EXE: Automatically Generating Inputs of Death

CRISTIAN CADAR, VIJAY GANESH, PETER M. PAWLOWSKI,
DAVID L. DILL and DAWSON R. ENGLER
Stanford University

This paper presents EXE, an effective bug-finding tool that automatically generates inputs that crash real code. Instead of running code on manually or randomly constructed input, EXE runs it on symbolic input initially allowed to be "anything." As checked code runs, EXE tracks the constraints on each symbolic (i.e., input-derived) memory location. If a statement uses a symbolic value, EXE does not run it, but instead adds it as an input-constraint; all other statements run as usual. If code conditionally checks a symbolic expression, EXE forks execution, constraining the expression to be true on the true branch and false on the other. Because EXE reasons about all possible values on a path, it has much more power than a traditional runtime tool: (1) it can force execution down any feasible program path and (2) at dangerous operations (e.g., a pointer dereference), it detects if the current path constraints allow *any* value that causes a bug. When a path terminates or hits a bug, EXE automatically generates a test case by solving the current path constraints to find concrete values using its own co-designed constraint solver, STP. Because EXE's constraints have no approximations, feeding this concrete input to an uninstrumented version of the checked code will cause it to follow the same path and hit the same bug (assuming deterministic code).

EXE works well on real code, finding bugs along with inputs that trigger them in: the BSD and Linux packet filter implementations, the `udhcpd` DHCP server, the `pcre` regular expression library, and three Linux file systems.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*; *Symbolic execution*

General Terms: Reliability, Languages

Additional Key Words and Phrases: Bug finding, test case generation, constraint solving, symbolic execution, dynamic analysis, attack generation.

1. INTRODUCTION

Attacker-exposed code is often a tangled mess of deeply-nested conditionals, labyrinthine call chains, huge amounts of code, and frequent, abusive use of casting and pointer operations. For safety, this code must exhaustively vet input received directly from potential attackers (such as system call parameters, network packets,

A shorter version of this paper appeared in the Proceedings of the 13th ACM Conference on Computer and Communications Security, October 30 - November 3, 2006 [Cadar et al. 2006].

Authors' addresses: Computer Systems Laboratory, Stanford University, 323 Serra Mall, Gates Building, Stanford, CA 94305, U.S.A.

even data from USB sticks). However, attempting to guard against all possible attacks adds significant code complexity and requires awareness of subtle issues such as arithmetic and buffer overflow conditions, which the historical record unequivocally shows programmers reason about poorly.

Currently, programmers check for such errors using a combination of code review, manual and random testing, dynamic tools, and static analysis. While helpful, these techniques have significant weaknesses. The code features described above make manual inspection even more challenging than usual. The number of possibilities makes manual testing far from exhaustive, and even less so when compounded by programmer's limited ability to reason about all these possibilities. While random "fuzz" testing [Miller et al. 1990] often finds interesting corner case errors, even a single equality conditional can derail it: satisfying a 32-bit equality in a branch condition requires correctly guessing one value out of four billion possibilities. Correctly getting a sequence of such conditions is hopeless. Dynamic tools require test cases to drive them, and thus have the same coverage problems as both random and manual testing. Finally, while static analysis can benefit from full path coverage, the fact that it inspects rather than executes code means that it reasons poorly about bugs that depend on accurate value information (the exact value of an index or size of an object), pointers, and heap layout, among many others.

This paper describes EXE ("EXecution generated Executions"), an unusual but effective bug-finding tool built to deeply check real code. The main insight behind EXE is that code can *automatically* generate its own (potentially highly complex) test cases. Instead of running code on manually or randomly constructed input, EXE runs it on *symbolic* input that is initially allowed to be "anything." As checked code runs, if it tries to operate on symbolic (i.e., input-derived) expressions, EXE replaces the operation with its corresponding input-constraint; it runs all other operations as usual. When code conditionally checks a symbolic expression, EXE forks execution, constraining the expression to be true on the true branch and false on the other. When a path terminates or hits a bug, EXE automatically generates a test case that will run this path by solving the path's constraints for concrete values using its co-designed constraint solver, STP ("Simple Theorem Prover").

EXE amplifies the effect of running a single code path since the use of STP 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. For instance, a dynamic memory checker such as Purify [Hastings and Joyce 1992] only catches an out-of-bounds array access if it is provided with a test case where the index (or pointer) has a specific concrete value that is out-of-bounds. In contrast, when EXE explores the same path (which it does automatically), it identifies this bug if there is any possible input value on the given path that can cause an out-of-bounds access to the array. Similarly, for an arithmetic expression that uses symbolic data, EXE can solve the associated constraints for values that cause an overflow or a division/modulo by zero. Moreover, for an `assert` statement, EXE can reason about all possible input values on the given path that may cause the assert to fail. If the `assert` does not fail, then either (1) no input on this path can cause it to fail, (2) EXE does not have the full set of constraints (see Section 5), or (3) there is a bug in EXE.

The ability to automatically generate concrete inputs to execute program paths has several nice features. First, EXE can test any code path it wishes (and given enough time, exhaust all of them), thereby getting coverage out of practical reach from random or manual testing. Second, EXE generates actual attacks. This ability lets it show that external forces can exploit a bug, improving on static analysis, which often cannot distinguish minor errors from showstoppers. Third, the presence of a concrete input allows the user to easily discard error reports due to bugs in EXE or STP: the user can confirm the error report by simply re-running an uninstrumented copy of the checked code on the concrete input to verify that it actually hits the bug (note that both EXE and STP are sound with respect to the test cases generated, and therefore false positives can only arise due to implementation bugs in EXE and/or STP).

Careful co-design of EXE and STP has resulted in a system with several novel features. First, STP primitives let EXE build constraints for all C expressions with perfect accuracy, down to a single bit. (The main exception is floating-point, which STP does not handle.) EXE handles pointers, unions, bit-fields, casts, and aggressive bit-operations such as shifting, masking, and byte swapping. Because EXE is dynamic (it executes the checked code), it has access to runtime information typically not available to static analyses. All non-symbolic (i.e., *concrete*) operations happen exactly as they would in uninstrumented code and produce exactly the same results: when these results appear in future constraints they are correct, not approximations. In our context, this accuracy means that if (1) EXE has the full set of constraints for a given path, (2) STP can produce a concrete solution from these constraints, and (3) the path is deterministic, then rerunning the checked system on these concrete values will force the program to follow the same exact path to the error or termination that generated this set of constraints.

In addition, STP provides the speed needed to make perfect accuracy useful. Aggressive customization makes STP often 100 times faster than more traditional constraint solvers while handling a broader class of examples. Crucially, STP efficiently reasons about constraints that refer to memory using symbolic pointer expressions, which presents more challenges than one may expect. For example, given a concrete pointer `a` and a symbolic variable `i` with the constraint $0 \leq i \leq n$, the conditional expression `if (a[i] == 10)` is essentially equivalent to a big disjunction: `if (a[0] == 10 || ... || a[n] == 10)`. Similarly, an assignment `a[i] = 42` represents a potential assignment to any element in the array between `0` and `n`.

The result of these features is that EXE finds bugs in real code, and automatically generates concrete inputs to trigger them. It generates evil packet filters that exploit buffer overruns in the very mature and audited Berkeley Packet Filter (BPF) code as well as its Linux equivalent (§ 6.1). It generates packets that cause invalid memory reads in the `udhcpd` DHCP server (§ 6.2), and bad regular expressions that compromise the `pcre` library (§ 6.3), previously audited for security holes. In prior work, it generated raw disk images that, when mounted by a Linux kernel, would crash it or cause a buffer overflow [Yang et al. 2006].

Both EXE and STP are contributions of this paper, which is organized as follows. We first give an overview of the entire system (§ 2), then describe STP and its key optimizations (§ 3), and do the same for EXE (§ 4). Finally, we summarize the main limitations(§ 5), present experimental results (§ 6), discuss related work (§ 7),

```
1 : #include <assert.h>
2 : int main(void) {
3 :   unsigned i, t, a[4] = { 1, 3, 5, 2 };
4 :   make_symbolic(&i);
5 :   if(i >= 4)
6 :     exit(0);
7 :   // cast + symbolic offset + symbolic mutation
8 :   char *p = (char *)a + i * 4;
9 :   *p = *p − 1; // Just modifies one byte!
10:
11:   // ERROR: EXE catches potential overflow i=2
12:   t = a[*p];
13:   // At this point i != 2.
14:
15:   // ERROR: EXE catches div by 0 when i = 0.
16:   t = t / a[i];
17:   // At this point: i != 0 && i != 2.
18:
19:   // EXE determines that neither assert fires.
20:   if(t == 2)
21:     assert(i == 1);
22:   else
23:     assert(i == 3);
24: }
```

Fig. 1. A contrived, but complete C program (*simple.c*) that generates five test cases when run under EXE, two of which trigger errors (a memory overflow at line 12 and a division by zero at line 16). This example is used heavily throughout the paper. We assume it runs on a 32-bit little-endian machine.

and conclude (§ 8).

2. EXE OVERVIEW

This section gives an overview of EXE. We illustrate EXE's main features by walking the reader through the simple code example in Figure 1. When EXE checks this code, it explores each of the three possible paths, and finds two errors: an illegal memory write (line 12) and a division by zero (line 16). Figure 2 gives a partial transcript of a checking run.

To check their code with EXE, programmers only need to mark which memory locations should be treated as holding *symbolic data* whose values are initially entirely unconstrained. These memory locations are typically the input to the program. In the example, the call `make_symbolic(&i)` (line 4) marks the four bytes associated with the 32-bit variable `i` as symbolic. The programmers then compile their code using the EXE compiler, `exe-cc`, which instruments it using the CIL source-to-source translator [Necula et al. 2002]. This instrumented code is then compiled with a normal compiler (e.g., `gcc`), linked with the EXE runtime system to produce an executable (in Figure 2, `./a.out`), and run.

As the program runs, EXE executes each feasible path, tracking the constraints on the input which will take it down each path. When a program path terminates, EXE calls STP to solve the path's constraints for concrete values. A path terminates when (1) `exit()` is called, (2) the program crashes, (3) an assertion fails, or (4)

```
% exe−cc simple.c
% ./a.out
% ls exe−last
   test1.forks test2.out      test3.forks  test4.out
   test1.out   test2.ptr.err test3.out     test5.forks
   test2.forks test3.div.err test4.forks  test5.out
% cat exe−last/test3.div.err
   ERROR: simple.c:16 Division/modulo by zero!
% cat exe−last/test3.out
   # concrete byte values:
   0 # i[0]
   0 # i[1]
   0 # i[2]
   0 # i[3]
% cat exe−last/test3.forks
   # take these choices to follow path
   0 # false branch (line 5)
   0 # false (implicit: pointer overflow check on line 9)
   1 # true (implicit: div−by−0 check on line 16)
% cat exe−last/test2.out
   # concrete byte values:
   2 # i[0]
   0 # i[1]
   0 # i[2]
   0 # i[3]
```

Fig. 2. Transcript of compiling and running the C program shown in Figure 1.

