Marple: A Demand-Driven Path-Sensitive Buffer Overflow Detector

Wei Le and Mary Lou Soffa
Department of Computer Science
University of Virginia
Charlottesville, VA 22904, USA
{weile, soffa}@cs.virginia.edu

ABSTRACT

Despite increasing efforts in detecting and managing software security vulnerabilities, the number of security attacks is still rising every year. As software becomes more complex, security vulnerabilities are more easily introduced into a system and more difficult to eliminate. Even though buffer overflow detection has been studied for more than 20 years, it is still the most commonly exploited vulnerability. In this paper, we develop a static analyzer for detecting and helping diagnose buffer overflows with the key idea of categorizing program paths as they relate to vulnerability. We combine path-sensitivity with a demand-driven analysis for precision and scalability. We first develop a vulnerability model for buffer overflow and then use the model in the development of the demand-driven path-sensitive analyzer. We detect and identify categories of paths including *infeasible*, *safe*, *vulnerable*, *overflow-input-independent* and *don't-know*. The categorization enables priorities to be set when searching for root causes of vulnerable paths. We implemented our analyzer, Marple, and compared its performance with existing tools. Our experiments show that Marple is able to detect buffer overflows that other tools cannot, and being path-sensitive with prioritization, Marple produces only 1 false positive out of 72 reported overflows. We also show that Marple scales to 570,000 lines of code, the largest benchmark we had.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Security, Verification, Algorithms, Experimentation

1. INTRODUCTION

It has been 20 years since buffer overflow was exploited by the Morris worm [22]. Over the years, buffer overflow has caused huge losses in terms of productivity and trust in information technology. Despite persistent efforts to manage buffer overflow, the statistics show that it is still the most common exploitable vulnerability. In 2007, SecuriTeam documented 482 newly discovered vulnerabilities, 207 of which are buffer overflows [21], and the US-CERT database records 183 buffer overflows out of total 344 vulnerabilities [6]. Due to legacy code and performance, many companies still heavily use C and C++ to develop software. For instance, about 80% revenue for Microsoft comes from products written in C or C++ code [1]. Therefore, much time and resources are spent trying to detect and remove buffer overflows.

Currently, there are dynamic and static approaches to detect buffer overflow. While helpful, both techniques have significant drawbacks. Dynamic tools slow down the execution and thus are not universally applicable. To prevent an exploit, dynamic detectors often halt the program, impacting the software availability. Software companies cannot simply rely on dynamic detection to isolate all potential buffer overflows, because it is very difficult and expensive to develop patches quickly and distribute them safely to a large number of deployed software systems.

Static analysis has been shown to be helpful in identifying buffer overflow before software release [11]. However, applying static analysis in practice requires considerable human effort, either for annotating code to help analysis or for confirming superfluous warnings. Often, the warnings are not systematically prioritized [10, 19, 23, 24]. Many tools, including Splint [10], ARCHER [24] and BOON [23], only report a potentially vulnerable point in the program, such as a statement or a buffer. The code inspectors do not have knowledge about the path that actually produces the overflow. Tools that provide path information include ESPx [11], Prefix [5] and Prefast [19]. The analysis in these tools is performed exhaustively along all paths, and as a result, scalability is an issue. In some cases, the analysis does not terminate in a reasonable amount of time even when constrained to a procedure. ESPx relies on manual annotations for scalability. However, not only is writing and verifying annotation costly, but the correctness of the annotation is not guaranteed [11]. Prefix and Prefast apply many heuristics to select and merge paths, resulting in an unacceptably high false positive rate [5, 19].

In this paper, we present an innovative technique that statically identifies the paths along which buffer overflow occurs in a program using a demand-driven path-sensitive analysis. Our overall goal is to reduce the manual effort required by improving the precision of the static detection and providing explanations for the causes of the identified overflows. Better precision and more helpful information can

SIGSOFT 2008/FSE-16, November 9–15, Atlanta, Georgia, USA

be provided if the path along which a buffer overflow might occur is identified. Hence, our approach is path-sensitive. To address the scalability of path-sensitive analysis for large software, we apply a demand-driven approach. To the best of our knowledge, this is the first research to develop techniques for buffer overflow detection that are demand-driven and path-sensitive.

By applying path-sensitive analysis, we improve the precision of the detection by first isolating infeasible paths that we find and excluding them from further analysis. We then classify individual paths as to whether they are safe or vulnerable. Instead of merging imprecise dataflow facts with precise facts and generating approximate results, we identify paths whose vulnerability cannot be statically determined as don't-know and the reason that makes them don't-know. For paths that are vulnerable, we further categorize them based on the severity of the vulnerability. The classification guides code inspectors to first address those warnings that contain the most severe vulnerabilities and are the least likely to be false positives. To further focus the code inspectors' attention, we report path segments that are relevant to the buffer overflow and pinpoint statements that change the buffer status on the path, with the goal of identifying the root cause of an overflow.

Despite the benefit, path-sensitive analysis can incur unreasonable overhead. Therefore, in our work, we collect path information using a demand-driven analysis. Our insight is that code is not equally vulnerable over the software. A buffer overflow can only occur when a string is written into or read from the buffer, and only statements that possibly update a buffer are relevant to the vulnerability. Therefore, our focus is the path segments starting from the entry of the program to a buffer access. Only statements that can reach the buffer access need to be examined to determine the vulnerability. In the demand-driven analysis, we construct a query at each buffer access specifying whether a buffer overflow could occur at the statement and whether the untrusted user is able to write to the buffer. Answers to the query are collected through a partial reversal of dataflow analysis. Previous research has demonstrated the scalability of demand-driven algorithms [8, 9, 13]. Experiments on a demand-driven copy constant propagation framework report speedups of 1.4–44.3 [9], and a demand-driven pointer analysis is able to scale up to millions of lines of code [13].

We implemented our technique in our tool, Marple, using Microsoft's Phoenix [18] and Disolver [12] infrastructures. We show that different types of paths can traverse a buffer statement. We identify a total of 71 overflows over 8 benchmark programs, 14 previously reported and 57 not reported overflows. We demonstrate the scalability of our tool through successfully analyzing a Microsoft online XBox game with over 570,000 lines of code within 35.4 minutes. We also compare Marple to several existing detectors in terms of precision of the detection, false positives and speed of analysis.

In summary, the contributions of this paper include:

1. the development of the first demand-driven path-sensitive analysis for detecting buffer overflow in large software,
2. the categorization of paths for prioritizing warnings based on infeasibility, severity of the overflow and confidence of the detection,
3. the identification of vulnerable path segments and the statements relevant to the root cause,
4. the implementation, evaluation and comparison of our technique with other buffer overflow detectors, and
5. the presentation of a technique that is scalable and can report buffer overflow with low false positive rates and rich diagnostic information.

This paper is organized as follows. Section 2 explains the benefit of path information and the feasibility of demand-driven analysis for buffer overflow detection. Section 3 describes our framework and algorithms. Experiments and results are given in Section 4, followed by related work in Section 5 and a summary and future work in Section 6.

