# ARCHER: Using Symbolic, Path-sensitive Analysis to Detect Memory Access Errors

Yichen Xie, Andy Chou, and Dawson Engler
Computer Systems Laboratory
Stanford University
Stanford, CA 94305, U.S.A.

## ABSTRACT

Memory corruption errors lead to non-deterministic, elusive crashes. This paper describes ARCHER (*ARray CHeckER*) a static, effective memory access checker. ARCHER uses path-sensitive, interprocedural symbolic analysis to bound the values of both variables and memory sizes. It evaluates known values using a constraint solver at every array access, pointer dereference, or call to a function that expects a size parameter. Accesses that violate constraints are flagged as errors. Those that are exploitable by malicious attackers are marked as security holes.

We carefully designed ARCHER to work well on large bodies of source code. It requires no annotations to use (though it can use them). Its solver has been built to be powerful in the ways that real code requires, while backing off on the places that were irrelevant. Selective power allows it to gain efficiency while avoiding classes of false positives that arise when a complex analysis interacts badly with statically undecidable program properties. ARCHER uses statistical code analysis to automatically infer the set of functions that it should track — this inference serves as a robust guard against omissions, especially in large systems which can have hundreds of such functions.

In practice ARCHER is effective: it finds many errors; its analysis scales to systems of millions of lines of code and the average false positive rate of our results is below 35%. We have run ARCHER over several large open source software projects — such as Linux, OpenBSD, Sendmail, and PostgreSQL — and have found errors in all of them (118 in the case of Linux, including 21 security holes).

## Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Assertion checkers, Reliability, Statistical methods*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Symbolic Execution*


*ESEC/FSE'03,* September 1–5, 2003, Helsinki, Finland.


## General Terms

Algorithms, Experimentation, Reliability, Security, Verification

## Keywords

Buffer overrun, buffer overflow, memory access errors, error detection, static analysis, security

## 1. INTRODUCTION

Unsafe languages such as C and C++ are vulnerable to memory corruption errors. Such errors lead to elusive, non-deterministic crashes or, worse, can be exploited by malicious attackers to compromise a system. Safe languages such as Java are also vulnerable. These languages check buffer accesses at runtime and raise exceptions when errors occur. However, many of these exceptions are unexpected and remain unhandled by the programmer, causing the program to abort.

This paper describes ARCHER (*ARray CHeckER*), an automatic tool that statically catches such memory access errors. ARCHER uses a path-sensitive, interprocedural analysis to derive and propagate memory bounds and variable values. It tracks constant relations (e.g., `i = 4`, $j \neq 10$ and $0 \leq k < 20$) as well as symbolic constraints between variables that have unknown values (e.g., $j < k < 2 \cdot l$). At every potentially dangerous access — such as an array index, pointer dereference, or call to a routine that takes a pointer and explicit size — it uses a custom constraint solver to evaluate the values used in the operation against known constraints. It flags constraint violations as potential memory errors. To increase its reach, ARCHER propagates these constraints across procedure boundaries on demand.

ARCHER was designed to work on large existing code bases. It has several key features that make this possible.

**No annotations needed**. Compared to annotation-based tools like ESC/Java [10] and LCLint [18], ARCHER uses statistical methods to extract necessary information automatically from the source program. Human intervention is minimal, making it suitable for checking existing code bases with little extra cost. Of course, ARCHER is ready to use any annotations that are provided (and makes providing them worthwhile). However, it does not require them to obtain good results.

**Speed**. Two aspects of ARCHER's design allow it to handle multi-million line programs. First, it uses bottom-up interprocedural analysis. By simulating the body of the C function, it is able to infer a set of constraints that must be satisfied by callers of the function without re-analyzing the function body in each calling context. Second, ARCHER uses a custom constraint solver targeted to linear relations. We have observed that the vast majority of pointer arithmetic operations and conditional operations are linear (e.g., array indices, loop bounds). We have identified a subset of essential linear relationships to track with our solver. Compared to more complex integer constraint solvers (e.g. Omega [21]) and theorem provers (e.g. SAT [23]), we have traded precision for speed and predictability. Nevertheless, preliminary results have shown that our solver is effective at finding a large class of buffer access errors in real code.

**Few false positives.** The solver has been carefully tuned to allow it to suppress common sets of false positives. In addition, we use error ranking to sort errors from most to least plausible. Finally, we have developed several heuristics to minimize the number of false positives reported by the checker.

**Drawbacks.** While ARCHER works well in practice, it has limitations. It is not a verifier. Code that contains errors can pass through it silently. Conversely, code without errors can still trigger warnings. The main strategy of ARCHER has been to find as many bugs as possible while minimizing the number of false positives. One limitation of the current system is that it does not understand of C string operations; these cause many errors, which it will miss. However, we believe the analysis framework and the solver can be readily used to analyze null-terminated strings, and a version that does so is under development.

The next section discusses related work in more detail. Section 3 gives two examples that illustrate the techniques needed to find buffer access errors. Section 4 gives an overview of ARCHER, and Sections 5 and 6 describe the implementation of the analysis in detail. Section 7 discusses how we use statistical analysis to infer size fields. Experimental results are given in Section 8. Finally, Section 9 concludes.

# 2. RELATED WORK

In this section, we compare ARCHER to existing dynamic and static approaches to finding memory errors.

## 2.1 Dynamic Analysis

Dynamic tools instrument the program (source code or binary) to add buffer bounds checks. This approach has the advantage that all values are concrete at runtime, so pointer offsets and buffer lengths are known, and only feasible paths are considered.

Static analysis has several advantages over a dynamic approach for large code bases. Unlike a dynamic approach, static analysis does not require executing code. It can immediately find errors on obscure paths that could otherwise take weeks of testing to trigger. Some serious security errors may not be triggered at all in testing, only to be uncovered after significant damage is incurred [22]. Further, static analysis finds bugs in code that cannot be run on a particular configuration. This feature is important for OS device drivers because only a small fraction of them can be tested at a typical site. Finally, dynamic tools typically slow down execution by a factor of 2-30, making it time consuming to run test cases and inapplicable to programs that have strict timing requirements.

