SUDS: An Infrastructure for Creating Bug 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 bug 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 bug detection tools created with SUDS and could be expanded to find defects statically. 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 is improved when static analysis is used to eliminated unnecessary instrumentation.

1. Introduction

It is increasingly important in the world today to have correctly working software. Implementation errors, many due to using an inherently unsafe language like C, can be exploited by malicious users to execute arbitrary code and to gain access private data. Therefore it is necessary to have high quality tools that can detect bugs before software is released.

Software bug detection can either be performed statically or dynamically. Static techniques allow the software developer to prove that a program correctly satisfies a given property. However, the number of possible states that must be searched to obtain such a proof is often extremely large even for a simple property. As a result, the verification of a property can be infeasible. Abstraction of code not relevant to the property can greatly reduce the search space but does not necessarily make searching feasible. Further abstractions that limit the scope or simplify the problem in a manner are often necessary. This typically not only increases the number of bugs detected but also the number of false bug reports, possibly requiring significant manual effort.

Other tools look for software errors at run-time or dynamically. While their effectiveness is obviously restricted by the particular input set, they are effective at finding bugs on the common paths of execution. Unlike static techniques, dynamic techniques do not need to limit the scope of the search. Dynamic checkers can find bugs that span multiple function boundaries, library functions, or even process boundaries. The largest drawback to dynamic bug detection is its dependence on the input. Faults will only be detected if a test is run that exposes the bug. Another downside is the performance overhead associated when running the program. This is particularly true for techniques that track additional state in order to find more defects.

This paper describes SUDS, an infrastructure designed to create a wide range of software bug detection tools. It contains support for both static analysis and dynamic instrumentation. This allows developers to create tools that take advantage of the benefits of both static and dynamic techniques that are effective and efficient. We demonstrate the effectiveness of SUDS by creating a memory and array access checker specific to input-derived variables. This checker found 17 bugs in 9 programs.

At the heart of SUDS is a source-to-source converter that takes C source code as an input and produces instrumented C source code. SUDS contains support for two analyses related to software bug detection: 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. Performance of our input checker improved 50% on average when the static analyses of SUDS are employed and used during instrumentation.

The remainder of the document is organized as follows. Section 2 provides an overview of SUDS, giving a brief description of each phase in SUDS. Sections 3 and 4 delve into the details of the static analysis and instrumentation phases respectively. Results of using SUDS are presented in Section 5. Section 6 outlines related work and Section 7 concludes.

2. Overview of SUDS

SUDS, like most compilers and bug detection tools, is organized as a series of different phases. The phases are displayed in Figure 1. SUDS takes preprocessed source code as

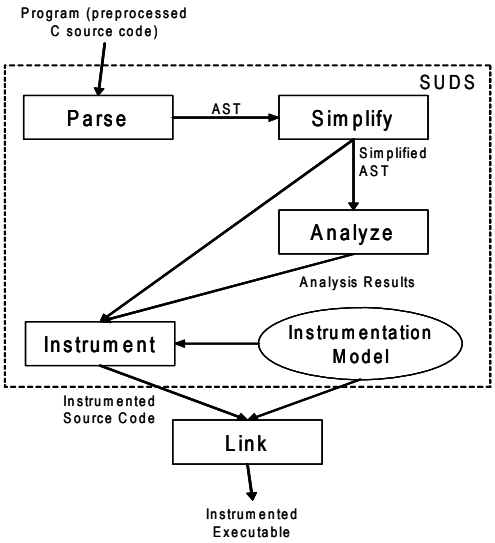

Figure 1: Overview of SUDS.

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 result of this phase is an AST of the simplified program, used by the static analysis and instrumentation phases. The static analysis phase consists of several subphases that each perform some sort of static analysis. The result of this phase can be used to improve the dynamic instrumentation. The last phase within SUDS instruments the program based on an instrumentation model. A user can either use an existing model or create their own. The output of SUDS is instrumented source code which than can be compiled and linked with an instrumentation model to form an instrumented program that can be used to detect bugs.

SUDS is written in C++ and processes programs written in the C programming language. The organization of SUDS is highly modular in that other it is relatively straight-forward to modify or add 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 [8]. The parser creates a list of top-level statements that form the AST. The list consists of global variable declarations, type declarations, function prototypes, and function definitions. Each function definition contains a list of statements that forms of the body of the function. As in most compilers, a symbol table is used to keep track of names of variables, types, and functions in the current scope. In addition, a separate table of variables is constructed and is used by the analyses to store attributes associated with the variables.

