SUDS: An Infrastructure for Creating Dynamic Software Defect Detection Tools

ERIC LARSON

Department of Computer Science and Software Engineering
Seattle University
elarson@seattleu.edu

Abstract SUDS is a powerful infrastructure for creating dynamic software defect detection tools. It contains phases for both static analysis and dynamic instrumentation allowing users to create tools that take advantage of both paradigms. The results of static analysis phases can be used to improve the quality of dynamic defect detection tools created with SUDS by focusing the instrumentation on types of defects, sources of data, or regions of code. The instrumentation engine is designed in a manner that allows users to create their own correctness models quickly but is flexible to support construction of a wide range of different tools. The effectiveness of SUDS is demonstrated by showing that it is capable of finding bugs and that performance improves when static analysis is used to eliminate unnecessary instrumentation.

Keywords software testing, software defect detection, software engineering, static analysis, instrumentation, testing tools

Appears in Automated Software Engineering, Volume 17, Issue 3 (September 2010). The original publication is available at www.springerlink.com.

1 Introduction

It is increasingly important in the world today to have correctly working software. In order for software systems to fully protect against malicious users, they must be perfectly implemented in software – a very difficult task. Most of the defects detected in secure systems are due to implementation mistakes, many due to using an inherently unsafe language like C. If programmers are not careful, it is possible for an attacker to overwrite regions of memory in such a manner they are able to execute arbitrary code and gain access to private data. Therefore it is necessary to have high quality tools that can detect defects before software is released.

Many tools look for software defects at run-time by inserting instrumentation into the source code. Often, additional information about the program is stored and used to locate defects. Dynamic checkers are capable of finding defects that span multiple function boundaries, library functions, or even process boundaries. The largest drawback to dynamic defect detection is its dependence on the input. Defects will only be detected if a test is run that exposes the defect. Another downside is the performance overhead associated when running the program.

This paper describes SUDS, an infrastructure designed to create dynamic software defect detection tools. Users can create their own defect tools by first determining which programming constructs or events are relevant to the property being checked. Then, users can insert function calls to instrumentation routines. A set of provided routines allows the user to pass run-time information as parameters to the instrumentation routines.

We demonstrate the effectiveness of SUDS by creating three checkers specific to input-derived variables. The first checker looks for array out-of-bounds and pointer dereference errors. The second checker detects arithmetic overflow for input data. The last checker makes sure that string functions are used safely. All three of these checkers keep track of additional state during run-time and can be extended to find different types of software defects. These checkers found 26 defects in 18 programs.

In addition to standard compiler analyses, SUDS contains support for tainted data propagation and program slicing. The results of these phases can be used to focus and/or improve the performance of the instrumented code. In particular, we used these phases to remove instrumentation that was not needed by our array out-of-bounds checker. We considered data coming from input to be tainted and we applied program slicing from all array references and

```
1    int a, b, c, d, x[5];
2    scanf("%d", &a);                          -∞ ≤ a ≤ ∞
3    scanf("%d", &b);                          -∞ ≤ b ≤ ∞
4    if ((a > 10) || (a < 0)) exit(-1);        0 ≤ a ≤ 10
5    c = 2;
6    d = b;
7    x[a] = 3;                                 ERROR!
8    x[c] = 6;
9    printf("%d\n", d);
```

Fig. 1 Example of an array reference error

pointer dereferences in the program. This improves the run-time performance of the array checker by 36% on average.

To illustrate the key features of SUDS, consider the example in Fig. 1. An array reference error occurs at line 7 because the array could be referenced out-of-bounds. The array checker find the error by assigning ranges to input values as shown in Fig. 1. On lines 2 and 3, variables a and b get assigned a value from input. The corresponding ranges are set from -∞ to ∞. Assume the user entered the value 3 for variable a[1]. Then, on line 4, the if statement evaluates to false. This will limit the range of variable a to be between 0 and 10. Finally on line 7, an error is detected since the upper bound of a is 10, greater than the maximum allowable index of 4.

In order to find the error, SUDS will instrument this program by adding calls to instrumentation routines using the instrumentation director interface described in Section 3.1. In the case of the array checker, the instrumentation routines will manipulate additional state associated with integers and arrays. This additional state is stored in a shadow state table. For integers, the table stores the allowable range of values. For arrays, the table stores the run-time size of the array. On array references, the tables are accessed to determine if the array can be accessed out-of-bounds.

Fig. 2 shows how instrumentation[2] is applied to the program in Fig. 1. The instrumentation call on line 2 will save the size of the array x. The calls on line 4 and 6 introduce ranges for variables a and b respectively. The range for a gets narrowed on lines 7 and 8 based on the actual run-time value of a. On line 14, the array will be checked based on the current range of a and the size of x.

The other instrumentation calls are unnecessary. The static analysis phases of SUDS can be used to focus the instrumentation. The array checker focuses on errors that are derived from

[1] Any value from 0 to 10 will work.
[2] Some details have been elided from this example for clarity. A more complete example is shown in Section 4.1.3.

```
1    int a, b, c, d, x[5];                          int a, b, c, d, x[5];
2    process_array_birth(x, 5*sizeof(int));         process_array_birth(x, 5*sizeof(int));
3    scanf("%d", &a);                               scanf("%d", &a);
4    process_new_input(&a, INT_MIN, INT_MAX);       process_new_input(&a, INT_MIN, INT_MAX);
5    scanf("%d", &b);                               scanf("%d", &b);
6    process_new_input(&b, INT_MIN, INT_MAX);       process_new_input(&b, INT_MIN, INT_MAX);
7    process_greater_than(&a, a, 10);               process_greater_than(&a, a, 10);
8    process_less_than(&a, a, 0);                   process_less_than(&a, a, 0);
9    if ((a > 10) || (a < 0)) exit(-1);             if ((a > 10) || (a < 0)) exit(-1);
10   c = 2;                                         c = 2;
11   process_int_remove(&c);                        process_int_remove(&c);
12   d = b;                                         d = b;
13   process_int_copy(&d, &b);                      process_int_copy(&d, &b);
14   process_array_ref(x, &a);                      process_array_ref(x, &a);
15   x[a] = 3;                                      x[a] = 3;
16   process_array_ref(x, &c);                      process_array_ref(x, &c);
17   x[c] = 6;                                      x[c] = 6;
18   printf("%d\n", d);                             printf("%d\n", d);
```

Fig. 2 Array checker instrumentation before static analysis (left) and after static analysis (right).

input values. The tainted data propagation algorithm recognizes that variable `c` in Fig. 2 never holds an input value, thus instrumentation associated with variable `c` is removed. The program slicing algorithm works backwards from array references and determines which statements the array references depend on. Statements that do not have any influence on an array reference do not need instrumentation. In Fig. 2, neither variables `b` nor `d` are used in an array reference; instrumentation calls for these variables can be removed. Fig. 2 shows the instrumentation before static analysis (left) and after static analysis is applied (right) with the unnecessary instrumentation calls blocked out.

The main contributions of this paper are as follows:

- Description of SUDS – an infrastructure for creating dynamic defect detection tools.
- A detailed description of three input checkers – all use a shadow state table to track additional state.
- The use of static analysis to focus dynamic defect detection. Instrumentation is only applied to relevant statements improving run-time performance.

The remainder of the document is organized as follows. Section 2 provides an overview of SUDS. Adding instrumentation is described in Section 3. Section 4 presents the instrumentation models of our three input checkers. Static analysis phases are outlined in Section 5. Results using SUDS are presented in Section 6. Section 7 contains related work and Section 8 concludes with directions for future work.

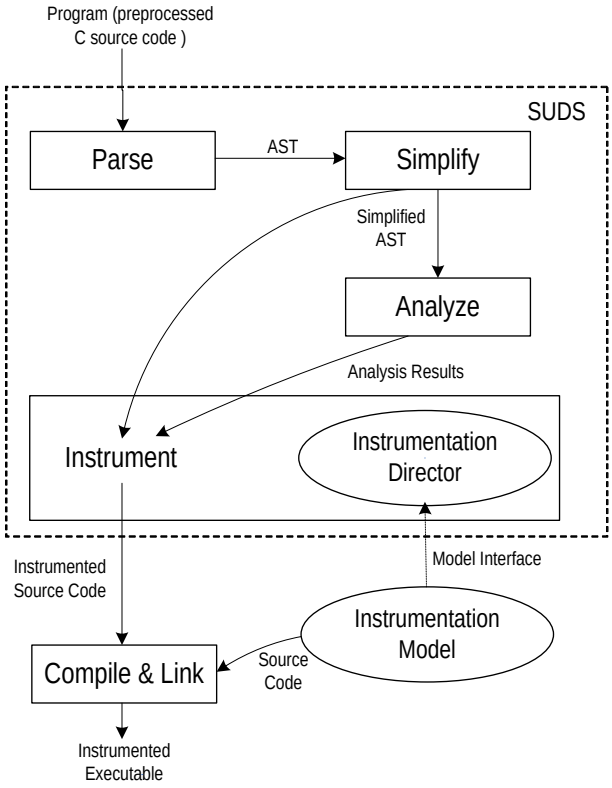

Fig. 3 Overview of SUDS

2 Overview of SUDS

SUDS, like most compilers and defect detection tools, is organized as a series of different phases. The primary phases are displayed in Fig. 3.

SUDS takes preprocessed source code as an input and parses the code to create an intermediate representation using an abstract syntax tree (AST). The code then goes through a simplification phase that converts the source code into an equivalent program that is easier to analyze in later phases. The static analysis phase consists of several subphases, each performing a different static analysis algorithm. The last phase within SUDS instruments the program using the instrumentation director interface. The output of SUDS is instrumented source code which then can be compiled and linked with an instrumentation model to form an instrumented program.

A user can either use an existing model or create their own. To create their own instrumentation model, a user only needs to write the instrumentation model, which is

completely separate from SUDS, and an instance of the instrumentation director interface (both drawn as ovals in Fig. 3).

SUDS is written in C++ and processes programs written in the C programming language. The organization of SUDS is highly modular in that it is relatively straight-forward to add a new phase or modify an existing phase without internal knowledge of how the other phases work. The remainder of this section briefly describes each of the phases.

2.1 Parsing

SUDS uses a modified version of the parser from cTool (2004) and operates on the whole program. The parser supports virtually all C programs with a few exceptions. The most prominent restriction is that it requires that all functions be declared before being used. In a vast majority of these exceptional cases, it is possible to rewrite the code with minimal effort. SUDS supports many, but not all, of the language extensions that exist in popular compilers such as gcc. Except for addressing these restrictions, no modifications to the program source code are needed. Functions without source code present, including system calls, are not analyzed by SUDS. However, instrumentation can be added to statements that call these functions, summarizing what these functions accomplish with respect to the checker[3].

2.2 Simplification

The next phase in SUDS is to transform the initial AST into an intermediate C representation similar to the grammar developed by Hendren et al. (1992) and the GIMPLE grammar used in the gcc compiler (Merrill 2003). It serves a similar purpose as CIL (Necula et al. 2002a). The intent of simplification is to reduce the complexity of identifying and analyzing relevant program elements. Complex C statements are broken down into simple statements with at most two operands and a single assignment to an l-value (such as `a=b+c`). Side effects and short-circuited operators (such as &&) are eliminated via program transformations.

There are two advantages to simplifying the program. It simplifies the static analysis phases – statements can either alter the control of a program or assign data to a memory location but they cannot do both. The other advantage of the simplification process is that instrumented function

[3] In the checkers described in Section 4, instrumentation is added in this manner for common system calls.

calls can be restricted to statement boundaries. This is not possible without simplification since there might be an important event in the middle of a long expression.

2.3 Static Analysis

The static analysis phase consists of several subphases. The initial subphases perform standard compiler analyses such as computing points-to information and performing data flow analysis. Two analyses, tainted data propagation and program slicing, allow the user to further focus the dynamic instrumentation.

The user can specify what data they deem to be tainted. The tainted data propagation algorithm will determine the set of statements that operate on tainted data. The users can also specify the slicing criterion – the set of statements that are initially in the slice. The program slicing algorithm will compute the set of statements that influence the statements in the slicing criterion. The instrumentation phase can use this information to only apply instrumentation to interesting statements – typically statements that operate on tainted data and are in the slice.

A more detailed discussion of the static analysis phase is in Section 5. This phase is optional if SUDS is used strictly as an instrumentation tool as it is not necessary to use any of the static analysis information when instrumenting the program.

2.4 Instrumentation

The instrumentation engine of SUDS automatically adds instrumentation using an instrumentation director. The instrumentation director interface of SUDS provides functions for different programming constructs including expressions, statements, and certain events (such as the start of a function). Users can add instrumentation to a program by implementing the functions that correspond to constructs and events that are interesting with respect to the instrumentation model. In most cases, the added instrumentation is a call to a routine in the instrumentation model. A suite of helper functions is provided to simplify the process of adding a call to an instrumentation function with different parameters.

This organization, described in Section 3, facilitates rapid creation of new defect detection tools and is flexible in that users have access to the internals of SUDS to allow for the creation of very powerful, specialized instrumentation. In addition, SUDS can be used to create profilers, coverage tools, debugging aids, and program analyzers.

2.5 Instrumentation Models

An instrumentation model consists of routines that are used to find defects. SUDS includes models for input checking (described in Section 4) that employ a shadow state table that keeps track of additional state associated with variables, pointers, and arrays. Different instrumentation routines add, remove, modify, or check entries in the table. The structure of these models allows users to easily create and implement their own models by using one of the provided models as a starting point. It is also possible to create much simpler models that merely output information or inject checks. For instance, an instrumentation model that traces function calls is described in Section 3.2.

3 Instrumentation Engine

This section describes the instrumentation engine used by SUDS. To accommodate users who want to design their own models, this phase has been designed in a fashion for both fast creation of simple models but also generic enough to support a wide range of instrumentation tasks. To promote the construction of a model quickly, several helper functions that perform common instrumentation tasks are provided.