2. OVERVIEW

In this section, we identify the path information we aim to compute, and discuss the value of the path information in detecting, diagnosing and removing buffer overflow through examples found in real-world applications. We also explore the feasibility of applying a demand-driven analysis to compute path information of interest.

2.1 Path-Sensitivity in our Analysis

We first show that paths across a buffer overflow statement can be distinct with regard to the feasibility, presence of the overflow, and the reasons that cause the overflow. Distinguishing the type of paths is important to achieve precise detection, to prioritize the inspection tasks and to guide the diagnosis and correction. We then define five types of paths that our analysis identifies.

2.1.1 The Value of Path Information

Compared to path-insensitive analysis, path-sensitive detection can achieve better precision because it can take the impact of infeasible paths into the consideration, and it does not merge the dataflow facts collected to determine an overflow. We give an example from `Sendmail-8.7.5` to show the impact of merge on the detection results. In Figure 1, the `strcpy()` at node 5 is not a buffer overflow. However, a path-insensitive analyzer would merge the facts of `buf = xalloc(i+1)` from path $\langle 1-3\rangle$, and `buf = buf0` from $\langle 1,2,4\rangle$, and gets the result of `buf = xalloc(i+1)` $\vee$ `buf = buf0` at node 5; since `buf0` is a buffer with the fixed length, and `a->q_user` gets the content from a network package, the analysis identifies node 5 as vulnerable. Whereas, path-sensitive analysis can distinguish that `buf` is set to be `buf0` only along the path $\langle 1,2,4\rangle$, while along this path, the length of `a->q_user` is always less than the buffer size of `buf0`, and thus the buffer is safe. We discovered this example from our experiments. Our path-sensitive analyzer was able to be aware of the impact of the bounds checking at node 2 and successfully excluded this false positive. But a path-insensitive detector, Splint, incorrectly identified it as vulnerable.

Paths also play an important role in reporting an overflow. As an example, consider a code snippet from `wu-ftpd-2.6.2` in Figure 2. For the `strcat()` statement at line 13, the path $\langle 3,4,7-9,13\rangle$ is always safe and $\langle 3,4,7-9,11-13\rangle$ is always infeasible. Only the path $\langle 3,5-9,11-13\rangle$ can overflow the buffer with a $'\backslash 0'$. If a tool only reports line 13 as an overflow, the code inspector may waste time by manually exploring the infeasible or safe paths. In our work, we

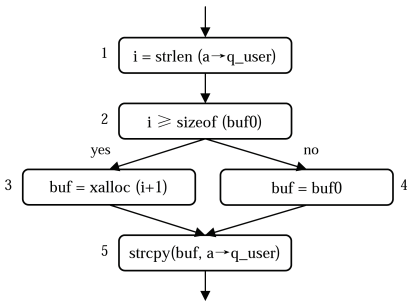

Figure 1: Path-sensitive vs. path-insensitive detection

```
1  /*size of resolved: MAXPATHLEN */
2  char *fb_realpath(const char *path,char* resolved){
3    if(resolved[0]=='/' && resolved[1]=='\0')
4      rootd = 1;
5    else
6      rootd = 0;
7    if(*wbuf){
8      if(strlen(resolved)+strlen(wbuf)+rootd+1
9          > MAXPATHLEN){ ...
10       goto err1; }
11     if(rootd == 0)
12       strcat(resolved, "/");
13     strcat(resolved, wbuf);
14   }
15   ...
16 }
```

Figure 2: Different paths cross an overflow statement

```
1  pw = finduser(buf, &fuzzy);
2  buildfname(pw->pw_gecos,pw->pw_name,nbuf);
3
4  void buildfname(char* gecos,char* login,char* buf){
5    char *bp = buf; char *p;
6    for(p = gecos; *p!='\0' && *p!=',',
7      && *p != ';' && *p != '%'; p++){
8      if(*p == '&'){
9        strcpy(bp, login);
10       *bp = toupper(*bp);
11       while (*bp != '\0')
12         bp++;
13     }
14     ...
15   }
16 }
```

Figure 3: Path-sensitive root causes

report the path, not a program point, along which a buffer overflow could occur. To further focus the code inspector's attention, we only report path segments that are relevant to the overflow, and pinpoint statements on the paths that likely explain the root cause. For example, in Figure 2, we highlight the statements at lines 6,8,9,12 and 13 on the overflow path segment.

Paths are not only helpful for detection and diagnosis, but also can provide guidance for correcting an overflow. We discovered in our research that the root cause for buffer overflow also can be path-sensitive, i.e., more than one root cause can impact the same buffer overflow statement and be located along different paths. Consider an example from `Sendmail-8.7.5` in Figure 3. There are two root causes responsible for the vulnerable `strcpy()` at line 9. First, a user is able to taint the string `login` through `pw->pw_name` at line 2, and there is no validation along the path. Also, the pointer `bp` might already reference an address outside of the buffer `buf` at line 9, if the user carefully constructs the input for `pw->pw_gecos`. This example shows that diagnosing one path for a fix is not always sufficient to correct the overflow.

2.1.2 Path Classification

A goal of our technique is to categorize paths that go through the potentially overflow statements. Through path classification, we aim 1) to distinguish faulty paths from safe or infeasible paths; 2) to prioritize vulnerable paths based on their potential for being exploited; and 3) to isolate paths whose vulnerability cannot be determined by static analysis.

Infeasible: Infeasible paths can never be executed. Identifying paths that go through an overflow statement but actually are infeasible is not helpful for precision or for understanding how a buffer overflow is produced. Previous research has shown that 9–40% of the paths in the program can be statically identified as infeasible paths [4]. Thus detecting them is important to achieve more precise detection.

Safe: Some paths that execute a potentially overflow statement can always guarantee the safety of the buffer regardless of the input. (See Figure 2).

Vulnerable: A path is vulnerable if a buffer may overflow along the path that allows users to write any content to the buffer. Knowing more about who can write to the overflow buffer, e.g., whether the user is anonymous or registered, or from local or network, we can further distinguish the degree of vulnerability for the paths. For instance, when some anonymous network user can write any content to a buffer through some path, it is the most vulnerable.

Overflow-Input-Independent: Not all buffer overflows are easily exploited. For example, when a buffer only can overflow with constant strings in the program, and users are not able to manipulate the content beyond the buffer bound, the chance of exploitation is low compared to a vulnerable buffer. More likely, a crash or corruption of the data will happen. We place these overflows in a lower priority, especially when we have to process a large number of warnings, and when there is a time limit imposed for correcting the code before shipping the software. Path $\langle 3, 5-9, 11-13\rangle$ in Figure 2 is an example of the overflow-input-independent path since the path always overflows the buffer with a $'\backslash 0'$.