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. Other restrictions are rare. 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 have exist in popular compilers such as GCC. Except for addressing these restrictions, no modifications to the program source code are needed. Source code is not needed for system libraries. Functions without source code are not analyzed by SUDS but can be instrumented at their entry and exit points.

2.2 Simplification

The next phase in SUDS is to transform the initial AST into an intermediate C representation similar to the simple grammar developed by Hendren *et al.* [14] and serves a similar purpose as CIL [22]. The intent of simplification is to reduce the complexity of identifying and instrumenting relevant program points. 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 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. A control flow graph is created for each function and a call graph is created for the entire program. A set of definitions and uses is created for each statement and points-to information is computed for each pointer. Two phases are geared toward software bug detection. One detects variables that could hold tainted data. This can be used to focus bug detection and testing on operations that manipulate tainted data. The definition of tainted can be altered by the user. Another phase performs program slicing and marks statements as either in or not in the slice. The slicing criterion is controlled by the user. A special slicing criterion is any dangerous statement (definition by dangerous also can be altered by the user). This allows SUDS to determine the set of statements that manipulate variables that are eventually used in a dan-

gerous statement.

A more detailed discussion of the static analysis phase is in Section 3. This phase is optional if SUDS is used strictly as an instrumentation tool. 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 based on an instrumentation model. An instrumentation model consists of code that directs SUDS how to instrument the program and the actual instrumented code itself. Models for input checking, memory access, and program tracing are provided. However, the structure of SUDS allows users to easily to create and implement their own models, especially when using one of the provided models as a starting point.

The instrumentation interface of SUDS provides a function for different programming constructs including expressions, statements, and certain events (such as the start of a function). Users can direct SUDS to add instrumentation by implementing the functions that correspond to constructs and events that are interesting with respect to the instrumentation model. A suite of helper functions are provided to simplify the process of adding a call to an instrumentation function with different parameters.

This organization, described in Section 4, facilitates rapid creation of new bug 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.

3. Static Analysis

The static analysis phase of SUDS works on the entire program and consists of these subphases in the order listed:

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.

Pointer analysis: The interprocedural pointer analysis developed by Hind *et. al.* [15] is used. It 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 is done intraprocedurally using a standard data-flow algorithm that iterates until steady state. For each call site, live definitions are propagated from the caller to the callee function on a parameter by parameter basis. At the end of the function, definitions that refer to global variables are propagated to all return points. 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 3.1.

Program slicing: Determines which statements and variables influence an operation or a particular set of operations. See section 3.2.

The output of these last two phases can be used by the instrumentation phase to improve the efficiency of dynamic bug detection. In the future, it is possible to add phases or incorporate external tools specifically designed to detect bugs statically. Then, the instrumented program can focus on operations that cannot be verified statically.

3.1 Detecting Tainted Data

This phase detects the set of variables that are considered tainted. The definition of tainted can be altered within the SUDS code. By default, data that is derived from input is considered tainted. This choice is based that input data from malicious users can compromise the security of the program and its system. DaCosta *et. al.* [9] observed that functions near an input source are more likely to be vulnerable to security exploits.

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 (derived from input data) or untainted (not derived from input data). The first step is to identify which definitions of integers[1] are derived from input directly from a system function that prompts for input (such as `scanf` or `getc`). In addition, this conservatively includes all return values from string conversion functions such as

1. The algorithm only works for integers currently but could easily be expanded to other data types.

`atoi` since our analysis does not track input attributes for strings. These definitions are marked tainted while all other definitions start in the untainted state. Now, the algorithm proceeds iteratively and processes each statement individually during each iteration. For each statement, if any of the definitions used by the statement are tainted, then all definitions generated by the statement are marked tainted. The analysis is interprocedural in a context-insensitive fashion. For function calls and returns, the tainted attribute is propagated from actual parameter to formal parameters 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. The result of this phase is a list of definitions that are tainted.

3.2 Program Slicing

Program slicing [32] 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 operation under investigation). All definitions that are used in the statement(s) in the criterion are added to the slice. Then, each statement is analyzed. 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. The algorithm is interprocedural and context-insensitive.