3.1 Instrumentation Director Interface

In order for SUDS to produce instrumented code, it needs to know what instrumentation to add and where to add it. The instrumentation director provides both of these capabilities. An instance of the instrumentation director interface is created for each instrumentation model. The base interface, from which instrumentation directors are derived from, consists of several member functions that correspond to various programming constructs and events. The instrumentation director is similar to the visitor design pattern in that it visits each node in the AST. For each node in the AST, the user decides what, if any, instrumentation needs to be added.

To create an instrumentation director, a user implements the functions that correspond to interesting constructs and events with respect to the instrumentation model. During the instrumentation phase, SUDS traverses the AST calling the appropriate interface function for each programming construct or event. This phase is static and makes only one pass through the program. The output of this phase is an AST containing instrumented source code. This is immediately followed by a phase that prints the instrumented source to a file.

The instrumentation director interface is illustrated using the following interface function:

```
void instrumentDerefExpr(DerefExpr *expr, bool isLhs,
  Stmt *&before, Stmt *&after);
```

This function refers to a pointer dereference operation (*) and is called whenever a dereference expression is encountered. The first two parameters `expr` and `isLhs` are input parameters. The parameter `expr` refers to the corresponding dereference expression that is under consideration. Most of the other interface functions will include a pointer to the corresponding programming construct. These internal data structures can be used in two ways. One use is to further analyze the data structure to determine which instrumentation function, if any, to call. Frequently, the type of the expression is analyzed when adding instrumentation. For instance, an addition operation involving two integers will typically be treated differently than an addition operation involving a pointer and an integer. The second use of the internal data structures is for specifying parameters to pass into the instrumentation functions. This process is described in Section 3.2. The second input parameter `isLhs` is true if the expression appears on the left-hand side of an assignment expression and false if it appears on the right-hand side of the expression. The `isLhs` parameter is only available for expressions that can appear on the left-hand side.

The last two parameters `before` and `after` are output parameters that point to a list of statements. In order to successfully add instrumentation, it is necessary to add the newly created statement to either the before list or the after list. If the instrumentation is added to the before list, it is added before the statement or event in question. When the instrumented program is executing, the added instrumentation will be executed before the statement. Similarly, it is added after the statement or event if it is added to the after list. Some interface functions only allow instrumentation to be added to the before list and others require instrumentation to be added to the after list only. For example, it is only permissible to add instrumentation after a variable declaration since it does not make sense to add instrumentation before the variable is born because the variable is inaccessible before it is born.

In addition to these four parameters, some interface functions have an additional parameter that represents an offset. This is used for instrumentation that deals with parameter passing and returning values. Parameter passing is described later in this section.

Table 1 Instrumentation Director Interface Functions

Statements	*Expressions*	*Function Calls*	*Other Events*
expression statement if statement switch statement for loop while loop do loop (start) do loop (end) goto statement break statement continue statement va_start statement va_end statement	constant string constant variable unary arithmetic / logical binary arithmetic / logical relational expression cast sizeof address of address-arrow `&(a->b)` dereference member selection arrow array reference built-in expression va_arg expression	function call (caller) start of function (callee) return statement (callee) end of function (callee) return from function (caller) function call argument birth of parameter death of parameter return value (callee) return value (caller) birth of parameter (main) death of parameter (main) system call	start of block end of block start of program end of program birth of variable death of variable label data member copy

Table 1 shows a table of the various constructs and events that have functions in the instrumentation director interface. The first two columns refer to statements and expressions respectively. A few notes regarding these two columns:

- There is a single interface function for all binary (two operands) arithmetic and logical operators. Within this function, different instrumentation routines may be called based on the operation.
- The address-arrow expression refers to expressions such as `&(a->b)`. It is an artifact of the simplification process.
- The va_start and va_end statements as well as the va_arg expression are used in processing variable length argument lists. In general, support for variable length argument lists is limited.

The third column of Table 1 refers to events that occur when calling and returning from a function. When calling a function there are interface functions for both the function call itself (added to the caller) and the start of the new function (added to the callee). Similarly, when returning from a function, interface functions exist for the end of the function (added to the callee) and when the function has returned (added to the caller). In addition, instrumentation can be added to return statements.

The interface function for a function call argument[4] is called for each argument passed into a function allowing the user to specify instrumentation for each argument independently. Similarly, the birth of parameter interface function is called for each parameter. Both of these interface functions have an offset parameter that refers to the position in which the parameters appear in the parameter list. Structure variables also use the offset field. Assume that a structure is passed into a function. Initially, the function call expression interface function will be called for the entire structure. Then the function is called for each data member that appears in the structure definition. Data members that refer to arrays or nested structures are traversed recursively, applying the interface function to each array entry or data member. The offset is incremented each time the interface function is applied to an array entry or data member. When the callee function starts, the same approach is used for the parameters. Together, they can be used to copy state from an argument to its corresponding parameter. A similar approach is employed when passing the return value back.

There is also an instrumentation routine that corresponds to the death of the parameter. Since it is implicitly called at the beginning of the program, the function `main` has separate parameter functions. This is important for our input checkers described in Section 4 since the parameters to `main` contain command-line information which is considered to come from input.

There is a separate interface function for system calls or any function where source code is not present. In most cases, this function will add different instrumentation based the function being called. The argument and return value interface functions are not called for system calls.

The fourth column describes other events that can occur in a program. It is largely self-explanatory except perhaps for the data member copy. This function is called for each data member when a struct is assigned to another struct.

The instrumentation director interface also contains data members that can be used while creating an instrumentation director instance. Here are a few of the data members that are available: the current statement, the current function, and the current line number.

3.2 Calling Instrumentation Functions

Once the programming constructs and events that need instrumentation has been determined, the next step is to write the instrumentation director routines to add the instrumentation. In most

[4] We use the terms *argument* and *parameter* to refer to actual parameters and formal parameters respectively.

Table 2 Helper routines for adding parameters to instrumentation function calls

Routine	*If x is ___*	*Then, add ___*	*Used For*
addStringParm(string x);	`foo`	`foo`	Any expression in string form. Useful for constructing your own expressions by concatenating different stings.
addStringConstantParm(string x);	`foo`	`"foo"`	Any string used as a constant within the instrumentation routine.
addIntegerParm(int x);	`23`	`23`	Any integer constant.
addExprParm(Expr *x);	`a[i]`	`a[i]`	The value of an expression or variable.
addStringExprParm(Expr *x);	`a[i]`	`"a[i]"`	The expression as a string - useful for debugging and diagnostics.
addAddrOfExprParm(Expr *x);	`a[i]`	`&(a[i])`	The address of an expression (assuming it is addressable); useful for models that track variables by address.
addDeclParm(Decl *x);	`a`	`a`	The variable associated with a declaration.
addSizeParm(Type *x);	`int`	`sizeof(int)`	The size of a type.
addObjSizeParm(Type *x);	`int *`	`sizeof(int)`	The size of the type pointed to (or the size of the element for arrays).
addFileNameParm();	`-`	`-`	The current file name.
addLineNumParm();	`-`	`-`	The current line number.
addUniqueIdParm();	`-`	`-`	A unique integer that is incremented after each call to an instrumentation routine. Used to distinguish between multiple calls that occur on the same source code line.

cases, the added instrumentation will merely be a single statement that calls an instrumentation function. Helper routines are provided to simplify this task. The basic flow is as follows:

1. Initialize the call by calling `initInstrCall`. This function takes no parameters.
2. Add parameters to the function. Table 2 outlines the different routines that are available depending on the type and use of parameter.
3. Complete the call by calling `addInstrCall`. This function takes two parameters: a list to add the instrumentation statement to (either the before or after list) and the name of the function to call.

This process is best illustrated using an example of an instrumentation director interface. Fig. 4 shows an instrumentation director that is used for an instrumentation model that traces function calls. In both cases the functions are called with one parameter - the name of the function passed

```
1    class instrTrace: public instrDirInterface {
2
3      public:
4
5      void instrumentBeginFunction(FunctionDef *fn, Stmt *&after)
6      {
7        initInstrCall();
8        addStringConstantParm(fn->FunctionName());
9        addInstrCall(after, "function_begin");
10     }
11
12     void instrumentEndFunction(FunctionDef *fn, Stmt *&before)
13     {
14       initInstrCall();
15       addStringConstantParm(fn->FunctionName());
16       addInstrCall(before, "function_end");
17     }
18   };
```

Fig. 4 Sample instrumentation director instance for tracing function calls

as a string. The corresponding instrumentation model consists of the instrumentation functions `function_begin` and `function_end`. These functions, written outside of SUDS, simply display an appropriate message with the function name.

Ultimately, the instrumentation call is stored in the AST as a string that is simply emitted during the output phase. Alternatively, we could have implemented the instrumentation call using AST nodes. However, this would require the user to have detailed knowledge of how the AST is constructed. This is not necessary if strings are used. In addition, users can take advantage of both the C++ string library and print routines that already exist for variables, expressions, and statements within SUDS. It is also possible to create inlined instrumentation or instrumentation code that is an arbitrary set of C statements instead of a single function call.

4 Dynamic Detection of Input-Related Software Defects

In this section, we illustrate how SUDS can be used to detect security defects caused by improperly bounded inputs. DaCosta et al. (2003) observed that functions near an input source are more likely to be vulnerable to security exploits. These checkers, based on earlier work (Larson and Austin 2003), relax the requirement that the user specify a precise set of inputs that exposes the defect. This is accomplished by shadowing variables derived from input with variables that represent the range of values. The additional shadow state is stored in a *shadow state table*. The table is similar to the object table used by Jones and Kelly (1997) to store additional information about the objects (such as the size) that pointers point to.

Three different checkers are presented here. The array checker, described in Section 4.1, looks for array reference and pointer dereference errors based on input data. An arithmetic checker is concerned with arithmetic overflow and other potentially dangerous uses of input data. This checker is presented in Section 4.2. Finally, the string checker, described in Section 4.3, ensures that all string operations are safe.

4.1 Array Checker

To prevent a buffer overflow exploit, it is necessary for the program to check input data to ensure it does not exceed the bounds of any buffer it may be used to reference. However, many programs either fail to check input data or check the data incorrectly. Such cases are often hard to find. For example, the code sequence in Fig. 5 contains an off-by-one error. Such a defect may be difficult to find if the programmer writing the code to check the reference is not aware that the index is incremented before it is referenced.

Data that comes from input is considered *tainted* and includes environment variables, command line inputs, data read from files, and network packets. For each variable that holds input data, the shadow state table stores the lower and upper bounds. In order for an array reference to be considered safe, the index must be checked to determine if it can exceed the bounds of the array. The initial value of the lower and upper bounds are the smallest and largest values respectively that can be stored in the variable based on the data type of the variable. For instance, an `unsigned int` would have an initial lower bound of 0 and an initial upper bound of `UINT_MAX` (4,294,967,295).

During execution, control tests and operators may narrow input interval constraints. When an access to an array occurs, the bounds of the array index are compared with the run-time size of the referenced array. An error is reported if there is an index that can exceed the bounds of the array. When a variable is assigned a value that is not dependent on input, the destination variable is considered untainted, releasing shadow state[5].

```
1    int a;
2    unsigned int x;
3    int array[5] = {1, 2, 3, 4, 5};
4    scanf("%d", &x);
5    if (x > 4) exit(-1);
6    x++;
7    a = array[x];
```

Fig. 5 Example of an array reference error. The array is accessed out of bounds if `x` is four.

Internally, the checker uses two shadow state tables. One table tracks tainted integers and the other tracks arrays (storing the run-time size of the array). The shadow state tables are implemented using hash tables. Except for array references and pointer arithmetic which access both tables, the routines used to manage the two tables are separate.

4.1.1 Tracking Integers

The integer shadow state table keeps track of tainted integers and is indexed by address. Integers that are not tainted (do not store input data) are not stored in the table. An integer becomes tainted if it comes directly from input (via a call to an input function such as `scanf`) or is assigned a value from an expression that involves tainted data.

The data structure used to store integer state contains the following fields:

- *address* – Address of the variable (same as the index to the shadow state table), stored for debugging purposes.
- *name* – Name of the variable. For integers stored in structures, arrays, or in the heap, it contains a string on how the variable was assigned (`"*p"` for example). Only used for debugging and diagnostic purposes.
- *lower bound* – Lower bound, lowest possible value for the variable.
- *upper bound* – Upper bound, highest possible value for the variable.

Initially, the upper and lower bounds are assigned based on the type. For unsigned integers, the lower bound is zero. Otherwise, it is the most negative value the variable can hold, based on type. Similarly, the upper bound is the largest possible value. Future expressions and comparisons adjust the bounds, while array references check the bounds against the size of the array. One potential weakness of modeling integers using ranges as described here is that it is not always possible to specify the set of possible values using a single range. However this is not an issue for array checks as it is only necessary to check the lowest and highest possible values.

For expression statements, there are five instrumentation routines for tracking integers that are listed in Table 3. The destination variable is the variable used on the left-hand side of the expression statement. Its shadow state may change as a result of the expression. The source variables are variables accessed on the right-hand side of the expression statement. An entry is considered not tainted and removed from the table if it comes from an expression that does not

Table 3 Instrumentation routines for tracking integers in expression statements

Instrumentation Routine	***Description***	***Called when the right-hand side is…***
process_int_remove	Removes the destination variable from the table.	a constant literal (`3`) a sizeof expression (`sizeof(int)`)
process_int_copy	If the source variable is tainted, the shadow state of the source variable is copied into the shadow state of the destination variable. If the source variable is not tainted, the destination variable is removed from the table.	a single variable (`x`) an array reference (`a[i]`) a data member of a struct (`b.z`) a pointer dereference (`*p`) a cast to another variable (`(int) y`)
process_int_unary_op	The precise functionality depends on the operator and the state of the source variable. If the source variable is not tainted, the destination variable is removed from the table.	any unary operator (`-z`)
process_int_binary_op	The precise functionality depends on the operator and the state of the source variable(s). If neither source variable is tainted, the destination variable is removed from the table.	any binary operator (`a + b`)
process_int_rel_op	Removes the destination variable from the table since it stores a Boolean result. The entries for the source operands may be adjusted based on the operator and whether the comparison is true or false.	any relational operator (`a < b`)

have tainted operands. A basic case occurs when the right-hand side is a constant literal (such as `x = 3`)[6]. Shadow state is copied for expressions that merely copy an integer from one memory location to another (such as `x = *p`).

