Bullseye: A System for Targeted Test Generation

Brandon Streiff
bstreiff@cs.utexas.edu

02 May 2008

Undergraduate Honors Thesis
Supervisor: Dr. Calvin Lin

Abstract

This paper describes the architecture and implementation of the Bullseye automatic test generation system. Bullseye differs from existing systems in that it is *targeted*: by using program analysis, it is able to guide testing to areas that influence or are influenced by areas deemed to be 'interesting': that is, locations that are more important to test. By not treating all paths equally and instead stressing these areas of influence, Bullseye guides branch selection to test only these areas. As a consequence, Bullseye deals with a manageable subset of program paths that still encapsulate relevant program behavior.

1 Introduction

Test generation is the development of inputs to a program or function to test its behavior. Manually developing tests is hard; it is difficult and time-consuming for humans to develop tests, and it is difficult to guarantee that human-developed tests exhibit full coverage without being redundant. Several automated techniques for generating tests have been proposed, such as *execution-driven testing*, in which test inputs are used in conjunction with concrete executions on those inputs to exhaustively test all possible program paths.

Two systems that perform execution-driven testing are DART [6] and CUTE [8]. Both of these systems instrument a program to collect branch conditions, then use random initial inputs to run the instrumented program. After execution, both look at all path conditions generated by the branches taken, invert the last one, and then attempt to repeat the process by generating a new input satisfying these new conditions. Essentially, this performs a depth-first search across the entire program, looking for possible bugs.

A major obstacle to execution-driven testing of entire programs is the path explosion program: The number of possible paths through a program grows exponentially. Each branch encountered may go in one of two ways; either the branch is taken, or the branch is not taken. Execution-driven testing systems such as CUTE are geared toward unit testing. Units tend to be small, with limited branch depth. Existing test coverage systems must bound the paths they check to a fixed number of branches. While this suffices for unit-testing, where paths are few and small, paths through entire programs are greater in number and in depth. A depth-first search across all paths does not scale.

Additionally, most of these paths result in correct behavior. The competent programmer hypothesis states that programmers write code that is almost perfect, and that those problems that do occur are syntactically small; these are logic errors that can be corrected with a few keystrokes, for instance, substitution of `>` for `>=`. It is reasonable to assume that a program behaves correctly most of the time, and that only a few paths through a program contain errors. It is these paths that we consider interesting, and if we can guide testing to these error-inducing paths, then we can test program behavior at a greater path depth because we do not need to test each and every path through the program.

In this paper, we introduce Bullseye, a targeted testing system that uses program analysis to guide branch searching to areas that influence or areas influenced by locations that we deem 'interesting'; that is, locations that are more important to test. We can define this interest in many potential ways, one of which we discuss later. By not treating all paths equally and instead stressing these areas of influence, Bullseye guides branch selection to test only these areas. Consequently, Bullseye is able to explore relevant paths earlier than if all branches were equal. In this way, we are able to detect bugs more quickly than a traditional execution-driven test system.

One way to define interest of program locations would be in program changes, for instance, between two successive versions of a program in a change management system. Given two versions of a program, we can identify the locations that have changed between them. By marking these as being interesting, Bullseye will prioritize its searching to changed code and areas affected by those changes. Because there is no need to retest areas not affected by those changes, Bullseye saves time by not checking unchanged paths.

In this thesis, we describe an implementation of the Bullseye system. We resolve and highlight some details in the implementation of prior work, such as CUTE, that do not appear in literature. Finally, we empirically evaluate this implementation of the Bullseye system.

This paper is structured as follows. In section 2, we give an overview of how our system expands on CUTE and the implementation details that were neglected in the description. In section 3, we give a high-level description of the overall architecture of our system. In section 4, we discuss a number of experiments that were performed to evaluate our system. In section 5, we draw conclusions about our system.

2 Related Work

Bullseye is inspired by the CUTE automatic test generation system [8]. CUTE instruments a program with symbolic tracking information, then runs the program on a random input. During execution, CUTE tracks the path taken through the program and the branch conditions that lead it through that path. At conclusion of the execution CUTE negates a condition at the end of the path. If negating this condition causes the modified list to be unsatisfiable, CUTE simply chooses a different branch earlier in the branch history until it finds one that can be satisfied. CUTE then takes these satisfiable conditions and uses them to generate a new input, then executes the program using it. This cycle causes CUTE to enumerate all possible paths through a program in a depth-first manner, using a bound on the number of branches to resolve issues with nontermination as well as to restrict the number of terms and variables for which to determine satisfiability.

