Refining Buffer Overflow Detection via Demand-Driven Path-Sensitive Analysis

Wei Le and Mary Lou Soffa

Department of Computer Science
University of Virginia
Charlottesville, VA 22904, USA
{weile, soffa}@cs.virginia.edu

Abstract

Although static analysis is an important technique for detecting buffer overflow before software deployment, current static tools rely on considerable human effort for annotating code to help analysis, or for diagnosing warnings, many of which are false positives. This paper presents an analysis technique that refines information about the paths that involve a potential buffer overflow to help in the diagnosis and debugging of vulnerabilities. Instead of only reporting a vulnerable buffer or statement in the program, which most tools do, our analysis categorizes paths of a possibly vulnerable statement into five types: Vulnerable, Overflow-User-Independent, Safe, Infeasible and Don't-Know. Thus, safe and infeasible paths can be excluded from being inspected, providing focus on problematic paths. For scalability, we designed and implemented our analysis as an interprocedural, demand-driven path-sensitive analysis. Our experiments demonstrate that various path types do go through a possibly vulnerable buffer statement. The results also indicate that our technique is efficient and practical.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.5 [*Software Engineering*]: Testing and Debugging

General Terms Algorithms, Reliability, Security, Verification

Keywords Path-Sensitive, Demand-Driven, Infeasible Paths

1. Introduction

Although much effort has been expended to detect and avoid buffer overflow in software, we are still plagued with exploits that are costly to fix, disruptive, and promote a general loss of trust in software. Since many applications are written in unsafe languages and it is difficult for programmers to correctly write applications that use buffers, buffer overflow is still being introduced into software and is the most commonly exploited vulnerability [5, 21]. In 2006, SecuriTeam reported 134 vulnerable overflows, a quarter of the total security warnings [21], and many of them have caused severe impact such as unauthorized access and denial of service. To detect vulnerabilities, dynamic detectors are used but they slow down the execution by a factor of 2 to 30 due to the increase of code size, branch mispredictions and data cache misses [24]. Therefore dynamic buffer overflow detection is difficult to apply for time constrained software. In addition, patches to fix the vulnerability are expensive due to the number of computers typically effected. For these reasons, a number of software companies rely on static analysis to detect buffer overflow before software release [12, 13].

However, current static tools require considerable human effort, either for diagnosing warnings or for annotating programs to help analysis [4, 11, 13, 18, 22, 24]. Many tools report warnings about potentially vulnerable program points, such as statements or buffers, for example, Splint, BOON and ARCHER [11, 22, 24]. The code reviewer has no knowledge about the paths through the program point that actually produce the vulnerability. Tools that report vulnerable paths instead of statements include Prefix, ESPx and Prefast [4, 13, 18]. The analysis is performed exhaustively along all program paths. The challenge for these tools is scalability, in particular when the vulnerability may cross procedure boundaries. As a result, the tools sometime have to give up after exploring a certain number of paths [18]. Although heuristics can be applied to select and merge paths, excessive warnings are produced [4]. Some tools address scalability by introducing annotations to specify the buffer contract between procedures and thus turn the buffer overflow detection intraprocedural [13]. But both writing and verification of annotations are costly, and thus, correctness of annotations is not guaranteed.

This paper presents an interprocedural demand-driven path-sensitive analysis with the goal of reducing the effort required to identify program paths that are vulnerable and providing more precise information about the vulnerability to help users find the root cause. Our analysis classifies paths as infeasible, safe, vulnerable with potential for exploits, overflow with little chance to be exploited, and don't-know. Our analysis is driven by statements that have a definition or redefinition of a buffer. By using a demand-driven algorithm, our analysis is directed to those paths that can be executed and maybe vulnerable, and the analysis terminates as soon as the vulnerability decision is discovered. Through our analysis, we exclude paths that are infeasible and safe, and prioritize paths that can overflow based on their chance of being exploited.

In summary, the contributions of the paper include:

1. A categorization and identification of five types of paths for buffer overflow.
2. An interprocedural demand-driven path-sensitive diagnosis tool for identifying the types of paths through a potential overflow buffer.
3. Experimental results that demonstrate the path types existing in real programs and the time and space costs of the analysis.

PASTE'07 June 13–14, 2007, San Diego, California, USA.