SUDS supports three modes for determining the slicing criteria: an individual statement, all dangerous operations in a given function, or all dangerous operations in the program. In the last two cases, the definition of dangerous can be modified within the SUDS code. By default, a dangerous operation is any operation with an array reference or pointer dereference. This allows SUDS to identify the set of statements and definitions that either directly or indirectly affect the operation of any statement considered dangerous.

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.

4. Instrumentation

The last phase of SUDS adds instrumentation to the source code. The instrumentation is directed by the particular model that is being used. Different models can be used to check for different correctness properties. Models can also be created for other software-related tasks such as profiling, debugging, and tracing.

Since users are likely to design their own models, this phase has been designed in a fashion for both fast creation of simple models but 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

4.1 Instrumentation Interface

All models must adhere to a common interface consisting of several member functions that correspond to various programming constructs and events. Figure 2 shows a partial listing of the instrumentation model interface - the entire list of interface functions is much longer. In the base interface, the functions perform no action. An instrumentation model is a class derived from this base interface. Functions corresponding to interesting constructs and events with respect to the model are overwritten to add instrumentation.

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 purpose of the interface functions is to add instrumentation to the program when necessary. The output of this phase (and SUDS) is instrumented source code.

Many of the parameters in the interface routines are pointers to internal data structures within SUDS. For instance, the class `Stmt` refers to a statement and `DerefExpr` refers to a dereference operation. The internal data structures are 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 4.2.

The parameters `before` and `after` each 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. Similarly, it is added after the statement or event if

```
1  class instrInterface {

     public:
       // Expr instrumentation functions
5      virtual void instrumentLhsVar(VarExpr *expr, Stmt *&before, Stmt *&after) {}
       virtual void instrumentRhsBinaryExpr(BinaryExpr *expr, Stmt *&before, Stmt *&after) {}
       virtual void instrumentLhsDerefExpr(DerefExpr *expr, Stmt *&before, Stmt *&after) {}
       virtual void instrumentRhsDerefExpr(DerefExpr *expr, Stmt *&before, Stmt *&after) {}

10     // Stmt instrumentation functions
       virtual void instrumentIfStmt(IfStmt *stmt, Stmt *&before) {}
       virtual void instrumentWhileStmt(WhileStmt *stmt, Stmt *&before) {}
       virtual void instrumentReturnStmt(ReturnStmt *stmt, Stmt *&before) {}

15     // Function call instrumentation functions
       virtual void instrumentCallStmt(CallStmt *stmt, Stmt *&before) {}
       virtual void instrumentSystemCall(CallStmt *stmt, Stmt *&before, Stmt *&after) {}
       virtual void instrumentReturnFrom(ReturnFromStmt *stmt, Stmt *&after) {}
       virtual void instrumentCallExprArg(Expr *expr, string exprStr, int offset, Stmt *&before) {}
20     virtual void instrumentBirthParm(Expr *expr, string exprStr, int offset, Stmt *&after) {}

       // Declaration instrumentation functions
       virtual void instrumentBirthVariable(Expr *expr, string exprStr, int offset, Stmt *&after) {}
       virtual void instrumentDeathVariable(Expr *expr, string exprStr, int offset, Stmt *&before) {}
25
       // Begin/End Functions
       virtual void instrumentBeginProgram(Stmt *&after) {}
       virtual void instrumentBeginFunction(FunctionDef *fn, Stmt *&after) {}
       virtual void instrumentEndFunction(FunctionDef *fn, Stmt *&before) {}
30     virtual void instrumentEndProgram(Stmt *&before) {}

       class Expr *currLhs;        // current value of lhs if processing rhs
   };
34
```

Figure 2: Instrumentation Interface (partial). All instrumentation models must adhere to this interface. These functions are called when SUDS traverses the AST during the instrumentation phase.

it is added to the after list. Notice how some of the functions require instrumentation to be added before, other require instrumentation to be added afterwards, and the rest allow instrumentation to be added either before or after or both.