For expressions with unary and binary operators, the function depends on the particular operation. Unary and binary operators can be divided into two categories: arithmetic operations and bitwise operations. For arithmetic operations, the destination is not tainted if neither source operand is tainted. An operand is considered not tainted if it is either a variable that is not tainted (not present in the shadow state table) or is a constant value. The arithmetic operations are summarized in Table 4 (addition, subtraction, and multiplication) and Table 5 (division and remainder). In these tables, ticked variables x' and y' refer to tainted integer variables while unticked variables x and y represent untainted operands. Using structure notation, $x'.lb$ and $x'.ub$ respectively represent the lower and upper bounds of tainted variable x'.

The negate operator inverts the bounds of the source. Addition and subtraction with an untainted operand results in adjusting the bounds by the current value of the untainted operand.

[6] Since this checker focuses on input-related faults, the checker only detects array out-of-bound errors if the index is derived from input. By tracking integers that are assigned constants, additional errors may be detected.

Table 4 Rules for computing bounds for addition, subtraction, and multiplication. *x'* and *y'* are tainted; *x* and *y* are untainted.

Op ***(d' = …)***	***Destination Lower Bound*** ***(d'.lb = …)***	***Destination Upper Bound*** ***(d'.ub = …)***
$-x'$	$-x'.ub$	$-x'.lb$
$x'+y$	$x'.lb+y$	$x'.ub+y$
$x'+y'$	$x'.lb+y'.lb$	$x'.ub+y'.ub$
$x'-y$	$x'.lb-y$	$x'.ub-y$
$x-y'$	$x-y'.ub$	$x-y'.lb$
$x'-y'$	$x'.lb-y'.ub$	$x'.ub-y'.lb$
$x'\times y$	$\begin{cases} x'.ub\times y & y<0 \\ 0 & y=0 \\ x'.lb\times y & y>0 \end{cases}$	$\begin{cases} x'.lb\times y & y<0 \\ 0 & y=0 \\ x'.ub\times y & y>0 \end{cases}$
$x'\times y'$	$MIN\begin{pmatrix} x'.ub\times y'.ub \\ x'.ub\times y'.lb \\ x'.lb\times y'.ub \\ x'.lb\times y'.lb \end{pmatrix}$	$MAX\begin{pmatrix} x'.ub\times y'.ub \\ x'.ub\times y'.lb \\ x'.lb\times y'.ub \\ x'.lb\times y'.lb \end{pmatrix}$

When adding two tainted variables, the bounds of the destination variable are formed by adding the bounds of the two source operands together. When subtracting two tainted variables, the upper bound represents the maximum difference and is found by subtracting the lower bound of the second operand from the upper bound of the first operand. Similarly, the lower bound represents the minimum difference and is computed by subtracting the upper bound of the second operand from the lower bound of the first operand.

Multiplying and dividing by an untainted operand need to take the sign of the untainted variable into account. If the value is positive, then the upper bound of the destination is computed using the upper bound of the tainted operand while the lower bound of the destination

is computed using the lower bound. The situation is reversed if the untainted variable is negative. If the untainted value is zero, the upper and lower bounds are both set to zero for multiplication and a divide-by-zero error occurs for division. Multiplying two tainted operands requires computing all four possible products involving the upper and lower bounds of the operands. The highest product is the upper bound of the destination, while the lowest product is the lowest bound of the destination.

In integer division, the highest and lowest possible values occur when dividing by either one or negative one. This is captured in Table 5 using the function *spanZero*. If the divisor's range includes both one and negative one, then the upper bound is formed by taking the maximum of the absolute value of the dividend's upper bound and the absolute value of the dividend's lower bound. The lower bound is the same value but negated. If the divisor's range does not include both one and negative one, it implies that the divisor is completely positive or completely negative. In this situation, the upper and lower bounds can be computed in a similar way to multiplication by computing the four quotients that involve the upper and lower bounds.

The bounds of the remainder operator can vary from zero to one less than the value of the divisor when dealing with positive numbers. Using negative numbers is more problematic. Based on the C99 standard (ISO/IEC 9899:1999), the magnitude of the remainder is computed using the absolute value of both the dividend and the divisor. The sign of the remainder is identical to the sign of the dividend.

For bitwise operators (and, or, not, shift), the checker treats the destination variable as untainted and removes it from the table. The reason for this decision is that the result of a bitwise operation is likely not to be used in an array operation. The logical and operator '`&&`' and logical or '`||`' operator are removed during simplification – converted to an appropriate if statement. The logical not operator '`!`' is still present after simplification and is treated identically to as not equal to 0 (`!x` is equivalent to `x != 0`) and is discussed with the other relational operators below.

Table 5 Rules for computing bounds for multiplication, division, and remainder. *x'* and *y'* are tainted; *x* and *y* are untainted. For division rules, *spanZero(y')* = $y'.lb < 0 \wedge y'.ub > 0$

***Op* (*d'* = …)**	***Destination Lower Bound* (*d'.lb* = …)**	***Destination Upper Bound* (*d'.ub* = …)**
x' / y	$\begin{cases} x'.ub / y & y < 0 \\ ERROR & y = 0 \\ x'.lb / y & y > 0 \end{cases}$	$\begin{cases} x'.lb / y & y < 0 \\ ERROR & y = 0 \\ x'.ub / y & y > 0 \end{cases}$
x / y'	$\begin{cases} -abs(x) & spanZero(y') \\ x / y'.lb & x < 0 \wedge \neg spanZero(y') \\ x / y'.ub & x \geq 0 \wedge \neg spanZero(y') \end{cases}$	$\begin{cases} abs(x) & spanZero(y') \\ x / y'.ub & x < 0 \wedge \neg spanZero(y') \\ x / y'.lb & x \geq 0 \wedge \neg spanZero(y') \end{cases}$
x' / y'	$\begin{cases} -MAX\begin{pmatrix} abs(x'.lb) \\ abs(x'.ub) \end{pmatrix} & spanZero(y') \\ MIN\begin{pmatrix} x'.ub / y'.ub \\ x'.ub / y'.lb \\ x'.lb / y'.ub \\ x'.lb / y'.lb \end{pmatrix} & \neg spanZero(y') \end{cases}$	$\begin{cases} MAX\begin{pmatrix} abs(x'.lb) \\ abs(x'.ub) \end{pmatrix} & spanZero(y') \\ MAX\begin{pmatrix} x'.ub / y'.ub \\ x'.ub / y'.lb \\ x'.lb / y'.ub \\ x'.lb / y'.lb \end{pmatrix} & \neg spanZero(y') \end{cases}$
$x' \% y$	$\begin{cases} -(abs(y)-1) & x'.lb < 0 \\ 0 & x'.lb \geq 0 \end{cases}$	$\begin{cases} abs(y)-1 & x'.ub > 0 \\ 0 & x'.ub \leq 0 \end{cases}$
$x \% y'$	$\begin{cases} -MAX\begin{pmatrix} abs(y'.lb) \\ abs(y'.ub) \end{pmatrix}+1 & x < 0 \\ 0 & x \geq 0 \end{cases}$	$\begin{cases} MAX\begin{pmatrix} abs(y'.lb) \\ abs(y'.ub) \end{pmatrix}-1 & x > 0 \\ 0 & x \leq 0 \end{cases}$
$x' \% y'$	$\begin{cases} -MAX\begin{pmatrix} abs(y'.lb) \\ abs(y'.ub) \end{pmatrix}+1 & x'.lb < 0 \\ 0 & x'.lb \geq 0 \end{cases}$	$\begin{cases} MAX\begin{pmatrix} abs(y'.lb) \\ abs(y'.ub) \end{pmatrix}-1 & x'.ub > 0 \\ 0 & x'.ub \leq 0 \end{cases}$

Table 6 Rules for computing bounds for relational operators. *x′* and *y′* are tainted; *y* is untainted.

Operation	***If true***	***If false***
$x' < y$	$if\,(x'.ub \geq y) : x'.ub = y - 1$	$if\,(x'.lb < y) : x'.lb = y$
$x' < y'$	$if\,(x'.ub \geq y'.ub) : x'.ub = y'.ub - 1$ $if\,(y'.lb \leq x'.lb) : y'.lb = x'.lb + 1$	$if\,(y'.ub > x'.ub) : y'.ub = x'.ub$ $if\,(x'.lb < y'.lb) : x'.lb = y'.lb$
$x' \leq y$	$if\,(x'.ub > y) : x'.ub = y$	$if\,(x'.lb \leq y) : x'.lb = y + 1$
$x' \leq y'$	$if\,(x'.ub > y'.ub) : x'.ub = y'.ub$ $if\,(y'.lb < x'.lb) : y'.lb = x'.lb$	$if\,(y'.ub \geq x'.ub) : y'.ub = x'.ub - 1$ $if\,(x'.lb \leq y'.lb) : x'.lb = y'.lb + 1$
$x' == y$	$x'.lb = y$ $x'.ub = y$	$if\,(x'.lb == y) : x'.lb = y + 1$ $if\,(x'.ub == y) : x'.ub = y - 1$
$x' == y'$	$x'.lb = y'.lb = MAX(x'.lb, y'.lb)$ $x'.ub = y'.ub = MIN(x'.ub, y'.ub)$	no changes

Relational operators are handled differently in that they narrow the constraints of the source operands. The destination variable of a relational operation is always considered untainted since the value is either true or false. Rules for some of the operators are shown in Table 6. These rules use the actual true or false result of the comparison in determining what bounds to adjust and how they are adjusted. Consider the less than operator $x' < y$ with a tainted integer x' and an untainted integer y. If this comparison is true, it is now known that $x' < y$. This means that the upper bound of x' must be no greater than one less than y. If the upper bound of x' is greater than or equal to y, the upper bound is then set to y-1. If the upper bound of x' is already lower than y, no change is made. If the less than operation is false, it implies that $x' \geq y$. In this situation, the lower bound of x' must be at least y. If the lower bound of x' is lower than y, it is set to y. Otherwise, no change is made. For comparisons, involving two tainted variables, it is possible for both source operands to get adjusted. In the case where $x' < y'$ is true, the upper bound of x' must be lower than the upper bound of y' and the lower bound of y' must be greater than the lower bound of x'.

The equality operators are handled differently. If an equality comparison to an untainted value is true, then the lower and upper bounds are set to that value. If the comparison is false, meaning the variable is not equal to the value, no changes are made unless the value happens to equal one of the bounds. When a test for equality between two tainted variables is true, the bounds of both variables are adjusted to the intersection of the ranges before the comparison. If two tainted variables are determined to be not equal, no changes are made to either variable.

No additional instrumentation is needed for control statements such as if statements and loops. This is due to the fact that comparisons are removed from control statements and replaced with a single Boolean variable during simplification. Adjustments to ranges are done at comparison expression statements which, after simplification, commonly precede control statements.

Integers can also be passed into functions as parameters and returned from functions as return values. The state of the function arguments must be propagated into the corresponding parameters. This is accomplished using a two-step process. First, the state of each function argument is stored into a separate table that is specifically used for passing parameters. Second, when the callee function begins, the state of each parameter is initialized using this table. This table is indexed by the offset value provided by SUDS. Described in Section 3.1, the offset is used to keep track of which parameter is being used and also accounts for integers within structs that are passed as parameters. A similar two-step process is used for propagating return values when the function has completed.

Instrumentation routines are also present to add input state to the table. These routines are called when a system library function that obtains input is called. In addition, instrumentation also copies state for functions like `memcpy`.

4.1.2 Tracking Arrays and Pointers

The array shadow state table stores additional state for arrays and pointers. The table is indexed by the base address and contains the following four fields:

- *base address* – Base address of the array or object being pointed to.
- *name* – Name of the variable, pointer, or expression at creation, used for debugging.
- *size* – Size of the array or object being pointed to in bytes.
- *illegal pointer* – Set to true if the pointer could be pointing out of bounds.

Table 7 Adding pointers to the input checker array and pointer shadow state table.

Pointer is assigned to …	***Size***	***Base Address***	***Illegal Pointer***
a string constant	size of the string	base address of the string	false
the result of a `malloc`/`calloc` call	size of the allocation	base address of the allocation	false
the address of a variable	size of the variable	address of the variable	false
the result of `p` + `i` or `p` – `i` where `p` is a pointer and `i` is an integer	size of `p`'s entry	base address of `p`'s entry	true if the operation, using `i`'s shadow state, exceeds the bounds

Entries in this table are added in two situations: when an array is declared (locally or globally) or when a pointer is assigned a new value. For arrays, the base address (and index of the shadow state table entry) is the starting address of the array and the size is the number of bytes in the array.

The different cases where pointers are added to the shadow state table are described in Table 7. In all cases, the base address, and index into the shadow state table, is the value of the pointer after the operation has completed. When a pointer is assigned to a string constant, the result of a dynamic memory allocation, or the address of a local variable, it points to the beginning of the object. The size is set based on how the pointer is initialized. The pointers are not flagged as illegal since they point to the beginning of the object and not out-of-bounds.

For pointer arithmetic operations, the information from the pointer used as a source operand is merely copied into the new entry. The index however is the resulting address after the arithmetic operation. Since pointer arithmetic operations are often used as the first step in referencing arrays, an additional check is made to see if the pointer could be pointing out-of-bounds. This is done using the shadow state of the integer *i* if *i* is tainted and is described later in this section.

No changes to the table occur when a pointer variable is assigned the result of another pointer variable. Consider the assignment $p = q$ and assume that q points to address 1000. Since q has already been assigned the address 1000 from an early statement, there already exists an entry for address 1000 in the shadow state table. The fact that p now points to address 1000 does not change the object at address 1000 in any fashion. Similarly, no instrumentation is needed when

passing pointers (or arrays which are treated as pointers) into functions[7] or in other cases where pointer values are simply copied from one memory location to another.

Entries are removed from the table when an array goes out of scope or when an object is deallocated. However, entries are not removed in other cases (string constant and address of variable) for performance purposes. The primary focus of this input checker is to detect out-of-bounds errors, opposed to temporal memory-access errors where a pointer is pointing to memory that has been deallocated. By being more careful with removing entries from the table, these types of errors can be detected.