The authors of CUTE describe many of its inner workings in implementation-level detail, which has made developing those components rather straightforward. However, in the discussion of their implementation, the authors omit a number of details that are required to fully implement CUTE. In CUTE, all that is mentioned regarding function call handling is a brief sentence on the use of a symbolic stack, without further details. In Bullseye, the data structure that maintains the symbolic state for variables also functions as a stack. CUTE also assumes the presence of sufficient runtime type information to determine the size, in bytes, of a variable; whether it is a pointer or a primative type; and, if a pointer, what type it points to. However, in the C programming language, runtime type information is generally not available at runtime. Bullseye solves this by constructing a data structure at compile time to represent all types seen by the compiler, which is then accessed during execution. Furthermore, CUTE assumes that a pointer to an object will only ever point to NULL or one such object. CUTE, as presented, does not have a mechanism to handle pointers that point to an array of objects; that is, arrays that have been dynamically allocated. (Statically allocated arrays are simply unrolled into a sequence of variables.) CUTE is designed primarily for data-structure testing and thus this restriction may not be disadvantageous for that purpose. On the other hand, we would like to be able to test programs for which a bug-causing input may include a dynamically-allocated C string, and so Bullseye must be able to handle dynamic arrays.

3 Overall system architecture

The Bullseye system consists of two major parts, an instrumenting compiler and a runtime library. Figure 1 shows the overall architecture of the system. The compiler instruments a program by adding code to perform runtime tracking, so that variables and constraints can be evaluated symbolically. The compiler generates instrumented source files, as well as an additional driver source file containing type data and entry code for the program. The instrumented source and the driver source file are compiled with an ordinary C compiler and are linked with the Bullseye runtime library.

When the resulting executable is run, code in the runtime library runs the program code in a loop. First, an input is generated and assigned into memory. The program is then run. Branch conditions encountered during execution are stored. At the program's end, a branch condition is selected to be negated to generate a new set of branch constraints. A constraint solver takes these constraints and uses them to generate a new input. Bullseye continues this generate-execute-negate-solve loop until the chosen branch selection algorithm indicates that there are no branches left to negate; this indicates that all paths have been explored.

To perform runtime tracking, we must add instrumentation to existing code. We do this by using a C-to-C source compiler to read in source files and insert instrumentation hooks inline with original source code. These hooks will, at

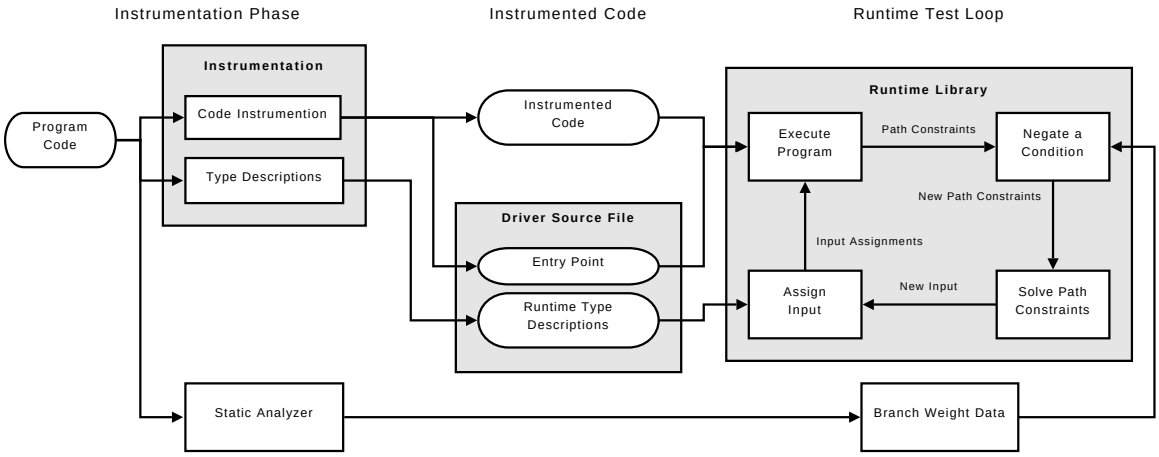

Figure 1: The architecture of Bullseye

runtime, invoke portions of the Bullseye runtime library. The compiler also constructs a new source file that contains a serialization of all types used in the original program, as well as a driver for the entry function.

The runtime library is implemented as a modular collection of C++ classes, with linkage code callable from instrumentation-added hooks written in C. The major subsystems of the runtime library are the logical input map, the runtime type system, the evaluation engine, the symbolic expression evaluator, and the symbolic constraint solver. The evaluation engine provides the main execution loop: it uses the input map and runtime type system in order to instantiate input variables, the symbolic expression evaluator to compute linear and pointer constraints for inputs, and the constraint solver to generate new inputs.

3.1 Compiler Instrumentation

To track values through the program at runtime, Bullseye uses an instrumenting compiler built on top of the C-Breeze compiler infrastructure [5]. C-Breeze provides a C (ANSI C89) parser front-end and a collection of C++ classes that represent the parsed syntax tree, as well as phases that allow for code transformation. Using the provided code transformation interfaces, the compiler inserts hooks to the Bullseye runtime library and outputs the resulting enhanced source files. These source files can then be compiled with a user's ordinary C compiler, such as `gcc`, and linked against the Bullseye runtime library.

