# EXE: A System for Automatically Generating Inputs of Death Using Symbolic Execution

Cristian Cadar, Paul Twohey, Vijay Ganesh, Dawson Engler
Stanford University
Computer Systems Laboratory
{cristic, twohey, vganesh, engler} @cs.stanford.edu

**Abstract**

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.

This paper presents EXE, a system designed to automatically find bugs in such code using symbolic execution. At a high level, rather than running the code on manually-constructed concrete input, EXE instead runs it on symbolic input that is initially allowed to be "anything." As input (and derived) values are observed through conditional statements and other checks, symbolic constraints are incrementally added to those values. EXE then generates concrete test cases by solving these symbolic constraints for concrete values with bit-level precision.

EXE has several novel features. First, it implements a complete, precise symbolic pointer theory that correctly handles both pointer arithmetic expressions and reads and writes to memory locations referenced by pointers with symbolic values. Second, it handles all of the C language with bit-level precision. Third, EXE greatly amplifies the effect of running a single code path since it uses a powerful constraint solver to 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.

EXE has been successfully applied to applications ranging from running the Linux kernel symbolically in order to find numerous security holes in the ext2, ext3, and JFS file systems [26] to detecting invalid memory reads and writes in a DHCPD server implementation to finding buffer overflow attacks in the BSD and Linux packet filter implementations.

## 1 Introduction

Systems code defines a tricky execution state space built from deeply nested conditionals, labyrinthine function call chains, massive amounts of code, and frequent use of casting and pointer operations. This code often must process input received directly from potential attackers (system call parameters, network packets, even memory from USB sticks), forcing it to exhaustively vet the input. These checks can be subtle, especially in the presence of arithmetic overflows or buffer overruns, both which programmers reason about poorly. All these features conspire to make manual inspection even more erratic than usual. Random testing faces its own difficulties. 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. Similarly, errors (such as mishandling overflow) often only occur for a narrow input range, making finding them with random test cases unlikely. Bugs that depend on the precise layout of objects in memory and complicated interactions with the heap are beyond the detection prowess of most static analysis tools.

This paper describes EXE ("EXecution generated Executions"), a practical symbolic execution system engineered for deep automatic checking of systems code. The central insight behind EXE is that code can be used to *automatically* generate its own (potentially highly complex) 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 checksumming). 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 Purify[18] 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.

Fourth, 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. Conceptually EXE is the offspring of both static (symbolic) and dynamic (concrete execution-based) bug-finding techniques; the aim is to achieve the precision of testing with the systematic coverage of static methods.

Finally, EXE works well on real code. It automatically found buffer overruns in the very mature and audited Berkeley Packet filter code, the Linux networking code, and a DHCPD server. EXE can scale up to large, real (and sometimes overly complex) systems code. In a separate application paper [26], we describe how to use EXE to symbolically check large chunks of the Linux kernel at user level, and have found numerous security holes in three widely-used Linux file systems — ext2, ext3 and JFS — where data mounted as a file system would crash the kernel or allow a buffer overflow attack.

We have described aspects of EXE in two previous papers. The application paper mentioned above [26], takes EXE as a given and focuses primarily on a new security attack and the use of EXE to find errors. In contrast, the main contribution of this paper is EXE itself, which it describes in much more detail, as well as focusing on a broader set of applications. The second paper, an unrefereed invited workshop paper [7], presented an initial, primitive version of the system that did not support: reads or writes of symbolic pointer expressions, symbolic arrays, bit-fields, casting, sign-extension, arithmetic overflow, and the universal checks we do. Further it was only applied to a few small benchmarks.

The main contribution of this paper is the EXE system. EXE is explicitly designed to work on complex system software and combines:

- Bit level accurate symbolic modeling of C code, including integer casting and sign extension, bitfield accesses, and arbitrary pointer casting (§ 3).
- Design of a sound, accurate pointer theory that handles symbolic pointers supporting arbitrary updates, and symbolic data blocks including data blocks of symbolic size (§ 4).
- A discussion of EXE's potential ability to verify certain properties on an execution path, and a set of built-in *universal* checks designed to catch common errors such as null pointer dereferences, buffer overflows and division or modulo by zero.
- A scalable implementation that uses mixed concrete and symbolic execution to generate test cases for real systems code.

Our experiments demonstrate EXE's applicability to real, complex systems code, such as the BSD and Linux packet filter implementations, or a DHCPD server implementation (§ 6).

The core of EXE is described in the next three sections: Section 2 gives an overview, Section 3 describes how we model memory with bit-level accuracy, and Section 4 presents EXE's support for symbolic pointers in detail. The remainder of the paper discusses practical issues and limitations (§ 5), results (§ 6), related work (§ 7), and then concludes.

## 2 Overview of EXE and its Implications

This section gives an overview of EXE. At a high level, using EXE to automatically generate test cases has five steps:

1. The user marks memory that EXE should treat as symbolic by manually inserting calls to either of the two functions:
   ```
   void make_symbolic(T *obj);
   void make_symbolic_bytes(void *bytes, unsigned nbytes);
   ```

   The first marks the entire object `obj` as symbolic, the second marks the byte range starting at `bytes` to `bytes + nbytes` as symbolic. In both cases, the contents of the symbolic memory are initially treated as entirely unconstrained (i.e., its values are "anything"). As the program executes using these values, constraints are added and then (finally) solved. There can be multiple calls to these functions; EXE emits concrete values in the order they are called.

2. The user compiles their program using the EXE compiler (`exe-cc`), which uses the CIL front-end [23] to instrument the program for symbolic execution. The EXE compiler has three main jobs. First, it inserts checks around every assignment, expression, and branch in the tested program to check

if its operands are concrete or symbolic. If all operands are concrete, the operation is executed concretely (i.e., identically to an operation in an uninstrumented program). If any operand is symbolic, the operation is not performed, instead it is passed to the EXE runtime system to be added as a constraint.

Second, it inserts code to fork program execution when a symbolic value could lead to one of several possible actions. Consider the if-statement `if(x*x + y*y == w*w)`. If `x`, `y`, and `w` 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 (using the UNIX `fork()` system call) 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.

3. The rewritten source code is then compiled with a normal compiler (`gcc`), linked with the EXE runtime system to produce an executable, and run.

4. As the instrumented code runs, it forks at each decision point. When the execution of a program path terminates, EXE solves the path's constraints for concrete values. A path terminates when (1) it calls `exit()`, (2) crashes, (3) an assertion fails, or (4) EXE detects an error. Constraint solutions are literally the concrete bit values for a symbolic input that will cause the given path to execute. When generated in response to an error, they provide the actual bit values of a concrete attacks that can be launched against the tested system. When not directly causing an error, the output induce from the system can be checked for correctness (e.g., by comparing to a reference implementation, or inverting their result).