Purify [14] performs binary rewriting to add instrumentation to check for buffer access errors, memory leaks, and many other errors. While Purify is very useful for debugging, it typically slows down program execution by 10x-30x and also increases memory use substantially. CCured [20] is a hybrid static-dynamic tool that requires source code. CCured uses a type inference algorithm to eliminate the need for many checks, which mitigates the speed penalty to between 10%-100%, but it also performs fewer checks than Purify. Jones and Kelly [16] show how dynamic bounds checking can be done without changing the representation of pointers, thus maintaining compatibility with binary-only libraries, but the technique typically slows down programs by a factor of five or six.

## 2.2 Static Analysis

Static analysis has traditionally been applied to the problem of inferring array bounds checks that can be safely eliminated [2]. This kind of analysis finds safety checks that are redundant in safe languages to improve performance. In contrast, ARCHER attempts to find memory accesses that are unsafe in an unsafe language in order to improve robustness and reliability of programs.

Several static tools have been developed to detect buffer access errors in C. However, existing tools either (1) depend heavily on user annotations that limit their applicability to existing code bases (e.g. LCLint [18], CSSV [6]), (2) produce a large number of false positives (e.g. LCLint [18], BOON [24]), or (3) use heavy-weight theorem provers that do not readily scale to large code bases (e.g. CSSV [6]). Furthermore, these tools do not statistically infer checking information as described in Section 7.

LCLint [18] detects buffer access errors using a simple static dataflow analysis along with annotations provided by the user. It relies on annotations for all interprocedural checking; this annotation burden makes it difficult to apply to large code bases. This can be seen in the relatively small number of errors it finds — three errors in early versions of `bind` and `wu-ftpd`, two of which were previously known. Further, the simple analysis employed by LCLint resulted in a large number of false positives despite the significant amount of information provided by the user.

BOON [24] employs an interprocedural analysis that finds buffer overruns caused by misuse of string manipulation functions. The tool performs a flow-insensitive pass over the abstract syntax tree to derive a set of constraints formulated by treating strings as an abstract data type. While the analysis is efficient, it loses a great deal of precision, which is reflected in the high false positive rate – BOON found 4 off-by-one bugs with 40 false alarms in Sendmail 8.9.3.

Dor et. al. [6] describe a technique that can handle string manipulation functions as well as pointer arithmetic and that aims to find all memory errors. The technique uses an expensive integer analysis algorithm and requires extensive annotations. The authors use the tool to find 8 unclean memory accesses with 8 false positives in some small string-intensive programs. The speed of the solver was a bottleneck – currently reported results indicate the analysis takes minutes of CPU time and hundreds of megabytes of memory to process C functions roughly 100 lines of code in size.

ESC/Java [10] is an annotation-based tool that uses a theorem prover to look for violations of specifications provided by the programmer. The tool has been used in [4] to annotate and rediscover two known security vulnerabilities.

```
1  /* 8.12.7/sendmail/daemon.c */
2  ...
3  /* get result */
4  p = &ibuf[0];
5  nleft = sizeof ibuf − 1;
6  while ((i = read(s, p, nleft)) > 0) {
7      p += i;
8      nleft −= i;
9      *p = '\0';
10     if (strchr(ibuf, '\n') != NULL || nleft <= 0)
11         break;
12 }
13 (void) close(s);
14 sm_clrevent(ev);
15 if (i < 0 || p == &ibuf[0])
16     goto noident;
17 if (*−−p == '\n' && *−−p == '\r')
18     p−−;
19 *++p = '\0';
20 ...
```

Figure 1: A sample bug from Sendmail version 8.12.7 that involves pointer `p` pointing into the buffer `ibuf`. The error occurs when a single '`\n`' character is read at line 6.

```
1  /* 2.5.53/include/linux/isdn.h */
2  #define ISDN_MAX_DRIVERS 32
3  #define ISDN_MAX_CHANNELS 64
4  /* 2.5.53/drivers/isdn/i4l/isdn_common.c */
5  static struct isdn_driver *drivers[ISDN_MAX_DRIVERS];
6  static struct isdn_driver *get_drv_by_nr(int di) {
7      unsigned long flags;
8      struct isdn_driver *drv;
9      if (di < 0)
10         return NULL;
11     spin_lock_irqsave(&drivers_lock, flags);
12     drv = drivers[di];
13     ...
14 }
15 static struct isdn_slot * get_slot_by_minor(int minor) {
16     int di, ch;
17     struct isdn_driver *drv;
18     for (di = 0; di < ISDN_MAX_CHANNELS; di++) {
19         drv = get_drv_by_nr(di);
20         ...
```

Figure 2: A sample error from Linux kernel version 2.5.53 that involves multiple functions and can potentially crash the system.

However, the annotation overhead of ESC/Java is relatively high: the authors estimate that 1 programmer hour is required for every 300 lines of code. A great deal of research has gone into deriving loop invariants [11] and automating the annotation process [9], but the manual cost of using the tool remains prohibitively high for large code bases.

PREfix [3] is a static tool that finds a variety of memory access errors without annotations. ARCHER's analysis style is similar to that of PREfix: both analyze functions in bottom-up order and generate a summary with constraints for each analyzed function. However, the analysis performed and the class of errors being targeted is different. PREfix targets a broader range of errors than ARCHER — references to uninitialized or freed memory regions, NULL pointer dereferences, and memory leaks by tracking memory regions of known size. However, it appears that ARCHER is more powerful for the memory overflow errors it targets, since it tracks properties such as buffer lengths and pointer offsets with a symbolic solver. Furthermore, PREfix relies on the user to provide models for library functions whose source code is not available; ARCHER needs much less modelling effort, and employs a statistical analysis to derive a large number of function interfaces automatically.

## 3. TWO MOTIVATING EXAMPLES

In this section, we use two representative samples of errors we have detected with ARCHER to illustrate the key features we find necessary in order for a non-annotation based analysis to be effective. These errors were found in recent releases of Sendmail and the Linux kernel. Both were reported to and confirmed by their developers and patches have been generated to correct them.

Figure 1 shows an example of an error we found in Sendmail 8.12.7. Here, the pointer `p` is initialized to point to the start of the buffer `ibuf` at line 4. If the call to `read()` at line 6 reads a single character '`\n`', `p` will be incremented by one at line 7 and then the loop will exit. The error occurs at line 17, where the second decrement of `p` causes it to point to `&ibuf[-1]`, resulting in an out-of-bounds read. If `ibuf[-1]` happens to be '`\r`' then it will be overwritten by '`\0`' at line 19, an out-of-bounds write.