The first four functions of Figure 2 refer to different expressions noting that are different functions for expressions on the left-hand side than expressions on the right-hand side. Since simplification removes all side effects, the only interesting expression statements are assignments statements and function calls. During the traversal, the function `instrumentLhsVar` is called when the left-hand side of an assignment is a single variable (such as "`a = b + c;`") and `instrumentRhsDerefExpr` is called when the right-hand side of an assignment contains a dereference (such as "`x = *p;`"). When processing the right-hand side, the data member `currLhs` stores the current expression that comprises the left-hand side of the assignment. This is done to add instrumentation that needs access to both sides of an assignment.

The next three instrumentation functions (lines 11-13) refer to different types of statements. They behave in a similar fashion to expressions except that instrumentation can only be added before the statement. In a `return` statement, statically adding a statement afterwards is pointless since it will never execute. In the case of the `if` and `while` statements, the precise meaning of after is ambiguous and since side-effects are removed, adding instrumentation before control statements suffices in almost all situations. Similar functions exist for other types of statements.

The instrumentation functions from lines 16-20 deal with function calls. The first two functions allow the user to insert instrumentation when functions are called - the second function is used for system calls or any function where source code is not present. The function `instrumentReturnFrom` can be used to add instrumentation just after a function call and is useful for adding instrumentation involving the return value. The function `instrumentCallExprArg` allows instrumentation to be added for each actual parameter used in a function call. Similarly, the function `instrumentBirthParm` is called each time a formal parameter is declared at the beginning of a function. Together, they can be used to copy state from the actual parameters to the formal parameters. These last two functions have an additional parameter `offset` that refers to the position in which the parameters appear in the parameter list. Structure variables also use the offset field. Initially `instrumentCallExprArg` will be called for the entire structure. Then it is called for each data member in the structure (called recursively for nested structures). For each data member, the offset is incremented. The same approach is used for the `instrumentBirthParm` function. The parameter `exprStr` contains the expression associated with the variable in string form and is provided for convenience when used as a parameter in an instrumentation function.

In lines 23-30, functions are provided for interesting events: variable declarations (birth), when variables go out of scope (death), the beginning and end of functions, and the

Table 1: Helper routines for adding parameters to calls to instrumentation routines.

Routine	*If x is ___*	*Then, add ___*	*Used For*
addStringParm(string x);	foo	foo	Passing any expression in string form. Use this function for passing values stored in `exprStr`, a parameter provided by some interface routines.
addStringConstantParm(string x);	foo	"foo"	Passing any string used as a constant within the instrumentation routine.
addIntegerParm(int x);	23	23	Passing any integer constant.
addExprParm(Expr *x);	a[i]	a[i]	Passing the value of an expression or variable.
addStringExprParm(Expr *x);	a[i]	"a[i]"	Passing the expression as a string - useful for debugging.
addAddrOfExprParm(Expr *x);	a[i]	&(a[i])	Passing the address of an expression (assuming it is addressable) - useful for tracking all variables since heap variables do not have names.

beginning and end of the program.

4.2 Calling Instrumentation Functions

Once a user has determined which constructs and events need instrumentation, the next step is to write the interface routines to add the instrumentation. In most cases, the added instrumentation will merely be one line to call a function likely with some parameters. 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 1 outlines the different routines 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 lists) and the name (string) of the function to call.

This process is best illustrated using an example of instrumentation model. Figure 3 shows a model used to trace function calls. In both cases the functions are called with one parameter - the name of the function passed as a string. The next step is to write the functions `function_begin` and `function_end`. The actual instrumentation functions simply display an appropriate message with the function name. Another example, an array checker, is presented in the next section.

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. Instrumentation is managed using string and string streams allowing users to take advantage of both the C++ string library and print routines that already exist for variables, expressions, and statements within SUDS.

4.3 Creating Instrumentation Models

Creating an instrumentation model requires the code to direct SUDS where to insert calls to instrumentation functions and the actual instrumentation routines themselves. The implementation of the instrumentation routines are dependent on the goal of the instrumentation model. When creating dynamic bug detection tools, the routines are used to track additional state using a table such as the object table used by Jones and Kelly [16]. We will describe the underlying code using the example model shown in Figure 4.

```
class instrTrace : public instrInterface {

  public:

  void instrumentBeginFunction(FunctionDef *fn,
      Stmt *&after)
  {
    initInstrCall();
    addStringConstantParm(fn->FunctionName());
    addInstrCall(after, "function_begin");
  }

  void instrumentEndFunction(FunctionDef *fn,
      Stmt *&before)
  {
    initInstrCall();
    addStringConstantParm(fn->FunctionName());
    addInstrCall(before, "function_end");
  }
};
```

Figure 3: Sample instrumentation model for tracing function calls.