5. The concrete inputs are fed back into the tested program as test cases. At this stage the program can be compiled with a normal compiler and the uninstrumented program run without any use of the EXE system at all. Thus, no false positives are possible: any error caught by the test cases is a true error, not a false positive due to EXE mistakes.

Since the analysis literally runs the program, all information is known. The main limit on the technique's power is the cost and generality of the constraint solver. We use the CVCL decision procedure solver [4], which supports (among other things) arrays and bit-vectors and most C arithmetic operations. While not the fastest solver, CVCL is powerful and has been successfully used in applications ranging from hardware verification to program analysis to mathematical theorem proving.

A natural question is EXE's cost. For our experiments, we needed less than an hour to generate tests that trigger the errors we discuss. 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 elide any discussion of the straightforward mechanics of EXE's front-end translation: it accepts the entire C language (including whatever GNU extensions are handled by CIL), which it lowers to a form that is easier for EXE's runtime system to handle. The front-end's main limitation is that it does not instrument "inline assembly" directives, but passes them through uninstrumented.

Below, we give more detail about one of the more interesting implications of symbolic execution: its ability to perform universal checks.

## 2.1 The power of symbolic execution: universal checks

The central difference between concrete and symbolic execution is that concrete execution operates on a single possible set of concrete values, whereas symbolic execution operates on all values that the current constraints allow (in practice, modulo the power of the constraint solver).

```
sym_expr div_transformation(sym_expr x, sym_expr y)
  if(query(y != 0) == satisfiable)
      if(fork())
          add_constraint(y != 0);
          return symbolic_expression(x / y);
      else
          add_constraint(y == 0);
          terminate_with_error("Found div by 0!\n");
```

Figure 2: Pseudo-code of how symbolic division rewritten for universal checking.

```
#include <assert.h>
#include <netinet/in.h>
void main(void) {
    int x;
    make_symbolic(x);
    assert(htonl(ntohl(x)) == x);
}
```

Figure 3: Trivial universal check of form $f^{-1}(f(x)) = x$ that verifies `ntohl` and `htonl` correctly invert each other for all 32-bit inputs.

**Universal checks.** EXE uses this ability to provide several "universal" symbolic checks. A symbolic check is much more powerful than a concrete check, since the latter only checks a single specific value, whereas symbolic checks check all possible values for a given program path. These checks are "universal" in the sense that if the check passes, there is no possible value that the symbolic input could take that would cause the check to ever fail on this program path. I.e., if EXE has the full set of symbolic constraints and CVCL can solve them, then the path has been *verified* as safe under all possible input values. To the best of our knowledge, no other approach can make guarantees this powerful. As discussed later, the main cases where EXE misses constraints are where (1) uninstrumented code has been called (§ 5) or (2) EXE has had to concretize a constraint to make progress (the two cases where this occurs are discussed in § 3.2 and § 4.4)

EXE performs the following three universal checks: (1) that an integer divisor or modulus is never zero, (2) that a dereferenced 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 action (division, modulus, pointer dereference) and calls the constraint solver to determine if the condition could occur. If so, it 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 and continues execution. Figure 2 gives pseudo-code showing mechanically how EXE rewrites `x/y`. The null-pointer check is similar, except that given a symbolic pointer `p` in a dereference `*p`, it queries the constraint $p \neq 0$. Memory overflow errors are more complicated and are described in detail in Section 4. Figure 1 gives an example of these universal checks.

```
#include <assert.h>
int main(void) {
  unsigned i, t, a[4] = { 1, 0, 5, 2 };
  make_symbolic(&i);

  // ERROR: EXE catches potential overflow i >= 4.
  t = a[i];
  // At this point i < 4.

  // ERROR: EXE catches div by 0 when i = 1.
  t = t / a[i];
  // At this point i != 1 && i < 4.

  // Demonstrate simple gross casting (EXE handles arbitrary
  // casting but for exposition we only show simple casting).
  t = (unsigned)&a[0];
  t += i*4;
  // the value of t is equal to &a[i] at this point.

  // ERROR: EXE catches buffer overflow when i = 2.
  return a[*(unsigned *)t];      // Same as: return a[a[i]];
}
```

Figure 1: A contrived example to illustrate the errors EXE's built in universal checks catch. When run it will emit four test cases: three with errors, one without. Because EXE performs accurate memory tracking, so (1) all errors are guaranteed to be found (despite the gross cast) and (2) no false positives will arise.

**Generalized universal checks.** EXE turns programmer `asserts` on a symbolic expression into universal checks on their values. Rather than having an `assert` that detects if the condition was violated on a specific input, when EXE hits an `assert` it will systematically search the set of constraints to try to reach the false path (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.

We can use this feature of conditionals to verify that a function $f^{-1}$ inverts another function $f$. To accomplish this, we simply ask EXE to search over all possible input values for any case where $f^{-1}(f(x)) \neq x$. For example, networking code uses the functions `ntohl` and `htonl` to byte-swap 32-bit values between "host" and "network" order. As Figure 3 shows, checking that a given implementation does this correctly for all inputs is trivial. Note that if the system terminates, then `ntohl` is proved to invert `htonl` for all 32-bit inputs (as is `htonl` in the opposite direction). This leads to the startling results that if either `ntohl` or `htonl` is correct, then passing the assertion equals full verification of total correctness! When applicable, such a verification method is much more practical than the traditional approach of theorem proving plus correctness specifications.

In a similar fashion, we can ask EXE to find places where two routines $f$ and $f'$ intended to implement the same function fail to do so, by making their inputs symbolic and asserting $f(x) \neq f'(x)$. This happens commonly with networking code: many different implementations of servers exist, as do implementations of core networking library functions. EXE can be used to cross-check these against each other to ruthlessly search for inputs that lead to incompatible outputs. If it cannot find any (and it terminates), then EXE has verified that no incompatibilities exist.

## 3 Bit-level Accurate Modeling of Memory

Two memory stores co-exist during a run of an EXE program: (1) the *concrete store*, which is just the memory of the underlying machine (e.g., 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, perhaps less obvious, (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. Since EXE is accurate down to the bit-level, any solution to its symbolic store is guaranteed to be a valid, legal concrete store.