Don't-Know: We identify paths as don't-know when the detection for buffer overflow is beyond the power of static analysis. Instead of introducing heuristics to estimate the don't-know facts and merging imprecise dataflow facts with precise ones to generate approximate results, we identify these paths as don't-know, the reason that makes them don't-know and the location of the don't-know factors. Therefore, code reviewers can be aware of them. Annotations or heuristics can be introduced to refine the static detection, and other techniques such as testing or dynamic detection can also be applied to address the unknown warnings. We summarize a set of factors that can make a path don't-know:

1. Library calls: the source of library calls is often not possibly known until link time. In our analysis, we model a set of library calls as to their potential for buffer overflow, e.g., `memcpy()` and `strcat()`. We identify all other library calls that might impact a query as don't-know. We can further model the commonly used library calls to reduce the number of don't-know messages of this type.

2. Loops and recursions: the number of iterations of a

loop or recursive call cannot always be determined statically. We identify three types of loops a query traverses: loops that have no impact on the query, loops where we can compute the symbolic update for the query, and loops where we cannot determine the update of the query. Our analyzer reports don't-know when a query encounters the third type of loops.

3. Non-linear operations: the capacity of a static analyzer is highly dependent on the constraint solver, since the security property under examination will be finally converted to constraints. Non-linear operations, such as bit operations, introduce non-linear constraints which cannot be handled well by practical constraint solvers.

4. Complex pointers and aliasing: pointer arithmetic and several levels of pointer indirection challenge the static analyzer to precisely reason about memory, especially heap operations. Imprecision of *points-to* information also can originate from the path-insensitivity, context-insensitivity or field-insensitivity of a particular alias analysis used in the detection. In our framework, we apply a pointer analysis integrated into the Microsoft Phoenix framework [18] to resolve memory indirection and aliasing, and report those that cannot be handled as don't-know.

5. Shared global variables: globals shared by multiple threads or by multiprocesses through shared memory are nondeterministic.

6. Environment: sometimes the safety of buffer accesses is dependent on the environment on which software will run. For example, in the experiment, we found cases where a buffer access is safe only when a command variable `argc` can never get the value 0, or when some uninitialized integers are always set to be 0 by the environment. We believe security analysis should not make such assumptions. Therefore, we report these results as don't-know.

2.2 Applying Demand-Driven Analysis

Computing path information is challenging since there can exist an exponential number of paths in a program, and determining the existence of buffer overflow requires the tracking of both control flow and value range information. In this subsection, we investigate the opportunities of applying a demand-driven analysis to compute properties of paths.

In demand-driven analysis, a demand is modeled as a set of queries originating at a statement of interest. For example, applying a demand-driven analysis to determine constants, the query is whether a certain variable in the program is a constant. To identify branch correlation, the query is whether the branch can always be evaluated as true or false.

Similarly, for modeling buffer overflow detection using demand-driven analysis, we construct queries as to whether each buffer access in the program is safe. We first raise a query at a statement where a buffer overflow possibly occurs. We then propagate the query backwards along the control flow towards the entry of the program; along the propagation, we collect value range information from the code to resolve the query. At the fork points of branches, the query is propagated to each predecessor node of the branch. At the points where branches merge, the queries from different branches are propagated to the single predecessor of the branch. The analysis terminates when the query propagates onto an infeasible path segment, the information collected can resolve the query, or the entry of the program is reached.

The demand-driven analysis can improve the scalability for buffer overflow detection because only nodes reachable from the buffer access are visited and only information relevant to the query is collected. Queries constructed from each potentially overflow statement are independent, and the analysis could be run in parallel.

3. BUFFER OVERFLOW ANALYSIS

We develop Marple, a demand-driven path-sensitive analyzer for computing buffer overflow paths. In this section, we introduce two important modules of the technique: the vulnerability model and the analyzer. We also briefly describe how a user can apply Marple to diagnose and eliminate buffer overflow.

To build a concrete demand-driven analyzer, we need to answer the following questions [9]: 1) What is the query? 2) How should the query be propagated? 3) What information is used for updating queries? 4) With the information, what are the updating rules for queries? and 5) How is the query resolved? The vulnerability model captures the information needed to answer the above questions and defines rules for constructing, updating and resolving queries for the analyzer. The analyzer drives the query in the code under a set of propagation rules. With the guidance of the vulnerability model, the analyzer takes information from the source program to resolve the query.

3.1 The Vulnerability Model

The vulnerability model for buffer overflow is a 5-tuple: $\langle POS, \delta, UPS, \gamma, r\rangle$, where

1. POS is a finite set of potentially overflow statements where queries are raised,
2. δ is the mapping $POS \rightarrow Q$, and Q is a finite set of buffer overflow queries,
3. UPS is a finite set of statements where buffer overflow queries are updated,
4. γ is the mapping $UPS \rightarrow E$, and E is a finite set of equations used for updating queries, and
5. r is the security policy to determine the resolution of the query.

POS: Buffer overflow can only manifest itself at certain statements, such as where a buffer is accessed. We define such program points as *potentially overflow statements*. Our analysis raises queries from these points and checks the safety for each of them. A program is free of buffer overflow if no violations are detected on any paths that lead to the potentially overflow statements in the program. We recognize that a buffer can be defined only through a string library call or a direct assignment via pointers or array indices. We therefore identify these types of statements as potentially overflow statements for write overflow. Table 1 presents a partial vulnerability model for buffer overflow. In the first column of the table, the first four expressions are types of potentially overflow statements. For the language dependent features, we use C. In the table, the notation $Len(x)$

Table 1: Partial buffer overflow vulnerability model

POS & UPS	Q: Constraints	E: Update Equations
`strcpy(a,b)`	$Size(a) > Len(b)$	$Len'(a) = Len(b)$
`strcat(a,b)`	$Size(a) > Len(a) + Len(b)$	$Len'(a) = Len(b) + Len(a)$
`strncpy(a,b,n)`	$Size(a) > Min(Len(b), n)$	$(Len'(a) = \infty \wedge Len(b) >= n) \vee$ $(Len'(a) = Len(b) \wedge Len(b) < n)$
`a[i] = 't'`	$Size(a) > i$	$Len'(a) = \infty$
`char a[x]`	N/A	$Size(a) = x$
`char *a = (char*)malloc(x)`	N/A	$Size(a) = x/8$
$r(x) : Size(x) > Len(x)$		

represents the location of $'\backslash 0'$ in buffer x, $Len'(x)$ indicates the location of $'\backslash 0'$ in buffer x after it is updated, $Size(x)$ is the buffer size of x, $Min(x, y)$ expresses the minimum value among x and y, and $r(x)$ is the security policy to determine if a write to buffer x is safe.

$\delta : POS \rightarrow Q$: The mapping provides rules for constructing a query from a potentially overflow statement in the code. We model the buffer overflow query for each potentially overflow statement using two elements. The first element specifies whether a buffer access at the statement would be safe, represented as an integer constraint of the buffer size and string length. The second element indicates whether the user input could write to the buffer, annotated as a taint flag. The second column in Table 1 shows the query constraints for the four types of potentially overflow statements listed in the first column.