The Bullseye instrumenting compiler performs AST transformations in the following order:

1. The C-Breeze dismantler is invoked to transform the source code to an intermediate representation.
2. Array and structure references are rewritten as pointer-plus-offset statements.
3. Symbolic instrumentation is added for evaluation expressions, branches, and procedure calls.
4. Variable declarations are added for each variable.
5. A source file containing additional C source code for the driver is output.

3.1.1 Dismantler

The C-Breeze compiler contains routines to transform a parsed C source file into a medium-level intermediate representation [5]. This representation consists of simplified assignment statements, basic control flow operations, and procedure calls, while still retaining higher-level constructs of C, such as array and structure references. This allows subsequent transformations to the AST to be considerably simpler, as there are fewer types of expressions that need to be handled.

3.1.2 Operand Changer

The `operandChanger` class converts the array and structure references retained by the dismantler into an even simpler pointer-plus-offset notation. Because C-Breeze operates at a high level and does not know what the size or

Source	After transformation
`x = y`	`cute_exec_symbolic_assign((void *) & x, "y");` `x = y;`
`*x = y`	`cute_exec_symbolic_assign((void *) x, "y");` `*x = y;`
`x = *y`	`cute_exec_symbolic_deref((void *) & x, "y");` `x = *y;`
`*x = *y`	`cute_exec_symbolic_deref((void *) x, "y");` `*x = *y;`
`x = y op z`	`cute_exec_symbolic_eval((void *) & x, "y", OPERATOR_op, "z");` `x = y op z;`
`*x = y op z`	`cute_exec_symbolic_eval((void *) x, "y", OPERATOR_op, "z");` `*x = y op z;`

`if (x op y) goto l;`	`cute_comparison_result = x op y;`
	`cute_evaluate_predicate("x", OPERATOR_op, "y",`
	`  cute_comparison_result, "branchIdentifer");`
	`if (cute_comparison_result != 0) goto l;`

Figure 2: Code instrumentation additions

offset of types will be, we employ the use of the C macro `offsetof(type, member)` which is used to determine the offset at compile time.

As an example, the expression `(object_t) x = r.a.b[13]` would be converted to pointer-plus-offset notation as `(void *) tmp = offsetof(object_t, a.b[13]); tmp = &r + tmp; x = *tmp;`

3.1.3 Symbolic Instrumentation

Symbolic instrumentation occurs inside the `cuteChanger` class. Figure 2 shows the hooks that are added for assignment statements and branches. The expressions in double quotes ("y") represent symbolic variables; at runtime, they are represented by strings. Branch identifiers are unique identifiers for each branch. Because the dismantler transforms branches with complex conditional expressions into a series of branches with simplified conditions, simply using file and line numbers is insufficient. Instead, we label each branch node in the AST with a unique identifier.

3.1.4 Variable Declarations

In order to use symbolic variables in an instrumented program, they must be first be declared. The `cuteDeclAdd` class adds variable declaration calls that initialize a symbolic variable with its address and its type. To handle local and parameter variables, we maintain a stack of variable contexts. For local variables, `cute_declare_variable` calls are added to create variables in the current context. For formal parameters `cute_declare_variable_from_stack` calls are added to simulate C's pass-by-value semantics; these both create a variable in the current context and copy the value from the associated parameter from the stack.

3.1.5 Driver

The instrumentation phase also results in an additional source file, `cute_driver.c`, containing type definitions and a stub function, `cute_program()`, which is used to initialize global variables before we run the entry function of the program we wish to test.

Type definitions are output in two formats. First, type declarations are printed in the driver as C source code. These are essentially copies of all types declared in the instrumented program. These types have been simplified somewhat, rewriting all occurrences of nested structures into multiple structure definitions. Secondly, the instrumentation outputs a function `cute_program_types()`, containing in its body a number of calls to the Bullseye runtime library to construct the runtime type graph. These calls then use the `sizeof()` and `offsetof()` operators on the C-language types in the driver.

C-Breeze makes use of several design patterns to traverse AST nodes, such as `Visitor`s, `Walker`s, and `Changer`s. These suffer from lack of fine-grained control over the traversal, thus we utilize a variant called (for