Concrete bytes holding concrete values have no corresponding storage in the symbolic store. When the user marks a concrete set of bytes as symbolic, the system creates a corresponding, identically-sized range of bytes in the symbolic store. EXE records this correspondence in a hash table $H$ that maps byte addresses to their corresponding symbolic bytes (if any). Initially, $H$ contains only the memory regions explicitly marked as symbolic. As the program executes, $H$ may grow as more bytes become symbolic, either by assigning a symbolic expression to a concrete variable or by indexing a data block by a symbolic index (this later case is discussed in Section 4).

### 3.1 CVCL support

CVCL is based on Nelson and Oppen's cooperating decision procedures framework [24], which allows decision procedures for quantifier-free theories to be merged into a decision procedure for the union of the theories. A theory is a deductively closed set of sentences starting from a set of axioms, where sentences are formulas in the language of the theory with no free variables.

EXE uses the theory of fixed-length bit-vectors and the theory of arrays. The theory of bit-vectors is a theory over finite non-empty strings over $\{0, 1\}$, whose length is known *a priori*. It includes all the standard

C arithmetic operations (except for division and modulo, which we discuss in the next subsection), bitwise boolean operations, relational operations (less than, less than or equal, etc.), and multiplexers, which provide an "if-then-else" construct that is converted into a logical formula.

The bit-vector theory is distinguished from other CVCL theories by its ability to efficiently support bit concatenation and bit extraction. The concatenation of two symbolic bit-vector variables $a_{sym}$ and $b_{sym}$ is denoted by $a_{sym}$ @ $b_{sym}$. For example, the concatenation of 0110 with 1100 is: 0110 @ 1100 = 01101100. The extraction of a sub-bit-vector of a bit-vector $a_{sym}$ starting at bit $i$ and ending at bit $j \leq i$ is denoted by $a_{sym}[i : j]$. For example, extracting the lower four bits of 01110010: $01110010[3 : 0] = 0010$.

We represent each byte of symbolic memory using an 8-bit bit-vector. We represent a data object as an array of such 8-bit elements. The main advantage of using CVCL bit-vectors is that they have complete fidelity with concrete memory: for each symbolic bit in the program, there is a corresponding bit in the constraint solver. This property lets us express constraints that refer to the same bits in memory in different ways. As a result, EXE views memory as completely untyped. Each read of memory generates constraints based on the static type of the read (e.g., `int`, `unsigned`, etc.) but these types do not persist. I.e., 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. Having multiple views of the same piece of memory is crucial to handling systems code, which frequently converts between pointers of different types and even between pointers and integers and back. Such chicanery does not confuse EXE. A further benefit of this bit-correspondence is that EXE has no approximations: the constraints on its symbolic store are completely accurate, which let it systematically explore all corner cases rather than resorting to giving up or random choice (as in [9, 15]).

## 3.2 Extensions to CVCL

CVCL's original theory of bit-vectors was missing direct support for several common operations encountered in systems code, such as signed comparators, sign-extension, and shifts by a non-constant bit-vector. We wrote wrappers to express these in terms of the base operations provided by the bit-vector theory, often exploiting the fact that we only cared about limited, finite quantities (of size 32-bits or smaller). For example, in the general case it is complicated to give constraints that correctly describe the shift of an unsigned value by a non-constant bit-vector. However, in the special case of 32-bit entities, it is trivial, since there are only 33 different cases (shift by 32 or greater, shift by 31, shift by 30, ..., shift by 0) and each case can be explicitly checked for and handled, by simply using a CVCL multiplexer.

Other than the occasional speed problems inherent in a constraint solver that can tackle NP-hard problems, the main limitation of CVCL's bit-vector theory is the omission of division and modulo by symbolic values. When these cases arise we constrain the operand to be a power of two (thus potentially discarding certain execution paths) and replace the division or modulo operation by a shift or bitwise mask, respectively. We plan to implement the full division and modulo circuits in a future version of the system.

## 3.3 Symbolic operations on untyped memory

For each operation in the program (assignment, branch, etc.), EXE queries the hash table `H` to check if all the operands are concrete. An operand is considered to be concrete if and only if all its constituent bytes are concrete. If all the operands are concrete, the operation is performed concretely, as in the original program. Otherwise, the operation is performed symbolically.

Each expression $e$ used in a symbolic operation is constructed in the following way. For each read of a storage location $l$ in $e$, EXE checks if $l$ is concrete. If so, the read of $l$ is replaced by its concrete value. Otherwise, for each byte in $ll$, EXE queries the hash table $H$ to get the symbolic expressions

$b_{sym_1}, b_{sym_2}, ..., b_{sym_n}$ associated with each of the $n$ bytes of $l$. These symbolic expressions are then concatenated, to obtain $b_{sym_1}@b_{sym_2}@...@b_{sym_n}$, the symbolic expression associated with $l$.

After this replacement, the expression is traversed in a bottom-up fashion. For an expression of the form $e = e1\ OP\ e2$, where $OP$ denotes an operator, and $e1$ and $e2$ have symbolic expressions $e1_{sym}$ and $e2_{sym}$ (which have been already computed), in order to construct $e_{sym}$, EXE may have to first cast $e1_{sym}$ and $e2_{sym}$ to the type of the operator (which is inferred by the front-end). For example, if we add an 8-bit bit-vector expression (obtained for example from a `char` variable in C) to a 32-bit bit-vector expression (obtained for example from an `int` variable in C), the 8-bit bit-vector expression needs to be converted to a 32-bit bit-vector expression.

As part of its strategy to drive execution into corner-cases, EXE attempts to force overflow in each symbolic arithmetic operation. EXE does so by performing each symbolic arithmetic operation twice. First, at the precision specified by the program being tested. Second, at a precision that cannot produce an overflow. If the results of the two operations are different, an arithmetic overflow is possible, and EXE forks execution, adding on one path that the overflow is possible, and on the other that it is not. Note that while an arithmetic overflow is not necessary an error, it is definitely an event worth exploring. In fact, this mechanism helped us find bugs in the Linux packet filter implementation, where an arithmetic overflow would cause a programmer-written bounds check to falsely succeed, leading to a simple-to-exploit buffer overflow of kernel memory.

Finally, note that an “assignment” in the symbolic realm is not destructive. Unlike a concrete assignment, which overwrites the previous value, a symbolic assignment $v = e$ to a variable $v$ creates a new, fresh symbolic variable and then sets an equality constraint. The variable $v$ may have been involved in previous constraints and we must distinguish the new value of $v$ from its use in any already generated constraints. For example, assume we have two assignments: (1) `v = y` and then (2) `y = 3`. The first assignment will generate the constraint that $v = y$. The second will generate the constraint $y = 3$. At this point, the constraints imply $v = 3$, which is false: the new value for `y` after its assignment `y = 3` has nothing to do with any prior constraints involving `y` and should have no impact on them.