Upon an array reference or a pointer arithmetic operation, both the integer and array shadow state tables are accessed. First, the array shadow state table is accessed to obtain the size and base address. The acceptable lower and upper bounds are computed using the size of the array in bytes (*size*), base address (*baseAddr*), current value of the pointer being used in the reference (*currAddr*), and the size of the elements of the array (*eltSize*):

$$lb = -\left(\frac{currAddr - baseAddr}{eltSize}\right) \qquad ub = \frac{size - (currAddr - baseAddr)}{eltSize} - 1$$

If the entry cannot be found in the shadow state table, it means that the pointer is no longer valid and an error message is emitted. A warning is also displayed if the element size does not divide evenly.

Once the acceptable bounds are determined, the next step is to get the entry from the integer shadow state table that corresponds to the index used in the array reference. If the entry is not present, then the integer is not tainted and no checks are performed. If the entry is present, the upper and lower bounds are compared to the allowable bounds for the array. For array references, if it is possible for the array to be accessed out-of-bounds, an error message is emitted and the program is (optionally) aborted. If the check passes, the program continues execution as normal. No shadow state for the array or integer is modified during the check.

For pointer arithmetic operations, if it is possible for the pointer to be out-of-bounds, the illegal pointer flag is set to true but an error is not reported. If a pointer dereference operation is executed with an illegal pointer, then an error is reported. The rationale for this rule is that a

[7] An exception to this rule is when array arguments are passed into `main`. This is why SUDS has different instrumentation interface functions for `main`.

program may intentionally have a pointer point out-of-bounds (commonly pointing just beyond the last array element). This is not a problem unless it is dereferenced.

4.1.3 Array Reference Example

Here we will show how our technique can find the off-by-one error in the code segment from the example in Fig. 5. In a conventional dynamic defect detection implementation, the defect will not be detected unless `x` is four. Fig. 6 displays the original code, the simplified code, and the simplified code with added instrumentation routines. The first three parameters to the instrumentation functions are the same for all routines: file name, line number (of the original source code file), and a unique identifier (in this example the unique ids range from 36 to 48).

In the instrumented version, arrays are added to the table at lines 8 and 21. At line 8, the array declared at line 3 is added to the table. The last three parameters are used to initialize the shadow state table entry for this array: name, base address, and the size. An array is added at line 21 for the string constant that is ultimately used as the format string for the `scanf` call. The array elements are initialized individually as a result of simplification starting at line 10. Each of these assignments is an integer assignment. Since the right-hand side contains a constant, the destination array element is considered untainted and removed from the table.

At line 23, `scanf` is executed, placing the value entered by the user into `x`. When the value is first read from input, it is given a range to span all possible values. Since `x` is an unsigned integer (or more technically `T_29` points to an unsigned integer), the initial lower bound is zero and the initial upper bound is `UINT_MAX`. At the comparison in line 26, the bounds will be adjusted depending on what value is actually entered by the user. If the user enters a number greater than four, the comparison will be true. This will cause the lower bound of `x` to be set to five. Control will then flow into the if statement and the program will exit. The more interesting case occurs when the if statement is false. This means that `x` must be less than or equal to four; the upper bound of `x` is set to four. The lower bound of `x` remains at zero.

Original Program

```
1    int a;
2    unsigned int x;
3    int array[5] = {1, 2, 3, 4, 5};
4    scanf("%d", &x);
5    if (x > 4) exit(-1);
6    x++;
7    a = array[x];
```

Program after simplification

```
1    int a;
2    unsigned int x;
3    int array[5];
4    char *T_28;
5    unsigned int *T_29;
6    int T_30;
7    char *T_31;
8
9    array[0] = 1;
10   array[1] = 2;
11   array[2] = 3;
12   array[3] = 4;
13   array[4] = 5;
14   T_28 = "%d";
15   T_29 = &x;
16   scanf(T_28,T_29);
17
18   T_30 = x > 4;
19   if (T_30) {
20     exit(-1);
21   }
22
23   x = x + 1;
24   a = array[x];
```

Program after instrumentation

```
1    int a;
2    unsigned int x;
3    int array[5];
4    char *T_28;
5    unsigned int *T_29;
6    int T_30;
7    char *T_31;
8    process_array_birth("test.c", 5, 36, "array", &(array[0]), 5 * (sizeof(int)));
9
10   array[0] = 1;
11   process_int_remove("test.c", 5, 37, &(array[0]));
12   array[1] = 2;
13   process_int_remove("test.c", 5, 38, &(array[1]));
14   array[2] = 3;
15   process_int_remove("test.c", 5, 39, &(array[2]));
16   array[3] = 4;
17   process_int_remove("test.c", 5, 40, &(array[3]));
18   array[4] = 5;
19   process_int_remove("test.c", 5, 41, &(array[4]));
20   T_28 = "%d";
21   process_array_birth("test.c", 9, 42, "T_28", T_28, 3);
22   T_29 = &x;
23   scanf(T_28,T_29);
24   process_new_input("test.c", 9, 43, "T_29", T_29, 0LL, 4294967295LL);
25
26   T_30 = x > 4;
27   process_int_rel_op("test.c", 10, 44, 36, "T_30", &(T_30), T_30, &(x), x, 0, 4);
28   if (T_30) {
29     exit(-1);
30     end_checker("test.c", 10, 45);
31   }
32
33   x = x + 1;
34   process_int_binary_op("test.c", 11, 46, 19, "x", &(x), &(x), x, 0, 1);
35   process_array_ref("test.c", 12, 47, array, sizeof(int), &(x));
36   a = array[x];
37   process_int_copy("test.c", 12, 48, "a", &(a), &(array[x]));
```

Fig. 6 Input checker example

When the value of `x` is incremented at line 33, the lower bound of `x` is incremented from zero to one and the upper bound is incremented from four to five. When the array check occurs at line 35, the interval of `x` is compared to the size of the array. Even though `x` may have a legal value, an error message is displayed since it is possible for the input to be five, which exceeds the bounds of the array.

In this example, the array reference defect could be found even if the user did not enter the precise failing value of four. However, the defect can only be detected if the array reference is actually executed. In this case, the defect would be missed if the user entered a value greater than four as the program exits. This problem is mitigated to a large extent if the program is run with a test suite that obtains high code coverage.

4.1.4 Limitations

Since our approach is dynamic and relies on the particular control path taken through a program, it is an unsound approach, meaning that it is possible to miss actual bugs. With respect to a particular control path, our approach is still unsound. One problem stems from the use of run-time information for arrays. An example, illustrated in Fig. 7, is where the actual size of the array is controlled by user input. If the user enters a size from one to nine, the array will overflow as the upper bound of `index` is nine on line 13. However if user enters a size input of ten, the defect will be missed since nine is a legal upper bound. This missed defect occurs despite executing the exact same control path.

Our technique also is incomplete in that it can produce false alarms, signaled bugs that are not actual bugs. Symbolic relationships between different variables are not tracked and this can cause operations that narrow bounds to be missed. Consider the example shown in Fig. 8. The

```
1    unsigned int size;
2    unsigned int index;
3    int *array;
4
5    size = getchar();
6    if (size <= 0 || size > 10) exit();
7    array = (int *) calloc(size, sizeof(int));
8
9    /* initialize array */
10
11   index = getchar();
12   if (index < 0 || index > 9) exit();
13   y = array[index];
```

Fig. 7 Example of an unsound path

```
1    int a;
2    int b;
3    int x;
4    int array[5];
5
6    scanf(“%d”, &a);
7    b = a;
8    if (b < 0 || b >= 5) exit();
9    x = array[a];
```

Fig. 8 Example of a false alarm

variable a is assigned a value from input at line 6 and thus has a maximal upper bound. Variable a will still have the same maximal upper bound at the array reference at line 9 since a is not used in a check or reassigned. This will cause the checker to report an error. However, this is not an actual defect since b is assigned the same value at line 7 and is checked at line 8. The checks at line 8 will update the bounds of b appropriately but the bounds of a are unchanged since the checker does not store the fact that a and b are equal to one another.

It is important to note that the unsoundness and incompleteness are functions of the array checker itself and not a property of SUDS itself. It is possible to create defect detection tools that are sound (with respect to the given input) and complete (no false alarms). Since SUDS is a dynamic defect detection tool, it is not possible to create a checker that is completely sound (catch every defect) unless every single input combination is tested – an impossible goal for most programs.

4.2 Arithmetic Checker

The arithmetic checker is used to check other situations where input integer data could be used dangerously and possibly lead to software defects. Here is a list of checks employed by the arithmetic checker:

- Unconstrained input is used an addition, subtraction, and multiplication. This likely leads to overflow.
- Unconstrained size of a memory allocation.
- Divide by zero errors for division and modulus.

The internals of this checker are very similar to that of the array checker detailed in the previous section. One key difference is that only an integer shadow state table is needed since none of the checks involve pointers and arrays.

```
1    int num;
2    int size;
3    int *array;
4    printf(“How many elements? “);
5    scanf(“%d”, &num);
6    size = num * sizeof(int);
7    array = malloc(size);
```

Fig. 9 Example of an unconstrained input error

The integer shadow state table stores tainted integers with each entry containing the following five fields:

- *address* – Address of the variable (same as the index to the shadow state table), stored for debugging purposes.
- *name* – Name of the variable, only used for debugging and diagnostic purposes.
- *lower bound* – Lower bound, lowest possible value for the variable.
- *upper bound* – Upper bound, highest possible value for the variable.
- *initial upper bound* - Initial upper bound, same as the upper bound when introduced.

The last field (*initial upper bound*) is unique to this checker. It tracks the initial upper bound when the input variable is introduced. It is initialized to the upper bound upon creation and does not change in subsequent operations. For the two unconstrained checks, an error message is reported if the upper bound is equal to the initial upper bound since the input has not been constrained in any way. An example of how this can lead to an error is shown in Fig. 9. The value of `num` comes from input on line 5. It is unconstrained in the multiply operation on line 6. This can lead to overflow if a large value is input by the user. This can cause memory issues when this overflow value is subsequently used by `malloc` in line 7.

The lower and upper bounds are updated using the same rules as the array checker (see Tables 4, 5, and 6). For divide and modulus operations, the bounds of the divisor are checked to determine if zero is included in the range. If zero is in the range, an error is reported.

4.2.1 Limitations

This checker emphasizes simplicity and performance. As a consequence, error reports emitted by the checker could very well be false alarms. The two unconstrained checks are likely sources of defects but they have to be checked manually to determine if it is an actual defect. The divide-by-zero check could also result in a false alarm if the range includes the value zero but the value was explicitly excluded separately in the program using the not-equal (`!=`) operator.

4.3 String Checker

Another common source of security bugs is the improper use of the string library functions in C. Since the functions provide no checking, the responsibility resides with the programmer. To complicate matters, there is little consistency on how the different string functions operate. For instance, `strcpy` always copies a null character but `strncpy` will not copy the null character unless one is present within the specified limit.

Input strings are shadowed by state variables that hold the maximum possible size of the string. Like integers, strings from external input sources are considered to have an unbounded maximum size. Comparisons that test the length of an input string can decrease the maximum string length. When a string copying function is called, the maximum length of the string is checked to ensure it will fit in the destination. In addition, the shadow state for strings contains a null flag which is set if the string is guaranteed to contain a null character. The null flag is checked for all string functions that expect a null-terminated string. On string copies, the null flag is only set on the destination string if it is copied in a manner that guarantees that a null character is copied or added to the string.

An example of a defect involving strings is shown in Fig. 10. On line 5, there is a check to filter out strings that are greater than 16 characters. However, `strlen` does not count the null character. This means that if the source string is exactly 16 characters (not including the null character), it will pass the check though it contains 17 characters, including the null character. As a result, the null character does not get copied by the `strncpy` in line 6, creating a potentially dangerous `strcpy` on line 8 because the source is not null terminated. This type of problem is difficult to catch during testing since it requires a source string of exactly 16 characters. Also, the defect may not manifest in incorrect output when such an input is presented; this is likely the case if the character after the `temp` array happens to be a null.

```
1    char *bad_string_copy(char *src)
2    {
3      char *dest;
4      char temp[16];
5      if (strlen(src) > 16) return NULL;
6      strncpy(temp, src, 16);
7      dest = (char *) malloc(16);
8      strcpy(dest, temp);
9      return dest;
10   }
```

Fig. 10 Example of a string defect

4.3.1 Tracking Strings

The shadow state table employed by the string checker tracks additional state for all strings and character arrays (which are potentially used as strings). Entries in the table are indexed by their base address.

All strings and arrays in the program are tracked with five fields:

- *base address* - Base address of the string (also the index).
- *name* - Name of the array or pointer used.
- *actual size* - Stores the actual run-time size of the array and cannot change.
- *maximum string size* - Stores the maximum size of the string in the array. It refers to the largest possible size of a string that a user can supply. Strings that come from input have an initial maximum string size of infinity (`INT_MAX`), unless constrained. For arrays that do not contain strings, the maximum string size is equal to the actual size.
- *known null* - A flag that is true if the string is known to contain a null character. If it is false, it is not known whether a null character is present or not. During checking, it is assumed that the string is not null terminated if this flag is false.

Arrays and pointers are added to the table based on the rules listed in Table 8. For array declarations and string constants, the actual size and the maximum string size are the same since they do not hold strings from input. The known null flag is false for locally declared arrays since they are uninitialized upon declaration. Strings that come from the command line are considered input and are marked as having an infinite maximum string size since the user could have supplied a string of any length. In most cases, dynamically allocated arrays are processed identically to the creation of arrays declared at compile-time. One exception is when the size of the allocation is dependent on the size of another string. This case is described in the next section. The result of a pointer arithmetic operation results in copying the state of the pointer to the destination pointer.

Except for string function calls, the only operation that can modify the state of an existing pointer is when zero or null is assigned to an array element. This causes the known null flag to be set. The flag is also set when functions `bzero` and `memset` (to zero) are called. Local arrays are removed from the table when they die; dynamically allocated arrays are removed when they are freed.

Table 8 Adding arrays and pointers to the string checker shadow state table.

