Classification of Static Analysis-Based Buffer Overflow Detectors

Article · June 2010

DOI: 10.1109/SSIRI-C.2010.28

2 authors, including:

Hossain Shahriar
Kennesaw State University
178 PUBLICATIONS **1,188** CITATIONS

SEE PROFILE

Classification of Static Analysis-based Buffer Overflow Detectors

Hossain Shahriar and Mohammad Zulkernine
School of Computing
Queen's University, Kingston, Canada
{shahriar, mzulker}@cs.queensu.ca

Abstract*— Buffer overflow is one of the most dangerous exploitable vulnerabilities in released software or programs. Many approaches are applied to mitigate buffer overflow (BOF) vulnerabilities such as testing and monitoring. However, BOF vulnerabilities are discovered in programs frequently which might be exploited to crash programs and execute arbitrary injected code. Static analysis is a popular approach for detecting BOF vulnerabilities before releasing programs. Many static analysis-based approaches are currently used in practice. However, there is no detailed classification of these approaches to understand their common characteristics, objectives, and limitations. In this paper, we classify static analysis-based BOF vulnerability detection approaches based on six features: inference technique, analysis sensitivity, analysis granularity, soundness, completeness, and language. We then classify static inference techniques into four types: tainted data flow, constraint, annotation, and string pattern matching. Moreover, we compare the approaches in terms of effectiveness, scalability, and required manual effort. The classification will enable researchers to differentiate among existing analysis approaches. We develop some guidelines to help in choosing approaches and building tools suitable for practitioners need.

Keywords: Static analysis, buffer overflow, sensitivity, completeness, soundness.

I. INTRODUCTION

The number of reported vulnerabilities (or security bugs) in released software (or programs) has been increasing over the last few years [5]. Among many of the known vulnerabilities, the buffer overflow (BOF) is considered to be one of the most notorious exploitable vulnerabilities. A buffer overflow allows attackers to overflow data buffers that might lead to program crashes or execute arbitrary code [1, 6]. The vulnerability has drawn a lot of interest among researchers to develop tools and techniques to mitigate vulnerabilities before (*e.g.*, testing [2]) and after (*e.g.*, monitoring [4], patching [24], fixing [3]) the deployment of programs. Nevertheless, BOF vulnerabilities are still widely found in both legacy code and newly developed programs on a daily basis (*e.g.*, [5]).

A common proactive approach is to detect security vulnerabilities in program code by applying static analysis technique [6]. The approach examines input program code (*e.g.*, source, intermediate object code), applies specific rules or algorithms (also known as inference), and derives a list of vulnerable code present in a program. The greater advantage of performing static analysis is that it does not require executing program code. As a result, the analysis can ignore the issues related to program executions such as the reachability of vulnerable code and the generation of input test cases to traverse the vulnerable code. Moreover, a static analysis can prove that a program is free from certain types of BOF under specific assumptions. The pioneer static analysis techniques (*e.g.*, control flow, data flow, inter-procedural) have been developed mainly for compiler generated code optimizations. As security breaches have become widespread in programming communities, many of these techniques have been leveraged to discover vulnerabilities in program code.

Many static analysis approaches have been introduced in the literature to detect BOF vulnerabilities (*e.g.*, [7, 8, 9]). Several works have evaluated the effectiveness of static analysis-based tools that detect BOF vulnerabilities on benchmark programs (*e.g.*, [22, 23]). However, there is no detailed classification to understand the common characteristics and limitations of these approaches. Moreover, the lack of a comprehensive comparative study provides little or no direction on choosing the appropriate BOF detection techniques.

In this paper, we survey the state of the art static analysis approaches that detect BOF vulnerabilities. We classify the approaches based on six common characteristics: *inference technique, analysis sensitivity, analysis granularity, soundness, completeness, and language*. Moreover, we further classify these characteristics to identify fine grained features. The effectiveness of any static analysis depends on how accurate an inference technique is in discovering potential vulnerable code. Thus, we are motivated to classify current static analysis related works based on the underlying inference techniques into four types: *tainted data flow*, *constraint*, *annotation*, and *string pattern matching*. Moreover, we comparatively analyze existing approaches for their effectiveness, scalability, and required manual effort to apply in practice. The survey will help researchers to differentiate existing static analysis approaches. Moreover, it will provide guidelines to choose a technique for their needs as well as to deal with the challenges in developing static analysis techniques related to BOF vulnerability detection.

This paper is organized as follows: Section II provides an overview of the BOF vulnerability. Section III provides the

classification of the static analysis work and some guidelines based on the analysis. Section IV provides the details of inference mechanisms, as well as some future guidelines. Section V discusses the comparison of approaches for practical use. Finally, Section VI draws conclusions and discusses current issues.