## 3.4 Modeling scalar casts

Modeling casts can be easily done in CVCL. We divide scalar casts into two categories: narrowing casts, namely casts from a larger to a shorter primitive type, and widening casts, namely casts from a shorter to a larger primitive type. Narrowing casts are translated into simple bit-extraction operations in the symbolic store. More precisely, if a variable $a$ of $a_{size}$ bits is cast to a variable $b$ of $b_{size} < a_{size}$ bits, then EXE adds the constraint:

$$b_{sym} = a_{sym}[b_{size} : 0]$$

In order to handle widening casts, we modified CVCL to support symbolic sign extension. We define the $sx(a_{sym}, n)$ operator to denote the sign extension of a symbolic bit-vector $a_{sym}$ to a larger number of bits $n > a_{size}$. If the variable is unsigned, sign-extension means padding with 0. For a signed variable, sign extension means padding with 0 if the sign bit is 0, or padding with 1 if the sign bit is 1. For example $sx(1010, 8) = 00001010$ if $1010_{sign} =$ unsigned, and $sx(1010, 8) = 11111010$ if $1010_{sign} =$ signed.

Casting can often lead to bugs, which are hard to diagnose since cast operations are often intentionally “unsafe.” Narrowing casts may introduce bugs because they can lead to information loss. Widening from a signed to an unsigned value may introduce bugs via sign-extension – if the signed variable is negative, the result of the cast is a very large positive number, which is sometimes not intended.

EXE can (optionally) drive execution into these corner cases caused by potentially “dangerous” casts. For narrowing casts, EXE checks if information can be lost by truncation, and if so, forks execution into two different execution paths: on one execution path it asserts that information is lost through truncation, and on the other path that no information is lost. More precisely, if a variable $a$ of $a_{size}$ bits is narrowed to a variable $b$ of $b_{size} < a_{size}$ bits, then EXE checks the constraint:

$$sx(a_{sym}[b_{size} - 1 : 0], a_{size}) = a_{sym}$$

For widening casts from a signed to an unsigned value, EXE checks if the sign bit of the signed variable 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.

## 3.5 Example: mapping memory operations to CVCL

```
struct my_header {
  unsigned len;
  char msg[8];
} *header;
unsigned char buf[12];
make_symbolic_bytes(buf, sizeof buf);
if (buf[0] == 123)
  exit(0);
header = (struct my_header*) buf;
if (header->len > 8) {
  printf("Invalid length\n");
  exit(0);
}
```

Figure 4: A contrived code fragment that refers to the same memory under differently typed views.

This subsection steps through the memory constraints generated from the contrived code example in Figure 4, which receives as input a symbolic buffer `buf` of size 12 bytes. The program implements a simple filter which rejects invalid buffers. It first checks if the 0th byte has value 123, and if so, it rejects the buffer. Otherwise it casts the buffer to the structure `my_header` and it rejects the buffer if the field `len` is bigger than 8.

The main difficulty of symbolically running this program is that the buffer `buf` is accessed in two different ways: first as an array of `char`, then as a structure of type `my_header`. However, as discussed above, modeling memory as bits makes it easy to add constraints derived from different typed-views of the same memory location(s). At the level of CVCL, when the code calls `make_symbolic_bytes`, EXE allocates an array of length 12 having as elements bit-vectors of size 8 (for readability, the following following examples use a simplified CVCL syntax; EXE uses CVCL's C API):

```
sym_buf:ARRAY[0..11] OF BITVECTOR(8);
```

When the program refers to `buf[0]`, our system first finds the symbolic CVCL bits associated with it (in this case `sym_buf[0]`) and emits the symbolic query:

```
QUERY(sym_buf[0] != 123)
```

A symbolic query can return two different results: `Valid` if CVCL can prove the proposition for all values of the symbolical inputs, and `Invalid` otherwise. [1] Because, at this point in the code there are no constraints on `buf`, then `sym_buf[0]` could be equal to 123, so the query is not always true, so the query result is `Invalid`. At this point, EXE forks execution, adding the symbolic constraint: `ASSERT(sym_buf[0] == 123)` on the true path and: `ASSERT(sym_buf[0] != 123)` on the false.

Since the true path then exits, we ask CVCL to generate concrete values for the current set of constraints. Currently CVCL happens to generate the value 123 for `buf[0]`, and 0 for the rest of the nine symbolic bytes.

When the false path continues, we query if "`header→len > 8`." As before, EXE finds finds the symbolic bits associated with `header→len`. The system first determines that `header→len` starts at the symbolic byte `sym_buf[0]` and is 4 bytes long. Since the comparison is on a multi-byte quantity we use the bit concatenation to construct the symbolic bit-vector corresponding to the entire four byte expression

[1]Note that if a query returns `Invalid` some of the values of the inputs may in fact satisfy the query, just not all. To show it is not true for any value, the query would have to be negated.

header→len. EXE emits the following query (note that the order in which we concatenate the four bytes reflects the little endian architecture of our machine):

```
QUERY(sym_buf[3] @ sym_buf[2] @ sym_buf[1] @ sym_buf[0] > 8)
```

This query is also Invalid. The only constraint on sym_buf is that its first byte is not 123, thus there are values that do not satisfy the query. Thus, our system forks again the execution, adding on one branch the constraint that:

```
ASSERT(sym_buf[3] @ sym_buf[2] @ sym_buf[1] @ sym_buf[0] > 8)
```

and on the other path the constraint that:

```
ASSERT(NOT (sym_buf[3] @ sym_buf[2] @ sym_buf[1] @ sym_buf[0] > 8))
```

Since both paths then reach exit points, EXE asks CVCL for concrete values. For the first path, CVCL generates the concrete values buf[0] = buf[1] = buf[2] = buf[3] = 255 (thus header→len = 4294967295), with the other bytes being 0. On the second, CVCL generates the concrete value buf[0] = 7 (thus header→len = 7) with the other bytes being 0.

## 4 Symbolic Pointers

This section describes how: (1) EXE implements symbolic memory overflow checks (from § 2), (2) maps memory blocks to the symbolic store, (3) implements reads and writes of symbolic pointer expressions, and (4) some corner-case issues.