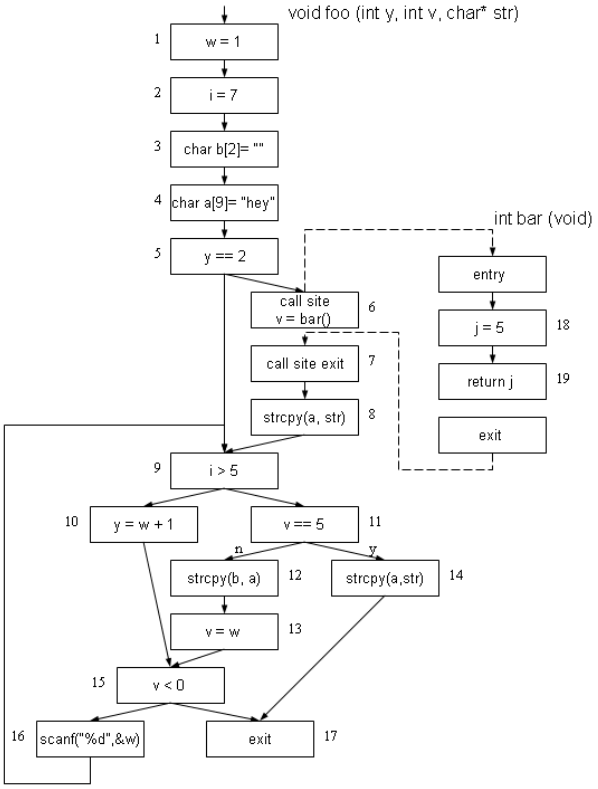

Figure 1. A Simple Example

```
1  void ftpBuildTitleUrl(FtpStateData *ftpState){
2    request_t *request = ftpState->request;
3    size_t len;
4    char *t;
5    len = 64
6          + strlen(ftpState->user)
7          + strlen(ftpState->password)
8          + strlen(request->host)
9          + strlen(request->urlpath);
10   ...
11   t = xcalloc (len, 1);
12   strcat(t, "ftp://");
13   if (strcmp(ftpState->user, "anonymous")) {
14         strcat(t, rfc1738_escape_part(
15                     ftpState->user));
16         if (ftpState->password_url) {
17           strcat(t, ":");
18           strcat(t, rfc1738_escape_part(
19                       ftpState->password));
20         }
21         strcat(t, "@");
22   }
23   strcat(t, request->host);
24   ...
25 }
```

Figure 2. Code Snippet from Squid-2.3 ftp.c.

Section 2 gives an overview of our approach using a simple example. Section 3 defines five path types. Section 4 describes the demand-driven model and framework. Experimental results are given in Section 5, followed by the related work in Section 6, and a summary in Section 7.

2. Overview and an Example

We present an overview of our technique through an example in Figure 1 that is based on the work of Bodik et al [3]. In the discussion of the example, we assume that a demand-driven path-sensitive infeasibility analysis has already been done [3].

In Figure 1, nodes 3, 4, 8, 12 and 14 write strings to a buffer. Overflow might occur at any of these five nodes. The nodes 3 and 4 are identified to be safe buffer definitions since both the buffer size and the content of the buffer can be determined locally. For the nodes 8, 12 and 14, we need more context to make a judgment as to their vulnerability.

Consider the buffer at node 12, which is a string copy. We first construct and raise the query $BSize(b) > TPos(a)$ at node 12, which means after the `strcpy`, if the size of buffer b, $BSize(b)$, is larger than the index of the null string terminator in the buffer, $TPos(a)$, the buffer access is safe (we assume the index starts at 0). The query is then propagated backwards to the nodes 11 and then 9. No information is collected at these two nodes to update the query as they have no impact on the query. At each step of propagation, we cache the query at the node for reuse. At node 9, we propagate the query along three paths to its predecessors, namely nodes 16, 8 and 5. At node 16, the query enters a loop, which does not update the query. Thus the query is merged at node 9 and not propagated further. The query from node 8 reaches an infeasible path segment, $\langle 8, 9, 11, 12\rangle$, and terminates. From node 5, the query is propagated to node 4. Here, it is discovered that $TPos(a) = 3$, and the query is updated to $BSize(b) > 3$. Meanwhile, a flag is set in the query to indicate the buffer content currently is constant. The propagation continues and the information at node 3 indicates $BSize(b) = 2$. Thus the query is resolved by $2 > 3$ as false. The constant flag shows the buffer is overflowed by some constant string and is not dependent on input. Note that propagation halts as soon as the query is resolved. We propagate this answer to the nodes we have visited to determine the path.