II. BUFFER OVERFLOW

A buffer overflow (BOF) vulnerability occurs while writing data to a program buffer exceeding the allocated size, and overwriting the contents of the neighboring memory locations. The overwriting corrupts sensitive neighboring variables of the buffer such as the return address of a function or the stack frame pointer. BOF can occur due to vulnerable ANSI C library function calls, lack of null characters at the end of buffers, accessing buffer through pointers and aliases, logic errors (off by one), and insufficient checks before accessing buffers in program code.

```
1. void foo (int a) {
2.       int var1;
3.       char buf [16];
         …
    }
```

Figure 1. C code snippet of *foo* function

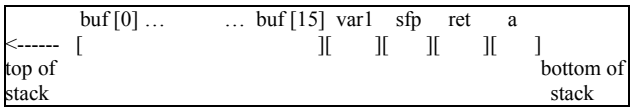

Figure 2. Stack layout of *foo* function

We provide an example of BOF in a C code snippet as shown in Figure 1. The function *foo* has a buffer named *buf* that is located inside the stack region. The valid location of this buffer is between *buf[0]* and *buf[15]*. The variable *var1* is located immediately after the ending location of the buffer followed by the stack frame pointer (*sfp*) and the return address (*ret*) of the function *foo* as shown in Figure 2. The return address indicates the memory location where the next instruction is stored and is read immediately after the function is executed.

A BOF might happen during reading or writing operations. Writing past the *buf* by at least one byte corrupts the value of *var1* (assuming no padding performed by a compiler). If overwriting spans more than one byte in stack, it might modify the return address (*ret*) of the function *foo*. As a result, when the function tries to retrieve the next instruction after its execution, the modified location might not fall within the valid address space of the program. This might result in a segmentation fault and the program crashes.

III. CLASSIFICATION OF BOF DETECTION APPROACHES

In this section, we compare static analysis approaches that detect BOF vulnerabilities based on six characteristics: *inference type, analysis sensitivity, analysis granularity, soundness, completeness, and language*. We describe these characteristics in Sections A to F, followed by some guidelines in Section G. We provide a brief summary of these works in Table I. A summary of the comparison is provided in Table II.

A. Inference technique

The core part of any static analysis is to infer potential vulnerabilities systematically by scanning program code. We divide inference types into four categories as shown in Table II. These are *tainted data flow-based* [19], *constraint-based* [10, 11, 12, 14], *annotation-based* [7, 8], and *string pattern matching-based* [9, 13, 15] approaches. We discuss these inference types in details in Section IV.

B. Analysis sensitivity

A common problem for a static analysis approach is that it might generate false positive warnings to be examined manually. Moreover, an analysis might suffer from false negatives (*i.e.*, actual BOF vulnerabilities might be unreported). To reduce the number of false positive or false negative, approaches might take advantage of pre-computed information based on program code before applying appropriate inference algorithms. We denote the dependency of such pre-computed information as analysis sensitivity. We identify five types of sensitivity: flow, path, context, points-to, and value range analysis, which we discuss in the followings.

TABLE I. A BRIEF SUMMARY OF STATIC ANALYSIS-BASED BOF VULNERABILITY DETECTION WORKS

Work	**Brief description**
Flawfinder [13]	Match known vulnerable function calls in source code.
Wagner *et al.* [10]	Detect BOF vulnerabilities by expressing buffer accesses as an integer range constraint problem.
Weber *et al.* [11]	Improve Wagner *et al.* [10] approach by considering BOF due to global variable, function, and recursions.
Viega *et al.* [9]	Tokenize source code and identify known vulnerable library function calls.
Evans *et al.* [17]	Annotate a function body with pre- and post-conditions and warn users if conditions are violated.
Dor *et al.* [7]	Detect all string manipulation errors through pointer analysis and contract checking on a subset of C language.
Ganapathy *et al.* [18]	Detect BOF by forming integer range constraints and apply linear programming to solve these constraints.
Xie *et al.* [12]	Detect BOF at each array and pointers dereference by applying path sensitive and inter-procedural symbolic execution.
Vulncheck [19]	Apply tainted data flow analysis to detect BOF vulnerabilities.
Tevis *et al.* [15]	Detect vulnerable library function calls in the section table of portable executable files.
Hackett *et al.* [8]	Apply annotation language to specify buffer interfaces for safe usage of buffers in functions.
Le *et al.* [14]	Apply query-based demand driven analysis to warn BOF and prioritize warnings based on path sensitivity.