### 4.1 Symbolic overflow checking

EXE checks for memory overflow whenever a pointer expression is dereferenced. It does so by tracking memory in a way similar to a bounds checking compiler (such as CRED [25]). For each data object, EXE records its size. For each pointer, it records the objects that the pointer should be contained within. Using this information, EXE can emit a warning if the program tries to dereference a pointer that is outside its intended object (while allowing a pointer to wander outside its object as long as it is not dereferenced).

EXE obtains the size of each data object by: (1) wrapping standard allocation calls to get the size of the returned object; (2) using the object file symbol table to get the size of global data; and (3) instrumenting function calls to get the size of local variables (erased when the function returns). The text below assumes a function size, which given the base address of an object (i.e., its first byte) returns the object's size.

The EXE front-end simplifies all pointer arithmetic expressions to be of the form p = q + k, where p and q are pointers and k is an integer. EXE tracks the base object that p points to by recording that it points inside the base object of q. We assume a function base which given the address of a pointer returns the object the pointer points to. Figure 5 shows how EXE updates base and size on a short example.

This information is sufficient to do bounds checking. Every pointer expression is composed of two parts: (1) a pointer p and (2) an offset k (which may be 0). Figure 6 gives the completely standard check EXE does when both p and k are concrete (as in [25, 18]). Figure 7 shows how EXE does checking when either p or k (or both) are symbolic.

### 4.2 Mapping memory blocks to CVCL arrays

Mirroring its promotion of scalars, EXE promotes a concrete n-byte memory block b to the symbolic domain by (1) allocating an n-byte CVCL array $b_{sym}$; and (2) inserting a mapping into the concrete-to-symbolic hash table $H$ that maps the start address of b to $b_{sym}$. Thus, given a pointer p EXE can always determine the corresponding CVCL array to work with: $b_{sym}$ = lookup(H, base(&p)).

CVCL does not provide pointers. In order to emulate symbolic pointer expressions we recast them as an array reference at some offset. We do so using the same machinery described in the previous section for

```
Original code | Instrumented code
              |
              | // base case: direct
              | // assignment of base object
p = malloc(4); | p = malloc(4);
              | size(p) = 4;
              | base(&p) = p;
              |
              | // assignment of possibly
              | // derived object.
q = p;        | q = p;
              | base(&q) = base(&p);
              |
p = q + 10;   | p = q + 10;
              | base(&p) = base(&q);
              |
q++;          | q = q + 1;
              | // unless optimized away.
              | base(&q) = base(&q);
```

Figure 5: Simple example of pointer rewrites.

```
// does p+k point outside base object?
char *b = base(&p), *ub = b + size(b);
if((p + k) < b || (p + k) >= ub)
    error("memory overflow");
```

Figure 6: Memory overflow check for concrete pointer expression.

```
// does p+k point outside base object?
char *b = base(&p), *ub = b + size(b);
if(query((p + k) < b || (p + k) >= ub) == exists)
    if(fork())
        // Assert error, solve constraints, terminate path.
        add_constraint((p + k) < b || (p + k) >= ub);
        terminate_with_error("array overflow!");
    else
        // Assert error does not happen, continue execution.
        add_constraint((p + k) >= b && (p + k) < ub);
```

Figure 7: Pseudo-code: memory overflow check for symbolic pointer expression.

tracking the sizes of memory blocks. Given a pointer expression of the form `*(p + i)`, we (1) determine the symbolic array $b_{sym}$ the pointer `p` refers to (as above, do: `lookup(H, base(&p))`); (2) compute the (possibly symbolic) offset of `p` from the base of the object it points to (i.e., $o = p - base(\&p)$); and (3) add this to the original (possibly symbolic) offset $i$. We can then evaluate the symbolic expression $b_{sym}[i + o]$.

## 4.3 Reads and writes of symbolic pointer expressions

The problem with symbolic pointer expressions is that, unlike scalars, they 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 boolean expression as simple as `(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] = 1` could update any value in `a`.

EXE handles both reads and writes of pointer expressions where the pointer or the offset expression are symbolic (or both). (Of course, reads and writes of concrete pointers and offsets are performed concretely, in the usual way.) As an example, consider the contrived programs in Figures 8 and 9. Both programs execute correctly in EXE and produce no assertion violations. The difficulty in Figure 8 is translating the constraint $a[i] + a[j] = 28$ into the symbolic domain. Since $i$ and $j$ are symbolic, $a[i]$ and $a[j]$ could potentially refer to any element of the array `a`. When EXE follows the true branch of the `if` statement, the constraint `(a[i] + a[j] == 28)` is added to the symbolic memory store. At this point, the symbolic store can refer to only two possible concrete stores: one in which $i = 0$ and $j = 2$, and one in which $i = 2$ and $j = 0$. The program checks this fact using `assert`, which EXE proves true.

In Figure 9, the assignment $a[i] = 1$ creates a symbolic store that can generate four different concrete memory stores, depending on what element of $a$ was overwritten. However, when EXE asserts that $(a[j] + a[k] == 14)$ on the `then` branch of the `if` statement, 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.

```
#include <assert.h>
int main() {
  unsigned char a[4] = {11, 13, 17, 19};
  unsigned char i, j;
  make_symbolic(&i);
  make_symbolic(&j);
  if(i >= 4 || j >= 4) // force in-bounds
    exit(0);
  if (a[i] + a[j] == 28)
    assert( (i == 0) && (j == 2) ||
            (i == 2) && (j == 0) );
}
```

Figure 8: A simple example using pointer reads

```
#include <assert.h>
int main() {
  unsigned char a[4] = {11, 13, 17, 19};
  unsigned char i, j, k;
  make_symbolic(&i); make_symbolic(&j); make_symbolic(&k);
  if(i >= 4 || j >= 4 || k >= 4) // force in-bounds
    exit(0);
  a[i] = 1;
  if ( (a[j] + a[k] == 14) )
    assert((i != 1));
}
```

Figure 9: A simple example using pointer writes

In the rest of this discussion, without loss of generality we assume that, using the techniques of the previous two subsections, all pointers expressions have been (1) checked for overflow and (2) translated to their corresponding array access `a[k]`. Further, for ease of exposition (and again without loss of generality), we only consider reads of the form `x = a[k]` and writes of the form `a[k] = x`. Other cases are similar.

There are three main cases for a read `a[k]`:

1 $a$ **symbolic,** $k$ **concrete**: the easy case. Add the constraint that the $k$-th element of the symbolic memory associated with `a` (i.e. $a_{sym}$) is equal to the symbolic memory associated with `x` (i.e. $x_{sym}$): $x_{sym} = a_{sym}[k]$. (Recall that because we are assigning a symbolic expression to `x`, `x` becomes symbolic even if it was not already.)