UPS: To update a query, the analysis extracts information from a set of program points. We identify two types of sources for information, including statements of buffer definitions and allocations, and statements where we are able to obtain values or ranges of the program variables that could impact a buffer access, such as constant assignment, conditional branch and the declaration of the type. In Table 1, the first four expressions in the first column are buffer definitions and the next two are buffer allocations, and they are all members of UPS.

$\gamma : UPS \rightarrow E$: The mapping formats the information as equations so that the analysis can apply substitution or inequality rules to update queries. In the third column of Table 1, we display the equations we derive from the corresponding UPS. The symbol ∞ is a conservative approximation for buffers where $'\backslash 0'$ may not be present.

r: The last part of the vulnerability model is a security policy defined for the analyzer to determine if an overflow could occur. We say a buffer definition is safe if after a write to the buffer, the declared buffer size is no less than the size of the string stored in the buffer. The last row of Table 1 expresses this policy. It should be noted that here we only specify the upper bound of the buffer and only model write overflows, but the technique can be easily extended to also include the lower bound and read overflow. Based on how a query conforms to this policy, the query can be resolved as safe, vulnerable, overflow-input-independent, infeasible or don't-know. These answers categorize the paths through which the query propagates.

3.2 Marple

Figure 4 summarizes Marple and shows the interaction between the vulnerability model and the analyzer. As the initial step, the analyzer identifies and labels the infeasible paths on the program. It then scans the code and identifies

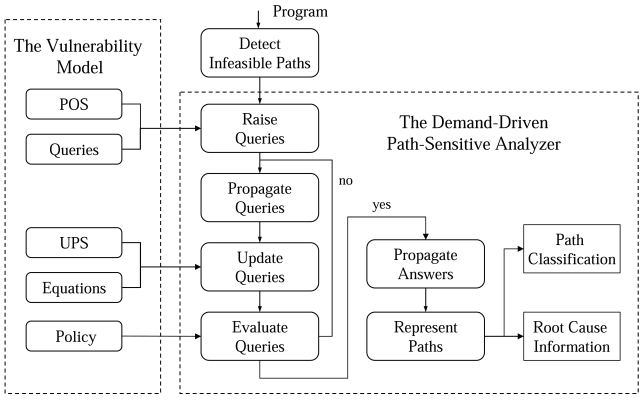

Figure 4: Marple

the statements that match the potentially overflow statements described in the vulnerability model. Queries are constructed from those statements based on the rules defined in the vulnerability model. The analyzer processes one query at a time. Each query is propagated backwards from where it is raised along feasible paths towards the program entry. A set of propagation rules are designed in the analyzer to guide the traversal. At the node where information could be collected, the query is updated using the equations. An evaluator follows to determine if the query can be resolved. If not, the propagation continues. If the query is resolved, the search is terminated. To present the detection results, the answers to the query are propagated to the visited nodes to identify path segments of certain types, and statements for understanding root causes are highlighted.

3.3 The Algorithm

We present the algorithm for computing buffer overflow paths in Algorithm 1. Due to the space limitation, we only describe the intraprocedural analysis here. Our actual framework is interprocedural, context-sensitive and path-sensitive. The side effects of globals are also modeled.

The analysis consists of two phases: *resolve query* and *identify paths*. In the first phase, the analysis first identifies the infeasible paths and marks them on an intermediate representation of the program, namely the Interprocedural Control Flow Graph (ICFG), at line 1 [4]. The analysis at line 2–15 examines the buffers from potentially overflow statements one by one and classifies paths that lead to the buffer access. At line 5, the query is constructed based on the query template stored in the vulnerability model `vm.Q`. The analysis uses a worklist to queue the queries under propagation, together with the node to which a query propagates. At line 6–13, each pair of the node and query is processed.

```
   Input  : ICFG (icfg), Vulnerability Model (vm)
   Output: four types of paths: safe, vulnerable,
           overflow-input-independent and don't-know
 1 Detect&MarkInfeasibleP(icfg)
 2 foreach s ∈ vm.POS do
 3 |  initialize each node n with Q[n] ={}
 4 |  set worklist to {}
 5 |  q = RaiseQ (s, vm.Q); add pair(s,q) to worklist
 6 |  while worklist ≠ ∅ do
 7 |  |  remove pair(node i, query q) from worklist
 8 |  |  UpdateQ(i, q, vm.S, vm.E)
 9 |  |  a = EvaluateQ(i, q)
10 |  |  if a ∈ {Vul, OCNST, Safe, Unknown}
11 |  |  then  add pair(i, a) to A[q]; else
12 |  |  foreach n ∈ Pred(i) do PropagateQ(i, n, q)
13 |  end
14 |  IdentifyP(A[q])
15 end

16 Procedure UpdateQ(node n, query q, ups S, rule E)
17 if n is unknown
18 then info = GetUnknown (n, q, E)
19 else if n ∈ S then info = CollectInfo(n, q, E)
20 Resolve(info, q)

21 Procedure EvaluateQ(node i, query q)
22 SimplifyC(q.c)
23 if q.c = true then a = Safe
24 else if q.c = false ∧ q.taint = CNST then a = OCNST
25 else if q.c = false ∧ q.taint = Userinput then a = Vul
26 else if q.c = undef ∧ q.unsolved = ∅ then a = Unknown
27 else a = Unsolved

28 Procedure PropagateQ(node i, node n, query q)
29 if NotLoop(i, n, q.loopinfo)
30 then
31 |  status = CheckFeasibility(i, n, q.ipp)
32 |  if status != Infeasible ∧ !MergeQ(q, Q[n])
33 |  then add q to Q[n]; add pair(n, q) to worklist
34 end
35 else ProcessLoop(i, n, q)
```

Algorithm 1: Categorizing buffer overflow paths

To update a query, the analysis first determines if the node could impact the buffer we are currently tracking. If so, we extract the information and format it into equations. Procedure `UpdateQ` at line 16–20 provides details. At line 17, the analysis encounters a node that defines a variable relevant to the current query, but the range or value of this variable is not able to be determined statically. We use `GetUnknown` to record this unknown factor based on rules E defined in the vulnerability model. Line 19 finds that node n is a member of UPS, and the analysis then computes $info$ from node n in `CollectInfo`. Finally, `Resolve` at line 20 consumes the information to update the query.

After the query is updated, `EvaluateQ` at line 9 checks if the query can be resolved as one of the defined answers. Line 21–27 describes `EvaluateQ` in a more detail. `SimplifyC` at line 22 first simplifies the constraints in the query. Based on the status of the query after the constraint solving, four types of answers can be drawn. For example, at line 26, `Unknown` is derived from the fact that the constraint $q.c$ is undetermined and the unresolved variable set, $q.unsolved$, is empty. If a query is resolved, its answer, together with the node where the query is resolved is recorded in `A[q]` (see line 11). If the query cannot be evaluated to be any of above four types of answers, `Unsolved` is returned and the query continues to propagate at line 12.