Program Construct	***Base Address***	***Actual Size***	***Maximum String Size***	***Known Null***
local character array declaration	base address of the array	size of the array	size of the array	false
global character array declaration	base address of the array	size of the array	size of the array	true
string constant	base address of the string	size of the string	size of the string	true
string from the command line (like `argv[2]`)	base address of the array	size of the string	`INT_MAX`	true
call to `malloc`/`calloc` (allocation size does not correspond to a string length)	base address of the allocation	size of the allocation	size of the allocation	false
call to `malloc`/`calloc` (allocation size corresponds to a string length)	base address of the allocation	size of the allocation	maximum string size of string based on the string length	false
`q` = `p` + `i` or `p` – `i` where `p` is a pointer and `i` is an integer	The entry for `p` is copied to a new entry for `q`.			

4.3.2 Tracking String Lengths

In order to properly adjust the maximum size of a string, it is necessary to track integers that store string lengths. A shadow state table is used to store any integer that is storing a string length. Each entry in the shadow state table stores the following:

- *name* - name of the integer
- *string address* - starting address of the corresponding string.
- *offset* - difference between the value in the integer and the actual string length.

This offset field is important as our approach includes the null character in the length of a string while the `strlen` function does not. Therefore, the initial offset for a variable assigned from `strlen` is -1. Addition and subtraction operations on the string length adjust the offset appropriately. For example, adding one to a `strlen` result to account for the null character will result in an offset of zero. While offsets of -1 and 0 are most common, variables occasionally store different offsets during concatenation operations.

The maximum length of the string can be reduced when a string length is used in a comparison. Let *n* be a variable holding a string length for string *s* with offset *d*. Let *c* be an integer constant (or any integer variable not holding a string length). Assume there is a

comparison ($n \leq c$). If ($n \leq c$) is true, the maximum size of string *s* is adjusted to $c + d$ unless the maximum size is already smaller. If ($n \leq c$) is false, no adjustment is made to *s* since there is no restriction on the maximum size.

If an integer *n* holding a string length for a string *s* is used as an argument to size to `malloc` or `calloc`, the maximum string size of the newly allocated string will be set using the maximum string size of *s*, adjusted by the offset of *n*. This is commonly done before a string copy to ensure the destination has enough space to hold the source string.

4.3.3 Processing String Functions

Table 9 shows how some common string library functions are handled. In the table, the variables *s* (source string), *d* (destination string), and *fmt* (format string) refer to strings. The variable *n* is an integer and refers to the size of a copy. The shadow state variables *actualSize*, *maxStringSize*, and *knownNull* are represented using structure notation such as *s.maxStringSize*. For functions that copy strings into a destination buffer, the size of the destination is conservatively determined by taking the maximum of the actual size and the maximum string size. For brevity, we use *SIZE(s) = MAX(s.actualSize, s.maxStringSize)* to represent the size of destination buffers.

Rules 1 and 2 illustrate how string copies are handled. In `strcpy`, the source must be null terminated and the size of the maximum string size must fit in the destination. If both the null check and size check pass, the destination will then have *knownNull* set to true and *maxStringSize* equal to that of the source. In `strncpy`, an additional check ensures that the destination size is less than the supplied size (*n*) parameter. This is done regardless of the size of the source string since nulls are padded at the end if the source is smaller. The *maxStringSize* is set to be the smaller of *n* and *maxStringSize* of the source. The *knownNull* flag is set true only if the entire string is copied. The `strcat` functions (Rules 3 and 4) are handled similarly to `strcpy`. The key differences are that the destination must be null terminated and the run-time size of the destination string is subtracted from the size comparison.

When getting a string from input, using `gets` (Rule 5) always results in an error since there is no limit placed on the number of characters that can be copied. If the safer `fgets` is used (Rule 6), the size parameter *n* is compared to the size of the destination buffer. For `scanf` and similar functions (Rule 7), the width is extracted from the format string *fmt*. An error is reported if no width is specified. Otherwise, the width is compared against the size of the destination buffer.

Table 9 Representative string function rules

	String Function	***Checks (Assertions)***	***Modified State***
1	`strcpy`(*d*, *s*)	*s.knownNull == TRUE* *s.maxStringSize ≤ SIZE(d)*	*d.maxStringSize = s.maxStringSize* *d.knownNull = TRUE*
2	`strncpy`(*d*, *s*, *n*)	*s.knownNull == TRUE* *n ≤ SIZE(d)*	*d.maxStringSize = MIN(s.maxStringSize, n)* *d.knownNull = s.maxStringSize ≤ n*
3	`strcat`(*d*, *s*)	*s.knownNull == TRUE* *d.knownNull == TRUE* *s.maxStringSize ≤ SIZE(d) - strlen(d)*	*d.maxStringSize = s.maxStringSize + strlen(d)* *d.knownNull = TRUE*
4	`strncat`(*d*, *s*, *n*)	*s.knownNull == TRUE* *d.knownNull == TRUE* *MIN(n + 1, s.maxStringSize) ≤ SIZE(d) - strlen(d)*	*d.maxStringSize = MIN(n + 1, s.maxStringSize) + strlen(d)* *d.knownNull = TRUE*
5	`gets`(*d*)	Automatic error!	
6	`fgets`(*d*, *n*, *stream*)	*n ≤ SIZE(d)*	*d.maxStringSize = n* *d.knownNull = TRUE*
7	`scanf`(*fmt*, *d*) Also: `fscanf`, `sscanf`	get *width* from *fmt* *width ≤ SIZE(d)*	*d.maxStringSize = width* *d.knownNull = TRUE*
8	`sprintf`(*d*, *fmt*, *s*)	*s.knownNull == TRUE* sum of all source strings ≤ *SIZE(d)*	*d.knownNull = TRUE*
9	`snprintf`(*d*, *n*, *fmt*, *s*)	*s.knownNull == TRUE* *n ≤ SIZE(d)*	*d.maxStringSize = n* *d.knownNull =* *(*sum of all source strings ≤ *SIZE(d))*
10	*d* = `strdup`(*s*)	*s.knownNull == TRUE*	*d.actualSize = s.maxStringSize* *d.maxStringSize = s.maxStringSize* *d.knownNull = TRUE*
11	String functions that only read strings such as `strcmp` and `atoi`	Check that all input source strings are null terminated.	

Legend:
- *s, d, fmt* are strings.
- *n* is an integer and refers to a parameter that restricts the number of characters written into a destination buffer.
- The macro *SIZE(s)* is equal to *MAX(s.actualSize, s.maxStringSize)*.

For `fgets` and `scanf`, the destination is guaranteed to be null terminated so *knownNull* is set to true.

The function `sprintf` (Rule 8) is implemented to ensure that the sum of the maximum sizes of all source strings does not exceed the destination size. In `snprintf` (Rule 9), the sum of the maximum sizes of the source strings is compared to the parameter *n*. If the sum is greater than *n*,

1	char buf0[12];	buf0: maxStringSize = 12, knownNull = FALSE
2	char *buf1;	
3	char buf2[18];	buf2: maxStringSize = 18, knownNull = FALSE
4	char *p;	
5		
6	strncpy(buf0, argv[1], 12);	buf0: maxStringSize = 12, knownNull = FALSE
7	buf1 = strdup(argv[2]);	buf1: maxStringSize = ∞, knownNull = TRUE
8		
9	if (value) {	
10	p = buf0 + 1;	p: maxStringSize = 12, knownNull = FALSE
11	strcpy(buf2, p);	p.knownNull == FALSE → ERROR
12	}	
13	else if (strlen(buf1) <= 6){	buf1: maxStringSize = 7, knownNull = TRUE
14	buf0[12] = 0;	buf0: maxStringSize = 12, knownNull = TRUE
15	sprintf(buf2, "%s%s", buf0, buf1);	(buf0.maxStringSize + buf1.maxStringSize) >
16	}	(buf2.maxStringSize) → 19 > 18 → ERROR

Fig. 11 Example of detecting string defects

then there is no guarantee that a null will be copied and *knownNull* is set to false. The function `strdup` (Rule 10) copies *maxStringSize* and *knownNull* from the source string to newly created destination string. String functions that only read strings check to see if all input strings are properly null terminated (Rule 11).

4.3.4 String Example

A detailed example illustrating how string defects can be found is shown in Fig. 11. The two buffers `buf0` and `buf2` have an initial maximum size equal to their static sizes. In line 6, the input value `argv[1]` is copied into `buf0` using `strncpy`. While the specified size of 12 does not cause an overflow, the *knownNull* flag for `buf0` remains false since a null would not have been copied if `argv[1]` has at least 12 characters. The `strdup` in line 7 copies the state values of `argv[2]` into `buf1`. If `value` is true at line 9, execution will continue to line 10 where the pointer `p` is assigned to point to the second element of `buf0`. This causes `p` to have the same shadow state `buf0`. When `p` is used in the `strcpy`, an error is signaled because `p` which is pointing to the same string as `buf0` may not be null terminated.

In the case where `value` is false at line 9, a comparison is made based on the length of `buf1`. Assuming it is less than or equal to 6, control will be taken to line 14 and the maximum size of `buf1` will be restricted to 7 (6 plus 1 for the null character that `strlen` does not count). The *knownNull* flag is set for `buf0` in line 14. In line 15, an error results because the sum of the maximum sizes of the two source buffers (19) can exceed the size of the destination (18).

4.3.5 Limitations

As with the other two checkers, missed bugs and false alarms are possible. Missed bugs are due to primarily two reasons. First, the concatenation checks (`strcat` and `strncat`) rely on the run-time length of the string currently in the destination buffer. The size of the string may be a factor if a defect is detected or not. Second, the checker only looks at the string functions. It is not able to handle homemade string functions that perform operations using pointer operations.

False alarms are possible due to limitations in the string length processing. As with the integers in the array checker, some symbolic relationships between variables may be missed resulting in lost opportunities to narrow the maximum string size of a string. It is also possible for the *knownNull* to be set to false incorrectly for similar reasons.

5 Static Analysis

The static analysis phase of SUDS consists of traditional compiler analyses and two additional analyses (taint analysis and program slicing) that allow the user to focus the instrumentation on particular types of defect and/or particular statements within a program.

Within the static analysis phase, these subphases are executed in the following order:

- **Dummy variable creation:** Dynamic memory is modeled by call-site. For each static call to a system function that allocates memory, a dummy variable is created and the return value points to the dummy variable.
- **Control flow graph:** The control flow graph is created intraprocedurally. Basic blocks that are not reachable from the starting point are excluded from future subphases.
- **Call graph:** A complete call graph is created including calls made by function pointers. The call graph and pointer analysis phase are performed together until both algorithms converge. Functions not reachable from `main` are excluded from future subphases.
- **Pointer analysis:** The interprocedural pointer analysis developed by Hind et al. (1999) is used to compute the set of variables that a pointer can point to. Both flow-sensitive and flow-insensitive versions of the algorithm are provided (selected by a command-line switch). Structs, unions, arrays, and dummy heap variables are treated as single variables. Updates to such variables are done in a weak fashion without killing prior relationships.
- **Data-flow analysis:** Data-flow analysis is used to determine the set of definitions generated, killed, and used by each statement. The analysis uses a standard data-flow

algorithm that iterates until steady state. For function calls, live definitions are propagated from the caller to the callee function on a parameter by parameter basis. At return points, definitions that refer to global variables are propagated to all callers. Other definitions are only propagated back to the caller if the definition refers to a variable that had at least one live definition prior to the call. Though the algorithm is not truly context-sensitive, this last rule restricts the flow of definitions from a call site to a completely different return point. Any definitions live in the caller just prior to the call are also considered live just after the call.

- **Tainted data propagation:** Determines which variables are tainted. See Section 5.1.
- **Program slicing:** Determines which statements and variables influence an operation or a particular set of operations. See Section 5.2.

The output of these last two phases can be used by the instrumentation phase to refine the added instrumentation. This allows the instrumented program to focus on particular types of defects and statements, decreasing the run-time overhead. This is described in more detail in Section 5.3.

5.1 Detecting Tainted Data

The tainted data propagation phase detects the set of variables that are considered tainted. The definition of what is considered tainted initially is easily altered within the source code. Here are some examples of what different users might consider to be tainted:

- Any data that comes from input
- Any data that comes from the network
- Data that comes from a particular system call
- Data returned from a function call
- Data produced by a particular statement

After the initial marking of tainted data, the tainted data propagation algorithm will mark variables tainted that are dependent on the initial tainted data.

The algorithm is similar to constant propagation, a compiler optimization that replaces variables that are known to be constant. The key difference is that SUDS propagates attributes instead of values. A variable definition will have one of two attributes: tainted or untainted. The algorithm proceeds iteratively and processes each statement individually during each iteration.

For arithmetic and copy operations, all definitions will be marked tainted if any of the uses are tainted. The definitions for bitwise and relational operations are always considered not tainted. The analysis is interprocedural in a context-insensitive fashion. For function calls and returns, the tainted attribute is propagated from argument to parameter and from return statements to return values.

Once a definition enters the tainted state, it remains in that state for the duration of the algorithm. The algorithm continues to iterate until steady state (no definitions enter the tainted state). In the worst case, the algorithm takes O(*ds*) time where *d* is the number of definitions and *s* is the number of statements. In practice, the algorithm takes less time, typically only needing between 5-15 iterations. The result of this phase is that definitions are either marked as tainted or untainted. In addition, statements that generate a tainted definition are marked as tainted. This information can be used by the instrumentation engine to aid in the instrumentation process.

5.2 Program Slicing

Program slicing (Weiser 1981) can be used to determine the statements that influence the operation of a particular operation. Like the tainted propagation algorithm, a definition can be either in one of two states: in the slice or not in the slice.

A backwards slicing algorithm is employed starting with the slicing criterion. The slicing criterion will differ depending on how it is used. Some examples:

- All array references
- All pointer dereferences
- All string function calls or a particular string function call
- A particular statement
- All statements within a particular function

By default, all definitions that are used in the statement(s) in the criterion are added to the slice. It is also possible to select only some of the uses to be in the slice. For instance, there are two sets of uses for a pointer dereference: one set for the pointer being dereferenced and one set for the memory locations that are potentially being pointed to by the pointer. A user may only want one of the two sets to be included in the slicing criterion.