2 $a$ **symbolic,** $k$ **symbolic**. In this case, we do not know the element $k$ references. Thus we add the constraint that if $k = 0$ then $x = a[0]$, if $k = 1$ then $x = a[1]$, etc. More precisely:

$$\bigvee_{0 \leq i < \text{size}(a)} k = i \wedge x_{sym} = a_{sym}[i]$$

3 $a$ **concrete,** $k$ **symbolic**. Conceptually this case can be implemented identically to the previous one, where the disjunction involves the concrete values (i.e., constants) of `a`:

$$\bigvee_{0 \leq i < \text{size}(a)} k = i \wedge x_{sym} = a[i]$$

However, in the current implementation of CVCL its easier (and faster) to promote `a` to a symbolic array whose contents are fixed constants (equal to those currently in `a`). Handling this case then reduces to the previous case of $a$ symbolic, $k$ symbolic.

Writes are more complicated than reads. As described in Section 3.3, assignment to a symbolic variable creates a new fresh copy. We must do the same thing when assigning to a symbolic array $a_{sym}$, with the complication that we may not know which array element is mutated. Assignment requires three steps: (1) allocate a new, fresh symbolic array $a'_{sym}$; (2) bind $a'_{sym}$ to the address of `a`; and (3) constrain all the elements in $a'_{sym}$ to be equal to the old elements in $a_{sym}$ with the exception of the one updated element. CVCL provides support to make such incremental updates efficient by using versioning (similar to functional languages).

There are three main cases for a write of the form `a[k] = x` that mirror those for reads (for space we elide formal notation):

1 $a$ **symbolic** $k$ **concrete**. Create $a'_{sym}$, a fresh copy of $a_{sym}$ whose elements are equal to $a_{sym}$ except for the one element $a'_{sym}[k]$ which has the constraint $a'_{sym}[k] = x$ (if $x$ is concrete) or $a'_{sym}[k] = x_{sym}$ (if

$x$ is symbolic).

2 $a$ **symbolic,** $k$ **symbolic**. Same as above, with a slightly more complicated disjunction expression.

3 $a$ **concrete,** $k$ **symbolic**. As with reads, this reduces to the previous case of $a$ symbolic and $k$ symbolic, by first promoting the concrete array a to the symbolic domain.

### 4.4 Concretizing pointers with unknown base object

There are several situations in which EXE cannot determine the base object of the pointer being dereferenced. First, the base object may have been allocated in uninstrumented code. Second, the pointer may have been initialized to the dereference of another symbolic pointer (such usage usually occurs when a symbolic collection such as a hash table is used). When the base object is not known, EXE cannot determine the symbolic expression associated with the symbolic dereference of the pointer, and thus needs to concretize the value of the symbolic pointer expression, and then perform the dereference concretely.

EXE concretizes an arbitrary symbolic pointer expression $p_{sym}$ by copying the current execution environment and asking the constraint solver for a solution $s$ for the values in $p_{sym}$. This solution is then copied back to the original environment where it is added as a constraint $p_{sym} = s$.

Concretizing the symbolic pointer expression means that EXE may discard certain execution paths. However this mechanism allows EXE to continue execution even when the base object of the pointer is unknown, which is obviously preferable to completely stopping execution.

We are currently exploring two options for extending EXE to support pointers initialized to the dereference of another symbolic pointer: one is to modify the constraint solver to keep track of the base object and the size of each symbolic data block; and another is to use a single symbolic memory array (of size $2^{32}$, but sparse) to represent memory.

### 4.5 Data blocks of symbolic size

EXE also supports data blocks of symbolic size. This lets it model memory whose size is determined by a symbolic input. Since CVCL provides efficient support for symbolic arrays of essentially arbitrary size, this construct translates directly into symbolic constraints. This feature's main challenge is in constructing a mapping back to the concrete store: we do not know how big the data block is, so it is not clear how to allocate memory for it in the concrete store. Fortunately, the way we track data objects lets us use a simple trick to solve this problem. EXE simply allocates one byte of concrete memory, which gives it a unique base address for the "object," as well as a unique name with which to refer to the symbolic array. All array updates and reads then proceed as normal.

## 5 Practical EXE Issues

This section discusses several practical aspects of EXE that need special consideration. We summarize the main limitations to verification; show how EXE interacts with instrumented code; present the search strategies employed by EXE; show how we validate the test suite produced by EXE; and discuss various configuration options.

**Limitations to verification.** EXE is a system designed to find bugs in critical systems code. While EXE can be potentially used to verify certain properties of the system being tested (through the universal checks discussed in Section 2.1), there are three main practical limitations that may impede verification: (1) the execution may not terminate (solving the constraints generated by a C application is NP-complete); (2) EXE may concretize certain symbolic inputs on some execution paths (for example, when it encounters a modulo operation by a non power of two); and (3) the execution may call uninstrumented code, for which EXE cannot generate constraints.

**Uninstrumented functions.** A program being tested often interacts with uninstrumented code that has not been compiled by `exe-cc`, such as system calls or calls to standard libraries. If these calls are made with concrete parameters, EXE simply calls the uninstrumented code, as in the original program. If these calls are made with symbolic parameters EXE has a problem, as it cannot generate constraints from code it has not instrumented. One possible strategy would be to force the generation of concrete values for the symbolic parameters before making such a call. However, our current methodology is different and designed to ensure more complete constraint generation. During execution, EXE logs all the calls to uninstrumented functions that are made with symbolic parameters. Thus our approach is to run EXE, inspect this log, and subsequently instrument all the functions recorded in the log. This iteration is repeated until there are no uninstrumented calls in the log. By default, EXE uses instrumented functions of the standard libraries.

**Search strategies.** When EXE forks execution it has a choice of which branch to follow. By default EXE uses depth-first search (DFS). DFS keeps the number of current children small (linear in the depth of the process chain) as opposed to breadth-first search. Unfortunately, DFS works poorly in some cases. For example, if EXE encounters a loop with a symbolic variable as a bound, DFS can get "stuck" since it will attempt to execute the loop as many times as possible, thus taking a very long time until it exits the loop. In order to overcome this problem, we use search heuristics to drive the execution along "interesting" execution paths.

After a `fork` call, each forked EXE process calls into a search server with a description of its current state (e.g., the file and line number it is at, its backtrace) and blocks until the server replies. The search server examines all blocked processes and picks the "best" one in terms of some heuristic. The server is structured so that new heuristics are simple to plug in. Our current heuristic uses a mixture of best-first and DFS search. It picks the process blocked at 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.