The model presented in Figure 4 checks array references for array variables - it does not check array references when pointers are used. Arrays that are declared call `add_array`. The function `add_array` adds the array to a table that is indexed by the base address of the array (first parameter to `add_array`). The table stores the size of the array (second parameter to `add_array`). When an array goes out of scope, `remove_array` is called, removing the array from the table.

During an array reference, the function `check_array` is called to make sure the subscript is within the bounds of the array. This function looks up the address in the table obtaining the size of the array. The size is compared to the index. If the index is outside the allowable bounds, an error is flagged.

In each instrumentation routine in Figure 4, notice how the expression is analyzed further and instrumentation is only added if the array reference refers to an array, opposed to a pointer. Instrumentation is not added otherwise, though

```
class InstrModel : public InstrInterface {

  public:

  void instrumentBirthVariable(class Expr *expr, string
    exprStr, int offset, class Stmt *&after)
  {
    if (expr->type->isArray()) {
      initInstrCall();
      addStringParm(exprStr);
      addIntegerParm(expr->type->getArraySize());
      addInstrCall(after, "add_array");
    }
  }

  void instrumentDeathVariable(class Expr *expr, string
    exprStr, int offset, class Stmt *&before)
  {
    if (type->isArray()) {
      initInstrCall();
      addStringParm(exprStr);
      addInstrCall(before, "remove_array");
    }
  }

  void instrumentLhsArrayExpr(class ArrayExpr *expr,
    class Stmt *&before, class Stmt *&after)
  {
    // if (!expr->index->isTainted()) return;
    if (expr->leftExpr->type->isArray()) {
      initInstrCall();
      addExprParm(expr->leftExpr);
      addExprParm(expr->index);
      addInstrCall(before, "check_array");
    }
  }

  void instrumentRhsArrayExpr(class ArrayExpr *expr,
    class Stmt *&before, class Stmt *&after)
  {
    instrumentLhsArrayExpr(expr, before, after);
  }
};
```

Figure 4: Sample instrumentation model for checking array references.

the model could easily be expanded to include pointers. It is also worth noting that there is no distinction between array references on the left-hand side and those on the right-hand side so the right-hand side routine merely calls the left-hand side routine.

SUDS can use the result of prior static analysis phases when instrumenting the program. Look at the commented out in the function `instrumentLhsArrayExpr` of Figure 4. Instrumentation is only added if the index used to reference the array is tainted. This allows the user to focus on array references where the index is tainted. If the index is not tainted, then no instrumentation is added. This notion could be expanded to safe and unsafe. Perhaps this array reference could be verified statically - then no instrumentation is necessary. This approach allows static analysis to fall back to dynamic instrumentation in cases where this is not enough information to determine if the operation is safe.

In addition to the two models described in this paper, additional models are provided including the input checker used in our results section. The input checker utilizes an object table and uses range analysis so that the entire range of possible values can be tracked. Both can be used in new models created by users. The object table has a debugging mode that is helpful both to debug newly created models and to pinpoint the source of an error once a defect

Table 2: Bugs detected during testing.

Program	SUDS	SPLINT		VALGRIND	
	Bugs	Same Bugs	New Bugs	Same Bugs	New Bugs
anagram	2	2	0	0	0
ft	2	0	0	0	0
ks	3	2	0	0	0
yacr2	2	0	0	0	0
betaftpd	2	0	0	0	1
gaim	1	1	1	0	0
ghttpd	3	1	0	1	0
openssh	2	1	0	didn't work	
thttpd	0	0	0	0	0

5. Results

To demonstrate the effectiveness of SUDS, we implemented and used the input checker described in our prior work [18]. The input checker uses ranges to track integers and strings that are derived from user-input. The checker checks for array overflows, pointer dereference errors, and string library errors such as overflowing strings and null termination problems. Since ranges are used, it is not necessary to have a precise data input to expose an error. Additional checks detect cases where unbounded input data is used in a loop or memory allocation.