EXE detects an error. Constraint solutions are literally the concrete bit values for an input that will cause the given path to execute. When generated in response to an error, they provide a concrete attack that can be launched against the tested system.

2.1 Instrumentation

The EXE compiler has the following main jobs. First, it inserts checks around every assignment, expression, and branch in the tested program to determine if its operands are concrete or symbolic. An operand is defined to be concrete if and only if all its constituent bits are concrete. If all operands are concrete, the operation is executed just as in the uninstrumented program. If any operand is symbolic, the operation is not performed; instead, the operation is passed to the EXE runtime system, which adds it as a constraint for the current path. For example, Figures 3 and 4 give the transformation rules for assignment involving primitive variables. Figure 3 covers the case when the right hand side is a single primitive variable, while Figure 4 covers the case when the right hand side is a binary operator involving exactly two primitive variables. All other assignments involving primitive variables can be recursively decomposed to these two main cases. Assignment involving array indexing and pointer dereferences is more complicated and is discussed in Section 3.

For the example's expression `p = (char *)a + i * 4` (line 8), EXE checks if the operands `a` and `i` on the right hand side of the assignment are concrete. If so, it executes the expression, assigning the result to `p`. However, since `i` is symbolic,

```
// rule for v = w
simple_assign_rule(T v, T w) {
   if sym(&w) == <null>
      v = w
      sym(&v) = <null>
   else
      sym(&v) = sym(&w)
}
```

Fig. 3. Transformation rule for assignment `v = w` for any primitive variables `v` and `w` of type `T`. `sym(&v)` returns the symbolic expression associated with variable `v`, or <`null`> if `v` is concrete.

```
// rule for v = x OP y
binary_assign_rule(OP, T v, T x, T y) {
   if sym(&x) == <null> && sym(&y) == <null>
      v = x OP y;
      sym(&v) = <null>
   else if sym(&x) == <null>
      sym(&v) = sym_exp(OP, x, sym(&y));
   else if sym(&y) == <null>
      sym(&v) = mk_sym_exp(OP, sym(&x), y);
   else
      sym(&v) = mk_sym_ex(OP, sym(&x), sym(&y));
}
```

Fig. 4. Transformation rule for `v = x OP y` where primitive variables `x` and `y` are of type `T`, and `OP` is a binary operator. `sym(&v)` returns the symbolic expression associated with variable `v`, or <`null`> if `v` is concrete. `mk_sym_exp` constructs a new symbolic expression.

EXE instead adds the constraint that p equals $(char*)a + i * 4$. Note that because i can have one of four values ($0 \leq i \leq 3$), p simultaneously refers to four different locations $a[0]$, $a[1]$, $a[2]$ and $a[3]$. In addition, EXE treats memory as untyped bytes (§ 3.2) and thus does not get confused by this (dubious) cast, nor the subsequent type-violating modification of a low-order byte at line 9.

Second, `exe-cc` inserts code to fork program execution when it reaches a symbolic branch point, so that it can explore each possibility. Figure 5 shows the transformation rule for conditional expressions. To understand how this transformation works, consider the if-statement at line 5, `if(i >= 4)`. Since `i` is symbolic, so is this expression. Thus, EXE forks execution (using the UNIX `fork()` system call) and on the true path asserts that $i \geq 4$ is true, and on the false path that it is not. Each time it adds a branch constraint (using `add_sym_constraint` in Figure 5), EXE queries STP to check that there exists at least one solution for the current path's constraints. If not, the path is impossible and EXE stops executing it. In our example, both branches are possible, so EXE explores both (though the true path exits immediately at line 6).

2.2 Default checks

In order to find generic program errors, `exe-cc` inserts code that calls to check if a symbolic expression could have any possible value that could cause either (1) a null or out-of-bounds memory reference or (2) a division or modulo by zero. If so,

```
// rule for if (e) s1 else s2
conditional_rule(e, s1, s2) {
   if is_sym(e) == <false>
      if e
         s1
      else
         s2
   else
      if fork() == child
         push_constraint(e = <true>)
         s1
      else
         push_constraint(e = <false>)
         s2
}

push_constraint(c) {
    add_sym_constraint(c)

    if path_feasible() == <false>
       kill()
}
```

Fig. 5. Transformation rule for conditional expressions `if(e) s1 else s2`. `is_sym(e)` returns <`true`> iff `e` is a symbolic expression, and returns <`false`> otherwise. `add_sym_constraint(c)` adds the constraint `c` to the set of constraints on the current path. `path_feasible()` determines whether the current set of constraints has a solution, and `kill()` terminates execution on the current path.

```
// rule for checking an error condition c
if fork() == child
   push_constraint(c)
   emit_error()
else
   push_constraint(!c)
   // continue normal execution
```

Fig. 6. Transformation template for checking error condition `c`. `emit_error()` emits an error, together with a corresponding test case, and exits. `push_constraint()` is defined in Figure 5.

EXE forks execution and (1) on the true path 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 (to find more bugs). Extending EXE to support other checks is easy. Figure 6 shows a template transformation that checks whether an error condition `c` occurs.

These checks are powerful because if EXE has the entire set of constraints on such expressions and STP can solve them, then EXE can detect if *any* input exists on that path that causes the error. Similarly, if the check passes, then no input exists that causes the error on that path — i.e., the path has been *verified* as safe under all possible input values.

These checks find two errors in our example. First, the symbolic index `*p` in the expression `a[*p]` (line 12) can cause an out-of-bounds error because `*p` can equal 4: the pointer `p` was computed using `i` with the constraint $0 \leq i < 4$ (line 8). Thus, $i = 2$ is legal, which means `p` can point to the low-order byte of `a[2]` (recall that each element of `a` has four bytes). The value of this byte is 4 after the subtraction at line 9. Since `a[4]` references an illegal location (one byte past the end of `a`), EXE forks execution and on one path asserts that $i = 2$ and emits an error (`test2.ptr.err`) and a test case (`test2.out`), and on the other asserts that $i \neq 2$ and continues.

Second, the symbolic expression `t / a[i]` (line 16) can generate a division by zero, which EXE detects by tracking and solving the constraints that (1) `i` can equal 0, 1, or 3 and (2) `a[0]` can equal 0 after the decrement at line 9. EXE again forks execution, emits an error (`test3.div.err`) and a test case (`test3.out`) and exits. The other path adds the constraint that $i \neq 0$ and continues.

Note, EXE automatically turns a programmer `assert(e)` on a symbolic expression e into a universal check of e simply because it tries to exhaust both paths of if-statements. If EXE determines that e can be false, it will go down the assertion's false path, hitting its error handling code. Further, if STP cannot find any such value, none exists on this path. In the example, EXE explores both branches at line 20, and proves that no input value exists that can cause either `assert` (line 21 and line 23) to fail. We leave working through this logic as an exercise for the more energetic reader. Even a cursory attempt should show the trickiness of manual reasoning about all-paths and all-values for even trivial code fragments. (We spent more time than we would like to admit puzzling over our own hand-crafted example and eventually gave up, resorting to using EXE to double-check our oft-wrong reasoning.)

Finally, EXE can be configured to insert additional forks at certain program points, in order to explore error prone states. For example, we can configure EXE to fork at each arithmetic operation, and add on one path the constraint that the operation triggers an arithmetic overflow (if possible), and on the other path that it doesn't. Similarly, we can ask EXE to fork at each cast: for narrowing casts, bugs may be introduced when the bits lost are not all zero, while for widening casts from a signed to an unsigned, bugs can be introduced via sign extension – if the signed variable is negative, the result of the cast is a very large number, which sometimes is not intended. While these extra forks are not necessary to find generic bugs such as buffer overflows and division/modulo by zero bugs, they can prove very useful in finding cross-checking errors (see § 6.4).

2.3 Mechanics of the EXE tool

The paths followed by EXE for our example are shown graphically in Figure 7. The branch points (both explicit and implicit) where EXE forks a new process are represented by rhombuses, and the test cases it generates by sequences of four bytes.

Mechanically, at each run of the instrumented code, EXE creates a new directory and, for each path, creates two files: one to hold the concrete bytes it generates, the other to hold the values for each decision (1 to take the true branch, 0 to take

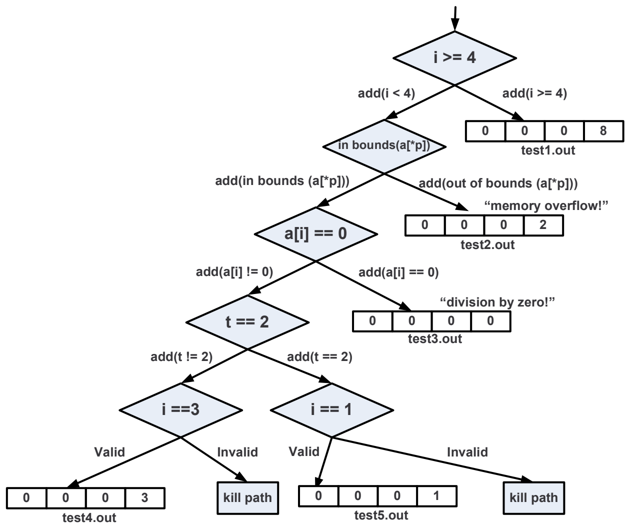

Fig. 7. Execution for the simple C program in Figure 1: EXE generates five test cases, two of which are errors.

the false). The choice points enable easy replay of a single path for debugging. The values can either be read back by using a trivial driver (which EXE provides) or used completely separately from EXE.

In our example, the three paths and two errors lead to five pairs of files that hold (1) concrete byte values for `i` (these files have the suffix `.out`) and (2) the branch decisions for that path (suffix `.forks`). EXE creates a symbolic link `exe-last` pointing to the most recent output directory. The two errors are in `.err` files. If we look at the contents of the file for the division bug (`test3.out`), it shows that each byte of `i` is zero, which when concatenated in the right order and treated as an unsigned 32-bit quantity equals `0`, as required. The branch decision states that we take the false branch at line 5, followed by the (implicit) false branch of the memory overflow check at line 9, and finally the (implicit) true branch of the division check at line 16. Similarly, the concrete values for the pointer error are byte 0 equals 2 and bytes 1, 2, 3 equal 0, which when concatenated yields the 32-bit value 2 as needed.

As expected, EXE is sound with respect to the tests it generates. That is, the concrete test case in each `.out` file is guaranteed to follow the branch points in the corresponding `.forks` file. However, errors in EXE and STP can lead to violations of this key property. Thus, EXE tracks the basic blocks visited when generating a given test case and automatically verifies that the same path is executed when the concrete values are rerun on the checked code. This check found many bugs inside EXE.

3. KEY FEATURES OF STP

This section gives a high-level overview of STP's key features, including the support it provides to EXE for accurately modeling memory. It then describes the optimizations STP performs, and shows experimental numbers evaluating their efficiency.

EXE's constraint solver is, more precisely, a *decision procedure* for bitvectors and arrays. Decision procedures are programs which determine the satisfiability of logical formulas that can express constraints relevant to software and hardware, and have been a mainstay of program verification for several decades. In the past, these decision procedures have been based on variations of Nelson and Oppen's *cooperating decision procedures framework* [Nelson and Oppen 1979] for combining a collection of specialized decision procedures into a more comprehensive decision procedure capable of handling a more expressive logic than any of the specialized procedures can do individually.