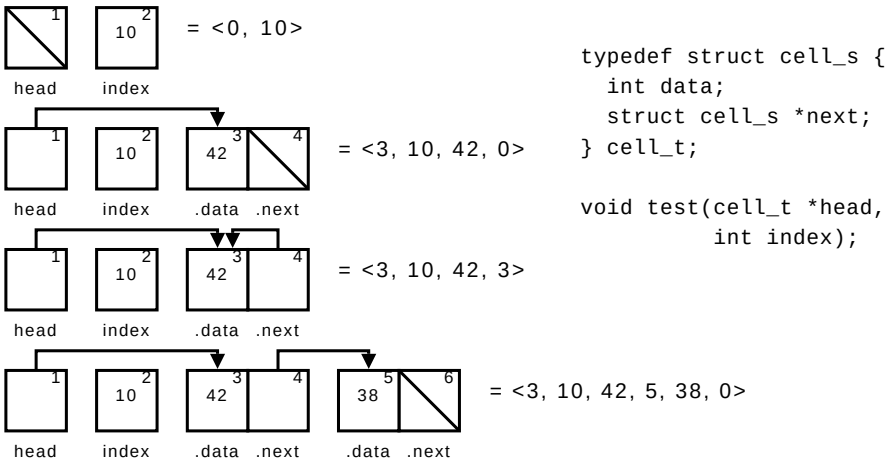

Figure 3: Example Memory Graphs and Corresponding Input Maps

lack of a better name) a `Dispatcher`. In a `Dispatcher`, however, the `dispatcher()` method invokes itself recursively on children of an AST node, as opposed to invoking the `visit()` method of a AST node. This was done to allow traversal to occur in the order necessary for the appropriate output format, while still permitting enough abstraction to allow for composition of `Dispatcher`s. The `TypeDispatcher` class is used for C-language type definitions, and the `CTypeDispatcher` class is used to create the function definitions to declare the runtime type graph. Each of these are composed with an `OrderedDispatcher`, which enforces that types are output in the correct order; for example, all the types used in a `struct` must be declared before the `struct` itself.

3.2 Runtime Library

In order to perform runtime tracking, Bullseye uses a compiler instrumentation phase to add hooks to a source program, resulting in an instrumented version of the program and a driver. These are then linked against the Bullseye runtime library, which contains the implementations for the added hooks. The resulting executable thus contains both the original program as well as the instrumentation with which to test it, running side-by-side. The execution of this resultant program follows the procedure below:

1. The entry point parses command-line options and instantiates the evaluation engine.
2. The evaluation engine initializes the runtime type system with all of the types defined in the program. These types were output by the instrumentation (see section 3.1.5).
3. The evaluation engine generates a blank input for the program.
4. The evaluation engine starts the execution loop:
    a. The instrumented program is run. Hooks in the program invoke library code to track symbolic execution and branches.
    b. After execution concludes, the evaluation engine chooses a branch to negate. If negating this condition results in an unsatisfiable input, the evaluation engine chooses a different branch until it gets one that is satisfiable.
    c. This modified list of branch conditions is passed to the constraint solver in order to generate a new input.
    d. The loop is repeated until the evaluation engine has explored all relevant paths.

The library contains the C entry point (`main()`), from which options are parsed, weight data is loaded from a file (if specified), and an instance of the evaluation engine (specifically, the `CuteEngine` class) is created and run.

3.2.1 Logical Input Map

Bullseye represents inputs to a program using a logical memory map, which is conceptually a serialized memory structure that associates logical addresses, represented using positive integer values, to a type and a value, which is

either a primitive value or another logical address. Logical addresses are used, rather than physical in-memory addresses, because physical addresses may change between successive executions of a program. Furthermore, physical addresses are not necessary to represent memory structures; all that is necessary is relationships between pointers and values. Figure 3 provides sample memory graphs and their corresponding input maps for a simple linked-list type.

Initially, all primitive values are assigned random values and all pointer values are assigned to be the NULL address. Whenever Bullseye runs a program under instrumentation, it transforms the logical input map into a concrete input, then runs the program with that input. After the program has executed, Bullseye then modifies the input map, changing entries or extending it as necessary (see Section 3.2.5).

To go from a logical input map to a concrete input for a program, Bullseye applies the following transformation. For primitive types, the value is simply copied from the input map to memory. For pointer types, Bullseye maintains a structure mapping logical addresses to concrete addresses of allocated blocks of memory; if Bullseye has determined that a pointer should point to the object at a particular logical address, this structure is used to determine where that object resides in memory. If Bullseye has determined that a pointer should point to some address that is not NULL (denoted by a special constant), Bullseye allocates a new block of memory and updates the allocation structure.

The logical input map is implemented in the `InputMap` class, which contains two STL maps, one from logical addresses to `InputCell`s representing value/type pairs and one from logical addresses to `MemoryBuffer`s representing handles for allocated blocks of memory.

To initialize a physical address P from a logical address L, Bullseye first checks to see if L has been initialized on a previous run. If not, then the cell at that address in the input map is set to a default value: primitives are set to a random integer, pointers are set to NULL. The cell's value is then copied to the value at P. If L had been initialized, then, if the entry at L in the input map is a primitive value, it will be simply copied to the value at P. If it is a pointer value, then if the pointer points to a logical address for which a memory buffer was allocated in a previous run, P will be initialized to the address of that buffer. Otherwise, a new buffer is created.