**Finding bugs in EXE.** When running a checked program, EXE records the basic blocks it visits. When reaches the end of a path and generates a test case, we rerun an instrumented version of the program on the test case and verify that it indeed causes the path to execute. This check has proved to be very useful during the development of EXE. It exposed bugs both in our system (due to errors in the constraint collection phase), and in the CVCL constraint solver (which sometimes did not generate correct concrete solutions).

**Configuration options.** EXE has several configuration options, which can be easily specified in a configuration file. The configuration options supported by EXE include: (1) the possibility to disable certain built-in checks (and so trade-off additional checks for number of visited paths); (2) enable various logging mechanisms; (3) enable a central server that caches satisfiability queries; (4) specify the search strategy; or (5) specify a timeout for each query sent to the constraint solver.

## 6 Results

This section describes our preliminary results. We applied EXE to (1) two packet filter implementations (where it found buffer overflows); (2) a DHCPD server implementation (where it found invalid memory reads and writes); (3) device drivers code; and (4) file system implementations (where it found kernel panic and buffer overflow attacks). All experiments were performed on a modern laptop and ran in less than two hours, with many completing in only a few minutes.

### 6.1 Packet fi lters

Many operating systems use some variant of packet filters to allow user-level programs that are typically privileged to specify the packets they desire to receive. The most prevalent packet filters are variants on the Berkeley Packet Filter (BPF) system. BPF filters are written in a pseudo-assembly language, downloaded into the kernel by applications, checked by the BPF system, and then (conceptually) applied to each incoming packet.

Filters can easily be modeled by EXE even though packet filters contain intricate internal structure and interpretation behavior which are difficult for standard testing to cover comprehensively. Each filter is represented as an array of instructions, each an instance of `bpf_isn`. The field `code` represents the opcode, while the other fields represent control-flow and other data related to the instruction. These arrays are lowered to flat ranges of bytes that can be easily marked by EXE as symbolic.

```
struct bpf_insn {
        u_short        code;
        u_char         jt;
        u_char         jf;
        bpf_u_int32    k;
};
```

We cross-checked two packet filter implementations: (1) the current BPF implementation from FreeBSD and (2) the Linux packet filter implementation which was initially derived from the FreeBSD implementation and subsequently heavily modified. EXE uncovered two buffer overflows in the FreeBSD implementation and four errors in the Linux implementation.

The procedure to check packet filters in EXE is straightforward: (1) we first mark both the buffer holding the filter and that holding the data (the packet) as symbolic, and then (2) call the packet filter checking routine on the buffer holding the filter. If this routine returns $true$, we (3) apply the valid symbolic filter on the symbolic data, thus generating all possible data packets that the valid symbolic filter accepts. The automatic generation of all packet filters of `flen` bytes and the testing of these filters on all packets of `dlen` bytes is performed by the following C code:

```
  // create a symbolic array that encodes a packet
  make_symbolic_bytes(filter, flen); // filter represented with up to flen bytes

  // create a symbolic array that contains the bytes of the packet
  make_symbolic_bytes(data, dlen);

  if(bpf_validate(filter, flen) != 0) {
     printf("accepted filter\n");
     if(bpf_filter(filter, data, dlen) != 0)
        printf("accepted data!\n");
     else printf("rejected data!\n");
  }
```

EXE discovered two conditions for a buffer overflow in the FreeBSD implementation. The errors occured if the opcode of a BPF instruction was either `code == BPF_STX` or `code == BPF_LDX | BPF_MEM`. For these opcodes, the `bpf_validate` function did not check that the given instruction's offset field was within a legal range. The code below shows how BPF checks that all instructions with opcodes `BPF_ST` and `BPF_LD | BPF_MEM` have valid offsets. Such a check is absent for instructions with opcodes `BPF_STX` and `BPF_LDX | BPF_MEM`:

```
 // Check that memory operations only uses valid addresses.
 // => Check forgets LDX,STX!
 if( (BPF_CLASS(p->code) == BPF_ST || (BPF_CLASS(p->code) == BPF_LD &&
      (p->code & 0xe0) == BPF_MEM)) && p->k >= BPF_MEMWORDS )
   return 0;
```

The elided check causes the BPF interpreter to allow such instructions to write or read arbitrary offsets off of a fixed sized buffer mem, thus crashing the kernel or allowing a trivial exploit (pc points to the current instruction):

```
case BPF_LDX|BPF_MEM:
    X = mem[pc−>k]; continue;
...
case BPF_STX:
    mem[pc−>k] = X; continue;
```

Linux had a trickier example. EXE found three filters that cause a kernel crash because of an arithmetic overflow in a bounds check. Because legal filters must end in a return, the smallest filters to cause these errors were two instructions long, using various forms of loads:

```
// other filters that cause this error...
//    => BPF_LD|BPF_B|BPF_IND
//    => BPF_LD|BPF_H|BPF_IND
s[0].code = BPF_LD|BPF_B|BPF_ABS;
s[0].k      = 0x7fffffffUL;
s[1].code = BPF_RET;
s[1].k      = 0xfffffff0UL;
```

As with BPF, the k field causes the problem. Here these instructions eventually lead to a call to the function:

```
static inline void * skb_header_pointer(struct sk_buff *skb, int offset, int len, void *buffer) {
        int hlen = skb_headlen(skb);
        if (offset + len <= hlen)
                return skb−>data + offset;
        ...
```

This routine is passed s[0].k as the offset parameter, and 4 or 2 as the len parameter. The routine extracts the size of the current message header hlen and checks that offset + len $\leq$ hlen. However, given a very large offset, the signed integer holding the sum of offset and len will overflow to a small value, passing the check, and then causing that very large value to be added to the message data pointer. This allows attackers to easily crash the machine. The error can be easily eliminated by fixing the bounds check, and the newest kernel versions have a correct bounds check.

This error is a good example of the power of EXE, because it would be very hard to hit with random testing. Furthermore, this error occurs in highly visible, widely-used code, demonstrating that such tricky cases can empirically withstand repeated manual inspection.

## 6.2 Complete server: udhcp-0.9.8

We also checked a complete networking server, udhcp-0.9.8, which implements a user-space DHCPD server, and whose code is very clean and well tested. To run this server through our system, we marked its input packet as symbolic, and restructured one loop that interacted badly with our current system. We then modified its network read call to return a packet less than or equal to 548 bytes. This generated 246 test cases. We reran these concrete test cases and checked the executions using valgrind, a tool that dynamically checks for some types of memory corruption and storage leaks. Valgrind flagged five different errors: one-byte read overflows at line 308 in dhcpd.c and three similar errors at lines 79, 90 and 111 in options.c. These errors were not found when we tested the code using random testing. Due to time constraints, we have currently not inspected these errors.