The Nelson-Oppen approach has two downsides. Whenever a specialized decision procedure can infer that two expressions are equal, it must do so explicitly and communicate the equality to the other specialized decision procedures, which can be expensive. Worse, the framework tends to lead to a web of complex dependencies, which makes its code difficult to understand, tune, or get right. These problems hampered CVCL [Barrett and Berezin 2004; Barrett et al. 2004], a state-of-the-art decision procedure that we implemented previously.

Our CVCL travails motivated us to simplify the design of STP by exploiting the extreme improvement in SAT solvers over the last decade. STP forgoes Nelson-Oppen contortions, and instead preprocesses the input through the application of mathematical and logical identities, and then eagerly translates constraints into a purely propositional logical formula that it feeds to an off-the-shelf SAT solver (we use MiniSAT [Een and Sorensson 2003]). As a result, the STP implementation is four times smaller than CVCL's, yet often runs orders of magnitude faster. STP is also more modular, because its pieces work in isolation. Modularity and simplicity help constraint solvers as they do everything else. In a sense, STP can be viewed as the result of applying the systems approach to constraint solving that has worked so well in the context of SAT: start simple, measure bottlenecks on real workloads, and tune to exactly these cases. STP was recently judged the co-winner of the 32-bit bitvector (QF_UFBV32) division of the SMTLIB competition [SMTLIB 2006] held as a satellite event of CAV 2006 [Ball and Jones 2006].

Recently, several other decision procedures have been based on eager translation to SAT, including Saturn [Xie and Aiken 2005], UCLID [Bryant et al. 2002], and Cogent [Cook et al. 2005]. Saturn is a static program analysis framework that translates C operations to SAT. It does not directly deal with arrays, so it avoids many interesting problems and optimizations. UCLID implements features such as arrays and arbitrary precision integer arithmetic, but does not focus on bitvector operations. Cogent is perhaps the most similar in architecture and purpose to STP. Judging from the published descriptions of these systems, STP's focus on optimizations for arrays is unique (and uniquely important for use with EXE). STP also has simplifications on word-level operations that are not discussed in the description of Cogent. (At this time, it is difficult to do side-by-side performance comparisons because of lack of common benchmarks and input syntax; Saturn, UCLID and Cogent also didn't participate in the SMTLIB competition.)

3.1 STP primitives

Systems code often treats memory as untyped bytes, and 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 a network packet, inode, packet filter, etc. through pointer casting.

As a result, STP also views memory as untyped bytes. It provides only three data types: booleans, bitvectors, and arrays of bitvectors. A bitvector is a fixed-length sequence of bits. For example, `0010` is a constant, 4-bit bitvector representing the constant 2. With the exception of floating-point, which STP does not support, all C operators have a corresponding STP operator that can be used to impose constraints on bitvectors. STP implements all arithmetic operations (even non-linear operations such as multiplication, division and modulo), 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 (similar to C's ternary operator). In addition, STP supports bit concatenation and bit extraction, features EXE makes extensive use of in order to translate untyped memory into properly-typed constraints.

STP implements its bitvector operations by translating them to operations on individual bits. There are two expression types: *terms*, which have bitvector values, and *formulas*, which have boolean values. If x and y are 32-bit bitvector values, $x + y$ is a term returning a 32-bit result, and $x + y < z$ is a formula. In the implementation, terms are converted into vectors of boolean formulas consisting entirely of single bit operations (AND, XOR, etc.). Each operation is converted in a fairly obvious way: for example, a 32-bit add is implemented as a ripple-carry adder. Formulas are converted into DAGs of single bit operations, where expressions with identical structure are represented uniquely (expression nodes are looked up in a hash table whenever they are created to see whether an identical node already exists). Simple boolean optimizations are applied as the nodes are created; for example, a call to create a node for `AND(x, FALSE)` will just return the `FALSE` node. The resulting boolean DAG is then converted to CNF by the standard method of naming intermediate nodes with new propositional variables.

3.2 Mapping C code to STP constraints

EXE represents each symbolic data block as an array of 8-bit bitvectors. The main advantage of using bitvectors is that they, like the C memory blocks that they represent, are essentially untyped. This property allows us to easily express constraints that refer to the same memory in different ways; 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.

EXE uses STP to solve constraints on input as follows. First, it tracks what memory locations in the checked code hold symbolic values. Second, it translates expressions to bitvector based constraints. We discuss each step below.

Initially, there are no symbolic bytes in the checked code. When the user marks a byte-range, `b`, as symbolic, EXE calls into STP to create a corresponding, identically-sized array b_{sym}, and records in a table that `b` corresponds to b_{sym}. In Figure 1 (line 4), the call to make the 32-bit variable `i` symbolic causes EXE to allocate a bitvector array i_{sym} with four 8-bit elements and record that the concrete address of `i` (`&i`) corresponds to it.

As the program executes, the table mapping concrete bytes to STP bitvectors grows in exactly two cases:

(1) `v = e`: where `e` is a symbolic expression (i.e., has at least one symbolic operand). EXE builds the symbolic expression e_{sym} representing `e`, and records that `&v` (which provides a unique identifier for `v`) maps to it. Note that EXE does not allocate a new STP variable in this case but instead will substitute e_{sym} for `v` in subsequent constraints. EXE removes this mapping when `v` is overwritten with a concrete value or deallocated. In Figure 1 (line 8), EXE records the fact that `p` maps to expression $(char*)a + i_{sym} * 4$ and substitutes any subsequent use of `p`'s value with this expression. (Note that a is replaced by the actual base address of array `a` in the program.)

(2) `b[e]`: where `e` is a symbolic expression and `b` is a concrete array. Since STP must reason about the set of values that `b[e]` could reference, EXE imports `b` into STP by allocating an identically-sized STP array b_{sym}, and initializing it to have the same (constant) contents as `b`. It then records that `b` maps to b_{sym} and removes this mapping only when the array is deallocated.
In Figure 1 (line 12), the array expression `a[*p]` causes EXE to allocate a_{sym}, a 16-element array of 8-bit bitvectors, and assert that:

$$a_{sym} = \{1, 0, 0, 0, 3, 0, 0, 0, 5, 0, 0, 0, 2, 0, 0, 0\}$$

Each expression `e` used in a symbolic operation is constructed in the following way. For each read of size n 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 (i.e., a constant). Otherwise, EXE breaks down `L` into its corresponding bytes $\mathtt{b}_0, \ldots, \mathtt{b}_{n-1}$. It then builds a symbolic expression with the same size as `L` by concatenating each byte's (possibly symbolic) value. For each byte $\mathtt{b}_i$ it queries its data structures to check if $\mathtt{b}_i$ is symbolic. If not, it uses its current concrete value (an 8-bit constant), otherwise it looks up and uses its symbolic expression $(b_i)_{sym}$.

For example, in Figure 1 (line 8), EXE builds the symbolic expression corresponding to `(char*)a + i*4` as follows. EXE determines that the first read of `a` is concrete and so replaces `a` with its concrete address (denoted a) represented as a 32-bit bitvector constant. It then determines that `i` is symbolic, and thus breaks it down into its four bytes, which are mapped to their corresponding STP bitvector array elements $i_{sym}[0]$, $i_{sym}[1]$, $i_{sym}[2]$, and $i_{sym}[3]$. Then, the four bitvectors are concatenated to obtain the expression $i_{sym}[3]$ @ $i_{sym}[2]$ @ $i_{sym}[1]$ @ $i_{sym}[0]$ (where "@" denotes bitvector concatenation, and we use little-endian order for multi-byte values), which corresponds to the four-byte read of `i`. Finally, the constant `4` is replaced by the corresponding 32-bit bitvector constant 0...00000100. The resulting expression is

$$a + (i_{sym}[3]@i_{sym}[2]@i_{sym}[1]@i_{sym}[0]) * 0...00000100$$

A limitation of STP is that it does not support pointers directly. EXE emulates symbolic pointer expressions by mapping them as an array reference at some offset. For each pointer `p` in the checked code, EXE tracks the data object to which `p` points by instrumenting all allocation and deallocation sites as well as all pointer arithmetic expressions (standard techniques developed by bounds-checking compilers [Ruwase and Lam 2004]). For example, in Figure 1 (line 4), EXE records that `p` points to the data block `a` of size 16. Then, when EXE encounters a pointer dereference `*p`: (1) it looks up the block `b` to which pointer `p` refers; (2) looks up the corresponding STP array b_{sym} associated with `b`; and (3) computes the (possibly symbolic) offset of `p` from the base of the object it points to (i.e., `o = p - b`). EXE can then use the symbolic expression $b_{sym}[i_{sym} + o_{sym}]$ in symbolic constraints.

However, STP's lack of pointer support means that when EXE encounters a double-dereference `**p` of a symbolic pointer `p` it *concretizes* the first dereference (`*p`), 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.) The user is also informed when this happens. This situation has rarely shown up in practice (see § 4.3), but we are working on removing it.

3.3 Concrete solutions

STP's ability to generate a concrete solution for a given set of constraints is essential for EXE. Whenever EXE terminates to execute a path thorough the program under checking, it asks STP for a concrete solution for the set of constraints gathered on that path. Effectively, this solution represents a test case that will drive an uninstrumented version of the checked program through the exact same path as the one explored by EXE. In the case of an error path, the test case represents an actual attack that can be mounted against the vulnerable program. The ability to independently confirm the attack is a significant advantage of EXE, which allows it to have absolutely no false positives (while EXE is sound with respect to the attacks it generates, there can still be false positives due to bugs in EXE and/or STP).

Currently, STP can generate a single solution for a given set of constraints. However, note that EXE can easily overcome this limitation by using the following simple algorithm: (1) given a set of constraints C on input I, we first ask STP for a solution s; (2) we add constraint $I \neq s$ to the constraint set C and ask STP for a new solution. We can repeat this algorithm to obtain more solutions (if they exist).

While in general EXE requires a single solution for a set of constraints, we do employ this algorithm to determine whether a symbolic variable has a single concrete solution (by simply checking if a solution can still be generated in step (2) of the algorithm above). If a symbolic variable has a single solution, we can replace it by a concrete variable with the right value. This optional check in EXE has a big impact on performance for certain benchmarks, but unfortunately it slows down other benchmarks, and for this reason we only enable it when it proves beneficial.

3.4 The key to speed: fast array constraints

Almost always, the main bottleneck in STP when used in EXE is reasoning about arrays. This subsection discusses STP's key array optimizations.

STP is an implementation of logic, so it is a purely functional language. The logic has one-dimensional arrays that are indexed by bitvectors and contain bitvectors. The operations on arrays are $read(A, i)$, which returns the value at location $A[i]$ where A is an array and i is an index expression of the correct type, and

$\mathit{write}(A, i, v)$, which returns a new array with the same value as A at all indexes except i, where it has the value v. Array reads and writes can appear as subexpressions of an `if-then-else` construct, denoted by $\mathit{ite}(c, a, b)$, where c is the condition, a the `then` expression, and b the `else` expression.