Input initialization is also recursive; for instance, when initializing a pointer value, the object that is pointed to should also be initialized so that it can be used by the program. Because we permit cyclic data types, input initialization also needs to perform cycle-detection so that Bullseye does not re-initialize addresses unnecessarily.

3.2.2 Runtime Type System

CUTE's technique, which we have adapted for Bullseye, operates on integer values when performing input initialization and symbolic execution, as opposed to direct bitfield settings such as in EXE [9]. The value of a variable is defined by several properties, such as its location in memory, the number of bits it occupies, and how those bits are to be interpreted. In order to do this effectively, we require the presence of sufficient runtime type information in order to determine the size of a variable in memory, whether it is a primitive or pointer type, and the type it points to if it is a pointer.

However, in the C programming language, this semantic information is typically lost in compilation and thus unavailable at runtime. In Bullseye, we use the instrumentation phase to create an additional source file containing code to generate a runtime-accessible type graph, which is then used by the Bullseye runtime library. We will first discuss this type graph before we discuss the construction via this source file.

Type Graph Representation The type graph is similar to the parsed form of a type expression used by compilers [2]. The type graph has directed edges, with primitive types at terminal nodes, such as `char`s and `int`s, and other types, such as pointers, arrays, `struct`s, and `union`s, at the nonterminal nodes.

Fields in a `struct` or `union` are stored in a list type. Each field also maintains its own offset in bytes, so that this information does not need to be recomputed for every runtime access for a field. Maintaining a list of offsets for each field also addresses the issue of structure padding, which we are only able to determine at compilation.

Type aliases, created with `typedef`, necessitate the creation of a table that maps type names to their corresponding location in the type graph. Such a mapping is demonstrated in Figure 4.

Type Graph Construction One of the initial phases of the C-Breeze compiler, which is used in Bullseye to perform instrumentation, is to parse ANSI C code. Conveniently, C-Breeze parses types into a form that it then uses in subsequent phases. In order to make this information available at runtime, Bullseye outputs a C function, `cute_program_types()`, containing a number of function calls to rebuild the information from C-Breeze's type graph at runtime. One such output is demonstrated in Figure 4.

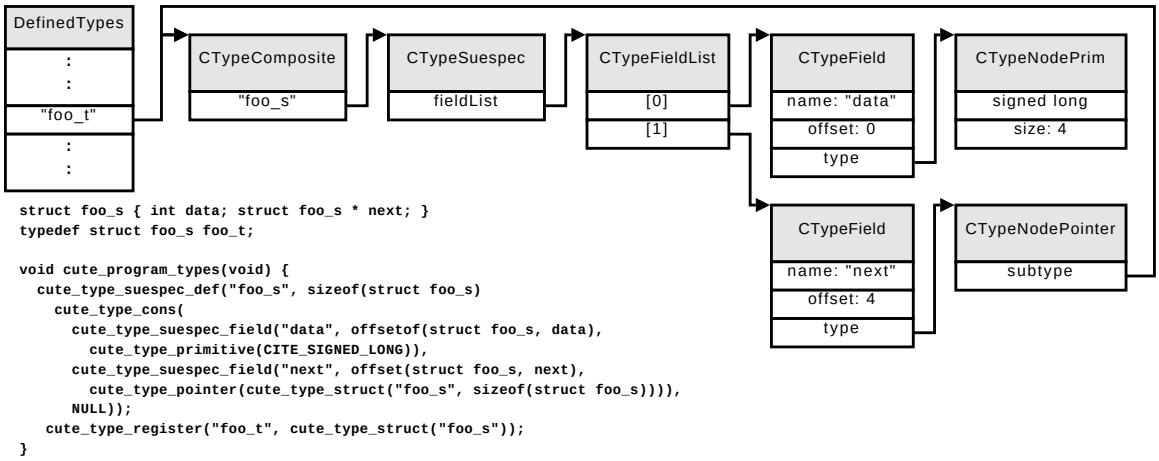

Figure 4: Type graph construction and representation for a simple linked-list type

C-Breeze only maintains machine-independent information; it has no notion of sizes or offsets of types. Therefore, we incorporate calls to the `sizeof()` operator and the `offsetof()` macro into the resultant source code, so that sizes and offsets can be evaluated when the instrumented program is compiled.

Because the type graph is cyclic due to recursive structures, C-Breeze has the notion of *suespecs*, "`struct`, `union`, and `enum` specifications." A suespec contains a list of fields in a composite type. When creating the function declaration to create a `struct` or `union`, we only refer to the name of the suespec; this resolves the cycles in the type graph, allowing us to serialize it in the form of function declarations.

Implementation In the implementation, the runtype type system is defined as a collection of C++ classes with C-linkage hooks. All classes inheriting from `CTypeNode` represent nodes in the type graph.