TABLE II. CLASSIFICATION OF STATIC ANALYSIS-BASED BOF VULNERABILITY DETECTION WORKS

Work	Inference technique	Analysis sensitivity	Analysis granularity	Soundness	Completeness	Language
Flawfinder [13]	String pattern matching	Context	Token	Yes (limited scope of the problem)	No (absence of semantic analysis)	C, C++
Wagner *et al.* [10]	Constraint	None	Statement	Yes (limited language feature)	No (absence of flow sensitive analysis)	C
Weber *et al.* [11]	Constraint	Control flow, context	Control flow, system dependence graph	Yes (absence of analysis sensitivity)	No (absence of points to sensitive analysis)	C
Viega *et al.* [9]	String pattern matching	None	Token	Yes (limited scope of the problem)	No (absence of semantic analysis)	C, C++
Evans *et al.* [17]	Annotation	Control flow	Statement	Yes (limited scope of the problem)	No (absence of analysis sensitivity, assumption on code)	C
Dor *et al.* [7]	Annotation	Points-to analysis	Intra-procedural	Yes (limited language feature)	No (absence of analysis sensitivity)	C
Ganapathy *et al.* [18]	Constraint	Context, points-to analysis	System dependence graph	Yes (limitation of analysis sensitivity)	No (absence of flow sensitive analysis)	C
Xie *et al.* [12]	Constraint	Path, context	Intra and inter-procedural	Yes (limited scope of the problem)	Yes	C
Vulncheck [19]	Tainted data flow	Value range	Statement, data flow, inter-procedural	Yes (limited scope of the problem)	Yes	C
Tevis *et al.* [15]	String pattern matching	N/A	Statement	Yes (limited scope of the problem)	No (absence of semantic analysis)	x86
Hackett *et al.* [8]	Annotation	N/A	Inter-procedural	Yes (limited scope of the problem)	No (assumption on code)	C
Le *et al.* [14]	Constraint	Path and context	Inter-procedural	Yes (absence of analysis sensitivity)	No (result interpretation)	C

1) Flow

An analysis is flow (*i.e.*, control flow) sensitive, if it performs inference based on statement execution sequence with respect to a control flow graph. Otherwise, an analysis is control flow insensitive and statements can be considered in any order. Applying flow sensitivity might increase the precision (or less false positive warnings) in BOF detection. For example, Weber *et al.* [11] improve the approach of Wagner *et al.* [10] by employing a flow sensitive analysis.

2) Path

Program execution paths can be derived from control flow graphs of a program. Some of the paths might not be feasible which can be determined statically. If an analysis explicitly excludes infeasible paths, we denote it as path sensitive. For example, Le *et al.* [14] detect BOF vulnerabilities that are reachable within feasible program paths. Xie *et al.* [12] detect BOF vulnerabilities due to pointer dereferences and buffer indexes in feasible program paths based on control flow graphs.

3) Context

A context sensitive approach differentiates multiple calls sites of a function with respect to supplied arguments [18]. In contrast, a context insensitive analysis ignores multiple calls of the same function with different arguments. Context insensitivity might result in false positive warnings. For example, a program might have two vulnerable library function calls of *strcpy (dest, src)*. Here, the function copies the buffer pointed by *src* to the buffer pointed by *dest*. The first function call might have *src* string whose value is provided by a user, and the second call might contain a constant string whose length can be determined statically to be less than that of *dest* buffer. A context insensitive analysis might report both of them as vulnerable. However, a context sensitive analysis (*e.g.*, Flawfinder [13]) reports the former call as vulnerable and the latter as non-vulnerable.

Several works apply context sensitivity in their analyses. Le *et al.* [14] apply context sensitive BOF detection by formulating BOF detections as query solving problems. While solving a query (*i.e.*, knowing unknown variable values), function calls present along paths are explored. In particular, it is checked whether any existing unknown variable's value (present in query) can be resolved within the supplied arguments of functions or not. Xie *et al.* [12] also consider arguments of the same function calls separately to perform accurate BOF detection.

4) Points-to analysis

A points-to analysis identifies a set of memory objects that might be pointed by a given pointer variable present in program code. Points-to analysis itself fits into another direction of research and interested readers might consult the literature related to points-to analysis (*e.g.*, [20]). We restrict our discussion on four concepts that might be relevant and used in detecting BOF vulnerabilities: flow sensitive, flow insensitive, context sensitive, and context insensitive points-to analysis.

In a flow sensitive points-to analysis, a program's control flow is taken into account. In contrast, in a flow insensitive points-to analysis, statements can be analyzed in any order. Obviously, a flow sensitive points-to analysis is more accurate than that of a flow insensitive analysis. In a context sensitive points-to analysis, function calls accepting pointer type arguments or returning pointers are analyzed separately. In contrast, in a context insensitive points-to analysis, these function calls are considered identical. It is more appropriated to apply context sensitive points-to analysis

than that of context insensitive analysis to obtain more precise information.

From the Table II, we observe that very few techniques apply *points-to* analysis [7, 18]. Dor *et al.* [7] apply flow insensitive points-to analysis. Most of the BOF detection approaches (as shown in Table II) do not incorporate points-to analysis explicitly. However, some approaches employ general assumptions on pointer data types. For example, Wagner *et al.* [10] assume that a pointer to a structure variable might point to all other similar structure variables present in a program. The assumption results in a huge number of false positive warnings. Xie *et al.* [12] consider a limited number of pointer information such as a pointer pointing to a buffer and the relative distance from the base size of a buffer. However, if a pointer points to an unknown type of memory object, their analysis does not include such information. As a result, some real BOF might be undetected (*i.e.*, increases false negative in the detection).

Points-to analysis might be used to reduce false negative (*i.e.*, detecting more vulnerabilities that would be otherwise not detected). For example, Ganapathy *et al.* [18] apply points-to analysis information while generating constraints on buffers that are being dereferenced by pointer variables. As a result, function calls having pointer to buffer arguments can be analyzed for detecting BOF vulnerabilities. Without including points-to analysis, BOF vulnerabilities caused by these function calls might be undetected.