Figure 2 shows another sample bug we found in a recent release of the Linux kernel. Here, `get_slot_by_minor` calls `get_drv_by_nr` with parameter `di`, which can go from 0 up to 63 (`ISDN_MAX_CHANNELS-1`). An out-of-bound memory access might occur at line 12, where `get_drv_by_nr` uses `di` as an index to access `drivers` — a global array with only 32 (`ISDN_MAX_DRIVERS`) elements. This is a potentially dangerous system-crashing bug that would be missed in testing except on certain hardware configurations with an abundance of ISDN devices.

The goal of ARCHER is to accurately pinpoint these errors statically. In order to achieve this goal, the analysis needs to be

- **Interprocedural** — in the Linux example in Figure 2, the definition and use sites of the buffer index `di` lie in two different functions. We have found over 30 instances of interprocedural buffer access errors — roughly one fourth of the total number of errors found in Linux.

- **Fully symbolic** — in Figure 2, to analyze the function `get_drv_by_nr`, we need to infer the constraints that the unknown parameter `di` must satisfy in order for the access `drivers[di]` to be safe. In order to do that, we need to represent the value of `di` symbolically and proceed with the analysis without knowing the exact value of it.

- **Path sensitive** — empirically, infeasible paths cause many false positives. We use path-sensitive analysis to eliminate some classes of false paths. Furthermore, the values of key properties of a typical buffer access (such as the offset of `p` at line 18 in Figure 1) often depend on specific program paths leading to it. A path-insensitive analysis would either lose these errors or flag a large number of false positives.

- **Context sensitive** — clearly, the range of possible values of the parameter `di` in `get_drv_by_nr` partly depends on its caller. For example, there are four calls to this function in that source file, only two of which can be confidently flagged as errors.

- **Aware of pointer aliases for buffers** — in order to detect the error in Figure 1, we need to know not only the fact that `p` is an alias to the buffer `ibuf` (thus both share the same length), but also the exact offset of `p` relative to `&ibuf[0]`.

# 4. ARCHER OVERVIEW

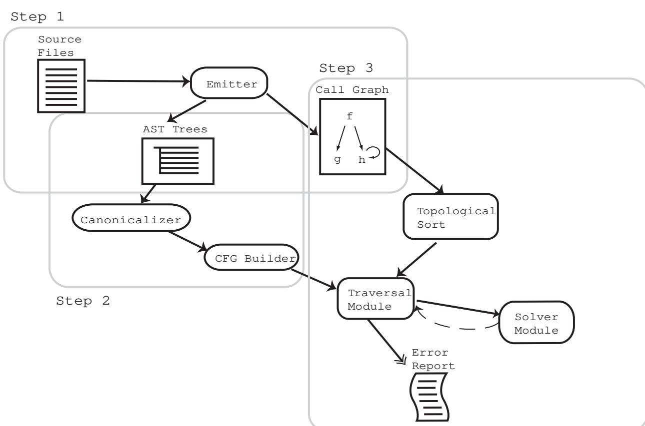


Figure 3: An Overview of the Analysis

This section gives an overview of ARCHER. The core of ARCHER is an interprocedural, path-sensitive and context-sensitive data-flow analysis. It consists of three pieces: (1) a translator that transforms C source code to a canonicalized intermediate representation; (2) a traversal module, which traverses this representation and, (3) a solver called by the traversal module to accumulate and solve constraints encountered during traversal. We give a brief overview of each stage below; the next two sections describe the traversal module and solver in more detail.

Figure 3 gives a flow-chart of the three steps involved in using ARCHER to check a program. The first step uses a modified version of the GNU C Compiler (gcc 3.1) to parse the source code into abstract syntax trees (AST). GCC was chosen because of the large amount of open source software it can process. The trees are then serialized and dumped onto the disk for further processing. The next two steps, described below, consist of 14KLOC in O'Caml.

The second step transforms the trees into a canonical representation. The transformation is inspired by tools such as CIL [19] and the Microsoft AST toolkit [5]. In this step, we introduce temporary variables to eliminate side-effects in expressions and flatten nested function calls. We also convert short-circuit operators such as &&, ||, and ? : into if-else statements to eliminate control flow within expressions. The result is a C program that is semantically equivalent to the original code, but with a reduced number of syntactic constructs. From these canonicalized trees we construct a control-flow graph (CFG) for each function with sets of AST trees as basic blocks and branching statements as edges. During this stage we also build an approximate program callgraph. The callgraph is approximate because we do not track function pointers. As will be shown in the next section, while the loss of precision may lead to missed errors (false negatives), it will not increase the number of false-positives.

The third step is where the action happens. The analysis is carried out in bottom-up order on the callgraph, inspecting each function for potential errors. Cycles in the call graph are broken at arbitrary places. As stated above, analysis partitions into two pieces, the traversal module and the solver. The traversal module performs a randomized, depth-first search to exhaustively explore each function's control flow graph. Each statement in the CFG causes the traversal module to do one of several actions, depending on the statement type:

1. A boolean condition: the solver is called to evaluate the conditional expression of an `if`-statement, loop, or `switch` statement. If the solver can determine the value, the traversal module will prune away any infeasible control flow edges controlled by the expression. When following each edge, the solver state is updated accordingly to reflect the constraint imposed by the path condition.

2. A memory access: the solver is queried to see if the memory access can potentially go out of bounds. If so, an error is emitted. If the solver cannot demonstrate that the access is unsafe, the traversal module emits no error. I.e., ARCHER is not conservative (sound). A sound tool would emit an error if it could not demonstrate the access was safe. If ARCHER treated accesses as "guilty until proven innocent" there would be an overwhelming number of false positives.

3. All other accesses: the traversal deconstructs the statement into a (possibly empty) set of constraints and calls the solver to update the solver state accordingly.

Interprocedural analysis introduces the complication that if a memory access involves a formal parameter, the solver only knows the possible values of the parameter after examining all call sites. Thus, the traversal module summarizes the memory access constraints in such a way that they can be evaluated as actual parameter values are encountered at each call site. We call the constraint representation under which an error could occur a *trigger* for the function. The analysis stores triggers in a global database. When it analyzes a function call, the triggers for the function (if any) are evaluated against the current set of known constraints. If a trigger is satisfied, the traversal module emits an error message. The next section describes traversal in more detail, and the subsequent one focuses on the solver.