- Primitive types, such as `char`s and `int`s, are represented with `CTypeNodePrim` objects. Enumerations are silently converted to integers.
- Static arrays are represented with `CTypeNodeArray` objects containing a dimension and a reference to a subtype node.
- Pointers are represented with `CTypeNodePointer` objects containing a reference to a subtype node.
- Composite types, e.g. `struct`s and `union`s, are represented with a `CTypeNodeComposite` that contains a reference to a `CTypeSuespec`. A `CTypeSuespec` contains a list of `CTypeField`s, each with a name, offset, and subtype reference. Because offsets are stored with each field, `struct`s and `union`s are represented identically.

Another class, `CTypeIterator`, is designed to ease iterating across all fields in a type. Given a base type and an initial offset, it traverses a type, exploring all nested fields and incrementing the offset.

3.2.3 Symbolic Execution

Bullseye executes a program both concretely (using the original source code) and symbolically (through added instrumentation. Bullseye maintains symbolic expressions for variables where possible, performing operations side-by-side with operations on concrete memory values.

Bullseye tracks two types of symbolic expressions: *Linear arithmetic expressions* are of the form $a_1x_1 + ... + a_nx_n + c$ where $n \geq 1$, each x_i is a variable, and each a_i and c are integer constants. *Pointer expressions* are either NULL or a symbolic variable. Because not every expression in a C program can be represented in one of these two forms, Bullseye uses the concrete value from memory when it cannot represent an expression symbolically. This can happen when Bullseye attempts to perform a non-linear operation, such as multiplication of two non-constant arithmetic expressions. Because the result of this operation is non-linear, Bullseye replaces one of the expressions with its concrete value to simplify it, giving us an approximate linear solution. Although this method does lead to a

loss of accuracy, we consider that most operations in a program tend to be addition or subtraction and thus expressions remain linear.

In the implementation, there are three `SymbolicExpr` classes to represent symbolic expressions. Only one object of the `SymbolicExpr` base class ever exists, `InvalidExpr`, to represent an expression that cannot be represented symbolically. All other expressions are either of the `ArithmeticExpr` or `PointerExpr` subclasses.

When Bullseye encounters a branch, it builds a constraint from the branch condition. For arithmetic expressions, these constraints take the form $a_1x_1 + ... + a_nx_n + c \bowtie 0$, where $\bowtie$ is one of $<$, $\leq$, $>$, $\geq$, $=$, or $\neq$. When two arithmetic expressions A and B are used to generate a constraint, the resulting constraint is $A - B \bowtie 0$, e.g., an expression compared to 0. Pointer constraints take the form $x_1 = x_2$, $x_1 \neq x_2$, $x_1 = NULL$, or $x_1 \neq NULL$. As an example, given the code on the left, the path constraints on the right might be generated based upon the input at the execution of the program. In the following, "$=$" represents concrete assignments and "$\rightarrow$" represents pointer relations.

```
int foo(int x, int * y) {
   x = x + 1;
   if (x < 10)
      if (y == NULL)
          return x + *y;
   return x;
}
```

- With $x = 0, y \rightarrow NULL$:
  $(1 \cdot x - 9 < 0) \wedge (y \rightarrow NULL)$
- With $x = 0, y \rightarrow \alpha, \alpha = 0$: ($\alpha$ is a new variable)
  $(1 \cdot x - 9 < 0) \wedge (y \nrightarrow NULL)$
- With $x = 10, y \rightarrow \alpha, \alpha = 0$:
  $(1 \cdot x - 9 \geq 0)$

In the implementation, the evaluation engine is defined by the `Engine` interface (`engine.h`) and implemented in the `CuteEngine` class. The evaluation engine provides the main execution loop. The engine's `run()` method sets up signal handlers for determining when the program under test throws an error, by trapping `abort()` calls and segmentation faults. The engine then clears symbolic variable states and path constraints, and executes the program.

As the program executes, it calls instrumentation-added hooks. The implementations for most of these hooks call `Engine` methods. Several of these are then delegated further; for instance, `Engine::initializeInput()` immediately invokes a method in the `Input` class in order to initialize input variables (see section 3.2.1).

The `VariableContexts` class, in `variablecontext.h`, provides a mapping from variable names to concrete addresses. Because this name-address mapping changes with function calls, the `VariableContexts` class also serves as a symbolic stack, which is pushed down and popped on function calls and returns, respectively. Two hooks are used to add variables to the current context: `cute_declare_variable`, for local variables, and `cute_declare_variable_from_stack`, which initializes a parameter with a copy of a symbolic expression from the caller.

There is also a mapping from concrete addresses to symbolic variable states, represented by `VariableState` class. Each `VariableState` consists of a type and a symbolic expression, and also provides helper methods for retrieving the concrete value as a particular address based on its runtime type.

The `Evaluator` class, handles symbolic evaluation, for addition, subtraction, and multiplication. Given a result address, the operator, and two variable names, the `Evaluator` looks up the symbolic expressions for the variables, computes an appropriate symbolic result, and assigns it to the symbolic state for the result address.