`PropagateQ` at line 28–35 interprets the rules we designed for guiding the query to propagate through infeasible paths, loops and branches. `CheckFeasibility` at line 31 checks if the propagation from the current node to its predecessor encounters an infeasible path and thus should be terminated. `MergeQ` at line 32 determines if the query has arrived at this node before. At line 35, the analysis processes the loop. We observe that when a query enters a loop, one of the following scenarios could occur: 1) the loop does not update the query, and the query remains the same after each iteration of the loop; 2) the query is updated in the loop and the number of loop iterations can be represented as a linear combination of some integer variables. A common example is that we can reason that the loop, `for (int i = 0; i < c; i++)`, iterates c times; and 3) the query is updated in the loop and the number of iterations cannot be simply represented using integer variables. For example, we are not able to express the iteration count for the loop `while(a[i] != '\\')` using integer variables. When the first type of loop is encountered, the analyzer identifies that the query will not change in the loop. The propagation stops traversing the loop after one iteration. To deal with the second and third cases, the analyzer reasons the impact of the loop on the query based on the update of the query per iteration, and the number of iterations of the loop. Since the initial query at the loop exit is known (note our analysis is backwards), obtaining the above two parameters, the analysis is able to compute the query at the loop entry. The third case is more complicated since the number of iterations cannot be symbolically expressed. We introduce a don't-know factor to represent the iteration count and use it to compute the query at the entry of the loop.

Line 14 shows the second phase of the analysis, *identify paths*. At this phase, the analysis propagates the answers from resolved nodes to every node that has been visited, and identifies paths and their types.

Optimizations for Scalability. We developed techniques to further speed up the analysis. One observation is that queries regarding local and global buffers are propagated in a different pattern during analysis. Queries that track local buffers cross into a new procedure only through function parameters or return variables, and the computation for local buffers often does not involve many procedures. However, global buffers can be accessed by any procedure in the program, and those procedures are not necessarily located close on the ICFG. In the worst case, the query cannot be resolved until the analysis visits almost every procedure on the ICFG, and the demand-driven approach cannot benefit much.

To address this challenge, we develop an optimization named *hop*. Our experience on analyzing real-world code demonstrates that although global variables can be defined at any procedure, the frequency of the accesses in a procedure is often low, i.e., the procedure possibly just updates the variable once or twice. Our approach is that when we build the ICFG of the program, we record the location of the global definitions in the procedures. Since the analysis is demand-driven, we are able to know before entering a new procedure the variables of interest. If all variables of interest are globals, we can simply search the global summaries at the procedure, and *hop* the query directly to the node that defines the unresolved variables in the query, skipping

most of the irrelevant code. This *hop* technique also can be applied intraprocedurally when we encounter a complex procedure with many branches and loops. Similar to the global *hop*, we can record the nodes that define local variables in the summary. Although the number of branch nodes could potentially be large, the number of nodes that define variables of interest often is relatively small. Therefore, guided by the demand, we are always able to resolve the query with a limited number of *hops*. In addition to *hop*, we apply optimizations of *advancing* and *caching* as developed by Duesterwald et al [9].

Threats to Validity. Although our framework introduces the concept of *don't-know* to handle the potential imprecision of the analysis, there is still untraceable imprecision that could impact the detection results. For example, we do not model control flows impacted by signal handlers or function pointers, and do not handle concurrency properties such as shared memory. Another example is that we use an intraprocedural field-sensitive and flow-sensitive alias analyzer from the Microsoft Phoenix infrastructure [18], which is conservative. We also can miss infeasible paths from our infeasible paths detection since identifying all infeasible paths is not computable [3].

3.4 User Scenario

To use Marple for diagnosis, the user inputs an application source code into Marple and requests to analyze the potentially overflow statement of interest. If Marple returns a vulnerable path segment, the user then follows the statements highlighted on the path segment to understand and correct the root cause. Our experiments show that a path segment often contains only about 20–30 basic blocks. After the fix is introduced into the code, Marple is run again to determine if all vulnerable paths are eliminated. If not, Marple returns another vulnerable path to the user for further diagnosis. The process iterates until all vulnerable paths are corrected. Similarly, Marple then helps users process overflow-input-independent paths, and lastly don't-know paths.

4. EXPERIMENTAL RESULTS

In order to investigate the scalability and capabilities of our analysis for detecting buffer overflow, we implemented Marple using the Microsoft Phoenix [18] and Disolver [12] infrastructures. The platform we used for experiments is the Dell Precision 490, one Intel Xeon 5140 2-core processor, 2.33 GHz, and 4 GB memory. We selected 8 benchmark programs from BugBench [16], the Buffer Overflow Benchmark [25] and a Microsoft Windows application [17]. All benchmarks are real-world code, and they all contain some known buffer overflows documented by the benchmark designers, which help estimate the false negative rate of Marple. We examined the scalability of our analysis using MechCommander2, a Microsoft online XBox game published in 2001 with 570.9 k lines of C++ code [17].

We conducted two sets of experiments. We first ran our analyzer over 8 benchmark programs and examined the detection results. For comparison, we also experimented with Splint, a path-insensitive static analysis tool developed by Evans [10] on the same set of the benchmarks. In the second set of experiment, we evaluated Marple using 28 programs from the Buffer Overflow Benchmark and compared our results with the data produced by 5 other representative static detectors [25]. We applied the metrics of probability of detection and false alarms developed by Zister et al. for comparison [25]. The results for these two sets of experiments are presented in the following subsections.

4.1 Experiment I

4.1.1 Path-Sensitive Detection

In this experiment, we ran Marple on every write to a buffer in the program to check for a potential overflow. For each buffer write, we excluded infeasible paths, and categorized paths of interest from program entry to the potentially overflow statement into vulnerable, overflow-input-independent, don't-know and safe types. We identified a total of 71 buffer overflows over 8 programs, of which 14 have been previously reported by the benchmark designers and 57 had not been reported before. Among all vulnerable and overflow-input-independent warnings Marple reports, only 1 message is a false positive, which we confirmed manually.

We show the detailed experimental results in Table 2. Column *Benchmark* lists the set of benchmarks we used, the first 4 from BugBench, `wu-ftp`, `Sendmail`, and `BIND` from the Buffer Overflow Benchmark, and the last XBox application MechCommander2. Column *POS* shows the number of potentially overflow statements identified in these programs. Column *Reported Bugs* records the number of overflow statements documented in the benchmarks.

Column *Detected Bugs* summarizes our detection results. It contains two subcolumns. Subcolumn *Reported* displays Marple's detection of previously reported overflows. Comparing the results from this subcolumn to the numbers listed in Column *Reported Bugs*, we show Marple detected 14 out of total 16 reported overflows. Marple identified 1 overflow in `BIND` as don't-know, because the analysis is blocked by some library call, and we missed 1 flaw in MechCommander2, because we do not model function pointers. Subcolumn *New* shows 57 previously not reported overflows we found in the experiment. We manually confirmed that these overflows are actually true buffer overflows. Many of these overflows are located in BugBench. For example, we found 11 previously not reported overflows in `ncompress-4.2.4` and 9 in `gzip-1.2.4`. Bugbench uses a set of dynamic error detectors such as Purify and CCured to detect overflow [16]. Those dynamic detectors terminate when the first buffer overflow on the path is encountered; therefore, other overflows on the same path can be missed. We inspected the overflows reported from Marple but not included in BugBench, and we found that many of new buffer overflow detections are actually located on the same path as other overflows. Furthermore, those overflows located on the same path do not always happen on the same buffer.