STP eliminates array expressions by translating them to bitvector primitives (which it then translates to SAT). This is accomplished through two main transformations. The first, **read-over-write**, eliminates all $\mathit{write}(A, i, v)$ expressions: [1]

$$\mathit{read}(\mathit{write}(A, i, v), j) \Rightarrow \mathit{ite}(i = j, v, \mathit{read}(A, j))$$

The second, **read elimination**, eliminates all *read* expressions via a transformation mentioned in [Bryant et al. 2002] that enforces the axiom that if two indexes i_s and i_t are the same, then $\mathit{read}(A, i_s)$ and $\mathit{read}(A, i_t)$ should return the same value. Mechanically, STP first replaces each occurrence of a read $\mathit{read}(A, i_j)$ with a new variable v_j, and then for each two terms i_s, i_t ever used to index into the same array A, it adds the *array axiom*:

$$i_s = i_t \Rightarrow v_s = v_t$$

For example, consider the formula:

$$(\mathit{read}(A, i_1) = e_1) \wedge (\mathit{read}(A, i_2) = e_2) \wedge (\mathit{read}(A, i_3) = e_3)$$

The transformed result would be:

$$(v_1 = e_1) \wedge (v_2 = e_2) \wedge (v_3 = e_3) \wedge (i_1 = i_2 \Rightarrow v_1 = v_2) \wedge$$

$$(i_1 = i_3 \Rightarrow v_1 = v_3) \wedge (i_2 = i_3 \Rightarrow v_2 = v_3)$$

Read elimination expands each formula by $n(n-1)/2$ nodes, where n is the number of syntactically distinct index expressions. Unfortunately, this blowup is lethal for arrays of a few thousand elements, which occur frequently in EXE. Fortunately, while finessing this problem appears hard in general, two optimizations we developed work well on the constraints generated by EXE.

The *array substitution optimization* reduces the number of array variables by substituting out all constraints of the form $\mathit{read}(A, c) = e$, where c is a constant and e does not contain another array read. Programs often index into arrays using constant indexes, so this is a case that occurs often in practice (see § 4.3). The optimization has two passes. The first pass builds a substitution table with the left-hand-side of each such equation ($\mathit{read}(A, c)$) as the key and the right-hand-side (e) as the value, and then deletes the equation from the EXE query. The second pass over the expression replaces each occurrence of a key by the corresponding table entry. Note that for soundness, if we encounter a second equation whose left-hand-side is already in the table, the second equation is not deleted and the table is not changed. For our example, if we saw a subsequent equation $\mathit{read}(A, i_1) = e_4$ we would leave it; the second pass of the algorithm would rewrite it as $e_1 = e_4$.

The second optimization, *array-based refinement*, delays the translation of array *read*s with non-constant indexes, in effect introducing some laziness into STP's

[1]Note that a *write* makes sense only inside a *read* node. A *write* node by itself has no effect, and can be ignored.

handling of arrays, in the hope of avoiding the $O(n^2)$ blowup from the read elimination transformation. Its main trick is to solve a less-expensive approximation of the formula, check the result in the original formula, and try again with a more accurate approximation if the result is incorrect.

Initially, all array read expressions are replaced by variables to yield an approximation of the original formula. The resulting logical formula is under-constrained, since it ignores the array axioms that require that array reads return the same values when indexes are the same. If the resulting under-constrained formula is not satisfiable, there is no solution for the original formula and STP returns unsatisfiable.

If, however, the SAT solver finds a solution to the under-constrained formula, then that solution is not guaranteed to be correct because it could violate one of the array axioms. For example, suppose STP is given the formula $(\mathit{read}(A, 0) = 0) \wedge (\mathit{read}(A, i) = 1)$. STP would first apply the substitution optimization by deleting the constraint $\mathit{read}(A, 0) = 0$ from the formula, and inserting the pair $(\mathit{read}(A, 0), 0))$ in the substitution table. Then, it would replace $\mathit{read}(A, i)$ by a new variable v_i, thus generating the under-constrained formula $v_i = 1$. Suppose STP finds the solution $i = 1$ and $v_i = 1$. STP then translates the solution to the variables of the original formula to get $(\mathit{read}(A, 0) = 0) \wedge (\mathit{read}(A, 1) = 1)$. This solution is satisfiable in the original formula as well, so STP terminates since it has found a true satisfying assignment.

However, suppose that STP finds the solution $i = 0$ and $v_i = 1$. Under this solution, the original formula evaluates to $(\mathit{read}(A, 0) = 0) \wedge (\mathit{read}(A, 0) = 1)$, which gives $0 = 1$. Hence, the solution to the under-constrained formula is not a solution to the original formula. When this happens, it must be because some array axiom was violated. STP adds array axioms to the formula and solves again until it gets a correct result. There are many policies for adding axioms, any of which is correct and will terminate so long as all of the axioms are added in the worst case. The current policy, which seems to work well, is to find an array index term for which at least one axiom is violated, then add all of the axioms involving that term. In our example, it will add the axiom $i = 0 \Rightarrow \mathit{read}(A, i) = \mathit{read}(A, 0)$. Then, the process of finding a satisfying assignment is repeated, by calling the SAT solver on the new under-constrained formula. The result must satisfy the newly added axioms, which the previous assignment violated, so the algorithm will not repeat assignments and will not violate previously added axioms. This process must terminate since there are only finitely many array axioms.

In the worst case, the algorithm will add all $n(n-1)/2$ array axioms, at which time it is guaranteed to return a correct result because there are no more axioms it can violate. However, in practice, this loop will often terminate quickly because the formula can be proved unsatisfiable without all the array axioms, or because it luckily finds a true satisfying assignment without adding all the axioms.

3.5 Boolean and mathematical simplifications

In addition to the above mentioned optimizations, STP implements several boolean and mathematical identities. These identities, or *simplifications*, also dramatically reduce the size of the input, before it is fed to the SAT solver. Some example identities include the associativity and commutativity laws for addition and multi-

Solver	**Total Time**	**Timeouts**
CVCL	60,366s	546
STP (no optimizations)	3,378s	36
STP (substitution)	1,216s	1
STP (refinement)	624s	1
STP (simplifications)	336s	0
STP (subst+refinement)	513s	1
STP (simplif+subst)	233s	0
STP (simplif+refinement)	220s	0
STP (all optimizations)	110s	0

Table I. Performance of STP and CVCL on a regression suite of 8495 test cases taken from our test programs. Queries time out (are aborted) after 60 seconds, which underestimates performance differences, since they could run for much longer. Using this conservative estimate, fully optimized STP is roughly 30X faster than the unoptimized version and 550X faster than CVCL and has no timeouts.

plication, distribution of multiplication by constants over addition, and the combination of like terms (e.g., $x + (-x)$ is simplified to 0). Simplifications play a crucial role in STP's efficiency and robustness. Instead of listing all the simplifications implemented in STP, we discuss the value added by simplifications, as well as their pitfalls, which may not be immediately apparent.

The simplifications in STP can be classified into two categories: (1) those that cause the input DAG (Directed Acyclic Graph) to blow-up (i.e. the number of nodes in the output DAG is typically much larger than in the input, often quadratic in size), and (2) those that typically do not cause such blow-up. We refer to the identities falling in the former category as *blow-up* simplifications, and we need to be very careful when deciding whether they should be applied. As an example, consider the quadratic blow-up of a thousand node DAG into a million node DAG. Such an increase can often be lethal for either the subsequent simplifications or for the SAT solver itself, and so in this case the simplification should not be applied. An example of a *blow-up* simplification is the distributivity law of multiplication over addition. This law often increases the DAG size. For example, consider the input DAG $(x + y)^n$, where $(x + y)$ is a shared expression. The output after the transformation is $x^n + nx^{n-1}y + ... + y^n$. This transformation breaks the sharing, which in turns increases the DAG significantly, becoming problematic for subsequent stages. On the other hand, the distributivity law is very useful when one of the multiplicands is a constant, often allowing STP to combine like terms. The lesson learned here is that the DAG size (usually highly determined by the amount of sub-expression sharing) plays a critical role in determining whether a simplification should be applied.

Consistent with this observation, simplifications that are guaranteed to reduce the size of the DAG should always be applied. Two such simplifications are constant folding and constant propagation. In constant folding, terms such as $2 + 3$ are simplified to 5 (another example is reducing 0@0 to 00, where @ is the concatenation operator). In constant folding, values of known constants are substituted in more complex expressions. For example, consider an input $(x = 5) \wedge (x + y + z = 7)$. Replacing x with 5 in $(x + y + z = 7)$ is an instance of constant propagation.

These simplification can have a huge impact on subsequent stages of STP, as shown by the experimental results in Section 3.6. For example, consider a constraint involving an array read, of the form $A[x] = 2$. If the constant folding and propagation simplifications can infer that x is a constant and this fact is propagated to the subsequent array optimizations, this constraint would go directly into the substitution map, instead of generating the more expensive array read axioms.

3.6 Measured performance

The optimizations outlined in Sections 3.4 and 3.5 have made it possible to deal with fairly large constant arrays when there are relatively few non-constant index expressions, which is sufficient to permit considerable progress in using EXE on real examples.

Table I gives experimental measurements for these optimizations. The experiment consists of running different versions of STP and our old solver, CVCL, over the performance regression suite we have built up of 8495 test cases taken from our test programs. The experiments for all solvers were run on a Pentium 4 machine at 3.2 GHz, with 2 GB of RAM and 512 KB of cache. The table gives the times taken by CVCL, baseline STP with no optimizations, STP with a subset of all optimizations enabled, and STP with full optimizations, i.e. substitution, array-based refinement, and simplifications. The third column shows the number of examples on which each solver timed out. The timeout was set at 60 seconds, and is added as penalty to the time taken by the solver (but in fact causes us to grossly underestimate the time taken by CVCL and earlier versions of STP since they could run for many minutes or even hours on some of the examples).

The baseline STP is nearly 20 times faster than CVCL, and more interestingly, times out in far fewer cases. The fully optimized version of STP is about 30 times faster than the unoptimized version, almost 550 times faster than CVCL, and there are no timeouts.

4. EXE OPTIMIZATIONS

This section presents optimizations EXE uses and measures their effectiveness on five benchmarks. We first present two optimizations: caching constraints to avoid calling STP (§ 4.1), and removing irrelevant constraints from the queries EXE sends to STP (§ 4.2). We then measure the cumulative improvement of these optimizations, and provide an empirical feel for what symbolic execution looks like, including the time spent in various parts of EXE, and a description of the symbolic slice through the code (§ 4.3). Finally, we discuss and measure EXE's search heuristics (§ 4.4).

4.1 Constraint caching

EXE caches the result of satisfiability queries and constraint solutions in order to avoid calling STP when possible. This cache is managed by a server process so that multiple EXE processes (created by forking at each conditional) can coordinate. Before invoking STP on a query `q`, an EXE process prints `q` as a string, computes an MD4 cryptographic hash of this string, and sends this hash to the server. The server checks its persistent cache (a file) and if it gets a hit, returns the result. If not, the EXE process does a local STP query and then sends the $(hash, result)$ pair back to the server. Constraint solutions are cached in a similar way.