# 5. TRAVERSAL IMPLEMENTATION

This section describes the traversal module in more detail. It first gives a brief overview of traversal, and then focuses on the exact errors flagged and the handling of loops and function calls.

## 5.1 Traversing executable paths

The goal of the traversal module is to enumerate and check all possible legal program execution paths. However, in practice it checks both a superset or a subset of legal paths. A superset because it only prunes impossible paths if it can determine that boolean conditions the path depends on are unsatisfiable. Since the problem is generally undecidable, and ARCHER's analysis has practical limitations, there will be cases where it cannot prune an impossible path. A subset because it ignores function pointers and usually does not simulate loops as many times as an actual run of the program would.

As stated in Section 4, ARCHER analyzes program functions in bottom-up order, doing a depth-first search over each function's control flow graph. Starting from the large scale to the smaller, it works as follows. The analysis minimizes the impact of missing summaries for functions yet to be analyzed by topologically sorting all functions based on the call graph, and starting the analysis from the leaves (functions that do not call others). If there is a cycle in the call graph, we pick the function with the least number of callees to be analyzed first. In case of a tie, we choose at random among all candidates.

The analysis processes a function's CFG using a simple DFS search starting from the entry block. Its goal is to build up a knowledge base (stored in the *solver state*) that tracks values of and relations between as many scalars, arrays, and pointers as possible, which it then uses to detect errors and prune infeasible paths. Initially, this state is empty. As traversal iterates over each statement in a block in order, it: (1) evaluates the statement using the current state and (2) records any side-effects of the statement, producing a new state. When it reaches the end of a block it either stops following the path (if the block has no valid successor) or explores the block's successors in random order. After it explores all of the successors, it backtracks to the last block with an unanalyzed successor and continues exploration. If there is no such block, the function has been completely explored and the analysis moves onto the next function.

We describe the handling of loops and conditional expressions in § 5.2 and § 5.4. Unlike some systems (e.g., *xgcc* [7]) we do not build a memoization cache that records the previous states a block was reached in. We found that such a cache was relatively ineffective for ARCHER. Traversal state includes all variables that we have observed a linear relation on, either through assignments or conditionals. This set includes a large number of program variables and, as a result, it rarely reaches a program point with the same information (e.g., at the join point after an if-statement). In reality, we observe that the performance benefit of using a cache does not justify its cost (extra memory footprint, time used to generate fingerprints and compare them, etc). Instead, traversal of a function stops in exactly one of two cases. First, if it exhausts all possible execution paths within a function (modulo some caveats about loops). This is the common case (ARCHER achieves full path coverage on 96% of the functions we analyze in Linux). Second, if it exceeds a pre-determined time limit, which defaults to five seconds.

## 5.2 Handling loops

The traversal algorithm described so far will only terminate for a small subset of loops — those that iterate a constant number of times. On each iteration the traversal module will follow the loop backedge, use the solver to evaluate the exit condition against the values it is tracking, and stop when the exit condition is false. For example, it will iterate the following loop exactly 100 times:

```
i = 100;
while (--i >= 0) {
  ... /* loop body */
}
```

Unfortunately, this simplistic approach will not terminate for loops that have either unknown termination conditions or conditions the solver just cannot evaluate. We have two heuristics to handle these cases.

The first method executes the loop exactly once, based on the heuristic that loops typically execute at least once. (This logic mirrors the PREfix's handling of loop iteration [15].) It works in three steps. First, it executes the loop body a single time. Second, on exit, it invalidates the values of all variables that were assigned to in the loop body. It does this to prevent false positives — since it does not know how many iterations the loop would really perform, the values after just one iteration cannot be trusted to be valid exit values. Third, it then continues the simulation of the current path. In the following example:

```
i = 0; tmp = f(); /* unknown */
while (tmp--) {
    i = i + 1;
}
/* we invalidate the value of i here */
```

the loop body will be executed exactly once, and `i` will be set to unknown on exit.

The second method specializes the first to "iterator loops" that have the following structure:

```
for (i = lb; i < ub; i++) {
    ... /* loop body where i is not modified */
}
```

Using the first technique on such loops needlessly misses many errors — iteration variables such as `i` are often used as indices or offsets for buffer accesses, yet would only be checked with the value `lb` (for the first iteration) and invalidated after the loop exits. Instead, we set the iteration variable to the range $[\mathsf{lb}, \mathsf{ub})$ when we simulate the body and to `ub` after the loop exits. (As with the first method, all other variables set in the loop are cleared on exit.)

In future work, we are planning to implement an induction variable detection algorithm to find a broader range of iterator loops and also to get information on other induction variables.

## 5.3 Errors

The traversal module attempts to check the safety of array accesses, pointer dereferences, or calls to routines that access memory regions specified by a pointer and a size parameter. It checks an array reference a[i] by querying the solver to find out the size of the array `a` and the bound on `i`. It issues an error if i is either (1) negative or (2) greater than or equal to the size of `a`. For a pointer dereference $*$`p`, it queries the solver to determine if the pointer references above or below the object boundary it was set to point to. If so, it emits an error. It checks calls to functions that take pointers and size parameters in a similar manner.

In general, ARCHER will not issue a warning if it does not know the object size or cannot determine the object offset of the reference. Since ARCHER only tracks information within a path, this means that all memory references that it flags descend from one of the following three sources:

1. An array variable declared earlier in the path.
2. A memory region that was dynamically allocated, or

whose size was otherwise inferred (e.g. by a heuristic) earlier in the path.

3. A memory region whose size is known from its static type (e.g., an array structure field).

Errors can either involve these memory objects directly, or through a pointer that can be traced back to them through a series of assignments, conditional branches, and pointer arithmetic operations that happen on the checked path. (The next section is a bit more precise about what exactly these operations can be.) These operations can span (multiple) procedure calls: we summarize unresolved safety conditions involving function parameters in a global per-function database. These conditions will be expressed as error *triggers*–conditions involving parameters and constants under which a memory access error could occur. Triggers of a function will be retrieved and evaluated at each of its calling contexts. If satisfied, an error will be emitted for the buffer access that occurs within the callee.

No other errors will be flagged in the current version of ARCHER. In particular, pointers to anonymous memory regions of unknown type from heap data structures will likely not be checked, since ARCHER will have no idea what their size is.