We used the nine programs shown in Table 2. All programs were compiled using GCC and with an -O4 optimization level. Four of the programs (*anagram*, *ft, ks*, and *yacr2*) are from the pointer-intensive benchmark suite [27] and were selected due to the difficulty of analyzing these programs statically. The other five programs are networking applications, including the popular secure shell program *openssh* (the server *sshd* specifically) and *gaim*, a popular instant messaging program. The other three server programs include a FTP server (*betaftpd*) and two web servers (*ghttpd* and *thttpd*).

5.1 Bugs Detected

The most important aspect of evaluating a bug detection infrastructure is its ability to find bugs. When applying our input checker we were able to find the 17 bugs in the 9 programs, as shown inTable 2. The 17 bugs were a mix of array overflows, use of unbounded data in loops, and string library

Table 3: Run-time performance results.

Program	Base line	Valgrind		SUDS Unoptimized		SUDS Optimized		Perf. Improve %
		Time	Ratio	Time	Ratio	Time	Ratio	
anagram	0.06	1.88	31.33	3.15	52.50	1.32	22.00	58.1%
ft	0.18	5.92	32.89	5.32	29.56	0.88	4.89	83.5%
ks	0.05	4.16	83.20	3.96	79.20	0.45	9.00	88.6%
yacr2	0.12	3.83	31.92	22.63	188.58	11.87	98.92	47.5%
betaftpd	0.07	0.45	6.43	0.53	7.57	0.27	3.86	49.1%
ghttpd	0.52	35.60	68.46	1.08	2.08	0.69	1.33	36.1%
openssh	0.70	didn't work		1.00	1.43	0.91	1.30	9.0%
thttpd	0.15	0.29	1.93	2.57	17.13	1.78	11.87	30.7%

function problems. We did detect six false alarms but all of these were attributed to an overly aggressive model rather than any shortcoming of SUDS. Users of SUDS can create models that are more conservative, reducing the number of false alarms but also possibly reducing the number of actual bugs found.

For comparison purposes, the same programs were run on different bug detection tools to see if they catch the same bugs. We used the static bug detection tool Splint [17] and the dynamic tool Valgrind [24] in our comparison. The results are shown in Table 2. Splint reported a large number of errors since we did not annotate the source code which would eliminate a large number of the false reports. As a result, we only manually looked for errors we found with our input checker and error reports that were marked “likely”; a vast majority of the errors were labeled as “possible”. It was able to find seven out of the seventeen bugs that we detected. It is important to note that these tools are designed to find similar but different type of bugs so a direct comparison is not entirely fair. The purpose of this experiment is to show that the SUDS infrastructure is capable of creating bug detection tools that are at least on par with existing tools.

5.2 Performance Results

To analyze performance, we compared the run-time performance with and without the static analysis features enabled[1]. For this particular model, we marked data coming from input as tainted and all operations that required a check as dangerous (includes array references, pointer dereferences, loop conditions, and malloc inputs). The results are shown in Table 3. The baseline column shows how fast the program runs without any instrumentation. The unoptimized and optimized SUDS columns refer to the speed of the instrumented programs without and with static analyses enabled respectively.

The amount of the amount of slow-down experienced by the unoptimized programs varies across the benchmarks. The four server programs exhibited the least amount of slow down with *thttpd* having the most with just over 17x. The four pointer intensive benchmarks suffered significant slowdown from a factor of 30x in *ft* to 186x in *yacr2*. The disparity in the results can be attributed to the fact that the pointer intensive benchmarks have more integer processing than the servers and have a significantly higher number of dynamic instrumentation calls.

When the static analyses are enabled, the performance improved (shown in the last column of Table 3) by 50% on average. In general, programs that had the worst impact on performance saw the best performance improvement. For instance, *yacr2*, which had a factor of 189x slowdown, saw a performance improvement of 48%. On the other end of the spectrum, *openssh* saw little performance improvement since it suffered very little performance degradation initially. The improvement in performance is shown in Table 4. Many programs had relatively few integers that were both tainted and dangerous. In fact, *ft* had no such integers. The instrumentation that remains is for arrays and strings which were untouched by these analyses.