4.2 Constraint independence optimization

This section describes one of EXE's most important optimizations, *constraint independence*, which exploits the fact that we can often divide the set of constraints EXE tracks into multiple independent subsets of constraints. Two constraints are considered to be independent if they have disjoint sets of operands (i.e. disjoint sets of array reads).

For example, assume EXE tracks the following set of three constraints:

$$(A[1] = A[2] + A[3]) \wedge (A[2] > A[4]) \wedge (A[7] = A[8])$$

We can divide this set into two subsets of independent constraints

$$(A[1] = A[2] + A[3]) \wedge (A[2] > A[4])$$

and

$$A[7] = A[8]$$

and solve them separately. Breaking a constraint into multiple independent subsets has two benefits. First, EXE can discard irrelevant constraints when it asks STP if a constraint c is satisfiable, with a corresponding decrease in cost. Instead of sending all the constraints collected so far to STP, EXE only sends the subset of constraints s_c to which c belongs, ignoring all other constraints. The worst case, when no irrelevant constraints are found, costs no more than the original query (omitting the small cost of computing the independent subsets).

Second, this optimization yields additional cache hits, since a given a subset of independent constraints may have appeared individually in previous runs. Conversely, including all constraints vastly increases the chance that at least one is different and so gets no cache hit. To illustrate, assume we have the following code fragment, which operates on two unconstrained symbolic arrays A and B:

```
if (A[i] > A[i+1]) {
   ...
}
if (B[j] + B[j-1] == B[j+1]) {
   ...
}
```

There are four paths through this code; EXE will thus create four processes. After forking and following each branch, EXE checks if the path is satisfiable. Without the constraint independence optimization, each of these four satisfiability queries will differ and miss in the cache. However, if the optimization is applied, some queries repeat. For example, when the second branch is reached, two of the four queries will be

$$(A[i] > A[i+1]) \wedge (B[j] + B[j-1] = B[j+1])$$

and

$$(A[i] \leq A[i+1]) \wedge (B[j] + B[j-1] = B[j+1])$$

which both devolve to

$$B[j] + B[j-1] = B[j+1]$$

since, in each query, the first constraint is unrelated to the last one, and its satisfiability was already determined when EXE reached the first branch.

Real programs often have many independent branches, which introduce many irrelevant constraints. These add up quickly. For example, assuming n consecutive independent branches (the example above is such an instance for $n = 2$), EXE will issue $2(2^n - 1)$ queries to STP (for each `if` statement, we issue two queries to check if both branches are possible). The optimization exponentially reduces this query count to $2n$ (two queries the first time we see each branch), since the rest of the time we find the result in the cache.

We compute the constraint independence subsets by constructing a graph G, whose nodes are the set of all array reads used in the given set of constraints. For the first example in the section, the set of nodes is $\{A[1], A[2], A[3], A[4], A[7], A[8]\}$. We add an edge between nodes n_i and n_j of G if and only if there exists a constraint c that contains both as operands. Once the graph G is constructed, we apply a standard algorithm to determine G's connected components. Finally, for each connected component, we create a corresponding independent subset of constraints by adding all the constraints that contain at least one of the nodes in that connected component. At the implementation level, we don't construct the graph G explicitly. Instead, we keep the nodes of G in a union-find structure (as described in Chapter 21 of [Cormen et al. 2001]), which we update each time a new constraint is added.

There are two additional issues that our algorithm has to take into account. First, an array read may contain a symbolic index. In this case, we are conservative, and merge all the elements of that array into a single subset. For example, if a constraint refers to $A[i]$, where i is a symbolic index, then the algorithm would merge all the elements of A into the same subset. We could optimize this in the future by looking at the constraints imposed on the symbolic index i. For example, if i could only have values 1 or 2, then only $A[1]$ and $A[2]$ need to be merged.

The second issue relates to array writes. Since EXE and STP arrays are functional, each array read explicitly contains an ordered list of all array writes performed so far. Each array write is remembered as a pair consisting of the location that was updated, and the expression that was written to that location. When processing this list of array writes, we are again conservative, and merge all the expressions written into the array (the right hand side of each array write) into the subset of the original read. In addition, if any array write is performed at a symbolic index, we merge all the elements of the array into a single subset.

4.3 Experiments

We evaluate our optimizations on five benchmarks. These benchmarks consist of the three applications discussed in Section 6, `bpf`, `pcre`, and `udhcpd`, to which we added two more: `expat`, an XML parser library, and `tcpdump`, a tool for printing out the headers of packets on a network interface that match a boolean expression.

We run each benchmark under four versions of EXE: no optimization, caching only, independence only, and finally with both optimizations turned on. As a baseline, we run each benchmark for roughly 30 minutes using the unoptimized version of EXE, and record the number of test cases n that this run generates. We then run the other versions until they generate n test cases. All experiments are

	`bpf`	`expat`	`pcre`	`tcpdump`	`udhcpd`
Test cases	7333	360	866	2140	328
None	30.6	28.4	31.3	28.2	30.4
Caching	32.6	30.8	34.4	27.0	36.4
Independence	17.8	25.2	10.0	24.9	30.5
All	10.3	26.3	7.5	23.6	32.1
STP cost	6.9	24.6	2.8	22.4	23.1

Table II. Optimization measurements, times in minutes. STP cost gives time spent in STP when all optimizations are enabled.Tables III, IV, and V explore the fully optimized run (All) in more detail.

		`bpf`	`expat`	`pcre`	`tcpdump`	`udhcpd`
1	Cache hit rate	92.8%	0%	83%	35%	9.1%
2	Hit rate w/o independence	0.1%	0%	17.5%	12.6%	9.1%
3	Avg. # of independent subsets	19	2,824	122	13	1
4	Independence overhead	0m	0m	.1m	0m	0m
5	Cache lookup cost	1.1m	1.2m	1.9m	0.4m	2.1m
6	% of lookup spent printing	72%	96%	84%	90%	95%

Table III. Optimization breakdown

performed on a dual-core 3.2 GHz Intel Pentium D machine with 2 GB of RAM, and 2048 KB of cache.

Table II gives the number of test cases generated, as well as the runtime for each optimization combination. Full optimization ("All") significantly sped up two of five benchmarks: `bpf` by roughly a factor of three, and `pcre` by more than a factor of four. Both `tcpdump` and `expat` had marginal improvements (20% and 7% faster respectively), but `udhcpd` slows down by 5.6%. As the last row shows, with the exception of `pcre`, the time spent in STP represents by far the dominant cost of EXE checking.

Table III breaks down the full optimization run. As its first three rows show, caching without independence is not a win — its overhead (see Table II) actually increases runtime for most applications, varying between 6.5% for `bpf` and 19.7% for `pcre`. With independence, the hit rate jumps sharply for both `bpf` and `pcre` (and, to a lesser extent, `tcpdump`), due to its removal of irrelevant constraints. The other two applications show no benefit from these optimizations — `udhcpd` has no independent constraints and `expat` has no cache hits. The average number of independent subsets (row 3) shows how interdependent our constraints are, varying from over 2,800 subsets for `expat` to only 1 (i.e., no independent constraints) for `udhcpd`.

The next three rows (4–6) measure the overhead spent in various parts of EXE. Reassuringly, the cost of independence is near zero. On the other hand, cache lookup overhead (row 5) is significant, due almost entirely to our naive implementation. On each cache lookup (§ 4.1), EXE prints the query as a string and then hashes it. As the table shows (row 6) the cost of printing the string dominates all other cache lookup overheads. Obviously, we plan to eliminate this inefficiency in the next version of the system.

		bpf	expat	pcre	tcpdump	udhcpd
1	# of queries (cache misses)	163K	5K	188K	22K	4K
2	Total # of constraints	0.4M	9.6M	3.5M	1.3M	0.6M
3	Total # of nodes	2.0M	32.7M	17.8M	20.7M	431.7M
4	# non-linear constraints	4K	11K	96K	343K	508K
5	% constraints non-linear	0.9%	0.1%	2.8%	27.1%	81.1%
6	Reads from symbolic array	0.4M	11.8M	3.8M	1.6M	4.0M
7	% sym. array reads with sym. index	0.3%	0.3%	2.9%	7.8%	62.9%
8	Writes to symbolic array	62	2.3M	0.7M	0	0
9	% sym. array writes with sym. index	100%	0%	1.8%	0%	0%

Table IV. Dynamic counts from queries sent to STP.

		bpf	expat	pcre	tcpdump	udhcpd
1	Symbolic input size (bytes)	96	10	16	84	548
2	Total statements run (not unique)	298,195	41,345	423,182	40,097	15,258
3	% of statements symbolic	29.2%	8.5%	34.7%	41.7%	23.6% %
4	Explicit symbolic branch points	77,024	1,969	98,138	11,425	888
5	% with both branches feasible	11.3%	19.3%	0.9%	19.4%	52.8%
6	Avg. # symbolic branches per path	38.33	43.44	55.72	103.37	200.14
7	Symbolic checks	1,490	904	4,451	552	1,535
8	Pointer concretizations	0	0	0	73	0
9	Symbolic args. to uninstr. calls	0	0	0	0	0

Table V. Dynamic counts from EXE execution runs.

Table IV breaks down the queries sent to STP. The first three rows give the total number of: queries, constraints, and nodes. These last two numbers give a feel for query complexity: `bpf` is the easiest case (a small number of constraints, with roughly five nodes per constraint), whereas `udhcpd` is the worst with 688 nodes per constraint.

The next two rows give the number of non-linear constraints (row 4) and their percentage (row 5) of the total constraints (from row 2). Non-linear constraints contain one or more non-linear operators — multiplication, division, or modulo — whose right hand side is not a constant power of two. In general, the more non-linear operations, the slower constraint solving gets, as the SAT circuits that STP constructs for these operations are expensive. For our benchmarks, only `udhcpd` has a large number of non-linear constraints, which translates into a large amount of time spent in STP.

The final four rows (6–9) give the number of reads and writes from and to symbolic data blocks, and the percentage of these that use symbolic indexes. While there are many array operations, with the exception of `udhcpd`, very few use symbolic indexes, which explains why the STP array substitution optimization (§ 3.4) was such a big win.

Table V gives more dynamic execution counts from the full optimization runs. The first row gives the number of bytes initially marked as symbolic; this represents the size of the symbolic filter and data in `bpf`, the size of the XML expression to be parsed in `expat`, the packet length in `udhcpd` and `tcpdump`, and the regular expression pattern length in `pcre`.

The next row (row 2) gives the total number of dynamic statements executed (assignments, branches, parameter and return value passing) across all paths executed by EXE, while the next (row 3) gives the percentage that are symbolic. For our benchmarks, this percentage varies from only 8.46% for `expat` to 41.70% for `tcpdump`. This numbers are encouraging and validate our approach of mixing concrete and symbolic execution, which lets us ignore a large amount of code in the programs we check.