5) Value range

An analysis might consider the value range of sensitive variables before deciding on BOF vulnerabilities. A value range analysis provides a lower and upper bound of a variable. The information might be useful when function calls are supplied with unsanitized arguments that represent buffer sizes. For example, Sotirov *et al.* [19] apply value range analysis on the *size* argument of *memcpy (dest, src, size)* function calls. The value range analysis might discover that the upper bound of the size argument of *memcpy* function call is MAXINT (maximum value of an integer). Their approach generates a BOF warning in this case.

We find that several works ignore sensitivity in analyses for the sake of achieving high level scalability (*e.g.*, [9, 10]). Note that several works do not explicitly discuss about sensitivity or insensitivity that are indicated as N/A in Table II (*e.g.*, [8, 15]).

C. Analysis granularity

This feature indicates the granularity level of program code at which an inference is performed. We identify five types of granularity levels: token, statement, intra-procedural, inter-procedural, and system dependence graph.

1) Token

A compiler performs lexical analysis which tokenizes program source code to identify keywords, variables, functions, etc. An approach might use these tokens to detect BOF vulnerabilities (*e.g.*, [9, 13]).

2) Statement

An analysis might infer the presence of vulnerabilities by analyzing each individual program statement sequentially. Most of the approaches analyze program code at statement level (*e.g.*, [10, 15]). Unlike traditional ways of analyzing on program code, an approach might analyze the executable program code such as *x86* [15]. However, the executable code is usually de-assembled first to make it partially readable.

3) Intra-procedural

A program can be analyzed based on either a control flow graph or a data flow graph (a graph which represents data dependencies between a number of operations). In this analysis, it is assumed that a program consists of only one function. Several works apply intra-procedural analysis to form a summary of either sensitive variables [7] or specific conditions that can (or cannot) be satisfied [12] related to BOF vulnerabilities.

4) Inter-procedural

In this case, an analysis examines a function body as well as accesses other function call sites present in programs. It is common to avoid analyzing the same function multiple times by following a bottom up analysis based on a call graph [12]. In a call graph, each node represents a unique function and a directed edge from node *a* to *b* indicates that the function represented as node *a* contains an invocation of the function represented by the node *b*.

5) System dependence graph (SDG)

In a SDG, a node represents a program point (*e.g.*, statement), and an edge represents dependency between two program points which can be two types: control flow and data flow. Control flow and data flow dependencies are identified by control and data flow analyses, respectively. Weber *et al.* [11] generate constraints for each node of the SDG and traverse these nodes from bottom to top (*i.e.*, the main function) to detect BOF vulnerabilities. Ganapathy *et al.* [18] also detect BOF based on SDG graph nodes generated by the Codesurfer tool [21].

D. Soundness

An approach is sound, if it has no false negative [16]. In other words, a sound approach does not leave any real program vulnerability unreported. From Table II, we notice that all the works can be considered sound under certain assumptions which are mentioned in the fifth column. We classify the assumptions into three types: limited language features, analysis sensitivity, and scope of the problem.

1) Limited language features

An approach might be sound with respect to certain data types and program structures. For example, the approach of Wagner *et al.* [10] is sound as long as there is no occurrence of BOF through pointer arithmetic or union data structures having buffers as member variables. The approach of Hackett *et al.* [8] is sound provided that a BOF is not caused by a global pointer or a structure field having pointer data type. Moreover, their approach does not support specification of safe buffer usage for conditional statements. Thus, the approach is sound, if no unsafe library function call (*e.g.*, *strcpy*) is present in a branch.

2) Analysis sensitivity

Some approaches do not employ points-to analysis [11, 14]. These approaches are sound provided that no BOF is caused

through arbitrary pointer dereferences. Moreover, the underlying analysis sensitivity has its own limitation, and soundness of an approach depends on the soundness of the points-to analysis algorithm (*e.g.*, [18]).

3) Scope of the problem

BOF might be caused through a variety of reasons such as unsafe ANSI C library function calls, pointer arithmetic, accessing buffers through arbitrary buffer indexes and pointers, lack of null characters at the end of buffers, and user defined functions containing flaws. To make an analysis approach manageable, some works explicitly limit the scope of detection. Thus, an approach can be considered sound with respect to the limited scope (or possible program code pattern causing BOF vulnerabilities). For example, Vulncheck [19] assumes that most BOF vulnerabilities occur through vulnerable library function calls with unsanitized arguments. Dor *et al.* [7] detect all BOF vulnerabilities present in a program caused by only string variables (*i.e.*, static or dynamic buffers) and those accessed after null byte characters. String pattern matching-based approaches [9, 13, 15] are sound, if BOF vulnerabilities occur only through known pattern of function calls and arguments. The approach of Evans *et al.* [17] can discover all BOF vulnerabilities in user defined functions, if annotations are provided appropriately. Xie *et al.* [12] assume that most pointer arithmetic and conditional expressions present in branches and loops are linear.