When comparing the performance results to that of Valgrind (in Table 3), our approach suffers a similar magnitude of slowdown for most of the benchmarks. In seven of the nine programs, the optimized programs from SUDS are faster. The two exceptions were *yacr2* (which was really slow in SUDS) and *thttpd* (which was really fast in Valgrind). It is important to note that Valgrind and our input checker look for different types of bugs making a precise performance comparison impossible.

1. Due to the interactive nature of *gaim*, it is not included in any performance experiments. There was not a noticeable slowdown when using the instrumented version of *gaim*.

Table 4: Breakdown of tainted and dangerous integers.

	anagram	ft	ks	yacr2	betaftpd	ghttpd	openssh	thttpd
Total Integers	257	233	244	1,508	950	420	11,727	2,493
Untainted	210	202	140	855	673	352	8,208	1,609
Tainted	47	31	104	653	277	68	3,519	885
Safe	202	213	179	1,163	854	365	8,871	2,106
Dangerous	55	20	65	345	96	55	2,856	387
Untainted and Safe	163	182	117	693	609	310	5,759	1,462
Untainted and Dangerous	47	20	23	162	64	42	2,449	147
Tainted and Safe	39	31	62	470	245	55	3,112	645
Tainted and Dangerous	8	0	42	183	32	13	407	240

6. Related Work

There are several tools used to instrument programs. ATOM [29] and Pin [19] 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 [31] is another infrastructure used to supervise programs and has specialized functionality for detecting software faults. The main disadvantage of these tools is that they all operate at the assembly-code level limiting the types of bugs they can detect. Tikir and Hollingsworth [30] describe an instrumentation technique for obtaining coverage for testing.

Recently, research groups have looked at techniques to incorporate static analysis in testing tools. CCured [23] 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. One key difference between SUDS and CCured is that SUDS is designed to be a general-purpose tool both in terms of static analysis and the instrumentation engine. Other research [7, 10, 28] has looked at using static analysis to automate testing. The premise is to use static techniques to analyze code. Based on this analysis, test cases are generated and run with the hope of detecting errors. This work is complementary to ours in that SUDS could assist in the process of detect errors dynamically with the generated test cases. One example of using analysis to improve performance is the work done by Bodik *et. al.* [2]. They use a lightweight static analysis to determine if array bounds checks are redundant with earlier checks. GrammaTech developed CodeSurfer [12], a whole program analysis tool with program slicing, and CodeSonar [11], a tool designed to catch a variety of memory access and other defects statically.

Memory access errors are a popular target for run-time or dynamic bug detection. Examples of dynamic bug detection systems include GNU's checker [5], Purify [13], Parasoft's Insure++ [26], and Jones and Kelly [16]. 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 [13], 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 errors that can be detected. Dynamic taint analysis has been used [25] to prevent unauthorized access or corruption.

There are many different approaches to verifying software statically or at compile-time. One approach is to use symbolic analysis to determine if a property has been violated. ARCHER [33] is a path-sensitive tool that uses symbolic analysis to find memory access errors. Coen-Porisini *et. al.* [4] use symbolic execution for a subset of the C programming language and have applied their technique to safety-critical software systems. Symbolic execution can be used to determine if a path is illegal. Zhang and Wang [34] describe BoNuS, a symbolic technique for determining path feasibility that supports both arithmetic inequalities and Boolean operators. Other common static approaches include using a constraint solver to detect violations [3, 17, 31] and model checking [1, 6, 20, 21].

7. Conclusions and Future Work

SUDS is a powerful infrastructure for creating bug detection tools that rely on both static analysis and dynamic instrumentation. It is effective in finding bugs dynamically using different correctness models.

Static analysis phases are used to improve dynamic bug detection by focusing on operations that can be exploited by malicious users. This improves performance 50% in the model we tested. The instrumentation engine is generic allowing users to create their own models of correctness.

SUDS, like all software tools, are constantly evolving. The next step for SUDS is to improve the static bug 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 errors. We also will explore the ability to plug in existing tools within the SUDS infrastructure.

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.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments.

References

[1] T. Ball and S. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. Workshop on Model Checking of Software, May 2001.

[2] R. Bodik, R. Gupta, and V. Sarkar. ABCD: Eliminating Array Bounds Checks on Demand. Proceedings of the Conference on Programming Language Design and Implementation, June 2000.

[3] W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic programming errors. Software Practice and Experience, July 2000.