## 5.4 Evaluating statements

Statements important for traversal fall into several partially-overlapping categories: buffer accesses, assignments, control flow expressions, and function calls. We described buffer accesses above. In this subsection, we briefly describe the effects of the latter three.

**Assignment** ($\mathsf{e}_1 = \mathsf{e}_2$)**:** is handled by first evaluating $\mathsf{e}_2$ against the current solver state, and then binding the result to $\mathsf{e}_1$.

**Control flow edges:** for blocks with multiple successors — in C an if, while, for, or switch — the traversal module uses the solver to evaluate the control-flow condition and skips any edges it determines to be impossible. It follows all edges it cannot demonstrate as infeasible, and adds the condition that would hold on this path to the current solver state.

Since interprocedural analysis happens in bottom-up order, in many cases we cannot evaluate control-flow conditions that involve or derive from function parameters. Whenever either an error would be flagged or a potential error stored in the "trigger database," the traversal module examines the current intraprocedural path and collects all conditionals that derive from function parameters into a *predicate set* for the current path. If the predicate set is non-empty it is stored along with the potential error in the trigger database so that it can be evaluated at call sites of the function to determine if the error path is feasible.

**Function calls** $\mathsf{f}(\mathsf{a}_1, \ldots \mathsf{a}_\mathsf{n})$**:** are processed in two steps. First, we check for potential violations of interface constraints. These can be function constraints built into the checker, such as that the call `bzero(p, size)` writes the memory range $[p, p + size)$. Or they can be the triggers described above that were calculated automatically when we analyzed f. We check triggers of f by (1) retrieving them from the database, (2) evaluating the parameters $\mathsf{a}_1$ through $\mathsf{a}_\mathsf{n}$ in the current solver state, and (3) issuing a warning for each trigger satisfied by the values produced.

Second, we conservatively model the side-effects of function calls on the solver state by invalidating all global variables and arguments passed by reference. For example, in

```
a = 3; global_var = 0;
p = &b;
r = f(&a, p);
```

we invalidate global_var, a, b, and r after the call to f. The points-to information is derived using a very simple per-path alias analysis algorithm. The accuracy of this step can be further improved by using an interprocedural side-effects analysis [17].

## 5.5 Trigger example

In this section, we walk through the bug example in Figure 2 to illustrate how we generate and test triggers.

The analysis starts from the callee `get_drv_by_nr`. At line 9, we encounter an `if` statement with condition `(di < 0)`. Since the value of `di` is unknown, we cannot evaluate the predicate so we schedule both branches for further exploration. The true branch returns control immediately to the caller, and we backtrack to explore the fall-through branch under the condition `(di >= 0)`. Since `di` is a parameter, we also collect this predicate in the path predicate set for trigger generation.

The buffer access occurs at line 12, where `di` is used as an index into the array `drivers`. The traversal module tests the following two constraints for this access

$$\begin{aligned} &1) \qquad \mathsf{di} \geq 0 \\ &2) \qquad \mathsf{di} < 32 \end{aligned}$$

The first constraint is satisfied by the path predicate from the `if` statement, but the second one is undetermined. Fortunately, since it only involves parameters and constants, it can be formulated as a constraint on the function calling interface and checked at the call sites.

To formulate the error trigger, we conjunct the path predicate set ($\mathsf{di} \geq 0$) with the logical not of the safety constraint ($\neg(\mathsf{di} < 32)$) to arrive at

$$(\mathsf{di} \geq 0) \wedge (\mathsf{di} \geq 32).$$

The trigger will be stored in the trigger database for `get_drv_by_nr`.

This trigger is tested at line 19, where `get_slot_by_minor` calls `get_drv_by_nr`. The actual parameter `di` in this case has range $[0, 64)$ (inferred from the enclosing iterator loop), which enthusiastically exceeds the allowed range. The trigger for `get_drv_by_nr` is retrieved, tested, and succeeds (i.e., the error's precondition is satisfied). As a result, we emit an error for this call.

# 6. SOLVER IMPLEMENTATION

At the heart of ARCHER is a linear constraint solver designed for common buffer access patterns. This section first gives an overview of the solver and then delves into the details of how it tracks the values of scalars (§ 6.2) and various properties of arrays and pointers (§ 6.3).

## 6.1 Overview of the Solver

As discussed previously, the solver has two main uses: (1) to detect infeasible paths and (2) to flag illegal memory references. Its main challenge is to do both effectively while scaling to millions of lines of code. We mainly get speed by aggressively specializing the solver to the common-case, while deliberately backing off on tasks empirically shown as less

| | | | |
|---|---|---|---|
| VariableBindings | : | var → scalar \| pointer \| array | |
| ConstantUpperBound | : | symbol → const | (1) |
| ConstantLowerBound | : | symbol → const | (2) |
| ConstantNotEq | : | symbol → const set | (3) |
| SymbolicUpperBound | : | symbol → linear_dev set | (4) |
| SymbolicLowerBound | : | symbol → linear_dev set | (5) |
| SymbolicNotEq | : | symbol → linear_dev set | (6) |
| linear_dev | = | $(a * \text{symbol} + b)/c$ | |
| scalar | = | const \| linear_dev | |
| pointer | = | {length : scalar; offset : scalar} | |
| array | = | {element_size : const; domain : scalar} | |

Figure 4: An operational view of the solver state. The solver state consists of seven mappings: VariableBindings maps program variables to either a scalar object, a pointer, or an array (each of the latter two is represented by a pair of scalar objects). A scalar object is either a constant or a linear deviation from a symbol. Symbols represent unknowns or partially-known values in the program (e.g. variables with known upper bounds). Mappings 2-7 keep track of this partial information (i.e. constant and symbolic bounds and disequality information) for each symbol.

relevant. For example, most boolean predicates in branch statements are simple and our solver can more than adequately prune most impossible paths (even a simpler solver is experimentally shown to be effective at this task [13]). While standard theorem provers and linear constraint solvers provide power, they are often both overly general (which costs), and yet insufficiently tuned to provide enough diagnostic information for common accesses (which results in false positives/negatives). A good example of the cost of power is the standard relation analysis package used in Dor et. al. [6]. While it effectively finds buffer access errors, it requires extensive user annotation, minutes of CPU time, and hundreds of megabytes of memory to process code fragments of no more than a hundred lines of code. We resort to a different strategy. The end result is that each query and update in our solver takes constant time. Although it is not as powerful, it tracks the most common relationships needed to find large numbers of real errors in large unannotated code bases.