3. Path Types

In this section, we describe the types of paths that we identify. We consider both feasibility and buffer overflow in the classification. Our goals for categorizing paths include: 1) distinguishing faulty paths from safe and infeasible paths, 2) prioritizing vulnerable paths based on their possible exploitation consequences, and 3) identifying what paths should be further explored to determine its vulnerability. We now classify paths that go through a potentially vulnerable statement, *PVS*.

Infeasible: Infeasible paths can never be executed. Therefore, the overflow property of buffers on those paths is meaningless for judging whether or not a buffer overflow exists. Infeasible paths occur when there exist branch correlations along a path that make a branch unexecutable. Previous work shows that there are 9–40% statically detectable branch correlations [2], which indicates that it is necessary to try to identify them. However, identifying all infeasible paths is not computable [1].

Safe: Given a PVS, some paths that execute the PVS are safe either because the bounds checking is properly done along the path or the overflow will not happen under any input that leads to traversal of these paths. For example, in Figure 2, code from Squid-2.3 ftp.c shows that the path $\langle 1 - 13, 23\rangle$ is always safe regardless of a possibly vulnerable `strcat` at the line 23.

Vulnerable: Many attacks through buffer overflow are conducted through external inputs, e.g., command line, file, network packets or environment variables. Attack incidents show that important attack data such as control transfer code in the control-data attack and data used to corrupt program variables in the non-control-data attack [7] are usually injected through the overflowed buffer [5, 21]. The data such as malicious payload for stack smashing [19] or parameters for system calls to launch return-into-lib attack [23] are also often located in the overflowed buffer [5, 21]. Therefore we consider a buffer that can overflow with user input as a likely exploitable buffer. If feasible paths reach these types of buffer, we call them vulnerable paths. In Figure 1, the path

$\langle 1-6, 18, 19, 7, 8\rangle$ is considered vulnerable if str gets a string from the user input.

Overflow-User-Independent: Not all buffer overflows are exploitable by unknown users, e.g., when the buffer can overflow only with constants in the program, the chance of exploitation is low compared to a buffer overflowed through external input. A crash or corruption of the data could still be possible. Paths containing these buffers are placed in a lower priority than vulnerable paths. This prioritization is useful when the message volume is large and there is a time limit imposed for correcting the code. In a large code base, it is impossible to fix every bug before releasing the software. $\langle 1-5, 9, 11, 12\rangle$ in Figure 1 is an overflow-user-independent path. It can overflow the buffer `b` with the C string `"y"` (the character '`y`' followed by the null terminator '\0').

Don't-Know: We identify paths as don't-know when their detection is beyond the power of static analysis, e.g., the library source will not be known until link time. Instead of merging imprecise dataflow facts with precise facts and generating conservative results, we identify those don't-know paths and the reason that makes them don't-know so that a code reviewer is aware of them and other detection facilities such as testing can be applied. In Figure 2, paths entering the `if` statement at line 13, such as $\langle 1-23\rangle$, encounter the library call `rfc1738_escape_part` at line 14 or line 18, which may define `ftpState->user` and `ftpState->password`. Thus the content written to the buffer `t` cannot be judged by the static analyzer.

4. Buffer Overflow Analysis

A demand-driven analysis has a number of advantages that lead to scalability. Firstly, each query of a PVS is independent and thus all queries can be performed in parallel. The intermediate queries generated for solving a query can be cached and reused for checking queries from other buffers. Also, the analysis only visits the nodes reachable from the PVS, collects information related to user queries, and terminates as soon as the query is resolved. Experiments on a demand-driven copy constant propagation framework reported the speedup of a factor of 1.4–44.3 for a set of benchmarks [10]. A demand-driven approach also provides a user with flexibility for diagnosing and debugging errors with regard to which buffer should be checked.