The next three rows (4–6) look at symbolic branches, including the implicit branches EXE does for checking. Row 4 gives the total number of explicit symbolic branch points and row 5 the percentage of these branch points that had both branches feasible. (EXE pruned the other branches because the path's constraints were not satisfiable.) On our benchmarks, EXE was able to prune more than 80% of the branches it encountered, with the exception of `udhcpd` where it pruned (only) 47.18% of the branches. These results are reassuring for scalability – while the potential number of paths in the search space grows exponentially with the number of symbolic branches, the actual growth is much smaller: real code appears to have many dependencies between program points.

Row 6 measures the average number of symbolic branches (both implicit and explicit) per path. This number is large: ranging from around 38 up to 200 branches, which means that random guessing would have a hard time satisfying all the branches to get to the end of one path, much less the hundreds or thousands that EXE can systematically explore.

Row 7 gives the total number of times EXE performed a symbolic check. (In addition to these checks, EXE performs many more similar concrete checks.) Row 8 shows how many times EXE had to concretize a pointer because it encountered a symbolic dereference of a symbolic pointer (§ 3.2). This situation occurs in only one of our five benchmarks, `tcpdump`. Finally, row 9 shows that no uninstrumented functions were called with symbolic data as arguments.

4.4 Search heuristics

When EXE forks execution, it must pick which branch to follow first. By default, EXE uses depth-first search (DFS), picking randomly between the two branches. DFS keeps the current number of processes small (linear in the depth of the process chain), but 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 attempts to execute the loop as many times as possible, thus potentially taking a very long time to exit the loop.

In order to overcome this problem, we use search heuristics to drive the execution along "interesting" execution paths (e.g., that cover unexplored statements). After a `fork` call, each forked EXE process calls into a search server with a description of its current state (e.g., its current file, line number, and backtrace) and blocks until the server replies. The search server examines all blocked processes and picks the best one in terms of some heuristic that is more global than simply picking a random branch to follow. Our current heuristic uses a mixture of best-first and depth-first search. The search server picks the process blocked at the line of code run the fewest number of times and then runs this process (and its children) in a DFS manner through four branches, picking a random branch at each point where

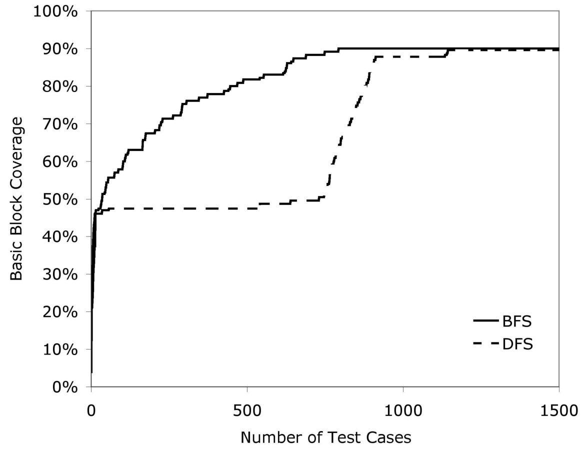

Fig. 8. Best-first search vs. depth-first search.

execution down both edges is feasible. It then picks another best-first candidate and iterates. This is just one of many possible heuristics, and the server is structured so that new heuristics are easy to plug in.

We experimentally evaluate our best-first search (BFS) heuristic in the context of one of our benchmarks, the Berkeley Packet Filter (BPF) (described in more detail in § 6.1). We start two separate executions of EXE, one using DFS and the other using BFS. We let both EXE executions run until they achieved full basic block coverage. Figure 8 compares BFS to DFS in terms of basic block coverage. (For visual clarity the graph only shows block coverage for the first 1500 test cases, as only a few blocks are missing from the coverage by these test cases.) BFS converges to full coverage more than twice as fast as DFS: 7,956 test cases versus 18,667. More precisely, EXE gets 91.74% block coverage, since there are several basic blocks in BPF that EXE cannot reach, such as dead code (e.g. the failure branch of asserts), or branches that do not depend on the input marked as symbolic.

Figure 9 then compares EXE against random testing, also in terms of basic block coverage. We generate one million random test cases of the same size as those generated by EXE, and run these random test cases through a lightly-instrumented version of BPF that records basic block coverage. These test cases only cover 56.96% of the blocks in BPF; EXE achieves the same coverage in only 75 tests when using BFS. Even more strikingly, these million random test cases yield only 131 unique paths through the code, while each of EXE's test cases represents a unique path. Most importantly, random testing did not have a wall clock time advantage over BFS: random testing with a million test cases took over four times as long as running BPF through EXE with BFS.

5. LIMITATIONS

This section summarizes the most important limitations of EXE and STP. On the constraint solving side, STP neither supports floating point arithmetic nor provides pointers. All other operations present in C are supported, albeit non-linear operations such as multiplication and modulo are often very slow.

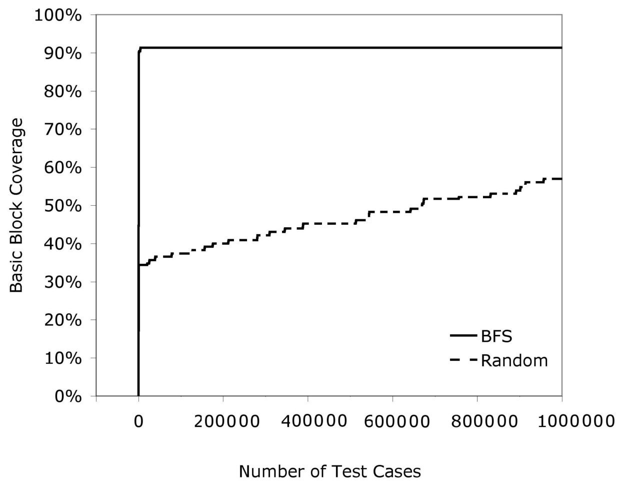

Fig. 9. EXE with best-first search vs. random testing.

Since STP does not provide pointers, EXE maps pointer dereferences to array references at some offset, by tracking the base object of each pointer in the program, as discussed in Section 3.2. As a consequence of this (but also because of the interaction with uninstrumented code, as discussed below), EXE may not be always able to determine the underlying object of the pointer being dereferenced, which in turn leads it to concretize part of the symbolic pointer expression, as discussed in more detail in Section 3.2. When this happens, EXE may discard certain execution paths, but will continue to make progress. Note that a straightforward remedy to this problem would be to model memory as a single STP array indexed by 32-bit bitvectors, but this approach is currently too slow to be practical.

Interaction with uninstrumented code may lead to EXE missing some constraints, which in turn may lead it to explore impossible paths. For this reason, our approach has been to instrument all the code on which the program being checked depends, including any standard libraries. The approach of mixed concrete and symbolic execution, combined with other optimizations presented in this paper, has made this approach feasible, as shown by our experimental results in Section 6.

Last but not least, an important limitation of EXE is that it does not directly support data blocks of symbolic size. To be precise, EXE technically does support symbolic sizes, but these are usually immediately "concretized" while running the program. As an illustration, consider the following `for` loop, where n is a symbolic unsigned integer representing the size of the array a:

```
for (i=0; i < n; i++)
    a[i] = i;
```

Before the first iteration through the loop, EXE will encounter the branch condition $0 < n$. As a result, it will fork execution, setting the value of n to 0 on one path, and adding the constraint $n >= 1$ on the other. On the latter path, after the first iteration is executed, EXE will encounter the branch condition $1 < n$, and again, it will fork execution, setting the value of n to 1 on one path, and adding the constraint $n >= 2$ on the other. Thus, the loop is explored in an iterative deepening

```
s[0].code = BPF_STX; // also: (BPF_LDX|BPF_MEM)
s[0].k    = 0xfffffff0UL;
s[1].code = BPF_RET;
```

Fig. 10. A BPF filter of death

manner, by successively setting the value of the length n to 0, 1, 2 and so on. Thus, when running our benchmarks, we usually set the length of the symbolic input to a fixed (larger) concrete value. Making symbolic execution effectively handle inputs of a symbolic size is still an open problem in the context of real applications. One direction that we plan to explore in the future is the inference of loop invariants combined with support for universal quantifiers in the constraint solving domain.

6. USING EXE TO FIND BUGS

This section presents three case studies that use EXE to find bugs in: (1) two packet filter implementations, (2) the `udhcpd` DHCP server, and (3) the `pcre` Perl compatible regular expressions library. We also summarize a previous effort of applying EXE to file system code.

6.1 Packet filters

Many operating systems allow programs to specify packet filters which describe the network packets they want to receive. Most packet filter implementations are variants of the Berkeley Packet Filter (BPF) system. BPF filters are written in a pseudo-assembly language, downloaded into the kernel, validated by the BPF system, and then applied to incoming packets. We used EXE to check the packet filter in both FreeBSD and Linux. FreeBSD uses BPF, while Linux uses a heavily modified version of it. EXE found two buffer overflows in the former and four errors in the latter. BPF is one particularly hard test of EXE — small, heavily-inspected and mature code, written by programmers known for their skill.

A filter is an array of instructions specifying an opcode (`code`), a possible memory offset to read or write (`k`), and several other fields. The BPF interpreter iterates over this filter, executing each opcode's corresponding action. This loop is the main source of vulnerabilities but is hard to test exhaustively (e.g., hitting all opcodes even once using random testing takes a long time).

We used a two-part checking process. First, we marked a fixed-sized array of filter instructions as symbolic and passed it to the packet filter validation routine `bpf_validate`, which returns `1` if it considers a filter legal. For each valid filter, we then mark a fixed-size byte array (representing a packet) as symbolic and run the filter interpreter `bpf_filter` on the symbolic filter with the symbolic packet, thus checking the filter against all possible data packets of that length.

This checking illustrates one of EXE's interesting features: it turns interpreters into generators of the programs they can interpret. In our example, running the BPF interpreter on a symbolic filter causes it to generate all possible filters of that length, since each branch of the interpreter will fork execution, adding a constraint corresponding to the opcode it checked.

Figure 10 shows one of the two filters EXE found that cause buffer overflows

```
// Code extracted from bpf_validate. Rejects
// filter if opcode's memory offset is more than
// BPF_MEMWORDS.
// Forgets to check opcodes LDX and 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;
 ...
// Code extracted from bpf_filter: pc points to current
// instruction. Both cases can overflow mem[pc->k].
   case BPF_LDX|BPF_MEM:
      X = mem[pc->k]; continue;
   ...
   case BPF_STX:
      mem[pc->k] = X; continue;
```

Fig. 11. The BPF code Figure 10's filter exploits.

```
// other filters that cause this error:
//      code = (BPF_LD|BPF_B|BPF_IND)
//      code = (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;
```

Fig. 12. A Linux filter of death

in FreeBSD's BPF. The bug can occur when the opcode of a BPF instruction is either `BPF_STX` or `BPF_LDX | BPF_MEM`. As shown in Figure 11, `bpf_validate` forgets to bounds check the memory offset given by these instructions, as it does for instructions with opcodes `BPF_ST` or `BPF_LD | BPF_MEM`. This missing check means these instructions can write or read arbitrary offsets off the fixed-sized buffer `mem`, thus crashing the kernel or allowing a trivial exploit.