Operationally, the traversal module uses the solver to keep track of *assumptions* and check safety properties in the form of *assertions*. Assumptions reflect information about program values collected by the traversal module along the current execution path (e.g. side-effects of assignments, value constraints imposed by `if` statements, etc). Assertions are conditions that must be satisfied in order for the program to be safe (e.g. the array index must be less than the number of elements in the array).

The solver collects available information in the *solver state*. Figure 4 gives a simplistic view of the representation of the state. Internally, the solver tracks three classes of objects in the source program: scalars, pointers, and arrays. The mapping from program variables to these objects is maintained in the simple VariableBindings table shown in Figure 4. Below, we describe the details of how the solver tracks each of these three object types.

## 6.2 Scalar objects

A scalar object (i.e. variables and structure fields of type int, short, or the length of a buffer, etc) in the source program is represented either as a constant, or a linear deviation from a *symbol* in the solver. We define the linear derivation from a symbol $\alpha$ as any $(a \cdot \alpha + b)/c$, where $a$, $b$, and $c$ are integer constants. The solver uses symbols when a variable's exact value is unknown. For example, on entering a function, the parameters are bound to fresh symbols to track the propagation of their values.

Scalar assumptions are passed to the solver in one of the following two forms: Sym op C or Sym op Sym where Sym is a linear derivation, C is a constant, and the operator op is one of $=, >, \geq, <, \leq, \neq$. The traversal module converts assumptions in the source program (e.g. `x < 5`) into one of these two forms using the VariableBindings mapping in the solver state. Although the solver is limited to handling assumptions relating at most two symbols, many variables are linearly related and can be reduced to the same symbol, and then to one of the two forms above. The core solver state is made of seven (initially empty) tables. The first, VariableBindings, maps program variable names to solver symbols. The other six tables, described below, record constraints on these symbols.

**Constant bounds** — the solver uses the ConstantUpperBound and ConstantLowerBound tables to record constant upper and lower bounds for symbols. For example, if variable `x` is represented by symbol $\alpha$, in the true branch of "if (x >= 0)," we will add the mapping $\alpha \mapsto 0$ to ConstantLowerBound. From then on, the solver can derive that $\alpha \geq n$ for all $n \leq 0$. As the solver collects more information it refines these bounds. If both bounds meet at a constant $c$, it derives that the symbol's value is $c$ and replaces each occurrence of that symbol in the solver state by $c$.

**Constant disequalities** — disequalities are common constraints found in C programs (e.g. if (x ! = NULL) {...}). The solver tracks all the constants a symbol is not equal to using the ConstantNotEq table (e.g. $\alpha \notin \{5, 6\}$). It sometimes combines this information with constant bound information to get more precise estimates of a symbol's value. For example, if the solver determines $\alpha <= 5$, and subsequently that $\alpha \neq 5$, it infers that $\alpha <= 4$ and updates ConstantUpperBound accordingly.

**Symbolic equalities** — equalities between program variables are handled by unifying (replacing) the symbols to which the variables are bound. During unification, the solver first expresses the "younger" symbol (determined by the order symbols are introduced) in terms of a linear deviation from the older one, and then uses that to replace each occurrence of the former in the solver state. For example, suppose $x$ is bound to $\alpha$, $y$ to $\beta$, and $z$ to $3 * \beta$, if we subsequently observed that $x = 3y$, the solver will replace $\beta$ by $\alpha/3$. The updated state will contain $x = \alpha$, $y = \alpha/3$, and $z = \alpha$. After unification, constraints on $\beta$ will be transformed and added as those for $\alpha$.

"Age discrimination" during unification is not an arbitrary decision. It simplifies propagation of values passed in from function parameters, which are used to generate triggers for interprocedural analysis. As mentioned above, parameters are bound to fresh symbols on entry, and the seniority rule ensures that they will not be replaced by younger symbols by unification. Therefore, to tell if a program value is derived from a parameter, we simply remember the set of symbols that are initially bound to function arguments.

**Symbolic bounds** — the solver also tracks a list of symbolic upper and lower bounds for each symbol. Since the upper bound of one symbol can be expressed as a lower bound of another, the solver maintains the bound information only for one of the two symbols involved. Again, we use the seniority rule to determine where this information is kept.

**Symbolic disequalities** — treatment of disequalities between symbols is similar to that of constant ones. Again, the seniority rule applies in this case.

## 6.3 Arrays and Pointers

The solver abstracts arrays and pointers as pairs of scalar objects. For arrays, it records the size of its base type in bytes and the number of elements in the array. For example, a 32-bit integer array buf with 30 elements will be represented as the pair $(4, 30)$. To check if the array access `buf[i]` is illegal, the traversal module will check

$$\begin{cases} \mathsf{i} >= 0 \\ \mathsf{i} < \mathsf{length(buf)}, \end{cases}$$

For pointers, the solver records the size of the pointed-to object in bytes and the offset of the pointer relative to the start of the object. For example, on a 32-bit machine, after

```
int *buf = (int*)malloc(len);
int *ptr = buf+1;
```

buf will be represented as (len, 0), and ptr will be (len, 4). To check the validity of $*$ptr, we test

$$\begin{cases} \mathsf{offset(ptr)} >= 0 \\ \mathsf{offset(ptr)} + \mathsf{sizeof}(*\mathsf{ptr}) <= \mathsf{length(ptr)} \end{cases}$$

which in this case translates to len $>= 8$. Notice we recognize "malloc" as an allocation function that returns a buffer of length equal to its first argument. We hardcode a small number of such functions in the checker, and use "size inference" (described in § 7) to derive the rest.

In addition to propagating lengths via assignments (as done above for buf to ptr) the solver also propagates buffer properties across pointer arithmetic operations. For example, if we see the statement "x = p2 − p1;", we know that the C language requires that p2 and p1 point to the same buffer and thus share the same length. Therefore, we unify the lengths of the pair after this program point.

Casting of arrays into pointers is implicit in C, but we need to do an explicit conversion in our representation. To cast a (size, domain) pair into a (length, offset) representation, we simply multiply the domain by the size of the base type of the array to form the length, and use 0 as the offset.