E. Completeness

An approach is said to be complete, if it generates no false positive warnings [16]. From the sixth column of Table II, we observe that very few approaches claim to be complete. However, it is entirely based on the underlying assumption. Vulncheck [19] is complete under the assumption that most BOF vulnerabilities occur through a set of known library function calls that accept tainted arguments.

In practice, it is hard to develop an analysis technique that results in no false positive warnings. We identify four common reasons that might contribute for an incomplete analysis: analysis sensitivity, result interpretation, absence of semantic analysis, and assumption on program code.

1) Analysis sensitivity

Inference approaches based on insensitive analysis of flow (*e.g.*, [10, 17, 18]), path (*e.g.*, [7]), context (*e.g.*, [10]), or points-to (*e.g.*, [8, 11, 17]) result in conservative analyses and generate a high number of false positive warning.

2) Result interpretation

Often false positives are due to the way results are to be interpreted. For example, the approach of Le *et al.* [14] provides a set of programs paths which do not include safe and infeasible paths. Thus, it is likely that some of the remaining suspected paths of the set might not be vulnerable.

3) Absence of semantic analysis

If an approach relies on lexical analysis and avoids semantic analysis, many warnings can be false positive [9, 13, 15]. For example, all the unsafe library function calls can be thought of as BOF vulnerabilities. However, an unsafe function call present in an infeasible path [13] cannot be exploited.

4) Assumption on code

An approach might assume that programmers write specific patterns of code in implementations. Breaking of these assumptions results in false positive warnings. For example, Evan *et al.* [17] infer how many times a loop runs based on a known set of loop structures that are usually written by programmers and use the result to detect BOF vulnerabilities. Moreover, an approach might rely on correct implementation of functionalities (*e.g.*, validation of inputs stored in buffers [7]) before inferring BOF vulnerabilities.

F. Language

This feature highlights the programming languages that are supported by an analysis approach. We notice that most techniques analyze programs implemented in C languages (*e.g.*, [7, 8, 9, 12]). Several works detect BOF vulnerabilities in C++ code (*e.g.*, [9, 13]). However, very few approaches analyze low level code such as *x86* (*e.g.*, [15]).

G. Guidelines based on classification features

Based on the classification, we notice that introducing analysis sensitivity and performing analysis on multiple granularity levels of program code result in better detection of vulnerabilities. Most of the current approaches lack completeness or soundness. There exists a tradeoff factor between analysis precision and scalability in the current analysis techniques. Detecting all possible BOFs through static analysis is still far from reality. Most of the works detect a limited type of BOF vulnerabilities under various assumptions. We also observe that most of the analysis techniques are geared towards high level languages such as C. Few works perform static analysis at intermediate object code and executable code level.

IV. INFERENCE TECHNIQUE

A. Tainted data flow-based technique

In this technique, program locations where untrusted inputs are obtained are marked as tainted. Tainted information propagation is computed through a fix point algorithm (*e.g.*, constant propagation). If tainted information is used in sensitive operations (*e.g.*, arguments of vulnerable library calls), related statements are flagged as vulnerable. Sotirov *et al.* [19] apply tainted data flow-based technique to detect suspected library function calls with arguments that might result in BOF vulnerabilities.

B. Constraint-based technique

Constraint-based techniques generate safety constraints from program code whose violations imply vulnerabilities. The constraints can be generated based on program statements [10, 11] and function calls [12]. Sometimes, constraints are propagated and updated while traversing a program. A program might be traversed based on a system dependence graph [11] and a control flow graph [12]. At the end, the analysis identifies whether any solution of the constraints exists or not. A solution indicates that vulnerabilities might be present. We divide constraint-based technique into three

categories: integer range analysis, symbolic execution, and demand-driven.

1) Integer range analysis

The idea of this technique is to formulate constraints by scanning each program statement containing buffer declarations and operations involving buffers. Each constraint is expressed in terms of a pair of integer ranges (buffer allocation size and the current size of a buffer) for each buffer defined or accessed. For each buffer, a set of constraints are solved to find a range of allocation and current size, which can be denoted as [a, b] and [c, d], respectively. Here, [a, b] implies that allocation size of a buffer can vary from a bytes to b bytes. Similarly, [c, d] implies that current size of a buffer can vary from c bytes to d bytes. These ranges are analyzed to identify non vulnerable (*e.g.*, $b > c$) and vulnerable (*e.g.*, $b \leq c$) statements. Wagner *et al.* [10] first apply integer range analysis to detect BOF vulnerabilities. Later, Weber *et al.* [11] improve Wagner's method by applying a flow sensitive analysis to reduce the false positive warnings. They also detect BOF due to global variable usage, function calls, and recursions. Ganapathy *et al.* [18] generate constraints on buffer sizes and allocations using an SDG and solve them using linear programming.