Linux had a trickier example. EXE found three filters that can crash the kernel because of an arithmetic overflow in a bounds check, shown in Figure 12. As with BPF, the offset field (`k`) causes the problem. Here, the code to interpret `BPF_LD` instructions eventually calls the function `skb_header_pointer`, which computes an offset into a given packet's data and returns it. This routine is passed `s[0].k` as the `offset` parameter, and values `4` or `2` as the `len` parameter. It extracts the size of the current message header into `hlen` and checks that `offset + len` $\leq$ `hlen`. However, the filter can cause `offset` to be very large, which means the signed addition `offset + len` will overflow to a small value, passing the check, but then causing that very large `offset` value to be added to the message `data` pointer. This allows attackers to easily crash the machine. This error would be hard to find with random testing. Its occurrence in highly-visible, widely-used code, demonstrates that such tricky cases can empirically withstand repeated manual inspection.

```
static inline void *
skb_header_pointer(struct sk_buff *skb,
            int offset, int len, void *buffer) {

   int hlen = skb_headlen(skb);

   // Memory overflow. offset=s[0].k; a filter
   // can make this value very large, causing
   // offset + len to overflow, trivially passing
   // the bounds check.
   if (offset + len <= hlen)
        return skb−>data + offset;
```

Fig. 13. The Linux code Figure 12's filter exploits.

Offset	*Hex value*
0000	0000 0000 0000 0000 0000 0000 0000 0000
0010	0000 0000 0000 0000 0000 0000 5A00 0000
....	....
00F0	2100 00F9 0000 0000 0000 0000 0000 0000
....	....
01E0	0000 0000 0000 0000 0000 0000 2734 0000
01F0	0000 0000 0000 0000 0000 0000 0000 0000
0200	0000 0000 0000 0000 0000 0000 0000 3500
0210	030F 0000 0000 0000 0000 0000 0000 0000
0220	0032 0036

Fig. 14. An EXE generated packet that causes an out-of-bounds read in `udhcpd`.

6.2 A complete server: udhcpd

We also checked `udhcpd-0.9.8`, a clean, well-tested user-level DHCP server. We marked its input packet as symbolic, and then modified its network read call to return a packet of at most 548 bytes. After running `udhcpd` long enough to generate 596 test cases, EXE detected five different memory errors: four-byte read overflows at lines 213 and 214 in `dhcpd.c` and three similar errors at lines 79, 94, and 99 in `options.c`. These errors were not found when we tested the code using random testing. EXE generated packets to trigger all of these errors, one of which is shown in Figure 14. We confirmed these errors by rerunning the concrete error packets on an uninstrumented version of `udhcpd` while monitoring it with `valgrind`, a tool that dynamically checks for some types of memory corruption and storage leaks [Nethercote and Seward 2003].

Upon investigating the cause of these errors, we discovered that the `get_option` method in `options.c` lacks several bounds checks. This method extracts the option with the given code from a given packet's option buffer. Let us consider one representative code snippet, shown in Figure 15. Note that options in DHCP packets are stored in one large buffer of variable-size entries, where the first byte of each entry stores the option's code, the second the length `len` of the option data, with the next `len` bytes being the option data itself. EXE automatically generated test packets which cause the code in Figure 15 to overflow the size of the originally allo-

```
1 : // optionptr points to the base of the packet's option buffer
2 : // this buffer is length bytes long
3 : // the below code attempts to return the data associated with the target_code
4 : while(i < length) { // effective behavior of loop
5 :     /* ... */
6 :     /* if we find an option entry with code being searched for... */
7 :     if (optionptr[i + OPT_CODE] == target_code) {        // OPT_CODE = 0
8 :       if (i + 1 + optionptr[i + OPT_LEN] >= length) {  // OPT_LEN = 1
9 :             /* log error */
10:         return NULL;
11:       }
12:       return optionptr + i + 2;
13:     }
14:     /* ... */
15: }
```

Fig. 15. Snippet of code from the `get_option` function in udhcpd.

```
[^[\0^\0]\*-?]{\0        [\-\`[\0^\0]\`]{\0            (?#)\?[[[\0\0][\0^\0]]\0
[\*-\`[\0^\0]\`-?]\0     [\*-\`[\0^\0]\`-?]\0          [\-\`[\0^\0]\`-]\0
(?#)\?[[[\0\0]\-]{\0     (?#)\?[[[\0\0]\-]\0           (?#)\?[[[\0\0]\[]\0
(?#)\?[:[[\0\0]\-]\0     (?#)\?[[[\0\0]\-]\0           (?#)\?[[[\0\0]\]\0
[\*-\`[\0^\0]\`-?]{\0    (?#)\?[[[\0\0][\0^\0]-]\0     (?#)\?[[[\0\0][\0^\0]\]\0
```

Fig. 16. EXE-generated regular expression patterns that cause out-of-bounds writes (leading to aborts in `glibc` on free) when passed as the first argument to `pcre_compile`.

cated packet in three different places. Conceptually, these three overflows stem from two errors. The first is that the loop invariant of `i < length` does not guarantee that `i + OPT_LEN = i + 1` will be in-bounds, hence such a bounds check should be included at the beginning of the conditional statement on line 7. More importantly, however, consider the case in which `optionptr[i + OPT_LEN] = 0` and `i = length - 2`. The conditional on line 8 will evaluate to false, but the function will return a pointer to the first byte past the end of the buffer. This particular case is indicative of a larger issue with trusting the length provided in the option entry. Even if the returned pointer does not point outside of the allocated buffer, the client of this function is expecting to receive a pointer to between one and four bytes of memory, depending on the code. Hence a valid returned pointer can still translate into an overflowing read after the function has returned. This is exactly why we found out-of-bounds reads: the caller of `get_option` memcpy-ed four bytes starting at the first byte past the end of the buffer. A potential fix might involve looking up the expected length of the option data given its code and comparing it to the one provided in the packet, e.g. enforcing the requirement that codes corresponding to IP address options always be four bytes long. Note that similar bounds issues were found in other parts of the `get_option` function.

6.3 Perl Compatible Regular Expressions

The `pcre` library [PCRE] is used by several popular open-source projects, including Apache, PHP, and Postfix. For speed, `pcre` provides a routine `pcre_compile`, which compiles a pattern string into a regular expression for later use. This routine has been the target of security advisories in the past [PCRE - CERT 2005].

We checked this routine by marking a null-terminated pattern string as symbolic and then passing it to `pcre_compile`. EXE quickly found a class of issues with this routine in a recent version of `pcre` (6.6). The function iterates over the provided pattern twice, first to do basic error checking and to estimate how much memory to allocate for the compiled pattern, and second to do actual compilation. The bugs found included overflowing reads in the `check_posix_syntax` helper function (pcre_compile.c:1361-1363), called during the first pass, as well as more dangerous overflowing reads and writes in the `compile_regex` and `compile_branch` helpers (illegal writes on pcre_compile.c lines 3400-3401 and 3515-3616), which are called during the compilation pass. While the first problem may appear to be an innocent read past the end of the buffer, it allows illegal expressions to enter the second pass, causing more serious issues. The substring "`[\0^\0]`" is especially dangerous because strings which end with this sequence will cause `pcre` to skip over both null characters and continue parsing unallocated or uninitialized memory. Figure 16 show a representative sample of EXE-generated patterns that trigger overflows in `pcre`, which in turn cause `glibc` aborts.

Let us discuss in greater detail the issue in which PCRE reads past the end of the pattern buffer. Consider the code snippet shown in Figure 17, in which `ptr` points to the indicated sequence of characters. Since the character at `ptr` is a `[`, the `check_posix_syntax` function is called. Inside this function, `ptr` is incremented and `terminator` is set to the current character, the null character. Without checking whether it has reached the end of a string, the function again increments the pointer. Finding a `^` there, it increments `ptr` yet again. Hence, `ptr` now points to the second null character in the pattern. The second conditional in the function evaluates to true, as the current character is equal to the terminator (both are `\0`) and the next character is a `]`. The new value of `ptr`, which has been incremented over one null character and which now points to a second, is then written back to the caller. Since `check_posix_syntax` now returns true, the original conditional evaluates to true. Therefore, `ptr` is incremented past the second null character and parsing continues.

In most cases, the error checking in the first pass later rejects this regular expression and hence the compilation pass never begins: EXE found many such test cases in which the extent of the damage was only a read off the end of the pattern buffer. However, EXE also found several test cases (shown in Figure 16) in which the characters following the string termination character were such that the pattern was not flagged as invalid in the first pass. In the compilation pass, these patterns then triggered several writes past the end of the buffer allocated to store the compiled regular expression. This caused sufficient heap corruption to cause glibc to abort when the buffer was later freed. PCRE reports errors like "PCRE compilation failed at offset 13: internal error: code overflow," but does not prevent the buffer overflow from occurring (and glibc from aborting).

The author of the library fixed the bug soon after being notified, and so the latest version of `pcre` as of this writing (7.0) does not exhibit this problem.

```
// ptr points to current location in the string being parsed
// consider: ptr == "[\0^\0]...";
if (*ptr == '[' && check_posix_syntax(ptr, &ptr, &compile_block)) // evaluates to true
{
  ptr++;       // ptr now points to ], and parsing continues
  /* ... */
}

static BOOL
check_posix_syntax(const uschar *ptr, const uschar **endptr, compile_data *cd)
{
  int terminator = *(++ptr);                // terminator set to \0
  if (*(++ptr) == '^') ptr++;               // ptr now points to second \0
  /*...*/
  if (*ptr == terminator && ptr[1] == ']') { // evaluates to true
    *endptr = ptr;                          // ptr local in caller now points to second \0
    return TRUE;
  }
  return FALSE;
}
```

Fig. 17. Snippet of code from PCRE.

6.4 Cross-checking applications with EXE

One interesting application of EXE is the cross-checking of a function and its supposed inverse, as well as the cross-checking of several implementations of the same function.

Given two routines `f` and $\texttt{f}^{-1}$, intended to be inverses of each other, we can check whether this is the case by making their inputs symbolic and writing an assert statement of the form $\texttt{assert(f(f}^{-1}\texttt{(x)) == x)}$. As mentioned in Section 2, when EXE hits an `assert` it will systematically search the set of constraints to try to violate the condition asserted (as with any conditional). An `assert` passes only if EXE could not find any input that would violate it. This means that if no errors occurred in EXE, and STP solved all gathered constraints, the two routines are proven to be inverses.

For example, networking code uses the functions `ntohl` and `htonl` to byte-swap 32-bit values between "host" and "network" order. As Figure 18 shows, using EXE to check that a given implementation does this correctly for all inputs is trivial. Note that if the system terminates, then `ntohl` is proven 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 specification.