Our analysis for buffer overflows instantiates and extends a general demand-driven framework based on Duesterwald et al's work [10]. The demand-driven approach has showed scalability for solving dataflow problems such as reaching definition and constant propagation [10]. However, according to our knowledge, it has not been investigated for detecting software errors or vulnerabilities. According to Duesterwald et al [10], in order to build a concrete demand-driven analyzer, we should answer the following questions: 1) What is the query and where is it raised? 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? 5) When is the search terminated?

4.1 The Demand-Driven Model

For designing a buffer overflow demand-driven analyzer, we develop a demand-driven model using the above questions as a guide. Some descriptions below are language dependent and we use C and C++ for explanation.

Query. We define a set of program points of interest as PVSs where queries are raised. Conservatively, we assume that every definition to a buffer (write to a buffer) is dangerous, thus is a PVS. A buffer overflow query is regarding whether a buffer access at the PVS would be safe and whether the user input could write to the buffer. These two parts are represented as a constraint of buffer size and string length, and a flag in the query. We designed a set of query templates for PVSs. The second column of Table 1 shows some example constraints for the selected PVSs.

Information for Updating Queries. There is a set of program points where information could be extracted to update queries. They include buffer definitions, buffer allocations, index definitions, alias operations and pointer arithmetic. Buffer definitions are PVSs, as we explained above. Buffer allocations often specify the size of a buffer. For example, stack buffer can be declared as `char a[10]`, and the heap buffer is usually allocated by the `malloc` family of library calls. The information also can come from constant assignment, branch conditions and the declared type. The extracted information is formatted as assertions so that the analysis can use substitution or inequality rules to update queries. The third column of Table 1 showed some assertions formatted from the node of the buffer definition and allocation.

Propagation Rules. Based on the work of Bodik and Duesterwald et al [2, 3, 10], we designed rules for propagating queries interprocedurally, incorporating feasibility, and handling loops.

We only propagate queries interprocedurally when we are confident that this call will update the query. To determine if a procedure impacts a query, we first check if any unknown variables in the query constraint defined by a global, a return or reference parameters of the call. If so, we perform a simple linear scan to determine if a statement in the procedure can possibly update a query. Our analysis is context sensitive. Therefore the query will be propagated back to the call site after it propagates out of the procedure. Only a newly raised query will be propagated to all call sites of its raised procedure.

In order to make sure a query is not propagated along an infeasible path, we first detect infeasible paths using branch correlation and mark infeasible path segments on the edges of an Interprocedural Control Flow Graph (ICFG) [3, 14]. During buffer overflow analysis, the query terminates when it encounters an infeasible path segment.

We also developed propagation rules for loops. In our analysis, users can specify the number of iterations they would like to compute for the loop. We track the query precisely when the iteration of the loop has not reached the threshold. Sometimes, the query can only be updated during the first iteration of the loop or even cannot be updated in the loop. In this case, queries from different iterations are merged. Sometimes, loop iteration is bounded by some constant integer, e.g., the loop `for (int i = 0; i < 100; i++)` will iterate 100 times. When this type of loop is detected, we compute the final query with the upper limit of loop iteration. There are also loops whose iterations are not regular and might be determined by the user input. We represent the query after these loops as unknown in terms of loop iterations, and continuously propagate the query to see if any user input can control this loop to overflow the buffer.

Resolving the Query. For buffer overflow detection, the general property we aim to check is: after a write to the buffer, the declared buffer size must be no less than the size of the string stored in the buffer. It should be noted that we only specify the upper limit of the buffer and for the buffer write overflow. But the technique can be easily extended to also include the lower bound and read overflow. Based on the general property, we further expand the overflow properties to be vulnerable, overflow-user-independent, safe and don't-know, each of which corresponds to a path type defined in Section 3. During the analysis, if the information collected in the query is enough to be evaluated as one of the above properties, the query is solved.