2) Symbolic value analysis

In this approach, constraint might contain program variables and whenever possible their values are assigned. Otherwise, they are treated as having symbolic values. Xie *et al.* [12] apply symbolic value analysis to detect BOF vulnerabilities caused by invalid buffer indexes, pointer dereferences, and invalid function arguments (buffer addresses and sizes). They traverse a call graph of a program using a bottom-up approach, where a function is analyzed through a control flow graph (CFG). During the CFG analysis, safety constraints (*i.e.*, the negation of valid conditions) are generated at every access of arrays, pointer dereferences, and function calls. Moreover, constant relations (*e.g.*, $x = 4$) and symbolic constraints between variables (*e.g.*, $x < y$) are captured and propagated. At every potentially dangerous access of arrays, pointer dereferences, or call to routines, a custom constraint solver is used to evaluate the values against known constraints. A warning flag is raised, if a constraint is violated. If the solver cannot solve the query, then no warning is generated.

3) Demand-driven

Most constraint-based techniques limit their scopes by providing a list of warnings based on built in constraint generation, propagation, and solution mechanisms. However, a recent direction is to provide a programmer the option to formulate queries on vulnerable program locations. This is denoted as demand-driven static analysis. Such approach relies on constraint generation, which starts from a location specified by developers. The end output is a set of prioritized paths that might trigger BOF.

Le *et al.* [14] apply this approach to detect different types paths (*e.g.*, infeasible, safe, user input dependant, vulnerable) vulnerable to BOF. A user specifies queries to know whether (i) buffer accesses at particular program points are safe, and (ii) user inputs can write to buffers. The queries are expressed with constraints in terms of buffer sizes, supplied string lengths, and flag values (to represent constant string values). The queries are propagated along program paths in backward directions (*i.e.*, starting from the point of query to the beginning of a program's main function call) through inter-procedural and context sensitive analyses. Finally, if queries can be resolved by checking whether a declared buffer size is less than the supplied input values, a BOF warning is issued.

C. Annotation-based technique

In an annotation-based technique, program code is annotated with desired properties in terms of pre- and post-conditions. Annotations can be specified at both function prototype declaration [7, 8, 17] and statement level [17]. After code is annotated, an algorithm checks if buffers can be used safely in functions based on the annotated conditions. A function call site is first evaluated to check whether it conforms to specified pre-conditions or not (*i.e.*, generate error messages). If pre-conditions are satisfied at the call site, the function body is further examined to ensure that the implementation meets specified post-conditions. If a pre-condition cannot be resolved from a previous statement's post-condition, then a warning message is generated. The warnings are issued by the checker are often prioritized based on how well the constraints for accessing buffers are understood. For example, no condition to access a buffer is listed at the top of a warning list, whereas buffer access based on a condition on buffer length is placed at the bottom of a warning list.

Annotation-based approaches generate constraints from the specified pre- and post-conditions. However, these constraints are evaluated to be true or false to identify whether a program location (*e.g.*, a function call invocation) is free from vulnerability or not. In contrast, a constraint-based approach solves a set of constraints for further analysis. We also observe that tainted data flow analysis might rely on annotating program code such function prototypes (*e.g.*, [19]). However, these annotations are not involved in generating constraints like the annotation-based approaches. Rather, the annotations facilitate comparing an expected data type (annotated) with an identified data type (through static analysis) to detect vulnerabilities.

Annotations can be specified by expressions supported by an implementation language. For example, Dor *et al.* [7] detect all string manipulation errors that might lead to BOF vulnerabilities by specifying contracts (or annotations) through C expressions. Contracts include pre-conditions, post-conditions, and specific side effects. After adding annotations, a source to source semantic preserving transformation of a given procedure is performed. The converted program code generates errors, if contracts are violated. However, the pre- and post-conditions expressed through implementation language expressions can detect limited type of BOF vulnerabilities such as accessing to buffers after the null-termination bytes.

Several works detect a wide range of vulnerabilities including BOF either by introducing an annotation language or by extending an implementation language. For example, Hackett *et al.* [8] develop an annotation language named *SAL*

that allows expressing buffer annotations to describe buffer interfaces through pointers. The annotation of buffer might include usage (*e.g.*, a buffer passed to a function and read from), option (*e.g.*, pointer can be NULL), extent of a buffer initialization (*i.e.*, a lower initialization bound), and capacity (*i.e.*, a lower capacity bound). In contrast, Evans *et al.* [17] annotate function parameters, return values, global variables, and structure fields inside tagged comments in C programs. BOF vulnerabilities are detected by adding pre- and post-conditions to user defined and ANSI C library functions. For example, the library function *strcpy(s_1, s_2)* copies the source buffer s_2 to the destination buffer s_1. The pre-condition */*@requires maxSet(s_1) ≥ maxRead(s_2) @*/* generates an error message, if it cannot be satisfied at the call site by a checker. Here, the pre-condition indicates that the maximum index value that can safely access s_1 (*i.e.*, *maxSet(s_1)*) during a write operation must be greater than or equal to the maximum index value that can safely access s_2 (*i.e.*, *maxRead(s_2)*) during a read operation.