In future work we will add two more properties to the representation of buffers: (1) whether pointers are null and (2) lengths for null-terminated strings. Initial experiments show promise in this approach.

# 7. SIZE INFERENCE

The more allocation and memory touching functions ARCHER knows of, the more effective it will be. A missing allocator means it will not track returned pointers; a missing memory function means it cannot flag parameter overflows. While ARCHER uses an interprocedural analysis to propagate information across procedure boundaries, it requires knowing a set of "root" allocation and memory functions such as "`malloc`," "`memset`," etc. While functions such as these are well-known, many others are domain- or even system-specific. And it is easy to miss even standard functions (did you forget "`wcsnlen`"?).

| Name | Ptr | Sz | S/N | Z |
|---|---|---|---|---|
| **kmalloc** | Ret | 0 | 1147/1176 | 15.0 |
| **copy_to_user** | 1 | 2 | 726/753 | 11.3 |
| **copy_from_user** | 0 | 2 | 742/772 | 11.2 |
| **copy_from_user** | 1 | 2 | 251/264 | 6.1 |
| **copy_to_user** | 0 | 2 | 238/255 | 5.3 |
| **snd_kcalloc** | Ret | 0 | 60/60 | 3.9 |
| **ckmalloc** | Ret | 0 | 6/6 | 1.2 |
| **pnp_alloc** | Ret | 0 | 5/5 | 1.1 |
| **stli_memalloc** | Ret | 0 | 2/2 | 0.7 |
| **slice_dma_loaf** | Ret | 1 | 22/22 | 2.3 |
| **stli_cmdwait** | 3 | 4 | 15/15 | 1.9 |
| **skb_put** | Ret | 1 | 80/91 | 1.9 |
| **BusLogic_Command** | 4 | 5 | 14/14 | 1.9 |
| **i2ob_query_device** | 3 | 4 | 10/10 | 1.6 |
| **SuperTraceWriteVar** | 3 | 5 | 8/8 | 1.4 |
| **fill_note** | 4 | 3 | 8/8 | 1.4 |
| **sock_kfree_s** | 1 | 2 | 7/7 | 1.3 |

Table 1: A partial listing of inferred allocation and memory touching functions. **Ptr** and **Sz** specify the parameter number of the pointer and its associated size respectively ("Ret" means the pointer is returned by the function). **S/N** is the ratio of successes to attempts and **Z** is the value of the computed test statistic. As an example, `kmalloc` returns a pointer whose size is specified by its 0th argument. It had 1147 successful pairings giving a z-rank of 15. The table is split into two populations: those with suggestive names (top half) and "everything else." (bottom half)

We must specify these root functions because, while traditional analysis can compute a transitive set of functions that call these roots directly or through other functions, it is too weak to infer the root set itself. A common practical reason for this is that it requires source for the functions it analyzes, which is often not available because: the code is in a third-party library, is a system call (such as `read` or `write`), or is written in assembly code (as many speed-critical memory touching functions are). Second, and more fundamentally, while many functions abstractly require correct sizes, it may be almost impossible to glean this from analysis of their implementation code. Consider the following function which initializes a structure:

```
void foo_init(struct foo *f, void *buf, unsigned buf_size) {
    f->buf = buf; f->buf_size = buf_size;
}
```

Intuition tells us it is a good bet that `buf_size` is the size of `buf`. Unfortunately, classic analysis cannot do similar inference. As a result, it will miss errors where an incorrect value is passed to `foo_init`. While this is a contrived example, more realistic ones abound — for example, from the compiler's point of view, a `malloc` implementation will look like some sort of complex linked-list manipulation routine and it is very unlikely that it can determine that the sole argument specifies the size of a requested memory block.

This section shows how to automatically infer which parameters specify sizes by turning the problem on its head. Rather than analyzing a routine's *implementation* to determine how it acts we instead analyze the routine's *calling contexts* to do so. The approach uses *statistical belief analysis* [8] to behaviorally infer how likely it is that a programmer believes a given parameter is a size parameter from code examples.

We divide functions into two populations. First, allocation functions that return a pointer of a requested size (such as `malloc`). Given such a function `foo` that takes an argument `n` and returns a pointer `p` we want to answer the question "is `n` the size of `p`?" Second, memory functions that take a pointer and a size parameter less than or equal to the size of the pointed to object (such as `memcpy` or `foo_init` from above). Given such a function `bar` that takes a pointer `p` and integer `n` as arguments we want to answer the question: "is `n` less than or equal to the size of `p`?"

The basic approach to answering either question is simple: for a function `foo` we count how often the correct value `size` is passed as the $i^{\text{th}}$ parameter `n` (a "success") versus how often an incorrect value is passed (a "failure"). Function parameters almost always passed the correct size are likely size parameters; those erratically correct are less likely. We use the z-test statistic [12] to order function-parameter pairs from most to least likely based on the frequency counts of successes and failures. Intuitively, this test statistic weighs both the accuracy of the observed data (the number of successes divided by attempts) as well as the population size (the total number of attempts). The larger the sample, the more confidence we have that the observed ratio of successes to attempts was not coincidental. A degenerate case is that one success and zero failures is 100% accurate, but gives us much less confidence than 9 successes and 1 failure (90% accuracy).

For example, assume that we see the following calls:

```
int *p, *q, *r;
struct foo { int *p, *q; } w;

p = kmalloc(sizeof(int), 0);
q = kmalloc(sizeof(int), 0);
w = kmalloc(sizeof(struct foo *), 0);
w->p = p; w->q = q;
r = kmalloc(x, 0);
memset(p, 0, sizeof(int));
```

The first two calls to `kmalloc` pass the correct size, the third an incorrect size (the size of the pointer rather than the pointed to object) and the fourth we do not know and ignore. In all four calls the second parameter is incorrect. The final call, to `memset`, passes the incorrect size as the second argument and the correct one as the third. Thus, we would rank the tuple `(kmalloc, ret, 0)` first, the tuple `(memset, 0, 2)` second, and finally the tuples `(memset, 0, 1)` and `(kmalloc, ret, 1)` last (if at all).

Making this scheme work in practice requires a few tweaks. First, allocation functions can return arrays of the given type rather than just a single instance. Thus, any non-zero size that is a multiple of the underlying object is counted as a success. We similarly count parameters for memory functions as successes if they are less than or equal to the needed value (e.g., memory copy functions need not copy the entire contents of buffers).