4.2 The Framework

Figure 3 presents the framework for the buffer overflow analyzer. Our goal is to compute types of paths we defined in Section 3. The demand-driven model in the previous section guides the analysis to

Code	Constraints	Assertions
`strcpy(a,b)`	$BSize(a) > TPos(b)$	$TPos'(a) = TPos(b)$
`strcat(a,b)`	$BSize(a) > TPos(a) + TPos(b)$	$TPos'(a) = TPos(b) + TPos(a)$
`strncpy(a,b,n)`	$BSize(a) > Min(TPos(b), n)$	$(TPos'(a) = \infty \&\& TPos(b) >= n) \parallel$ $(TPos'(a) = TPos(b) \&\& TPos(b) < n)$
`a[i] = 't'`	$BSize(a) > i$	$TPos'(a) = \infty$
`char a[x]`	N/A	$L(a) = x$
`char *a = (char*)malloc(x)`	N/A	$L(a) = x/8$

Table 1. Examples of Buffer Overflow Constraints and Assertions for C (TPos(x): index of the null terminator in buffer x; TPos′(x): index of the null terminator after the buffer definition; Min(x,y): minimum value among x and y; BSize(x): buffer size of x)

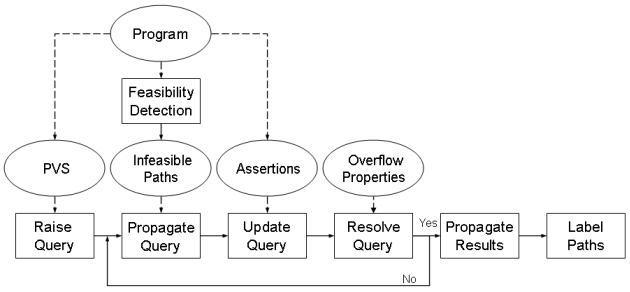

Figure 3. Framework for Buffer Overflow Analysis

identify a PVS, raise and solve the query with the proper information abstracted from source code. As the first step, the analysis detects infeasible paths and mark them on the ICFG. Second, a PVS is identified and a query is raised at this PVS. Then the query is propagated backwards under a set of propagation rules, and when it reaches a node, the information is collected to update the query. Every time a query is updated, the analysis judges if the query is solved to be one of the overflow properties. If not, the query will be continuously propagated. This process continues iteratively until the query is solved. After all queries are solved, the results are propagated from solved nodes to all previously visited nodes, and the path types are labeled on the edge. We can then identify the path based on the edge markings.

5. Experimentation

To investigate the existence of the five types of paths in the real programs, we implemented our demand-driven algorithm using Microsoft Phoenix APIs [17]. We measured the cost of the analysis on a set of benchmark programs selected from the BugBench [15] and the Buffer Overflow Benchmark [25]. The set consists of 9 programs, each of which contains known buffer overflow. Our experiments compute buffer overflow paths for these known vulnerabilities, identify the type of path, and determine the performance and space usage of analysis.

In our experiments, we compute buffer overflow paths for one PVS in each program. We would check every PVS of the program to make sure the software is secure. Our experiments consist of two steps. In the first step, we compute paths for a PVS in a benchmark program without considering infeasibility of paths. We then integrate our infeasibility detection module to check the impact of the infeasible paths on the query. In Tables 2, 3 and 4, we use a prime ($'$) symbol for results after integrating infeasible paths.

We summarize the generated paths for benchmarks including path types and path segment lengths in Table 2. The path segment consists of all nodes between when the query was raised and when it was resolved. Under the *Path Types* column, there are vulnerable (*Vul*), overflow-user-independent (*CNST*), don't-know (*UnK*) and safe (*Safe*) subcolumns. Each subcolumn lists the number of the path generated for the specified type. The results show that all five types of paths do exist in real programs. Six of nine programs are detected to have vulnerable paths, and two programs have don't-know paths due to an external library. One program contains overflow-user-independent paths. Seven out of nine programs have safe paths. Without our path detection, the code debuggers might explore safe paths which will not be successful in finding the vulnerability. For the program bc-1.06, the total number of overflow-user-independent paths is very large and we ran out of memory when we traversed the marked ICFG to print paths. Actually, the number of paths is not important because it is not necessary for a code reviewer to inspect every path for diagnosis. With our framework, users can specify the number of paths to be output. After fixing them, the framework would be used again to determine if this fix corrects all vulnerable paths of the PVS.

The column, *Inf*, under *Path Types* shows whether infeasible paths are detected in the programs [3]. We identify that six out of nine programs have infeasible paths. Using the infeasible information, the number of safe paths in three programs and the number of unknown paths in one program are able to be reduced. The length of the path segments is given by the number of different procedures (not including library calls) and number of basic blocks that are traversed by the path. These numbers are shown in the *Average Path Size* column.