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 `assert(f(x) == f'(x))`. Routines with identical functionality but different implementations appear commonly in different implementations of core libraries. EXE can be used to cross-check these against each other to ruth-

```
#include <assert.h>
#include <netinet/in.h>

void main(void) {
   int x;
   make_symbolic(x);
   assert(htonl(ntohl(x)) == x);
}
```

Fig. 18. Cross-checking of the form `f`$^{-1}$`(f(x)) = x` that verifies `ntohl` and `htonl` correctly invert each other for all 32-bit inputs.

lessly search for inputs that lead to incompatible outputs. If it cannot find any (and it terminates), then EXE has verified that no incompatibilities exist.

In a previous paper [Cadar and Engler 2005], we used EGT, a primitive version of EXE, to cross-check three different implementations of printf. All implementations (intentionally) implemented only a subset of the ANSI C99 standard (e.g., one version was written for embedded devices). Our cross-checking methodology was to mark the format string specifier as symbolic, generate test cases for each implementation, and then cross-check the concrete test cases against `glibc`'s printf. We found a total of 579 inputs that produced different behavior. As a single example, one implementation of printf incorrectly handled the "`'`" specifier which should comma-separate integer digits into groups of three. The exact test case was:

```
printf("%'d", -155209728);
// correct:  -155,209,728
// observed: -15,5209,728
```

6.5 Generating disks of death

We previously used EXE to generate disk images for three file systems (`ext2`, `ext3`, and `JFS`) that when mounted would crash or compromise the Linux kernel [Yang et al. 2006]. At a high level, the checking worked as follows. We wrote a special device driver that returned symbolic blocks to its callers. We then compiled Linux using EXE and ran it as a user-level process (so `fork` would work) and invoked the `mount` system call, which caused the file system to read symbolic blocks, thereby driving checking.

We found bugs in all three file systems, demonstrating that EXE can handle complex systems code. Further, these errors would almost certainly be beyond the reach of random testing. For example, the Linux `ext2` "read super block" routine has over forty if-statements to check 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 error checking, much less triggering the execution of "real code" that performs actual file system operations.

7. RELATED WORK

A shorter version of this paper appeared in the Proceedings of the 13th ACM Conference on Computer and Communications Security, October 30 - November 3, 2006 [Cadar et al. 2006]. We described an initial, primitive version of EXE (then called EGT) in an invited workshop paper [Cadar and Engler 2005]. EGT did not support reads or writes of symbolic pointer expressions, symbolic arrays, bit-fields, casting, sign-extension, arithmetic overflow, and our symbolic checks. We also gave an overview of EXE in the file system checking paper [Yang et al. 2006] discussed in Section 6.5. That paper took EXE as a given and used it to find bugs. In contrast, both STP and EXE are contributions of this paper (and its preliminary version [Cadar et al. 2006]), which we describe in more detail as well as focus on a broader set of applications.

Simultaneously with our initial work [Cadar and Engler 2005], DART [Godefroid et al. 2005] also generated test cases from symbolic inputs. DART runs the tested unit code on random input and symbolically gathers constraints at decision points that use input values. Then, DART negates one of these symbolic constraints to generate the next test case. DART only handles integer constraints and devolves to random testing when pointer constraints are used, with the usual problems of missed paths.

The CUTE project [Sen et al. 2005] extends DART by tracking symbolic pointer constraints of the form: `p = NULL`, `p` $\neq$ `NULL`, `p = q`, or `p` $\neq$ `q`. In addition, CUTE tracks constraints formed by reading or writing symbolic memory at constant offsets (such as a field dereference `p`→`field`), but unlike EXE it cannot handle 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 find the case when `i` equals `j`, which would have driven the code down both paths. In contrast to both DART and CUTE, EXE has completely accurate constraints on memory, and thus can (potentially) check code much more thoroughly.

CBMC is a bounded model checker for ANSI-C programs [Clarke and Kroening 2003] 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, as well as reads and writes of symbolic memory. However, it has several serious limitations. First, it has a strongly-typed view of memory, which prevents it from checking code that accesses memory through pointers of different types. 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 system calls or even calling code for which there is no source). Both of these limits seem to prevent CBMC from checking the applications in this paper. 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.

Larson and Austin [Larson and Austin 2003] 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 null-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. Further, they cannot generate inputs to cause paths to be executed; users must provide test cases and they can only check paths covered by these test cases.

Static checking and static input generation. There has been much recent work on static bug finding, including better type systems [DeLine and Fähndrich 2001; Foster et al. 2002; Flanagan and Freund 2000], static analysis tools [Foster et al. 2002; Ball and Rajamani 2001; Coverity ; Das et al. 2002; Flanagan et al. 2002; Bush et al. 2000; Wagner et al. 2000], and statically solving constraints to generate inputs that would cause execution to reach a specific program point or path [Boyer et al. 1975; Gotlieb et al. 1998; Ball 2004; Ball et al. 2001; Brumley et al. 2006]. The insides of these tools look dramatically different from EXE. An exception is Saturn [Xie and Aiken], which expresses program properties as boolean constraints and models pointers and heap data down to the bit level. Dynamic analysis requires running code, static analysis does not. Thus, static tools often take less work to apply (just compile the source and skip what cannot be handled), can check all paths (rather than only executed ones), and can find bugs in code it cannot run (such as operating systems code). However, because EXE runs code, it can check much deeper properties, such as complex expressions in assertions, or properties that depend on accurate value information (the exact value of an index or size of an object), pointers, and heap layout, among many others. Further, unlike static analysis, EXE has no false positives. However, we view the two approaches as complementary: there is no reason not to use lightweight static techniques and then use EXE.

Software Model Checking. Model checkers have been used to find bugs in both the design and the implementation of software [Holzmann 1997; 2001; Brat et al. 2000; Corbett et al. 2000; Ball and Rajamani 2001; Godefroid 1997; Yang et al. 2004]. These approaches often require a lot of manual effort to build test harnesses. However, to some degree, the approaches are complementary to EXE: the tests EXE generates could be used to drive the model checked code, similar to the approach embraced by the Java PathFinder (JPF) project [Khurshid et al. 2003]. 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 dynamic input generation work seem to focus on generating an input to follow a specific path, motivated by the problem of answering programmer queries as to whether control can reach a specific statement or not [Ferguson and Korel 1996; Gupta et al. 1998]. EXE instead focuses on bug finding, in particular the problems of exhausting all input-controlled paths and universal checking, neither addressed by prior work.

8. CONCLUSION

We have presented EXE, which uses robust, bit-level accurate symbolic execution to find deep errors in code and automatically generate inputs that will hit these errors. A key aspect of EXE is its modeling of memory and its co-designed, fast constraint solver STP. We have applied EXE to a variety of real, tested programs where it was powerful enough to uncover subtle and surprising bugs.

9. ACKNOWLEDGMENTS

We would like to thank Paul Twohey for his work on the regression suite, Martin Casado for providing us `tcpdump` in an easy to check form, and Suhabe Bugrara, Ted Kremenek, Darko Marinov, Adam Oliner, Ben Pfaff, and Paul Twohey for their valuable comments.

This research was supported by National Science Foundation (NSF) CAREER award CNS-0238570-001, Department of Homeland Security (DHS) grant FA8750-05-2-0142, NSF grant CCR-0121403, and a Junglee Corporation Stanford Graduate Fellowship.

REFERENCES

Ball, T. 2004. A theory of predicate-complete test coverage and generation. In *Proceedings of the Third International Symposium on Formal Methods for Components and Objects*.

Ball, T. and Jones, R. B., Eds. 2006. *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*. Lecture Notes in Computer Science, vol. 4144. Springer.

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

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

Barrett, C. and Berezin, S. 2004. CVC Lite: A new implementation of the cooperating validity checker. In *CAV*, R. Alur and D. A. Peled, Eds. Lecture Notes in Computer Science. Springer.

Barrett, C., Berezin, S., Shikanian, I., Chechik, M., Gurfinkel, A., and Dill, D. L. 2004. A practical approach to partial functions in CVC Lite. In *PDPAR'04 Workshop, Cork, Ireland*.

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

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

Brumley, D., Newsome, J., Song, D., Wang, H., and Jha, S. 2006. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*.

Bryant, R. E., Lahiri, S. K., and Seshia, S. A. 2002. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Proc. Computer-Aided Verification (CAV)*, E. Brinksma and K. G. Larsen, Eds. Springer-Verlaag, 78–92.

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

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

Cadar, C., Ganesh, V., Pawlowski, P., Dill, D., and Engler, D. 2006. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security.*

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

Cook, B., Kroening, D., and Sharygina, N. 2005. Cogent: Accurate theorem proving for program verification. In *Proceedings of CAV 2005*, K. Etessami and S. K. Rajamani, Eds. Lecture Notes in Computer Science, vol. 3576. Springer Verlag, 296–300.

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

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. 2001. *Introduction to Algorithms.* The MIT Electrical Engineering and Computer Science Series. MIT Press/McGraw Hill.

Coverity. SWAT: the Coverity software analysis toolset. `http://coverity.com`.

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

DeLine, R. and Fähndrich, M. 2001. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation.*

Een, N. and Sorensson, N. 2003. An extensible SAT-solver. In *Proc. of the Sixth International Conference on Theory and Applications of Satisfiability Testing.* 78–92.

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

Flanagan, C. and Freund, S. N. 2000. Type-based race detection for Java. In *SIGPLAN Conference on Programming Language Design and Implementation.* 219–232.

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

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

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

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

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

Gupta, N., Mathur, A. P., and Soffa, M. L. 1998. 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.* ACM Press, 231–244.

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

Holzmann, G. J. 1997. The model checker SPIN. *Software Engineering 23,* 5, 279–295.

Holzmann, G. J. 2001. From code to models. In Proc. 2nd Int. Conf. on Applications of Concurrency to System Design. *Proc. 2nd Int. Conf. on Applications of Concurrency to System Design*, 3–10.

Khurshid, S., Pasareanu, C. S., and Visser, W. 2003. 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.*

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

Miller, B. P., Fredriksen, L., and So, B. 1990. An empirical study of the reliability of UNIX utilities. *Communications of the Association for Computing Machinery 33,* 12, 32–44.

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

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

Nethercote, N. and Seward, J. 2003. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science 89,* 2.

PCRE. PCRE - Perl Compatible Regular Expressions. `http://www.pcre.org/`.

PCRE - CERT 2005. PCRE Regular Expression Heap Overflow. US-CERT Cyber Security Bulletin SB05-334. `http://www.us-cert.gov/cas/bulletins/SB05-334.html#pcre`.

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

Sen, K., Marinov, D., and Agha, G. 2005. 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).*

SMTLIB 2006. SMTLIB competition. `http://www.csl.sri.com/users/demoura/smt-comp`.

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

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

Xie, Y. and Aiken, A. 2005. Saturn: A SAT-based tool for bug detection. In *CAV*, K. Etessami and S. K. Rajamani, Eds. Lecture Notes in Computer Science, vol. 3576. Springer, 139–143.

Yang, J., Sar, C., Twohey, P., Cadar, C., and Engler, D. 2006. Automatically generating malicious disks using symbolic execution. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy.*

Yang, J., Twohey, P., Engler, D., and Musuvathi, M. 2004. Using model checking to find serious file system errors. In *Symposium on Operating Systems Design and Implementation.*

...