The above results show that Marple not only identified most of the documented overflows, but also discovered buffer overflows that have not been reported by the benchmark designers.

Column *Path Prioritization* presents the results of our path classifications. Subcolumns V, O and U show the number of statements Marple reported in the program that contain paths of vulnerable, overflow-input-independent and don't-know. We manually inspected the vulnerable and overflow-input-independent warnings and identified 1 false positive in MechCommander2. The false positive results from the insufficient range analysis for the integer parameters of a `sprintf()`. Marple can properly suppress false positives

Table 2: Detection results from Marple

Benchmark	POS	Reported Bugs	Detected Bugs		Path Prioritization			Root Cause Info	
			Reported	New	V	O	U	Stmt	Ave No.
polymorph-0.4.0	15	3	3	4	6	1	2	2.9	1.7
ncompress-4.2.4	38	1	1	11	8	4	12	3.9	1.0
gzip-1.2.4	38	1	1	9	7	3	18	4.2	1.7
bc-1.06	245	3	3	3	3	3	108	7.1	1.0
wu-ftp	13	4	4	0	3	1	4	6.8	1.0
Sendmail	21	2	2	2	3	1	6	6.5	1.2
BIND	48	1	0	0	0	0	22	N/A	N/A
MechCommander2	1512	1	0	28	28/1	0	487	9.4	1.0

because we use a relatively precise path-sensitive analysis, and we successfully prioritized warnings that are truly buffer overflows by categorizing the low confidence results into the don't-know set. For the don't-knows reported in Subcolumn *U*, we explain what factors cause the don't-know and where the reason for the don't-know appears in the source code.

Consider the benchmark `bc-1.06` as an example to illustrate the don't-know warnings we generate. Among a total of 108 statements that contain don't-know paths, 43 are marked with the factor of complex pointers, 28 result from recursive calls, 15 are caused by loops and 12 are due to non-linear operations. There are also 8 blocked by library calls and 6 dependent on environmental factors such as uninitialized variables. One statement could be labeled with more than one type of don't-know factor, since paths with different don't-know factors can go through the same statement. The computed factors indicate that we can further improve the analysis by applying better memory modeling to resolve pointers, trying to convert non-linear constraints to linear constraints, or annotating the library calls that affect the analysis. The results also help in manual inspections to follow up the don't-know warnings.

The above results validate our hypothesis that although real errors may be in the don't-know set, we are able to report a good number of buffer overflows with very low false positives.

The last column of the table, *Root Cause Info*, presents the assistance of our analysis for helping identify root causes. In our bug report, we highlight statements that update the query during analysis. We count the number of those statements for each overflow path segment. In Subcolumn *Stmt*, we report the average count over all overflow path segments in the program. The results suggest that to understand an overflow, the number of statements that the user has to focus on are actually less than 10 on average. We have shown in Section 2.1.1 that root cause can be path-sensitive. Subcolumn *Ave No.* displays the average number of root causes per overflow for all overflow statements in the program. If the result is larger than 1, there must exist some overflow in the program resulting from more than one root cause. We manually inspected overflow paths and discovered 3 out of 8 programs containing such overflows, and the different root causes for the overflow are all located on different paths.

4.1.2 Two Examples from Results

Here, we show three buffer overflows from two examples which we discovered but had not been previously reported. The first example is from `bc-1.06`. In Figure 5, the overflow occurs at line 8, since the number of elements written to the buffer `env_argv` is determined by the number of iterations of the `while` loop at line 6 and the `if` condition at line 7. However, the execution of both the `while` loop and the `if` condition are controlled by `env_value`, a string that can be manipulated by untrusted users through the environment variable at line 2.

```
1  char* env_argv[30];
2  env_value = getenv("BC_ENV_ARGS");
3  if(env_value != NULL){
4    env_argc = 1;
5    env_argv[0] = "BC_ENV_ARGS";
6    while(*env_value != 0){
7      if(*env_value != ' '){
8        env_argv[env_argc++] = env_value;
9        while(*env_value != ' ' && *env_value != 0)
10         env_value++;
11       if(*env_value != 0){
12         *env_value = 0;
13         env_value++; }
14     }
15     else env_value++; } ...
16 }
```

Figure 5: An overflow in bc-1.06, main.c

```
1  char SourceFiles[256][256];
2  void languageDirective (void) {
3    char fileName[128]; char fullPath[255];
4    while((curChar!='"') && (fileNameLength<127)){
5      fileName[fileNameLength++] = curChar;
6      getChar();
7    }
8    fileName[fileNameLength] = NULL; ...
9    if(curChar==-1) strcpy(fullPath,fileName);
10   else{
11     strcpy(fullPath,SourceFiles[0]);
12     fullPath[curChar+1] = NULL;
13     strcat(fullPath,fileName); }
14   if((openErr = openSourceFile(fullPath))...)
15 }
16 long openSourceFile (char* sourceFileName){ ...
17   strcpy(SourceFiles[NumSourceFiles],sourceFileName);
18 }
```

Figure 6: Overflows in MechCommander2, ablscan.cpp

The second example in Figure 6 presents two overflows we identified in MechCommander2. At line 13, two strings are concatenated into the buffer `fullPath`: the string `fileName`, with the possible length of 127 bytes, and `SourceFiles[0]`, whose maximum length could reach 255 bytes. Both the buffers of `fileName` and `SourceFile` are accessible to the user, e.g., `getChar()` at line 6 gets the input from a file that users can access, to the global `curChar`, which is then copied into `fileName` at line 5. Therefore, given the size of 255 bytes for `fullPath` at line 3, the overflow can occur at line 13 with the user input. This overflow further propagates to the procedure `openSourceFile` at line 14, and makes the buffer `SourceFiles[NumSourceFiles]` at line 17 also unsafe.

4.1.3 Comparison with Splint

We experimentally compared Marple with Splint [10], a path-insensitive static analyzer for buffer overflow detection, and the only static buffer overflow detector we found that can run through most of our benchmarks and report relatively reasonable false positives and false negatives [25]. We ran Splint over the same set of benchmarks, and collected statements that Splint reports as buffer overflow. We examined those warnings against statements Marple identified as containing paths of vulnerable, overflow-input-independent and don't-know. We summarized the comparison in Table 3. For programs of `polymorph-0.4.0`, `BIND` and `MechCommander2`, Splint terminates with parse errors, so we are not able to report a comparison.