In Table 3, we present data to evaluate the demand-driven approach. Under the *Basic Blocks* column, subcolumn *All Blocks* reports the total number of basic blocks in the program. The column labeled *V Blocks* reports the number of visited blocks during the analysis. Similarly, the *All Procs* column lists the total number of non-library procedures. *V Procs* lists the number of procedure visited. There are two worklists which are representative to report the memory usage of the analysis. *Max S* shows the maximum number of elements in the worklist during the solve-query step while *Max P* gives the maximum number of elements in the worklist during the propagate-results step. With the infeasible paths integration, the number of visited blocks and procedures is usually reduced because blocks that are on the infeasible paths are no longer visited. The total number of elements in the worklist of the solve-query step often increases because, in the presence of infeasible information, queries are less likely merged.

Table 4 shows the time of our analysis. Performance is reported by Phoenix's time report functionality [17]. For the nine programs, the performance varies from .24 to 102.6 seconds for detecting infeasible paths and resolving a buffer overflow query along all paths. The memory usage ranges from 9 to 65MB and the average is 18MB. We also report the memory usage by the size of the worklist queue (see Table 3 column *Work List*).

The above results demonstrate that the path types we defined all exist in the real code. For the vulnerable paths we generated, many cross procedural boundaries, involve global buffers, or are located in loops. Without the identification of the actual paths, these features will make manual inspection very difficult and time-consuming.

Benchmark	Lines of Code	Path Types					Average Path Size	
		Inf	Vul/Vul′	CNST/CNST′	UnK/UnK′	Safe/Safe′	# P/#P′	# B/#B′
polymorph-0.4.0	0.7K	yes	966/966	0/0	0/0	434/0	2.6/2.5	26.1/25.9
ncompress-4.2.4	1.9K	yes	288/288	0/0	0/0	2016/0	2.0/2.0	29.3/27.8
man-1.5h1	4.7K	yes	16/16	0/0	0/0	24/24	1.8/1.8	14.3/14.3
gzip-1.2.4	8.2K	no	1/1	0/0	0/0	0/0	3/3	5/5
bc-1.06	17.0K	yes	0/0	>50,000/>50,000	0/0	>30,000/>30,000	-	-
squid-2.3	93.5K	yes	0/0	0/0	8/4	4/2	1/1	6.7/6.8
wu-ftp: mapping-chdir	0.4K	yes	4320/4320	0/0	0/0	18624/18624	3.8/3.8	33.6/33.6
sendmail: ge-bad	0.7K	no	48/48	0/0	0/0	648/648	2.0/2.0	35.5/35.5
BIND: nxt-bad	1.3K	no	0/0	0/0	2/2	0/0	2.0/2.0	23.5/23.5

Table 2. Experiment Results: Computed Paths for Benchmarks

Benchmark	Basic Blocks		Procedures		Work List	
	All Blocks	V Blocks/V Blocks′	All Procs	V Procs/V Procs′	Max S/Max S′	Max P/Max P′
polymorph-0.4.0	740	34/34	22	3/3	10/9	13/13
ncompress-4.2.4	654	48/47	14	2/2	12/12	16/15
man-1.5h1	2593	100/100	78	8/8	23/23	25/25
gzip-1.2.4	3436	5/5	102	3/3	2/2	2/2
bc-1.06	3090	228/226	102	12/11	102/115	54/50
squid-2.3	35189	10/10	1423	1/1	5/4	3/3
wu-ftp:mapping chdir	129	40/39	6	4/3	46/46	10/10
sendmail: ge-bad	187	34/34	8	3/3	6/7	5/5
BIND: nxt-bad	423	31/31	14	2/2	8/8	6/6

Table 3. Evaluating Demand-Driven Analysis

Benchmark	Time(s)	Time′(s)
polymorph-0.4.0	12.19	12.40
ncompress-4.2.4	0.69	0.24
man-1.5h1	2.05	2.16
gzip-1.2.4	0.24	0.24
bc-1.06	98.3	102.6
squid-2.3	1.14	1.32
wu-ftp:mapping chdir	13.51	13.36
sendmail: ge-bad	1.64	1.70
BIND: nxt-bad	2.40	2.65