D. String pattern matching-based technique

The pioneer static analysis approaches are based on string pattern matching (*e.g.*, [9, 13]). These approaches rely on a known set of library function calls that might cause BOF. A set of rules are developed which represent signature of vulnerable code patterns. This technique tokenizes program code and scans them to identify vulnerable pattern of strings that represent vulnerable function calls and arguments.

A recent variation of string pattern matching-based technique is to scan executable program code to detect vulnerable function calls. For example, Tevis *et al.* [15] analyze portable executable (PE) files that run on Windows NT/XP to detect BOF vulnerabilities. They detect BOF vulnerabilities by examining symbol tables, if there are occurrences of vulnerable library function names (*e.g.*, ANSI C libraries) as well as tracing zero filled regions of 50 bytes or more. These regions can be used to load malicious code during BOF attacks.

E. Guidelines based on inference techniques

We observe that each inference mechanism is valuable from certain perspectives. For example, annotation-based mechanism is useful to verify that certain vulnerabilities are not present, whereas query-based inference can help in prioritizing vulnerabilities that must be addressed immediately. Our study indicates that constraint, annotation, and string pattern matching-based inferences have been widely applied to detect a limited type of BOF vulnerabilities. Thus, future works might combine multiple techniques or apply other inference mechanisms (*e.g.*, tainted data flow) to detect vulnerabilities. Another research direction could be the investigation of appropriate granularity and sensitivity for different inference techniques to improve completeness and soundness as much as possible.

V. PERFORMANCE COMPARISON OF BOF VULNERABILITY DETECTION APPROACHES

In this section, we perform a comparison of different static analysis-based BOF detection works with respect to the reported false positive rate (FP), LOC analyzed, and manual effort required. We report these features based on the evaluations performed by the original authors of each work. Table III shows a summary of comparison of approaches which are categorized based on inference techniques. If an approach analyzes several programs, we consider the lowest rate of FP. However, we consider the maximum for analyzed LOC. The manual effort column is reported in terms of "Yes" and "No". Most of the works do not report false negative rate, which prohibit us to include this important feature in our comparison.

TABLE III. PERFORMANCE COMPARSION OF STATIC ANALYSIS-BASED BOF DETECTORS

Inference	Work	FP	LOC	Manual effort
Tainted data flow	Vulncheck [19]	0%	16	Yes
Constraint	Wagner *et al.* [10]	90%	32K	No
	Weber *et al.* [11]	N/A	18K	No
	Ganapathy et al. [18]	N/A	68K	No
	Xie *et al.* [12]	35%	1.6M	No
	Le *et al.* [14]	N/A	93.5K	No
Annotation	Evans *et al.* [17]	75%	20K	Yes
	Dor *et al.* [7]	50%	228	Yes
	Hackett et al. [8]	10%	400K	Yes
String pattern matching	Viega *et al.* [9]	57%	68K	No
	Flawfinder [13]	50%	16K	No
	Tevis *et al.* [15]	N/A	9.3M	No

From the third column of Table III, it is obvious that FP rates vary widely which ranges from zero to 90% among different approaches. Among all the inference techniques, constraint-based approaches suffer from a large amount of false positive warnings followed by the annotation-based approaches. Tainted data flow and string pattern matching suffer from less number of FP warnings. Most of the constraint-based approaches do not report FP (denoted as N/A) due to overwhelming efforts required for examining the warnings. Most of the approaches selectively examine a subset of the generated warnings.

The amount of LOC that different approaches can analyze varies widely. String pattern matching-based approaches are highly scalable that can analyze up to 9.3 million LOC (x86 assembly code). Constraint-based approaches also have high scalability that can reach up to 1.6 million LOC. However, annotation-based approaches have relatively low scalability, except the approach of Hackettt *et al.* [8]. They incrementally annotate 400K LOC over a long period of time. Tainted data flow-based analysis has not been applied for large scale programs for detecting BOF.

We also notice that tainted data flow and annotation-based techniques require significant manual effort. These include marking tainted and untainted data sources and writing contracts for function prototypes. On the other hand, constraint-based and string pattern matching-based techniques require no manual effort. Given that, practitioners can choose constraint or string pattern matching-based tools to avoid manual effort and obtain high scalability in their analysis.

VI. CONCLUSIONS

Currently, many static analysis-based approaches are used to detect BOF vulnerabilities in source code. However, the lack of a classification provides us little or no information to understand the pros and cons of these approaches. This paper presents a detailed classification based on an extensive survey on the state of the art static analysis tools that detect BOF vulnerabilities. We classify the approaches based on six common characteristics namely inference technique, analysis sensitivity, analysis granularity, soundness, completeness, and language. We have discussed the approaches based four inference techniques: tainted data flow, constraint, annotation, and string pattern matching. We have also compared the current techniques in terms of effectiveness (false positive rate), scalability (lines of code), and required manual effort. The survey shows that the current static analysis-based approaches detect limited types of BOF vulnerabilities. Moreover, they are focused on increasing scalability and decreasing false positives. Future works can emphasize on reducing false negatives, introducing novel inference techniques, and combining multiple inference techniques.