Second, coincidental constants can cause a problem. For example, if we pass in a constant 32 to a routine that often allocates objects of two, four and eight bytes we will falsely think that this flag variable could be their size (since it is larger than them and they evenly divide into it). We refine the behavioral signature we are looking for by noticing that in practice such functions take a range of sizes and thus that a true size parameter will also take on a range of values. We filter out any routines that take on less than 5% new values (this handles a small number of integer typed flags that often have power-of-two values).

Finally, we use the idea of latent specifications [8] to split the results in two populations. The first population are routines that contain names that are common substrings of memory functions: `alloc`, `copy`, `cpy`, etc. These tend to be easier to reason about in general and we also tend to need fewer samples to have reasonable confidence. The second is everything else.

Without the three additional additional techniques described above, the statistical inference proved hopelessly noisy. With the three techniques above, it became happily effective. We found over 70 valid functions in Linux (a good improvement to our manual specification of four root functions) with 16 false positives. Table 1 gives a representative set.

| System | File | FN | LOC | Bug | FP |
|---|---|---|---|---|---|
| **Sendmail**$_{8.12.7}$ | 134 | 829 | 97K | 2 | 4 |
| **PostgreSQL**$_{7.3.1}$ | 404 | 5.7K | 296K | 9 | 0 |
| **OpenBSD**$_{3.2}$ | 850 | 10.7K | 628K | 31 | 12 |
| **Linux**$_{2.5.53}$ | 2,158 | 36.5K | 1.6M | 118 | 39 |
| local | | | | 87 | 33 |
| interprocedural | | | | 31 | 6 |
| security | | | | 21 | 0 |
| **Total** | 1388 | 53.7K | 2.6M | 160 | 55 |

Table 2: Experimental results: **Bugs** is the total number of bugs found; **FP** the number of false positives. **File** is the number of files checked; **FN** the number of functions; **LOC** the number of lines of code.

| Sendmail | PostgreSQL | OpenBSD | Linux |
|---|---|---|---|
| 5m23s | 18m3s | 1hr26m5s | 4hr10m4s |

Table 3: CPU time consumed in analyzing each of the four code bases. Format is in hours, minutes, and seconds. The longest run was Linux, at four hours, ten minutes and four seconds.

## 8. RESULTS

We applied ARCHER to the most recent releases of four large open-source software projects that were then available: Sendmail 8.12.7, PostgreSQL 7.3.1, OpenBSD 3.2, and Linux 2.5.53 (ordered by their respective size), all of which have undergone years of development and they are well-known, high quality systems that are widely deployed for production use. We believe they represent a good mix of security sensitive system software being used today.

We summarize the analysis results in Table 2. The security errors in Linux are found by intersecting buffer access errors with results from a modified tainting analysis [1]: we flag values that could come from a malicious user and any error involving those values could be a potential security hole.

We ran our tests on a single processor Xeon 2.8G system with 512M of memory. The running time of the analysis (excluding parsing) is given in Table 3. On average, ARCHER can analyze 121.4 lines of code per second, making it suitable to be used in a nightly build process for systems of significant size.

Our results are encouraging. ARCHER issued a total of 215 warnings. We manually inspected each of them and identified 160 potential errors. We have filed bug reports with their respective developers, and most of the errors have been confirmed and patched in subsequent releases. The 55 false positives were largely due to errors in our solver, which is in an early development stage. We are eliminating implementation errors daily and we expect a lower false positive count in the future.

# 9. CONCLUSION

This paper has described ARCHER, an automatic tool that uses powerful, simulation-based analysis to find memory access errors. ARCHER works well in practice. It needs no annotations, scales to millions of lines of code, and has found errors in every system we have checked (including hundreds of errors in Linux). It uses a novel, effective statistical approach to infer functions that it should check, guarding against programmer omissions.

## Acknowledgements

This research was supported in part by DARPA contract MDA904-98-C-A933 and by a grant from the Stanford Networking Research Center. Dawson Engler is partially supported by an NSF Career Award. We are also grateful for helpful comments from Ted Kremenek, Sriram Rajamani, Junfeng Yang, Zhe Yang, and the anonymous reviewers.

# 10. REFERENCES

[1] K. Ashcraft and D.R. Engler. Using programmer-written compiler extensions to catch security holes. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 2002.

[2] R. Bodik, R. Gupta, and V. Sarkar. ABCD: Eliminating array bounds checks on demand. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 321–333, June 2000.

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

[4] B. Chess. Improving computer security using extended static checking. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 2002.

[5] Microsoft Corporation. AST Toolkit. http://research.microsoft.com/sbt/.

[6] N. Dor, M. Rodeh, and M. Sagiv. CSSV: towards a realistic tool for statically detecting all buffer overflows in c. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 155–167. ACM Press, June 2003.

[7] D.R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of Operating Systems Design and Implementation (OSDI)*, September 2000.

[8] D.R. Engler, D.Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, 2001.

[9] C. Flanagan and K.R.M. Leino. Houdini, an annotation assistant for ESC/Java. In *Symposium of Formal Methods Europe*, pages 500–517, March 2001.

[10] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 234–245. ACM Press, 2002.

[11] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *Proceedings of the 29th Annual Symposium on Principles of Programming Languages*, June 2002.

[12] D. Freedman, R. Pisani, and R. Purves. *Statistics*. W W Norton & Co., third edition, September 1997.

[13] S. Hallem, B. Chelf, Y. Xie, and D.R. Engler. A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

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

[15] Intrinsa. A technical introduction to PREfix/Enterprise. Technical report, Intrinsa Corporation, 1998.

[16] R.W.M. Jones and P.H.J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Automated and Algorithmic Debugging*, pages 13–26, May 1997.

[17] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 56–67. ACM Press, 1993.

[18] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *10th USENIX Security Symposium*, August 2001.

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

[20] G.C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *Symposium on Principles of Programming Languages*, pages 128–139, January 2002.

[21] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, pages 4–13, November 1991.

[22] B. Schneier. Risks to cybersecurity. Congressional Testimony by Federal Document Clearing House, June 2003.

[23] M.N. Velev and R.E. Bryant. Effective use of boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors. *Journal of Symbolic Computation, special issue on Integration of Automated Reasoning and Computer Algebra Systems*, 2002.

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