Table 4. Performance of Analysis

6. Related Work

Many approaches for detecting buffer overflow have been proposed, including compilers, languages, dynamic detectors and static analysis. Static analysis has the advantage that the overflow can be detected and fixed before software release. The drawbacks include high false positive rates and required human efforts for confirmation, prioritization and diagnoses of the bug. General static approaches include mapping of buffer bounds checking to integer range analysis, abstract interpretation, symbolic execution or type inference [11, 13, 22, 24]. Most of the existing static tools report high false positives, require annotations, or do not report or characterize paths.

Path-sensitive analysis aims to check if the property holds for every path. It reduces false positives by excluding infeasible paths. ARCHER [24] is path-sensitive, but it does not compute faulty paths, and only reports statements where the access of the buffer is violated. ESP [8] generates a set of paths where typestate violation can occur. MOPS [6] adapts model checking technology for computing a set of traces that violate security properties.

Demand-driven analysis aims to reduce time and space overhead by only collecting information that is needed, and thus improving scalability [10]. Duesterwald et al. designed a general framework for interprocedural dataflow analysis [10], which has been used to infeasible path detection and dataflow testing [3, 9]. Demand-driven analysis has also been applied to reproduce traces to explain program failure caused by typestate errors and to detect memory leaks [16, 20].

7. Conclusions and Future Work

This paper presents a demand-driven path-sensitive analysis framework for detecting and categorizing paths along which a buffer overflow may occur. The analysis is flexible, scalable and fully automatic. Its major contributions are:

- Reducing false positives by eliminating infeasible and safe paths that go through a vulnerable statement.
- Providing information on paths with overflow for directing manual diagnosis.
- Categorizing paths based on their chance of being exploited.

In the future, we plan to more fully explore the usage of the information provided by our technique in finding and correcting bugs. We also plan to use more sophisticated constraint solvers and alias detectors to provide more precise categorization.

8. Acknowledgment

We thank the Microsoft External Research & Programs group for supporting this project, especially Yan Xu and John Lefor. We also thank Andy Ayers and Chris McKinsey from the Phoenix group for their help in using Phoenix.

References

[1] T. Ball and J. R. Larus. Programs follow paths. Microsoft Technical Report MSR-TR-99-01, 1999.

[2] R. Bodik, R. Gupta, and M. L. Soffa. Interprocedural conditional branch elimination. In *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, 1997.

[3] R. Bodik, R. Gupta, and M. L. Soffa. Refining data flow information using infeasible paths. In *Proceedings of the 5th ACM SIGSOFT international symposium on Foundations of software engineering*, 1997.

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

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

[6] 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.

[7] S. Chen, J. Xu, E. Sezer, P. Gauriar, and R. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th USENIX Security Symposium*, 2005.

[8] M. Das, S. Lerner, and M. Seigle. ESP: path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 conference on Programming language design and implementation*, 2002.

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

[10] 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.

[11] D. Evans. Static detection of dynamic memory errors. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, 1996.

[12] Fortify. `http://www.fortifysoftware.com`.

[13] B. Hackett, M. Das, D. Wang, and Z. Yang. Modular checking for buffer overflows in the large. In *Proceedings of the 28th international conference on Software engineering*, 2006.

[14] M. J. Harrold and M. L. Soffa. Efficient computation of interprocedural definition-use chains. *ACM Transactions on Programming Languages and Systems*, 1994.

[15] 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.

[16] R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang. PSE: explaining program failures via postmortem static analysis. In *Proceedings of the 12th ACM SIGSOFT international symposium on Foundations of software engineering*, 2004.

[17] Microsoft. Phoenix: A software optimization and analysis framework. `http://research.microsoft.com/phoenix/`.

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

[19] A. One. Smashing the stack for fun and profit. `http://www.phrack.org/archives/49/P49-14`.

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

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

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

[23] R. Wojtczuk. The advanced return-into-lib(c) exploits. `http://www.phrack.org`, 2001.

[24] Y. Xie, A. Chou, and D. Engler. ARCHER: Using symbolic, path-sensitive analysis to detect memory access errors. In *Proceedings of the 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.