3.2.4 Branch Selection

After the program finishes executing, it is the engine's responsibility to choose a new input. To do this, it queries a `BranchSelector` for a potential condition to flip. `BranchSelector`s are created with access to the path history, path constraints, and the list of weights, and provide an iterator-like decision API. The selector gives the index of the 'best' branch to try to negate; the evaluation engine then tries to negate it and use it to generate a new input. If this fails (because the resulting conditions are unsatisfiable), then the selector is queried for the next-best choice, and so on, until either a satisfiable input is reached or there are no more possibilities. (In the latter case, we conclude testing.)

Three basic branch selection strategies are provided and can be specified using the `--selector` command-line option.

- The `CuteBranchSelector` performs CUTE's depth-first-search strategy.
- The `RandomBranchSelector` chooses the branch to negate entirely at random.

- The `ProbabilisticBranchSelector` uses the weighting data as probabilities for whether it should flip the branch or not. To do this, it begins at the deepest branch, then chooses (with weighting) which way the branch should go. If the direction chosen is different from the direction that the branch actually went, then the branch is negated. Otherwise, the selector moves up to the next deepest branch and continues.

After the engine has generated an input that can be satisfied, it uses a constraint solver (see section 3.2.5) to generate concrete values belonging to the class of inputs represented by the path conditions. The input is then loaded into the input map, and the execution loop continues.

3.2.5 Solving Constraints

Any path through a program can be represented by the conjunction of constraints. Because each input to the program is given a symbolic variable, a list of path constraints represents a class of possible variable assignments that will cause execution to follow a particular path. Therefore, by negating particular constraints and choosing a satisfying assignment for the input variables, we generate inputs that are in different classes and thus cause execution to follow different paths.

CUTE creates a symbolic constraint for every branch traversed, appending it to a list of branch conditions that represents a path through the program. When the program has concluded execution, CUTE negates one constraint and solves for new inputs.

Because CUTE tracks both symbolic arithmetic and symbolic pointer expressions, CUTE naturally must solve both arithmetic and pointer constraints. To solve arithmetic constraints, CUTE uses `lp_solve` [4], a library for linear integer programming.

To solve pointer constraints, CUTE looks at all constraints except for the negated one and uses equality constraints to partition variables into equivalence classes. To determine satisfiability, CUTE builds an undirected graph, where vertices are equivalence classes and edges are disequalities (determined from disequality constraints). If the negated constraint is of the form $x_1 = x_2$ and there is no edge between the classes $[x_1]_=$ and $[x_2]_=$, then the new set of constraints is satisfiable. Otherwise, if the negated constraint is of the form $x_1 \neq x_2$, then the new set of constraints is satisfiable if $[x_1]_=$ and $[x_2]_=$ are different equivalence classes.

Once satisfiability is determined, the input map is updated. If the negated constraint is of the form $x \neq NULL$, all variables in $[x]_=$ in the input map are set to a constant denoting a non-NULL value. Similarly, if the negated constraint is of the form $x \neq y$, all variables in $[y]_=$ in the input map are set to non-NULL. This non-NULL constant is used during input initialization to indicate that a new object should be created. Otherwise, if the negated constraint is of the form $x = NULL$, all variables in $[x]_=$ in the input map are set to NULL. If the negated constraint is $x = y$, all variables in $[y]_=$ in the input map are set to the current value of x in the input map.

As an example, assume we have the pointer constraints $x_1 = x_2$, $x_2 = x_3$, $x_3 \neq x_4$, and $x_2 = y$. Let the last of the four be the one we have chosen to negate. First, we look at all of the constraints excepting the last to discover equivalency classes $[x_1]_= = \{x_1, x_2, x_3\}$ and $[x_1]_= \neq [x_4]_=$. Because the negated constraint is $x_2 \neq y$, we set all variables in the equivalency class $[y]_=$ (consisting of only y itself) to non-NULL.

Bullseye, much like CUTE, also tracks arithmetic and symbolic expressions and constraints. However, Bullseye solves for these constraints using the CVC3 Satisfiability Modulo Theories (SMT) solver [3]. A SMT solver improves on a traditional boolean satisfiability (SAT) solver by utilizing binary-valued predicates rather than simple boolean variables. Predicates can be solved using a number of more expressive theories; for instance, a SMT solver can solve a predicate such as $x + 2 \leq y$ using the theory of linear arithmetic, whereas a SAT solver would have to encode the predicate as a series of boolean logic operations [7].

Bullseye uses the theory of linear arithmetic to solve arithmetic expressions, and the theory of equality to solve pointer constraints. CVC3 is able to understand more theories, such as theories of arrays, list structures, and bitvectors, although Bullseye does not utilize these at the present time. Future work could include using these theories to extend Bullseye's symbolic tracking beyond CUTE's linear arithmetic and pointers, for instance, to add support for bitwise operations commonly seen in hand-optimized C programs and for floating-point expressions.