Table 3: Comparison of Splint and Marple

Benchmark	Tm	Ts	$(V \cup O) \cap Ts$	$U \cap T's$
ncompress-4.2.4	24	14	1/11	7/5/8
gzip-1.2.4	21	95	8/2	15/3/84
bc-1.06	110	133	2/4	72/28/105
wu-ftp	6	6	4/0	2/1/0
Sendmail	6	8	2/2	3/1/5

The first column Tm in Table 3 shows the total number of statements Marple reports as containing vulnerable, overflow-input-independent and don't-know paths. Since our analysis is path-sensitive, different types of paths can go through an overflow statement [14], e.g., a statement can be counted both in Column V and U in Table 2. Column Ts in Table 3 presents the total number of warnings Splint generates for buffer overflow. Comparing these two columns, we discovered that even if we do not use any further techniques such as heuristics or modeling of library calls to remove don't-knows, Marple generated less warnings, except for the benchmark `ncompress-4.2.4`; however, Splint reported 10 less warnings than our detection on this benchmark because it missed 11 statements we identified as overflow. We manually inspected the warnings missed by Splint, and found that those 11 overflows are correlated in that the overflow of the first buffer consecutively causes the 10 overflows on 3 other different buffers.

The second column $(V \cup O) \cap Ts$ lists the intersection of statements containing paths of overflow-input-independent and vulnerable reported from Marple and the overflow messages generated by Splint. The number before "/" is the number of statements that are listed in both Splint and Marple results, while the number after "/" is the total number of confirmed overflows generated by Marple but missed by Splint. Column $U \cap T's$ compares our don't-know set, U, with the warning set Splint produced excluding the confirmed overflows, annotated as $T's$. In each cell, we present three numbers separated by "/". The first number is the number of statements listed in U but not reported by $T's$. The second number counts the elements in both sets. The third number reports the number of statements from $T's$, but not in U.

Figure 7 presents a summary of the comparison in Table 3. The diagram shows that for the 5 programs listed in Table 3, Splint and Marple identified a total of 17 common overflows (see set A in Figure 7), and Marple detected 19 more flaws that Splint is not able to report (B). There are 38 warnings both reported by Splint and the don't-know set from Marple (C), and thus those statements are very likely to be

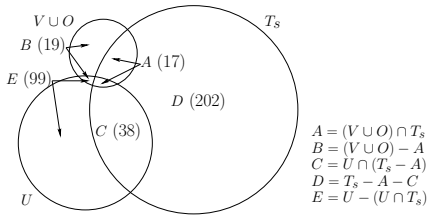

Figure 7: Summary of comparison

an overflow. There are a total of 202 warnings generated by Splint but not included in our conservative don't-know set (D). We manually diagnosed some of these warnings including all sets from `ncompress-4.2.4` and `Sendmail`, 10 from `gzip-1.2.4` and 10 from `bc-1.06` that belong to D; we found that all of these inspected warnings are false positives. The number of statements that are in our don't-know set but not reported by Splint is 99 (E), which suggests that Splint either ignored or applied heuristics to process certain don't-know elements in the program.

4.2 Experiment II

We also compared Marple with other static buffer overflow detectors using the Buffer Overflow Benchmark developed by Zister et al. [25] in terms of detection and false positive rates. The Buffer Overflow Benchmark contains a total of 14 benchmarks constructed from real-world applications including `wu-ftpd`, `Sendmail` and `BIND`. Each benchmark contains a "bad" program, where several overflows are marked, and a corresponding "ok" version, where overflows in the "bad" program are fixed. Zister et al. evaluated five static buffer overflow detectors: ARCHER, BOON, UNO, Splint and PolySpace (a commercial tool), with the Buffer Overflow Benchmark. The results show that 3 out of the 5 above detectors report less than 5% of the overflows in the benchmarks, and the other 2 have higher detection rates, but the false positive rates are unacceptably high at 1 false alarm in every 12 lines of code and 1 in every 46 lines of code.

The results of the evaluation have been plotted on the ROC (Receiver Operating Characteristic) curve shown in Figure 8 [25]. The y-axis $p(d)$ shows the probability of detection, computed by the formula $C(d)/T(d)$, where $C(d)$ is the number of marked overflows detected by the tool and $T(d)$ is the total number of overflows highlighted in the "bad" program. Similarly, the x-axis $p(f)$ represents the probability of false alarms, computed by $C(f)/T(f)$, where $C(f)$ is the number of "ok" statements identified by the tool as an overflow, and $T(f)$ is the total number of fixed overflow statements in the "ok" version of the program. The diagonal line in the figure suggests where a static analyzer based on random guessing would be located. The uppermost and leftmost corner of the plot represents an ideal detector with 100% detection and 0% false positive rates.

We ran Marple over the Buffer Overflow Benchmark and rendered our results of $p(f)$ and $p(d)$ on the plot. In Figure 8, we computed two points for Marple. $Marple\_A$ is computed using only overflow-input-independent and vulnerable warnings, while $Marple\_B$ is derived also using don't-know messages, i.e., a don't-know warning is counted both into $C(d)$ as a detection and into $C(f)$ as a false positive. $Marple\_A$ shows that we can detect 49% of overflows with

Table 4: Benefit of demand-driven analysis

Benchmark	Size (LOC)	Blocks		Procedures		WorkList Max Size	Time
		Total	Visited	Total	Visited		
polymorph-0.4.0	1.7 k	323	41	11	4	6	1.3 s
ncompress-4.2.4	2.0 k	473	269	13	4	56	1.3 s
gzip-1.2.4	8.2 k	1,218	482	42	17	110	26.2 s
bc-1.06	17.7 k	3,035	1,489	119	77	677	3.5 min
wu-ftp	0.4 k	84	50	5	5	31	2.1 s
Sendmail	0.7 k	140	81	7	4	12	1.1 s
BIND	1.3 k	226	83	9	3	1	0.9 s
MechCommander2	570.9 k	57,883	25,069	3,259	1,689	944	35.4 min

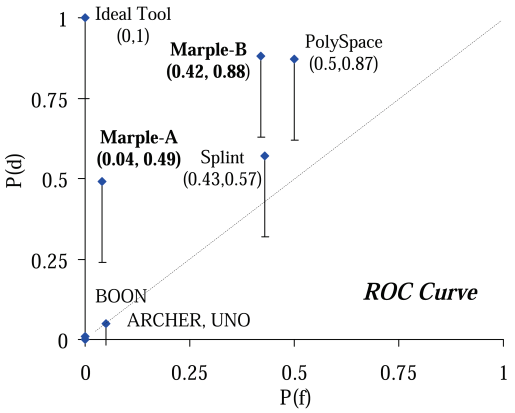

Figure 8: Comparison of Marple with other five static detectors on ROC plot

a 4% false positive rate. *Marple_B* achieves better results both in false positive and negative rates than PolySpace and Splint. Our results indicate that Marple can more precisely detect buffer overflows with high detection and low false positive rates. We discovered that although the don't-know warnings should not miss overflows since they are computed conservatively, we obtained 88% detection rate. The reason for this is that some overflows in the benchmarks are caused by integer errors or they are read overflows, and we have not yet modeled these in our analysis.