[4] A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezzè. Using Symbolic Execution for Verifying Safety-Critical Systems. Proceedings of the 9th International Symposium on Foundations of Software Engineering, Sept. 2001.

[5] Checker. http://www.gnu.org/software/checker/checker.html

[6] H. Chen and D. Wagner. MOPS: an Infrastructure for Examining Security Properties of Software. Proceedings of the Conference on Computer and Communications Security, November 2002.

[7] C. Csnaller and Y. Smaragdakis. Check 'n' Crash: Combining Static Checking and Testing. Proceedings of the International Conference on Software Engineering, May 2005.

[8] cTool. http://sourceforge.net/projects/ctool/

[9] D. DaCosta, C. Dahn, S. Mancoridis, V. Prevelakis. Characterizing the 'Security Vulnerability Likelihood' of Software Functions. Proc. of the 2003 International Conference on Software Maintenance, Sept. 2003.

[10] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. Proceedings of the Conference on Programming Language Design and Implementation, June 2005.

[11] GrammaTech, Inc. CodeSonar. http://www.grammatech.com/products/codesonar

[12] GrammaTech, Inc. CodeSurfer. http://www.grammatech.com/products/codesurfer

[13] R. Hastings and B. Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. Proceedings of the 1992 Winter Usenix Conference, January 1992.

[14] L. Hendren, C. Donawa, M. Emami, G. Gao, Justiani, and B. Sridharan. Designing the McCAT Compiler Based on a Family of Structured Intermediate Representations. Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing, August 1992.

[15] M. Hind, M. Burke, P. Carini, and J. Choi. Interprocedural Pointer Alias Analysis. ACM Transactions on Programming Languages and Systems. July 1999.

[16] R. Jones and P. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. Proc. of the 3rd International Workshop on Automated Debugging, May 1997.

[17] D. Larochelle and D. Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. Proceedings of the 2001 USENIX Security Symposium, 2001.

[18] E. Larson and T. Austin. High Coverage Detection of Input-Related Security Faults. 12th USENIX Security Symposium, August 2003.

[19] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. Proceedings of the Conference on Programming Language Design and Implementation, June 2005.

[20] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[21] M. Musuvathi, D. Park, A. Chou, D. Engler, and D. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. Proceedings of the Conference on Operating System Design and Implementation, December 2002.

[22] G. Necula, S. McPeak, S. P. Rahul, W.Weimer. Cil: Intermediate Language and Tools for Analysis and Transformation of C Programs. International Conference on Compiler Construction, Apr. 2002

[23] G. Necula, S. McPeak, and W. Weimer. CCured: Type-Safe Retrofitting of Legacy Code. Proceedings of the Symposium on Principles of Programming Languages, January 2002.

[24] N. Nethercote and J. Seward. Valgrind: A Program Supervision Framework. Electronic Notes in Theoretical Computer Science, 2003.

[25] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis and Signature Generation of Exploits on Commodity Software. Proceedings of the 12th Annual Network and Distributed System Security Symposium, Feb. 2005.

[26] Parasoft Corporation. Insure++: An Automatic Runtime Error Detection Tool. Technical Report PS961-INS1.

[27] Pointer-Intensive Benchmark Suite <http://www.cs.wisc.edu/~austin/ptr-dist.html>

[28] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. Proceedings of the Symposium on the Foundations of Software Engineering, Sept. 2005.

[29] A. Srivastava and A. Eustace. ATOM: A System for Building Customized Program Analysis Tools. Proceedings of the Conference on Programming Language Design and Implementation, June 1994.

[30] M. Tikir and J. Hollingsworth. Efficient Instrumentation for Code Coverage Testing. ACM SIGSOFT Software Engineering Notes, 2002.

[31] D. Wagner, J. Foster, E. Brewer, A. Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. Network and Distributed Security Symposium, February 2000.

[32] M. Weiser. Programmers use slices when debugging. Communications of the ACM, 1982.

[33] Y. Xie, A. Chou, D. Engler. ARCHER: Using Symbolic, Path-sensitive Analysis to Detect Memory Access Errors. Proceedings of the 11th International Symposium on the Foundations of Software Engineering, Sep. 2003.

[34] J. Zhang and X. Wang. A constraint solver and its application to path feasibility analysis. International Journal of Software Engineering and Knowledge, Volume 11, 2001.