### 6.3 Device Drivers

The commonly accepted wisdom is that device driver code has a much higher defect rate than core operating system code[28, 8]. This is unfortunate from a reliability perspective given that device drivers make up the vast majority of the code in Linux. To make matters worse, individual kernel developers can each only test a handful of drivers because they are limited by their available hardware.

As a first step toward improving driver testing we have created a user-space test harness where any device driver's initialization and cleanup functions can be tested. We interpose on routines (like `inb` and `pci_read_config_word`) that device drivers use to read from hardware and leverage the power of symbolic execution to make arbitrary values for reads. Because we interpose across the visible hardware interface, we are able to simulate any hardware which the driver could be reading.

We also use symbolic execution as a primitive form of model checking to model potential choices in the kernel environment such as memory allocation (e.g. `kmalloc`) failing or not, `down_trylock` acquiring a lock or not, and `capable` (which checks if a user has a privilege to perform an action) failing or not.

Together, these symbolic inputs constitute the control information for device driver initialization and cleanup routines and thus drives the code through all interesting execution paths.

Specifically, we test that a driver initialization routine either (1) completes successfully and allocates all necessary kernel resources and that these resources are freed by the driver unload function or (2) that initialization fails and no kernel resource are leaked. Currently we track memory allocations, lock state, and interrupt request line reservations.

We have run 26 device driver initialization and teardown routines through our testing framework, including 22 watchdog timer drivers, two cryptography modules (arc4 and cast5), one joystick driver (stinger), and one ethernet driver (de620). We are able to successfully load each kernel module without having the device present and generate failure cases for every device except the cryptography modules, which cannot fail to load. We had to add only ten lines of code to all the device drivers to get them to run in our system. For all generated tests we apply the resource usage checks discussed previously, in addition to running the code under valgrind. To date we have not found any errors, but our results are preliminary.

### 6.4 File system implementations: ext2, ext3 and JFS

In a previous application paper [26], we have used EXE to check the mount operation of several file system implementations, ext2, ext3 and JFS. Our approach was to mark the disk as symbolic input, and then run the mount code of each file system inside a version of Linux, which EXE instrumented as well. We found bugs in all file systems, where malicious data could either cause a kernel panic or form the basis of a buffer overflow attack. The most dangerous exploit that we found was in the ext2/ext3 code and allows a potential attacker to read from or write to arbitrary regions in memory. Similar to the bug we found in BPF, this bug involves a combination of two events: an arithmetic overflow, exposed with the help of the arithmetic overflow mechanism described in Section 3.3, and a buffer overflow, detected by the checker described in Section 4. This illustrates that EXE is very good at detecting complex errors that require multiple successive events in order to be triggered.

## 7 Related Work

Simultaneously with our initial work [7], the DART project [15] developed a similar approach of generating test cases from symbolic inputs. DART runs the unit being tested on a concrete random input, while symbolically gathering constraints at all decision points which depend on the input values. Then, DART negates one of these symbolic constraints to generate the next test case. DART only handles constraints on integers and devolves to random testing when pointer constraints are used, where it has the usual problems of missed paths. It also uses randomization to initialize pointers: when an pointer is symbolic, DART randomly sets its value to either `NULL` or the address of a newly allocated memory block.

The CUTE project [27], which splintered from DART, extends the DART approach by tracking symbolic pointer constraints of the form: `p = NULL`, $p \neq$ `NULL`, `p = q`, or $p \neq q$. In addition, CUTE correctly tracks constraints formed by reading or writing symbolic memory at constant offsets (such as a field dereference `p→field`), but cannot deal with symbolic offsets. For example, the paper on CUTE shows that on the code snippet `a[i] = 0; a[j] = 1; if (a[i] == 0) ERROR`, CUTE fails to generate the case in which `i == j`, which would have driven the code down both paths.

In contrast to both DART and CUTE, EXE has completely accurate constraints on memory, which lets it check code (potentially) much more thoroughly.

CBMC is a bounded model checker for ANSI-C programs [9] designed to cross-check an ANSI C re-implementation of a circuit against its Verilog implementation. 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 prevent 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 code in which we found bugs. Finally, CBMC unrolls all loops and recursive calls, which means that it may miss bugs that EXE can find and also that it may execute some symbolic loops more times than the current set of constraints allows it to.

Larson and Todd [21] 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. They associate tainted integers with an upper and lower bound and tainted strings with their maximum length and whether the string is 0-terminated. At potentially dangerous uses of inputs, such as array references or calls to the string library, they check whether the integer 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.

We next compare EXE to past static checking and test generation work, and then to dynamic bug finding methods.

**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 [5, 16, 1, 2]. A nice features 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 [13, 3, 11, 6, 29]. The insides of these tools look dramatically different than EXE. The Saturn tool [30] 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.

**Software Model Checking.** Model checkers have been previously used to find errors in both the design and the implementation of software systems [19, 10, 3, 14, 22]. 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 [20]. JPF combines model checking and symbolic execution to check applications that manipulate complex data structures written in Java. JPF differs from EXE in that it does not have support for untyped memory (not needed because Java is a strongly typed language), and does not support symbolic pointers.

**Dynamic techniques for test and input generation.** 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 [12, 17]. 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.

## 8 Conclusion

This paper has presented key aspects of how EXE achieves accurate, exact, bit-level symbolic execution: (1) the complete pointer theory (including mutation and symbolic sizes), (2) an operational view of EXE's semantics and (3) its support for universal checks. We have applied EXE to a variety of widely-used, mature systems, where it was powerful enough to uncover subtle and surprising bugs.

## References

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

[2] 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.

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

[4] C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating valid ity checker. In R. Alur and D. A. Peled, editors, *CAV*, Lecture Notes in Computer Science. Springer, 2004.

[5] 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.

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

[7] 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.

[8] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, 2001.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

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

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

[20] 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.

[21] 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).*

[22] 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.

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

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

[25] 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.

[26] C. Sar, P. Twohey, J. Yang, C. Cadar, and D. Engler. Discovering malicious disks with symbolic execution. In *IEEE Symposium on Security and Privacy*, May 2006.

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

[28] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. In *OSDI*, pages 1–16, Dec. 2004.

[29] 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.

[30] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *Proceedings of the 32nd Annual Symposium on Principles of Programming Languages (POPL 2005), January 2005.*