After the initial slicing definitions are marked, the slicing algorithm proceeds iteratively. If any definition generated by a statement is in the slice, then all definitions used by the statement

are added to the slice. Indirect uses from control statements are also added to the slice by default: if any statement in a control statement is in the slice, then the control statement and its uses are added to the slice. The user can exclude indirect uses by use of a command-line switch. Since the resulting slice is only used for analysis purposes, no attempt is made to keep the slice an executable program. The algorithm is interprocedural and context-insensitive.

Once a definition is added to the slice, it remains in the slice for the duration of the algorithm. The algorithm continues to iterate until steady state (no definitions enter the slice). In the worst case, the algorithm takes O(*ds*) time where *d* is the number of definitions and *s* is the number of statements. The result of this phase is a list of definitions and statements in the slice.

5.3 Using Static Analysis Results During Instrumentation

The instrumentation engine can use the results of the static analysis phases to focus the instrumentation. If both taint propagation and program slicing are used, then an interesting statement is one that is both tainted and in the slice. During the instrumentation phase, instrumentation calls are added as normal for interesting statements. However for statements that are not interesting, a user may decide to call a different instrumentation routine or omit the call altogether, depending on the instrumentation model. Some checkers may only use tainted data, others may only use the slicing algorithm, and some may use neither. In the latter case, the static analysis phase is not needed and subsequently not run.

5.4 Static Analysis Example

To illustrate how static analysis can be used to focus the instrumentation engine, we describe how the tainted propagation and program slicing algorithms are used in conjunction with the input array checker described in Section 4.1.

In the tainted propagation algorithm, variables that directly come from a system function that prompts for input or obtains data from an input source such as a file or network (examples: `scanf`, `getc`, `read`) are marked as tainted. In addition, we conservatively include return values from string conversion functions such as `atoi` since many integers from input start out as strings and then get converted to integers (integers passed in as command line arguments are a good example). The initial slicing criterion is the set of statements that contain array references

or pointer dereferences since that is what is being checked. We exclude indirect uses from control statements when performing the slicing.

A statement is considered interesting if it is both tainted (potentially derived from input) and in the slice (potentially could be used in an array reference or pointer dereference). Since indirect uses are excluded from the slice, a statement that compares integers is marked as interesting if it is tainted regardless if it is in the slice or not. Integer instrumentation is only applied to interesting integer statements. Statements that operate on integers that are deemed not interesting do not need instrumentation. Since only integers are analyzed, none of the instrumentation used in the management of the array shadow state table is removed. Left as future work, it would be possible to extend these analyses to remove unnecessary array instrumentation too

The primary benefit of focusing is the instrumentation for the input checker is to reduce the amount of run-time overhead. Results of applying the static analyses described in this section can be found in Section 6.3.

5.5 SUDS as a Static Analysis Tool

The static analysis phases of SUDS can also be used to create static analysis tools. Phases that attempt to detect defects in the code could be added to SUDS taking advantage of the existing analyses. Since SUDS has both static analysis and dynamic instrumentation capabilities, it is possible to create powerful tools that combine static and dynamic techniques. SUDS could also be used to create other types of static analysis tools. As an example, we implemented a phase that counts the number of paths in a function (Larson 2009). Using the slicing phase provided by SUDS, we analyzed the effect of slicing has the number of paths through a program.

6 Results

To demonstrate the effectiveness of SUDS, 18 different programs were used as test subjects for our three checkers as shown in Table 10. Four of the programs (*anagram*, *ft*, *ks*, and *yacr2*) are from the Pointer-Intensive Benchmark Suite (1995). Ten programs are from the SIR repository (Do et al. 2004), specifically the seven Siemens benchmarks (*printtokens, printtokens2, replace, schedule, schedule2, tcas,* and *totinfo*) and the programs *flex, gzip*, and *space*. These programs each include several versions that contain seeded defects. The remaining four programs include a calculator (*bc*), an FTP server (*betaftpd*) and two web servers (*ghttpd* and *thttpd*). Table 10 also

Table 10 Programs used as test subjects

Program	Description	Lines	Coverage
anagram	anagram generator	354	90%
bc	calculator	5,457	52%
betaftpd	file transfer daemon	2,949	61%
flex	lexical analyzer	8,830	88%
ft	spanning tree	1,105	67%
ghttpd	web server	608	58%
gzip	compression utility	5,232	65%
ks	graph partitioning	552	99%
printtokens	lexical analyzer	477	96%
printtokens2	lexical analyzer	402	100%
replace	pattern matching / substitution	514	96%
schedule	priority scheduler	295	99%
schedule2	priority scheduler	298	99%
space	specialized interpreter	6,200	92%
tcas	aircraft collision avoidance system	136	99%
thttpd	web server	6,380	41%
totinfo	statistics calculator	346	95%
yacr2	channel router	2,591	88%

includes the number of source code lines as computed using SLOCcount (2004) and statement coverage of all tests run using gcov.

Results for defect detection, presented in Section 6.1, include defects found in any test or, in the case of the SIR tests, in any version. A defect is only counted once even if it appears in multiple tests or versions (for the SIR tests). For the tests in the SIR repository and the Pointer-Intensive Benchmark Suite, we used the provided test suites to exercise the code. The other programs (*betaftpd*, *ghttpd*, *thttpd*, and *bc*) were primarily tested manually and had low code coverage (41-61%). If a defect was detected by one of our checkers, it was manually inspected to make sure that it indeed was an actual defect or a false alarm.

The performance results, outlined in Section 6.2, were taken using a representative and sufficiently long test input. For the SIR tests that had multiple versions, a representative non-seeded version was used. The same version and input was used throughout the experiments. We did not use the program *ghttpd* for performance testing since it forked off multiple processes. We also did not use *tcas* or *totinfo* which executed too fast, regardless of the input, to get meaningful data. All programs were compiled using GCC with an -O4 optimization level and were run on a 3.2 GHz Xeon processor with 4 GB of RAM. For comparison purposes, we also ran the benchmarks with Valgrind (Nethercote and Seward 2007), a popular dynamic tool that is used to catch memory related errors.

Table 11 Defects Detected

	Defects detected			False alarms detected		
	array	arith	string	array	arith	string
anagram	0	0	1	0	0	0
bc	0	0	0	1	0	0
betaftpd	0	0	2	0	0	1
flex	0	0	2	1	0	3
ft	0	2	0	0	0	0
ghttpd	0	0	4	0	0	1
gzip	0	0	2	0	0	0
ks	2	0	0	0	0	0
printtokens	0	0	0	0	0	0
printtokens2	0	0	0	0	0	0
replace	0	0	1	0	0	0
schedule	0	1	0	0	0	0
schedule2	1	1	0	0	0	0
space	0	0	1	0	1	0
tcas	1	1	0	0	0	0
thttpd	0	0	0	0	0	1
totinfo	0	2	0	0	0	0
yacr2	0	2	0	1	0	0
TOTAL	**4**	**9**	**13**	**3**	**1**	**6**
	TOTAL DEFECTS: 26			**TOTAL FALSE ALARMS: 10**		

6.1 Defects Detected

Using our three checkers, we were able to detect 26 defects in the 18 programs, as shown in Table 11. The string checker found 13 defects, the arithmetic checker detected 9 defects, and the array checker accounted for 4 defects. There were also 10 false alarms, 6 of them coming from the string checker.

Of the 13 string checker defects, 8 were due to string copy operations that exceeded the bounds of the destination buffer. Two defects were the result of a string not being properly null terminated. Two other defects were due to using `fgets` and `gets` improperly; simply using `gets` is a defect. The last defect was due to using uninitialized data to guide a loop. The six false alarms were quite different in how they were triggered. Two cases were due to manual string copy operations that caused the checker to think that a string was not null terminated when it was. Two other false alarms were due to relationships between pointers that are not tracked. Another false alarm occurred when the size of the destination buffer was based on the sum of two string lengths – our checker does not support the addition of two string length variables. Finally, our checker always assumes that the null character is in the last position; this assumption led to another false alarm.

Seven of the nine bugs reported by the arithmetic checker were due to arithmetic overflows. The other two defects were due to use of a negative divisor in a modulus operation and a dangerous use of `malloc`. The lone false alarm was due to an `atoi` conversion. The checker assumes that integers from string conversion functions such as `atoi` come from input and start unconstrained. In this particular case, the number of digits in the string supplied to `atoi` was properly constrained such that arithmetic overflow could not occur.

The array checker detected four array overflow bugs that were related to input. In all of these cases, the indices were not constrained prior to the check. In two of the three false alarms, the limits were not reduced when casting from a larger-sized integer to a smaller-size integer. Our checker currently does not handle this situation and can be addressed by making appropriate modifications to the checker. The third false alarm is due to reading the input file twice. It is read once to set the array sizes and a second time to initialize the array values. Our tool has no mechanism for determining that the input is actually constrained when it is read the second time.

Valgrind found 20 defects using the programs, not including memory leaks. Of the 20 defects, 19 were not detected by any of the SUDS checkers. This clearly indicates the checkers for SUDS and Valgrind are complimentary. Most of the errors detected by Valgrind were either using uninitialized values or illegal pointer dereferences. The three checkers do not specifically look for uninitialized values like Valgrind. The one bug that was detected both by Valgrind and our checker was the use of an uninitialized string. The string checker caught this particular bug because the uninitialized string was used in a string function and was not null terminated. The pointer dereference errors mostly occurred in operations that are in loops or sections of code that have several pointer operations. The string checker would miss such bugs as it looks primarily at the string functions. The array checker would only catch these bugs if the pointer arithmetic operations used input data.

On the other hand, Valgrind did not find 25 of the bugs we found. First of all, Valgrind does not check for arithmetic overflow so it is not expected it missed those bugs. In the other cases, there was not a test case that exposed the particular fault. For instance, to find a string copy bug, it is necessary to execute the string copy with a string that is longer than the destination in order for Valgrind to find it. Since our string checker uses ranges, it is not necessary to have a precise test case to expose the bug - the primary advantage of using our input checkers.

Table 12 Run-time performance results

	Base line (secs)	array (unopt)		array (opt)		arith		string		Valgrind	
		Time (secs)	Ratio	Time (secs)	Ratio	Time (secs)	Ratio	Time (secs)	Ratio	Time (secs)	Ratio
anagram	0.06	3.36	56.0	2.65	44.2	2.10	35.0	2.25	37.5	2.95	49.2
bc	0.40	10.44	26.1	9.38	23.5	6.70	16.8	7.65	19.1	17.62	44.1
betaftpd	0.04	0.54	13.5	0.49	12.3	0.42	10.5	0.48	12.0	2.41	60.3
flex	0.01	0.44	44.0	0.34	34.0	0.38	38.0	0.35	35.0	1.12	112.0
ft	0.17	7.24	42.6	4.80	28.2	5.37	31.6	4.56	26.8	5.23	30.8
gzip	0.01	1.02	102.0	0.18	18.0	0.83	83.0	1.02	102.0	0.82	82.0
ks	0.04	4.34	108.5	3.68	92.0	3.59	89.8	2.47	61.8	2.18	54.5
printtokens	0.01	0.28	28.0	0.06	6.0	0.25	25.0	0.23	23.0	0.67	67.0
printtokens2	0.01	0.36	36.0	0.18	18.0	0.36	36.0	0.34	34.0	0.78	78.0
replace	0.01	0.34	34.0	0.12	12.0	0.28	28.0	0.25	25.0	0.91	91.0
schedule	0.02	0.10	5.0	0.06	3.0	0.07	3.5	0.08	4.0	2.42	121.0
schedule2	0.03	0.28	9.3	0.20	6.7	0.21	7.0	0.21	7.0	4.01	133.7
space	0.05	0.12	2.4	0.10	2.0	0.10	2.0	0.10	2.0	5.26	105.2
thttpd	0.14	2.00	14.3	1.53	10.9	1.46	10.4	1.32	9.4	4.24	30.3
yacr2	0.07	20.10	287.1	12.18	174.0	19.01	271.6	12.29	175.6	4.17	59.6
		Mean	53.9	Mean	32.3	Mean	45.9	Mean	38.3	Mean	74.6
		Max	287.1	Max	174.0	Max	271.6	Max	175.6	Max	133.7
		Min	2.4	Min	2.0	Min	2.0	Min	2.0	Min	30.3

Tests in the SIR repository contain seeded faults. The SUDS checkers and Valgrind missed most of these seeded faults as many of these faults are program logic errors (incorrect output for a particular input). This is expected as the SUDS checkers and Valgrind are looking for common programming mistakes that can apply to any program. However, SUDS can be used to create checkers that look for errors that are specific to a particular program.

6.2 Performance Results

To analyze performance, we compared the run-time performance of the three checkers with an uninstrumented baseline version and with Valgrind. For the array checker, we used two versions: an unoptimized version and an optimized version that removed unnecessary instrumentation based on the static analysis phases of SUDS. The results are shown in Table 12. The baseline column shows how fast the program runs, in seconds, without any instrumentation. For each of the checkers and Valgrind, the running time and ratio with respect to the baseline are displayed.

The amount of slow-down experienced by the checkers varied more by the program than by the type of checker. Except for *gzip* and *yacr2*, the arithmetic and string checker had similar performance slowdowns with average slowdowns of 45.9x and 38.3x respectively. Not

surprisingly, the unoptimized array checker was the slowest with an average slowdown of 53.9x while the optimized array checker was the fastest exhibiting an average slowdown of 32.3x. The array checker is slower because it requires heavy instrumentation for both arrays and integers. The string checker tracks information for arrays but only tracks the relatively few integers that hold string lengths. The arithmetic checker only requires instrumentation for integers. The optimized array checker reduces the amount of instrumentation and is discussed in more detail in Section 6.3.

The programs *schedule, schedule2,* and *space* exhibited the least amount of slowdown in all three of the checkers, running two to nine times as slow as the uninstrumented versions. On the other end of the spectrum, *gzip* and *ks* saw their run time increase 80-100 times and *yacr2* slowed down by a factor of over 270x for both the array (unoptimized) and arithmetic checkers. There was more disparity between the three checkers with these slower running programs. The reason for the major slowdown is that these programs have many instrumentation calls in processing loops that not only take time to process but also prevent the compiler from optimizing the code within these loops.