ACKNOWLEDGMENT

This work is partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Ontario Graduate Scholarship (OGS). We also thank to the anonymous reviewers for their thoughtful comments to improve this paper

REFERENCES

[1] Aleph One, "Smashing the Stack for Fun and Profit", *Phrack Magazine,* Volume 7, Issue 49, November 1996, Accessed from insecure.org/stf/smashstack.html (March 2010).

[2] R. Xu, P. Godefroid, and R. Majumdar, "Testing for Buffer Overflows with Length Abstraction," *Proceedings of the International Symposium on Software Testing and Analysis*, Seattle, July 2008, pp. 27-38.

[3] C. Dahn and S. Mancoridis, "Using Program Transformation to Secure C Programs Against Buffer Overflows," *Proceedings of the 10th Working Conference on Reverse Engineering*, Canada, November 2003, pp. 323-332.

[4] R. Jones and P. Kelly, "Backwards-compatible Bounds Checking for Arrays and Pointers in C Programs," *Proceedings of Automated and Algorithmic Debugging*, Sweden, 1997, pp. 13-26.

[5] Common Vulnerabilities and Exposures (CVE), Accessed from http://cve.mitre.org (March 2010)

[6] B. Chess and G. McGraw, "Static Analysis for Security," *IEEE Security and Privacy*, Volume 2, Issue 6, December 2004, pp. 76-79.

[7] N. Dor, M. Rodeh, and M. Sagiv, "CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C," *Proceedings of the Conference on Programming Language Design and Implementation*, California, USA, June 2003, pp. 155-167.

[8] B. Hackett, M. Das, D. Wang, and Z. Yang, "Modular Checking for Buffer Overflows in the Large," *Proceedings of the 28th International Conference on Software Engineering*, Shanghai, China, May 2006, pp. 232-241.

[9] J. Viega, J. Bloch, T. Kohno, and G. McGraw, "Token-based scanning of source code for security problems," *ACM Transactions on Information and System Security (TISSEC)*, Volume 5, Issue 3, August 2002, pp. 238-261.

[10] D. Wagner, J. Foster, E. Brewer, and A. Aiken, "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities," *Proc. of Network and Distributed System Security Symposium*, San Diego, USA, February 2000, pp. 3-17.

[11] M. Weber, V. Shah, and C. Ren, "A Case Study in Detecting Software Security Vulnerabilities Using Constraint Optimization," *Proceedings of the Workshop on Source Code Analysis and Manipulation*, Italy, November 2001, pp. 3-13.

[12] Y. Xie, A. Chou, and D. Engler, "ARCHER: Using Symbolic, "Path-sensitive Analysis to Detect Memory Access Errors," *Proceedings of the 9th European Software Engineering Conference*, Helsinki, Finland, 2003, pp. 327-336.

[13] FlawFinder, http://www.dwheeler.com/flawfinder

[14] W. Le and M. Soffa, "Marple: a demand-driven path-sensitive buffer overflow detector," *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'08)*, Atlanta, Georgia, pp. 272-282.

[15] J. Tevis and J. Hamilton, "Static Analysis of Anomalies and Security Vulnerabilities in Executable Files," *Proceedings of the 44th Annual Southeast Regional Conference*, Melbourne, Florida, 2006, pp. 560-565.

[16] K. Chen and D. Wagner, "Large-Scale Analysis of Format String Vulnerabilities in Debian Linux," *Proceedings of the Workshop on Programming Languages and Analysis for Security (PLAS' 07)*, San Diego, USA, June 2007, pp. 75-84.

[17] D. Evans and D. Larochelle, "Improving Security Using Extensible Lightweight Static Analysis," *IEEE Software*, Volume 19, Issue 1, January 2002, pp. 42-51.

[18] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek, "Buffer Overrun Detection using Linear Programming and Static Analysis," *Proceedings of the 10th ACM conference on Computer and Communications Security*, Washington D.C., USA, October 2003, pp. 345-354.

[19] A. Sotirov, *Automatic Vulnerability Detection Using Static Analysis*, MSc Thesis, The University of Alabama, 2005, Accessed from gcc.vulncheck.org/sotirov05automatic.pdf

[20] M. Hind, "Pointer Analysis: Haven't We Solve This Problem Yet?," *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, June 2001, Snowbird, Utah, pp. 54-61.

[21] CodeSurfer, Accessed on March 2010 from www.grammatech.com/products/codesurfer

[22] M. Zitser, R. Lippmann, and T. Leek, "Testing Static Analysis Tools using Exploitable Buffer Overflows from Open Source Code," *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, California, USA, November 2004, pp. 97-106.

[23] D. Pozza, R. Sisto, L. Durante, and A. Valenzano, "Comparing Lexical Analysis Tools for Buffer Overflow Detection in Network Software," *Proceedings of the 1st International Conference on Communication System Software and Middleware*, New Delhi, India, January 2006, pp. 1-7.

[24] A. Smirnov and T. Chiueh, "Automatic Patch Generation for Buffer Overflow Attacks," *Proceedings of the 3rd International Symposium on Information Assurance and Security*, Manchester, UK, August 2007, pp. 165-170.