4.3 Benefit of Demand-Driven Analysis

To evaluate the scalability of our analysis, we measured both the time and memory of analyzing 8 programs. Table 4 Column *Size* lists the size of benchmark programs in terms of lines of code. Columns *Blocks* and *Procedures* compare the number of total blocks and procedures on the ICFG of the program, listed under Subcolumns *Total*, to the number of blocks and procedures Marple visited during analysis, displayed in Subcolumns *Visited*. The results show that because we direct the analysis only to the code relevant to buffer overflow, the analysis only visited an average of 43% nodes and 52% procedures on the ICFG for 8 programs. Column *WorkList Size* shows the maximum number of elements in the major worklist in analysis. The actual memory measurement reports that all 8 benchmark programs can be analyzed using less than 2.5 GB memory.

Column *Time* reports the time that our analysis uses for each program. The results show that analyses for all benchmarks can finish within a reasonable time, and we successfully analyzed MechCommander2 within 35.4 minutes. We compared the performance of our analysis with two path-sensitive tools, ARCHER [24] and IPSSA [15]. ARCHER uses an exhaustive based search and achieves the speed of analyzing 121.4 lines of code per second [24]. IPSSA detects buffer overflows on the SSA annotated with path-sensitive alias information; its average speed for 10 programs in the experiments is 155.3 lines per second [15]. Marple reports the speed of analyzing 254.7 lines per second over our benchmark programs.

5. RELATED WORK

Buffer overflow can cause critical consequences in software systems. Many static tools have been developed to detect buffer overflow before software release. The general approach is to map memory safety problems to integer range analysis, abstract interpretation, symbolic execution or type inference, and apply annotations or heuristics to achieve scalability. Important tools for buffer overflows include Prefast [19], Prefix [5], ESPx [11], BOON [23], Splint [10], ARCHER [24], and PolySpace [2]. Most of these detectors apply an exhaustive search over the program for potentially violations of buffer access, and often report high false positives or require annotations to achieve precision.

Path-sensitive tools for software vulnerability detection include ARCHER [24], IPSSA [15], and MOPS [7]. ARCHER exhaustively evaluates every path using symbolic execution and takes infeasible paths into consideration. Although the analysis is path-sensitive, ARCHER does not report the concrete faulty paths as detection results, but only the statement where the buffer overflow occurs [24]. The heuristics used to make the analysis scalable and suppress false positives can potentially cause the low detection rate as reported in the previous evaluation [25]. IPSSA performs a hybrid pointer analysis and annotates the results on SSA. IPSSA based buffer overflow detection uses a forward analysis starting from where a user input can be introduced into the system. The detector cannot reason about the dynamic buffer allocations or bounds checks located before the user inputs, if any, which could either cause the detection to fail or produce false positives. The analysis also does not consider any assignment to the buffer through array characters, resulting in another source of imprecision. The scalability of this method is not evaluated in the paper [15]. MOPS is targeted to identify vulnerabilities that can be represented in a finite state machine. It models a program as a push down automaton and applies the push down model checker to determine the reachability of vulnerable states in the PDA. However, traces on PDAs are not always feasible paths in the program. According to the author, MOPS generates many false positives, e.g., among 1378 warnings for misuse

of `strncpy()`, only 11 are confirmed as real flaws [7].

The demand-driven approach has been used to compute dataflow problems such as reaching definitions and constant propagation [9]. There has also been work on demand-driven pointer analysis, infeasible paths computation, and memory leak detection [4, 13, 20]. In our previous paper [14], we used a demand-driven algorithm to identify the five types of paths for a buffer overflow statement, and experimentally showed that all categories exist in applications.

6. CONCLUSION AND FUTURE WORK

This paper presents a demand-driven path-sensitive analysis that leverages the capabilities of the static analysis for achieving both scalability and precision in buffer overflow detection. Since the analysis and the reports of detected bugs are both based on paths, we are able to provide better explanations for the detection results. We prioritize warnings that contain severe vulnerabilities and are more likely true errors, and highlight statements that are important in understanding root causes. We experimentally show that Marple is scalable and can detect buffer overflows that have not previously been reported in the benchmarks and also not found by Splint. We also find in our experiments that 99% of overflows reported by Marple are real buffer overflows. Our future work is to develop techniques that can further address the don't-know warnings generated from our analysis.

7. ACKNOWLEDGMENTS

This research was supported in part by grants from the Microsoft External Research & Programs. We especially thank Yan Xu for the support of this project, and we are very grateful for the help from Andy Ayers, Chris McKinsey and Joseph Tremoulet in using Phoenix. We also thank Manuvir Das for his helpful comments.

8. REFERENCES

[1] Personal communication with John Lin from Microsoft.

[2] Polyspace. `http://www.polyspace.com`.

[3] T. Ball and J. R. Larus. Program flow path. Microsoft Technical Report MSR-TR-99-01, 1999.

[4] R. Bodik, R. Gupta, and M. L. Soffa. Refining data flow information using infeasible paths. In *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 1997.

[5] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 2000.

[6] CERT. `http://www.cert.org/`.

[7] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, 2002.

[8] E. Duesterwald, R. Gupta, and M. L. Soffa. A demand-driven analyzer for data flow testing at the integration level. In *Proceedings of 18th International Conference on Software Engineering*, 1996.

[9] E. Duesterwald, R. Gupta, and M. L. Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM Transactions on Programming Languages and Systems*, 1997.

[10] D. Evans. Static detection of dynamic memory errors. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, 1996.

[11] B. Hackett, M. Das, D. Wang, and Z. Yang. Modular checking for buffer overflows in the large. In *Proceeding of the 28th International Conference on Software Engineering*, 2006.

[12] Y. Hamadi. Disolver : A Distributed Constraint Solver. Technical Report MSR-TR-2003-91, Microsoft Research.

[13] N. Heintze and O. Tardieu. Demand-driven pointer analysis. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, 2001.

[14] W. Le and M. L. Soffa. Refining buffer overflow detection via demand-driven path-sensitive analysis. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2007.

[15] V. B. Livshits and M. S. Lam. Tracking pointers with path and context sensitivity for bug detection in c programs. In *Proceedings of the 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2003.

[16] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *Proceedings of Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[17] Microsoft Game Studio MechCommander2. `http://www.microsoft.com/games/mechcommander2/`.

[18] Microsoft Phoenix. `http://research.microsoft.com/phoenix/`.

[19] Microsoft Prefast. `http://www.microsoft.com/whdc/devtools/tools/prefast.mspx`.

[20] M. Orlovich and R. Rugina. Memory leak analysis by contradiction. In *Static Analysis, 13th International Symposium*, 2006.

[21] SecurityTeam. `http://www.securiteam.com/`.

[22] E. Spafford. A failure to learn from the past. `http://citeseer.ist.psu.edu/spafford03failure.html`.

[23] D. Wagner, J. S. Foster, and E. A. B. hand Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of Network and Distributed System Security Symposium*, 2000.

[24] Y. Xie, A. Chou, and D. Engler. ARCHER: Using symbolic, path-sensitive analysis to detect memory access errors. In *Proceedings of 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2003.

[25] M. Zitser, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2004.