When comparing the performance results to that of Valgrind, our checkers suffer a similar order of magnitude of slowdown for most of the programs. In the cases where our checkers exhibited the least amount of slowdown, Valgrind had very high ratios. This is partly due that Valgrind requires some start-up time to set up its memory checking. The amount of set-up in our checkers is minimal. Conversely, Valgrind was much faster in the programs that saw the most slowdown with our checkers. We hypothesize that this is due to the fact that since Valgrind operates at the assembly-code level, it can process an optimized executable produced by the compiler. On the other hand, SUDS adds instrumentation calls before the compiler optimizes the code. This prevents many compiler optimizations since the compiler must be conservative when optimizing code across the unknown instrumentation calls.

6.3 Performance Optimizations

When static analyses are used to eliminate unnecessary instrumentation for the array checker, the performance improved by 36% on average. The program *gzip* saw the biggest improvement going from a slowdown of 102x to 18x, an improvement of 82%. Both print token programs and *replace* saw increases of over 50%. Programs that exhibited less slowdown in the unoptimized

Table 13 Breakdown of tainted and dangerous integers

	Tainted	In Slice	Interesting (Tainted & In Slice)
anagram	18.8%	70.1%	12.5%
bc	42.9%	81.0%	35.6%
betaftpd	35.6%	73.1%	28.5%
flex	28.9%	83.8%	23.7%
ft	23.9%	51.0%	12.6%
gzip	0.02%	76.2%	0.02%
ks	29.6%	74.7%	23.8%
printtokens	0.0%	83.4%	0.0%
printtokens2	45.0%	71.1%	29.9%
replace	0.0%	72.1%	0.0%
schedule	22.2%	55.6%	16.0%
schedule2	23.6%	50.0%	20.8%
space	14.2%	71.8%	10.7%
thttpd	40.5%	80.3%	29.1%
yacr2	26.4%	81.4%	20.6%
Average	**23.4%**	**71.7%**	**17.6%**
Maximum	**45.0%**	**83.8%**	**35.6%**
Minimum	**0.0%**	**50.0%**	**0.0%**

version tended to have little improvement: *betaftpd* went from 13.5x to 12.3x and *space* went from 2.4x to only 2.0x.

To better explain these results, Table 13 shows the breakdown of how many integer definitions (from a compiler point of view) are tainted (come from input data) and how many are used in the slice (lead to an array reference or pointer dereference). The last column shows the percentage of interesting statements – statements that are both tainted and in the slice. On average, 17.6% of integers are both tainted and in the slice. This means that the remaining 82.4% of the integers do not require any instrumentation. In two of the programs (*printtokens* and *replace*), none of the integers were tainted meaning that all of the instrumentation associated with integers was useless and not necessary. These benchmarks saw large performance improvements when the analyses were applied within SUDS.

Table 14 summarizes the number of instrumentation calls that were added statically (compile-time) and executed dynamically (run-time). The first column shows the number of statements in each program after the simplification process. By looking at the static number of instrumentation calls added, it is quite apparent where the slowdown is coming from. In each program, over half of the simple statements require instrumentation on average. In *printtokens2*, 1,018

Table 14 Instrumentation Counts

	Simple	Static Instrumentation Sites			Dynamic Instrumentation Sites		
	Stmts	Unopt.	Opt.	Impr	Unoptimized	Optimized	Impr
anagram	830	497	249	49.9%	47,932,765	33,817,372	29.4%
bc	14,503	9,773	3,719	61.9%	196,133,882	168,515,562	14.1%
betaftpd	4,782	3,038	2,440	19.7%	4,676,847	3,730,672	20.2%
flex	22,453	14,121	6,045	57.2%	7,968,578	6,296,356	21.0%
ft	1,886	937	475	49.3%	98,234,559	51,688,831	47.4%
gzip	12,566	7,419	2,146	71.1%	20,493,901	3,072,848	85.0%
ks	1,326	685	430	37.2%	93,319,877	74,045,793	20.7%
printtokens	1,745	1,314	295	77.5%	5,804,428	1,460,252	74.8%
printtokens2	1,048	1,018	900	11.6%	6,126,142	3,352,032	45.3%
replace	1,394	848	346	59.2%	6,692,472	2,523,927	62.3%
schedule	541	209	149	28.7%	1,460,728	767,920	47.4%
schedule2	571	244	175	28.3%	4,763,440	3,080,383	35.3%
space	11,652	5,509	4,114	25.3%	1,127,569	718,052	36.3%
thttpd	12,503	7,911	5,791	26.8%	23,149,256	16,000,043	30.9%
yacr2	5,612	3,481	1,849	46.9%	301,184,379	180,742,566	40.0%
			Avg	43.4%		Avg	40.7%
			Max	77.5%		Max	85.0%
			Min	11.6%		Min	14.1%

instrumentation calls were added to the 1,048 simplified statements. The static size of these programs is approaching double the size of the original program. Dynamically, this results in millions of additional function calls to instrumentation routines.

When unnecessary instrumentation is removed, the number of static instrumentation sites and dynamic instrumentation is greatly reduced. 43.4% of the static sites and 40.7% of the dynamic calls are removed from this process. The greatest improvement was shown by *gzip*: 71% of the static sites and 85% of the dynamic instrumentation calls were removed. It only had one integer that was interesting. The programs *printtokens* and *replace* also saw large decreases. These programs had no tainted integers. The remaining instrumentation is due to tracking arrays as the static analyses were not run with respect to arrays[8].

[8] One could argue that it is not even necessary to run the array checker for *printtokens* and *replace* since they had no tainted integers.

7 Related Work

7.1 Instrumentation Infrastructures

The most similar tool to SUDS is CIL (Necula et. al. 2002a). CIL is a source-to-source code translator outputting intermediate level code that is similar to the simplified code produced by SUDS. CIL is often used a pre-processor for other tools simplifying the analysis needed by these tools. However, CIL can also be extended to create a custom source-to-source translator in a similar way as SUDS. Unlike CIL, SUDS has support for adding instrumentation making it easy for user to add instrumentation to the outputted code. SUDS also has built-in data analyses, including taint analysis and slicing.

There are several tools used to instrument programs. ATOM (Srivastava and Eustace 1994) and Pin (Luk et al. 2005) are primarily used for performance analysis and gathering statistics about programs but could be used to detect lower-level software bugs such as invalid memory accesses. Valgrind (Nethercote and Seward 2007) is another infrastructure used to supervise programs and has specialized functionality for detecting software defects. The Phoenix compiler framework (Microsoft Corporation) allows developers to implement plug-ins that can insert code. The buffer overflow detector Marple (Le and Soffa 2008) and an efficient memory protection scheme using "baggy bounds" (Akritidis et. al. 2009) were both implemented using Phoenix. Unlike SUDS, these tools operate on either assembly code or low level intermediate code. Such tools typically do not require source code and thus do not suffer from a lack of source code from libraries. However, the bug-finding capabilities of these tools are limited to properties that can be expressed easily using assembly code semantics. Since SUDS operates close to the source code level, it is possible to create checkers for high-level programming constructs and program specific properties.

7.2 Software Defect Detection

Software defect detection tools are either dynamic (run-time analysis), static (compile-time analysis), or use both. Memory access errors are a popular target for dynamic defect detection. These tools detect memory bugs by keeping track of the state of dynamically allocated memory using a table to keep track of the state of memory. In Purify (Hastings and Joyce 1992), the table is implemented using a bitmap array making accesses fast. However, the limited amount of

information gained from a small number of bits restricts in the types of defects that can be detected. Jones and Kelly (1997) use an object table, similar to our shadow state table, to store state on which region of memory a pointer is allowed to point to. Haugh and Bishop (2003) check all of the interesting string library functions by comparing the allocated sizes of the arrays. This approach is similar to our string library maximum size checks. Their tool tracks coverage to ensure that each interesting string function is executed once. Our technique can potentially find more defects because it checks for proper null termination and array references. Lhee and Chapin (2002) intercept array references and checks to see if they exceed the bounds. Sizes of heap allocated arrays are stored in a table. Safe C (Austin et al. 1994) stores extra state with each pointer that stores the bounds of the object the pointer is referring to. Accesses are compared to the bounds to see if there is a defect. Insure++ (Parasoft Corporation 2006) is a commercially available product that detects memory errors dynamically.

A weakness of dynamic bug detection is that the effectiveness of the checker is dependent on the input. This weakness can be mitigated by running the checker on a test suite that obtains high coverage. The three checkers implemented in SUDS reduce this weakness further by modeling variables using ranges making it not necessary to have the precise data value to cause the failure.

In static software bug detection, one approach is to use symbolic execution (King 1976). In symbolic execution, the symbolic execution engine traverses all of the paths representing values derived from input symbolically. PREfix uses symbolic execution (Bush et. al. 2000) to detect memory faults. The array checker ARCHER (Xie et al. 2003) also employs symbolic execution. An extension that provides symbolic execution support (Anand et al. 2007) has been added to Java Pathfinder (Java Pathfinder, Visser et al. 2003).

Constraint systems are common in detecting string-related errors. Wagner et al. (2000) developed a system that collects constraints on buffers in C programs and then uses a constraint solver to determine if any of the constraints have been violated. CSSV (Dor et al. 2003) converts the program into an integer program and then analyzes the program for potential errors. HAMPI (Kieżun et al. 2009) is a solver for string constraints that can be used to find SQL injection vulnerabilities.

Another popular technique used in static bug detection is model checking (McMillian 1993). In model checking, the program is modeled as a finite state machine. The model checker then determines if there is a legal set of transitions that cause the program to enter an error state.

Examples of model checking systems include SPIN (Holzmann 1997), SLAM (Ball and Rajamani 2002), Java Pathfinder (Java Pathfinder, Visser et al. 2003), MOPS (Chen and Wagner 2002), and CMC (Musuvathi et al. 2002).

Commercially available products that perform static bug detection include CodeSonar (GrammaTech) and Coverity Static Analysis (Coverity). Employees at Coverity recently discussed (Bessey et al. 2010) some of the challenges of using static analysis in the real world with real customers.

A primary issue with static techniques is scalability. In symbolic execution, there are too many paths to simulate. In constraint solving, the resulting constraint system may be too large. In model checking, the finite state machine may either be too large to explore or too simple resulting in many false alarms. To address this issue, research groups have looked at combining static and dynamic bug detection techniques to create powerful tools. KLEE (Cadar et al. 2008) uses symbolic execution to find errors in UNIX utility programs. Upon finding an error, KLEE will generate actual concrete input values that will trigger the failure. Then, the original program can be run with the actual inputs to determine if the bug was real or a false positives. CCured (Necula et al. 2002b) uses a static verifier to prove as many dangerous operations safe as possible using a type system. Then instrumentation is added to catch any bugs for operations that cannot be proved safe. In work by Bodden et al. (2008), a static verification phase can prove that a run-time invariant or assertion will never result in an error condition. In absence of a proof, locations that where violations can occur are marked. Ringenburg and Grossman (2005) combine static analysis with run-time information to determine format-string attacks. White-lists, a set of allowable addresses, is generated at run-time by adding instrumentation calls. The instrumentation calls can either be added manually or automatically using an extension of CIL (Necula et al. 2002a). Bodik et al. (2000) use a lightweight static analysis to determine if run-time array bounds checks are redundant with earlier checks.

7.3 Dynamic Test Generation

Another way to combine static analysis is to generate test cases based on the analysis. DART (Godefroid et al. 2005), CUTE (Sen et al. 2005), PathCrawler (Williams et al. 2005), and Pex (Tillmann and de Halleux 2008) use a combination of symbolic and concrete execution. Instrumentation calls are added to perform symbolic execution that tracks constraints for integers

and pointers. Splat (Xu et al. 2008) targets string operations. Lengths of strings are represented symbolically. Symbolic execution is used to generate input tests. In Symbolic Pathfinder for Java (C. Păsăreanu et al. 2008), programs execute in concrete mode and can transition into symbolic mode based on specified conditions. Godefroid et al. (2008a) make a distinction between active property checking which uses symbolic values and passive property checking which uses concrete values. When implemented within a dynamic test generation environment, active property checkers were able to find defects in several Windows programs. Tikir and Hollingsworth (2002) describe an instrumentation technique for obtaining coverage for testing.

Fuzz testing (Forrester et al. 2000) has also benefited from run-time information and analysis. Fuzz testing randomly applies input values to different programs. It can find many errors in handling invalid input but struggles at producing legal complex inputs. Whitebox fuzz testing, starting with an input, will track control decisions made during the execution. Subsequent inputs are generated by negating one of the predicates used in making a control decision. The goal is to have tests that exercise most of the program paths. SAGE (Godefroid et al. 2008b) applies whitebox fuzzing to Windows programs and uses a sophisticated pointer reasoning algorithm (Elkarablieh et al. 2009). BuzzFuzz (Ganesh et al. 2009) uses whitebox testing for programs that have complex structured files as input. They use tainting to determine which control points in the program are influenced by the input file. SmartFuzz (Molnar et. al 2009) uses directed fuzz testing to find integer arithmetic and conversion errors.

Dynamic test generation and whitebox fuzzing are complimentary to the checkers used by SUDS. The goal of dynamic test generation is to generate interesting inputs. It is still necessary to check if the generated tests pass or fail – checkers and/or test oracles are still necessary. Dynamic test generation tends to focus on the control flow of the program in their attempt to cover all interesting paths. They still may miss bugs that depend on a specific data value. By using the range analyses provided by the array and string checkers, this issue can be minimized.

7.4 Taint Analysis

Taint analysis is a specialized information flow problem (Myers 1999, Myers and Liskov 2000). Support for taint analysis has been developed for many programming languages including Perl (perlsec 2009), Java (Haldar et al. 2005), and PHP (Nguyen-Tuong et al. 2005). Finding security property violations is a common use for taint analysis. Shankar et al. (2001) uses static taint

analysis to find string errors. Their approach is implemented by extending the C type system using type qualifiers. Dynamic taint analysis has been used by Newsome and Song (2005) to prevent unauthorized access or corruption and by Chess and West (2008) to detect potential misuse of strings. Dytan (Clause, Li, and Orso 2007b) is a dynamic taint analysis framework that works on the assembly code level. The framework has been used to find illegal memory accesses (Clause, Doudalis et al. 2007a) and to aid debugging by finding relevant inputs given a particular failure (Clause and Orso 2009).

