is lightweight enough that there is no reason not to apply it and then use EXE.

7 Conclusion

Currently, it is easy to attack file systems by mounting data produced by a malicious source. This paper has shown how aggressive symbolic execution can be used to find such security holes in real systems code and other interesting errors. We used the EXE symbolic execution system to find nine such errors in three Linux file systems: one error in JFS and four in ext2, which were replicated via cut-and-paste to ext3.

We plan to extend these techniques to automatically "harden" code by generating filters that reject bad data. When EXE finds a path that leads to an error, if it has the complete, accurate set of path constraints then it knows exactly what input data will cause the error to occur. It can easily translate these constraints to if-statements that reject any concrete input that satisfy these constraints. For the `mount` code, these checks can be inserted around disk read calls and reject bad blocks with a "cannot mount" error. The same basic approach can be used to generate filters that reject dangerous network packets.

Conversely, we also plan to use EXE to automatically generate attacks. A natural domain is operating systems code: (1) run EXE-instrumented system calls that mark all input from the user (system call parameters, data copied manually from user-space) as unconstrained, (2) when a crash is found, solve for concrete input values, and (3) synthesize a small program that will call into the kernel with these values. (EXE can, of course, be used to generate filters for these attacks as above.)

8 Acknowledgments

We thank Vijay Ganesh and David Dill for their outstanding work on our constraint solvers CVCL and STP, Derek Chan for pointing us toward Linspire, Lea Kissner for her editing help, and our shepherd David Wagner for his patience and extensive comments. This research was supported by NSF CAREER award CNS-0238570-001, a Department of Homeland Security (DHS) grant, and a Junglee Corporation Stanford Graduate Fellowship.

References

[1] Distributing software with internet-enabled disk images. `http://developer.apple.com/documentation/DeveloperTools/Conceptual/Soft%wareDistribution/Concepts/sd_disk_images.html`.

[2] Linspire - the world's easiest desktop linux. `http://www.linspire.com`.

[3] The user-mode linux kernel home page. `http://user-mode-linux.sourceforge.net`.

[4] T. Ball. A theory of predicate-complete test coverage and generation. In *FMCO'2004: Symp. on Formal Methods for Components and Objects*. SpringerPress, 2004.

[5] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 203–213. ACM Press, 2001.

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

[7] R. S. Boyer, B. Elspas, and K. N. Levitt. Select – a formal system for testing and debugging programs by symbolic execution. *ACM SIGPLAN Notices*, 10(6):234–45, June 1975.

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

[9] C. Cadar and D. Engler. Execution generated test cases: How to make systems code crash itself. In *Proceedings of the 12th International SPIN Workshop on Model Checking of Software*, August 2005. A longer version of this paper appeared as Technical Report CSTR-2005-04, Computer Systems Laboratory, Stanford University.

[10] C. Cadar, P. Twohey, V. Ganesh, and D. Engler. EXE: A system for automatically generating inputs of death using symbolic execution. Technical Report CSTR 2006-01, Stanford, 2006.

[11] E. Clarke and D. Kroening. Hardware verification using ANSI-C programs as a reference. In *Proceedings of ASP-DAC 2003*, pages 308–311. IEEE Computer Society Press, January 2003.

[12] 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.

[13] 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.

[14] N. Eén and N. Sörensson. An extensible sat-solver. In E. Giunchiglia and A. Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[15] The ext2/ext3 File system. `http://e2fsprogs.sf.net`.

[16] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol.*, 5(1):63–86, 1996.

[17] J. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, June 2002.

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

[19] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, Chicago, IL USA, June 2005. ACM Press.

[20] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. In *ISSTA '98: Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, pages 53–62. ACM Press, 1998.

[21] N. Gupta, A. P. Mathur, and M. L. Soffa. Automated test data generation using an iterative relaxation method. In *SIGSOFT '98/FSE-6: Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 231–244. ACM Press, 1998.

[22] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, Dec. 1992.

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

[24] The IBM Journaling File System for Linux. `http://www-124.ibm.com/jfs`.

[25] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of the Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2003.

[26] E. Larson and T. Austin. High coverage detection of input-related security faults. In *Proceedings of the 12th USENIX Security Symposium (Security 2003)*, August 2003.

[27] Linux Test Project. `http://ltp.sf.net`.

[28] M. Musuvathi and D. R. Engler. Model checking large network protocol implementations. In *Proceedings of the First Symposium on Networked Systems Design and Implementation*, 2004.

[29] G. C. Necula, S. McPeak, S. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of Conference on Compilier Construction*, March 2002.

[30] G. Nelson and D. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–57, 1979.

[31] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Iron file systems. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 206–220, New York, NY, USA, 2005. ACM Press.

[32] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, pages 159–169, 2004.

[33] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *5th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'05)*, Sept. 2005.

[34] stress. `http://weather.ou.edu/~apw/projects/stress`.

[35] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *The 2000 Network and Distributed Systems Security Conference. San Diego, CA*, Feb. 2000.

[36] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 351–363, New York, NY, USA, 2005. ACM Press.

[37] 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.