In the implementation, there are three classes for constraints: `ArithmeticConstraint`s represent relational or equality constraints between `ArithmeticExpr`s, `PointerConstraint`s represent equality constraints between `PointerExpr`s (or between a `PointerExpr` and `NULL` or `non-NULL`), and `ConcreteConstraint`s simply serve as a wrapper for a simple true or false value for which we are unable to compute symbolically. All three are implementations of the `Constraint` interface.

The `ConstraintSolver` class provides an interface for the constraint solver in Bullseye. It is implemented by the `CVC3Solver` class; this implementation uses CVC3's `ValidityChecker` API. Variables in the constraint

GLib Binary Tree		
Max Depth	Time	Paths Explored
10	1.086 s	50
11	1.873 s	73
12	3.461 s	102
13	6.241 s	142
14	12.108 s	198
15	23.733 s	288

Singly-Linked List		
Max Depth	Time	Paths Explored
10	0.052 s	12
20	0.124 s	22
30	0.231 s	32
50	0.557 s	52
100	2.083 s	102
200	8.992 s	202

Figure 5: Performance Results

solver are represented as integers using CVC3's `INT` type, and methods exist to convert Bullseye `Constraint`s into CVC3's constraint types.

4 Experiments

One way we could evaluate the performance of our system is through *mutant-kill testing*. Mutants are variations in a program that indicate a bug or some other deviation in behavior. Our performance metric in this instance is how long it takes for our system to detect the variations between a source program and a mutant. One experiment to perform, therefore, would be to test both an original program and a mutant using Bullseye. Bullseye should be able to determine the point of change as being interesting, and thus should be able to detect the mutation at runtime faster than a traditional bounded depth-first approach.

As the time of this writing, we are still evaluating our system. Currently, we are able to use Bullseye to test the balanced binary tree structures provided in the GLib data structures library [1]. In GLib, the `g_tree_node_check` function performs invariant checking. By convention, a function such as this is called a `repOk` function. When Bullseye tests this function, it produces inputs that correspond to valid binary trees.

When operating with CUTE-style branch prediction (that is, a bounded depth-first strategy), Bullseye yields similar performance results to CUTE's results on similar data structures in the SGLIB data structures library [8]. The results of using Bullseye to test the GLib binary tree structure and a singly-linked list data structure at various maximum branch depths are summarized in Figure 5.

5 Conclusions

At this time, the basic system for Bullseye is complete and can be used for testing programs in a depth-first manner. The system is also able to employ weighting data to guide a search, though it is not yet able to generate this weighting data itself. Work in adding the static analysis portion to generate weighting data is still ongoing.

There are still numerous areas of future work where Bullseye could see improvement. Currently, Bullseye only performs symbolic evaluation on linear arithmetic expressions and pointers, yet the SMT solver that it employs is capable of many more types of expressions. Bullseye also has difficulty with dynamically-generated arrays, such as C-style strings represented with `char *` types. Bullseye cannot generate inputs using call sequences as CUTE is able to; such a feature would be of considerable utility in testing data structures.

6 Acknowledgments

I would like to thank Dr. Calvin Lin and Walter Chang for their guidance and expertise, particularly in learning how to use the C-Breeze framework and for critiquing a not-inconsiderable number of prior drafts of this thesis. I would also like to thank Dr. Sarfraz Khurshid and Dr. Vitaly Shmatikov for serving on my thesis committee and providing feedback. Additionally, I would thank the Department of Computer Sciences for funding one summer semester of this work through an Undergraduate Research Opportunities Program award.

References

[1] GLib Reference Manual: Balanced Binary Trees. http://library.gnome.org/devel/glib/stable/glib-Balanced-Binary-Trees.html.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*, pages 370–378. Pearson Education, Inc., Boston, MA, USA, second edition, 2007.

[3] Clark Barrett and Cesare Tinelli. CVC3: An automatic theorem prover for satisfiability modulo theories problems. http://www.cs.nyu.edu/acsys/cvc3/.

[4] Michel Berkelaar, Kjell Eikland, and Peter Notebaert. lp_solve: a mixed integer linear programming solver. http://lpsolve.sourceforge.net/5.5/.

[5] Adam Brown, Samuel Z. Guyer, Daniel Jiménez, and Calvin Lin. The C-Breeze Compiler Infrastructure. http://www.cs.utexas.edu/users/c-breeze/tr/c_breeze.ps, 2006.

[6] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI'05)*, pages 213–223, 2005.

[7] Karem Sakallah. What is SMT (satisfiability modulo theories)? *ACM/SIGDA E-Newsletter*, 36(24), December 2006. http://www.sigda.org/newsletter/2006/eNews_061215.html.

[8] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *5th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'05)*, pages 263–272. ACM, 2005.

[9] Junfeng Yang, Can Sar, Paul Twokey, Cristian Cadar, and Dawson Engler. Automatically generating malicious disks using symbolic execution. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 243–257, 2006.