7.5 Program Slicing

Program slicing (Weiser 1981) is a popular program analysis technique. Tip (1995) provides a survey on the different types of slicing techniques while Binkley et al. (2007) provide results from a large scale empirical study on slicing. CodeSurfer (GrammaTech) is a commercial product that performs program slicing. Program slicing is commonly used once a defect has been detected. Pan et al. (2006) use slicing metrics to classify defects. Memory slicing (Xin and Zhang 2009) is a dynamic slicing technique looks at the dependencies between memory locations. This results in fewer, but more precise, dependencies making it easier for debuggers to comprehend. Delta debugging (Zeller 2002) relies on the use of computing cause-effect chains which borrows many ideas from program slicing.

8 Conclusions and Future Work

SUDS is a powerful and flexible infrastructure for creating dynamic defect detection tools. An instrumentation interface allows users to create defect detection tools. Static analysis phases, tainted data propagation and program slicing, allow the tools to focus on types of defects, sources of data, and/or particular statements. Using three checkers that look for input-related defects, SUDS was able to find 26 defects across 18 programs. Using the static analyses, the run-time performance of the array checker improved by 36%.

SUDS, like all software tools, is constantly evolving. The next step for SUDS is to incorporate static defect detection capabilities by including technique(s) that are more sophisticated than data-flow analysis. One possibility is to create a new phase that generates constraints and uses a solver to detect defects. We also will explore the ability to plug in existing tools within the SUDS infrastructure. Another direction is to modify SUDS to that it can process other languages in addition to C. Since it is similar to C, C++ would be a likely first candidate.

The key challenges of analyzing C++ code is handling polymorphism and templates when instrumenting the program. Since the instrumentation is applied statically, it has to be generic with respect to dynamic binding. The instrumentation routines will be responsible for looking at the run-time type and applying the appropriate instrumentation based on that type. Other object-oriented languages such as Java and C# could be handled in a similar fashion.

The ultimate goal of SUDS is to explore new techniques that effectively combine static and dynamic approaches. As a first step in that work, we feel it is necessary to analyze what makes certain bugs hard to find (or prove the absence of bugs) statically and what makes certain bugs hard to find dynamically. By understanding what lies at the heart of static and dynamic techniques, we can take advantage of the best of both paradigms.

References

Akritidis, P., Costa, M., Castro, M., Hand, S.: Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors. In: 2009 USENIX Security Conference. (2009)

Anand, S., Păsăreanu, C., Visser, W.: JPF-SE: A Symbolic Execution Extension to Java Pathfinder. In: Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems. (2007)

Austin, T., Breach, S., Sohi, G.: Efficient Detection of All Pointer and Array Access Errors. In: Proceedings of the conference on Programming Language Design and Implementation. (1994)

Ball, T., and Rajamani, S. K.: The SLAM Project: Debugging System Software via Static Analysis. In: Proceedings of the symposium on Principles of Programming Languages. (2002)

Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, H., Kamsky, A., McPeak, S., Engler, D.: A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. In: Communications of the ACM. (2010)

Binkley, S., Gold, N., Harman, M.: An Empirical Study of Static Program Slice Size. In: ACM Transactions on Software Engineering and Methodology (TOSEM). (2007)

Bodden, E., Lam, P., Hendren, L.: Finding Programming Errors Earlier by Evaluating Runtime Monitors Ahead-of-Time. In: Proceedings of the International Symposium on Foundations of Software Engineering. (2008)

Bodik, R., Gupta, R., Sarkar, V.: ABCD: Eliminating Array Bounds Checks on Demand. In: Proceedings of the Conference on Programming Language Design and Implementation. (2000)

Bush, W., Pincus, J., Sielaff, D.: A static analyzer for finding dynamic programming errors. In: Software Practice and Experience. (2000)

Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation. (2008)

Chen, H., Wagner, D.: MOPS: an Infrastructure for Examining Security Properties of Software. In: Proceedings of the 9th ACM conference on Computer and Communications Security. (2002)

Chess, B.,West, J.: Dynamic taint propagation: Finding vulnerabilities without attacking. In: Information Security Technical Report, Vol. 13, No. 1. (2008)

Clause, J., Doudalis, I., Orso, A., Prvulovic, M.: Effective Memory Protection Using Dynamic Tainting. In: Proceedings of the international conference on Automated Software Engineering. (2007a)

Clause, J., Li, W., Orso, A.: Dytan: A Generic Dynamic Taint Analysis Framework. In: Proceedings of the International Symposium on Software Testing and Analysis. (2007b)

Clause, J., Orso, A.: PENUMBRA: Automatically Identifying Failure-Relevant Inputs Using Dynamic Tainting. In: Proceedings of the International Symposium on Software Testing and Analysis. (2009)

Coverity http://www.coverity.com

cTool http://ctool.sourceforge.net

Do, H., Elbaum, S., Rothermel, G.: Infrastructure support for controlled experimentation with software testing and regression testing techniques. In: Proceedings of the International Symposium on Empirical Software Engineering. (2004)
Dor, N., Rodeh, M., Sagiv, M.: CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C. In: Proceedings of the conference on Programming Language Design and Implementation. (2003)
DaCosta, D., Dahn, C., Mancoridis, S., Prevelakis, V.: Characterizing the 'Security Vulnerability Likelihood' of Software Functions. In: Proceedings of the International Conference on Software Maintenance. (2003)
Elkarablieh, B., Godefroid, P., and Levin, M.: Precise Pointer Reasoning for Dynamic Test Generation. In: Proceedings of the International Symposium on Software Testing and Analysis. (2009)
Forrester, J., Miller, B.: An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. In: Proceedings of the 4th USENIX Windows Systems Symposium. (2000).
Ganesh, V., Leek, T., Rinard, M.: Taint-based Directed Whitebox Fuzzing. In: Proceedings of the International Conference on Software Engineering. (2009).
Godefroid, P., Klarlund, N., Sen, K.: DART: Directed Automated Random Testing. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. (2005)
Godefroid, P., Levin, M., Molnar, D.: Active property checking In: Proceedings of the 8th ACM International Conference on Embedded Software. (2008a)
Godefroid, P., Levin, M., Molnar, D.: Automated Whitebox Fuzz Testing In: Proceedings of the 15th Annual Network and Distributed System Security Symposium. (2008b)
GrammaTech, Inc. http://www.grammatech.com
Hastings, R., Joyce, B: Purify: Fast Detection of Memory Leaks and Access Errors. In: Proceedings of the 1992 Winter Usenix Conference. (1992)
Haldar, V., Chandra, D., and Franz, M.: Dynamic Taint Propagation for Java. In: Proceedings of the 21st Annual Computer Security Applications Conference. (2005).
Haugh, E., Bishop, M.: Testing C Programs for Buffer Overflow Vulnerabilities. In: Proceedings of the 10th Network and Distributed System Security Symposium. (2003)
Hendren, L., Donawa, C., Emami, M., Gao, G., Justiani, Sridharan, B.: Designing the McCAT Compiler Based on a Family of Structured Intermediate Representations. In: Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing. (1992)
Hind, M., Burke, M., Carini, P., Choi, J.: Interprocedural Pointer Alias Analysis. In: ACM Transactions on Programming Languages and Systems. (1999)
Holzmann, G.: The Model Checker SPIN. In: IEEE Transactions of Software Engineering. (1997)
ISO/IEC 9899:1999: Programming languages — C (1999)
Java Pathfinder http://babelfish.arc.nasa.gov/trac/jpf
Jones, R., Kelly, P.: Backwards-compatible bounds checking for arrays and pointers in C programs. In: Proceedings of the 3rd International Workshop on Automated Debugging. (1997)
Kieżun, A., Ganesh, V., Guo, P., Hooimeijer, P., Ernst, M.: HAMPI: A Solver for String Constraints. In: Proceedings of the International Symposium on Software Testing and Analysis. (2009)
King, J.: Symbolic execution and program testing. In: Communications of the ACM. (1976)
Larson, E., Austin, T.: High Coverage Detection of Input-Related Security Faults. In: Proceedings of the 12th USENIX Security Symposium. (2003)
Larson, E: A Plethora of Paths. In: Proceedings of the 17th IEEE International Conference on Program Comprehension. (2009)
Le, W., Soffa, M. L.: Marple: a Demand-Driven Path-Sensitive Buffer Overflow Detector. In: Proceedings of the International Symposium on Foundations of Software Engineering. (2008)
Lhee, K., Chapin, S.: Type-Assisted Dynamic Buffer Overflow Detection. In: Proceedings of the 11th USENIX Security Symposium. (2002)
Luk, C., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V., Hazelwood, K.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In: Proceedings of the Conference on Programming Language Design and Implementation. (2005)
McMillan, K.: Symbolic Model Checking. (1993)
Merrill, J.: GENERIC and GIMPLE: A new tree representation for entire functions. In: Proceedings of the GCC Developers Summit. (2003)
Microsoft Corporation: Phoenix Compiler and Shared Source Common Language Infrastructure. http://research.microsoft.com/en-us/collaboration/focus/cs/phoenix.aspx

Molnar, D., Li, X., Wagner, D.: Dynamic Test Generation To Find Integer Bugs in x86 Binary Linux Programs. In: Proceedings of the 2009 USENIX Security Conference. (2009)
Musuvathi, M., Park, D., Chou, A., Engler, D., Dill, D. CMC: A Pragmatic Approach to Model Checking Real Code. In: Proceedings of the 5th symposium on Operating Systems Design and Implementation (2002).
Myers, A. C.: JFlow: practical mostly-static information flow control. In: Proceedings of the symposium on Principles of Programming Languages. (1999)
Myers, A. C., Liskov, B.: Protecting privacy using the decentralized label model In: ACM Transactions of Software Engineering Methodology, Vol. 9, No. 4. (2000)
Necula, G., McPeak, S., Rahul, P., Weimer, W.: CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In: Proceedings of the International Conference on Compiler Construction. (2002a)
Necula, G., McPeak, S., Weimer, W.: CCured: Type-Safe Retrofitting of Legacy Code. In: Proceedings of the Symposium on Principles of Programming Languages. (2002b)
Nethercote, N. and Seward, J.: Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In: Proceedings of the conference on Programming Language Design and Implementation. (2007)
Newsome, J., Song, D.: Dynamic Taint Analysis for Automatic Detection, Analysis and Signature Generation of Exploits on Commodity Software. In: Proceedings of the 12th Annual Network and Distributed System Security Symposium. (2005)
Nguyen-Tuong, A., Guarnieri, S., Greene, D., Shirley, J., Evans, D.: Automatically hardening web applications using precise tainting In: Proceedings of the 20th IFIP International Information Security Conference (2005)
Pan, K., Kim, S., Whitehead, E. J.: Bug classification using program slicing metrics In: Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation, pp. 31-42 (2006)
Parasoft Corporation: Automating C/C++ Runtime Error Detection with Parasoft Insure++ http://www.parasoft.com/jsp/printables/InsureWhitePaper.pdf
Păsăreanu, C., Mehlitz, P., Bushnell, D., Gundy-Burlet, K., Lowry, M., Person, S., Pape, M.: Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing NASA Software. In: Proceedings of the International Symposium on Software Testing and Analysis. (2008)
perlsec - Perl Security http://search.cpan.org/dist/perl/pod/perlsec.pod
Pointer-Intensive Benchmark Suite (1995) http://www.cs.wisc.edu/~austin/ptr-dist.html
Ringenburg, M., Grossman, D.: Preventing Format-String Attacks via Automatic and Efficient Dynamic Checking. In: Proceedings of the conference on Computer and Communications Security. (2005)
Sen, K., Marinov, D., Agha, G.: CUTE: A Concolic Unit Testing Engine for C. In: Proceedings of the Symposium on the Foundations of Software Engineering. (2005)
Shankar, U., Talwar, K., Foster, J., Wagner, D.: Detecting Format-String Vulnerabilities with Type Qualifiers In: Proceedings of the 10th USENIX Security Symposium. (2001)
SLOCcount http://www.dwheeler.com/sloccount
Srivastava, A., Eustace, A.: ATOM: A System for Building Customized Program Analysis Tools. In: Proceedings of the Conference on Programming Language Design and Implementation. (1994)
Tikir, M., Hollingsworth, J.: Efficient Instrumentation for Code Coverage Testing. In: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis. (2002)
Tillmann, N., de Halleux, J.: Pex – White Box Test Generation for .NET. In: Proceedings of the 2nd International Conference on Tests and Proofs. (2008)
Tip, F.: A Survey of Program Slicing Techniques. In: Journal of Programming Languages (1995).
Visser, W., Havelend, K., Brat, G., Park S., Lerda, F. Model Checking Programs. In: Automated Software Engineering Journal (2003).
Wagner, D., Foster, J., Brewer, E., Aiken, A. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In: Proceedings of the Network and Distributed System Security Symposium. (2000)
Weiser, M: Program slicing. In: Proceedings of the 5th International Conference on Software Engineering (1981)
Williams, N., Marre, B., Mouy, P., Roger, M.: PathCrawler: Automatic generation of path tests by combining static and dynamic analysis. In: Proceedings of Fifth European Dependable Computing Conference. (2005)
Xie, Y., Chou, A., Engler, D. ARCHER: Using Symbolic, Path-sensitive Analysis to Detect Memory Access Errors. In: Proceedings of the 9th European Software Engineering Conference held jointly with 11th international symposium on Foundations of Software Engineering. (2003)
Xin, B. and Zhang, X.: Memory Slicing. In: Proceedings of the International Symposium on Software Testing and Analysis. (2009)

Xu, R., Godefroid, P., Majumdar, R.: Testing for buffer overflows with length abstraction. In: Proceedings of the International Symposium on Software Testing and Analysis. (2008)
Zeller, A.: Isolating cause-effect chains from computer programs In: Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering, pp. 1-10. (2002)