Loop-Extended Symbolic Execution on Binary Programs

Prateek Saxena   Pongsin Poosankam[†]   Stephen McCamant   Dawn Song
University of California, Berkeley and [†]Carnegie Mellon University
Berkeley, California, USA
{prateeks,ppoosank,smcc,dawnsong}@cs.berkeley.edu

ABSTRACT

Mixed concrete and symbolic execution is an important technique for finding and understanding software bugs, including security-relevant ones. However, existing symbolic execution techniques are limited to examining one execution path at a time, in which symbolic variables reflect only direct data dependencies. We introduce loop-extended symbolic execution, a generalization that broadens the coverage of symbolic results in programs with loops. It introduces symbolic variables for the number of times each loop executes, and links these with features of a known input grammar such as variable-length or repeating fields. This allows the symbolic constraints to cover a class of paths that includes different numbers of loop iterations, expressing loop-dependent program values in terms of properties of the input. By performing more reasoning symbolically, instead of by undirected exploration, applications of loop-extended symbolic execution can achieve better results and/or require fewer program executions. To demonstrate our technique, we apply it to the problem of discovering and diagnosing buffer-overflow vulnerabilities in software given only in binary form. Our tool finds vulnerabilities in both a standard benchmark suite and 3 real-world applications, after generating only a handful of candidate inputs, and also diagnoses general vulnerability conditions.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection; D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Security, Reliability, Verification

1. INTRODUCTION

Mixed concrete and symbolic execution generalizes a single concrete execution by representing inputs as variables and performing operations on values dependent on them symbolically (such as [13, 25]). This approach enables automated tools to reason about properties of all the program executions that follow the same control flow path, and has been successfully applied to a wide range of different applications in software engineering and security [7, 10, 14, 26, 41, 46]. However, this approach generalizes an execution only to a set of executions that follow exactly the same control-flow path. We therefore call this approach *single-path symbolic execution* (SPSE for short).

A key limitation of single-path symbolic execution is that it interacts poorly with loops, a common programming construct. Specifically, the generalized program executions all follow the same number of loop iterations for each loop as in the original concrete execution. For instance when used for bug-finding, in one iteration starting from a benign execution, single-path symbolic execution will be unable to expose the bug if it is only triggered with a different number of loop iterations as in the original execution. Similarly, when single-path symbolic execution is applied to test case generation to increase coverage, it will be unable (in one iteration) to generate an input that forces execution down a different branch than in the original execution, if taking that branch is only feasible with a different number of loop iterations. In other words, in single-path symbolic execution, the values of a symbolic variable reflect only the data dependencies on the symbolic inputs—control dependencies, including loop dependencies, are ignored.

In this paper we propose a new symbolic execution technique, *loop-extended symbolic execution* (or LESE for short), which generalizes from a concrete execution to a set of program executions which may contain a different number of iterations for each loop as in the original execution. In loop-extended symbolic execution, in addition to the data dependencies on inputs, the value of a symbolic variable also captures certain loop dependent effects.

At a high level, our approach works by introducing new symbolic variables to represent the number of times each loop in the program has executed. In addition to maintaining the data dependencies of program state variables on inputs as in SPSE, LESE performs a more detailed analysis to identify loop-dependent variables, for instance finding variables whose value is a linear function of one or more loop execution counts. It also relates loop execution counts to features of the program input, introducing auxiliary variables to capture the lengths and repetition counts of fields in a known input grammar. Together, these constraints allow LESE to additionally express how loop-dependent variables relate to the lengths and counts of elements in the program input.

Loop-extended symbolic execution can be used to get better results from symbolic execution whenever it is used with programs in which loops are important. It can make bug finding tools more effective and allow test-case generation to reach high coverage more quickly. Capturing more program logic in symbolic constraints allows LESE to reason about loop-related constraints with a decision

procedure, rather than requiring iterative undirected search as with SPSE.

The power of LESE is crucial for several important applications. As sample applications, this paper uses loop-extended symbolic execution to discover and diagnose buffer-overflow vulnerabilities, one of the most important classes of software errors that allow attackers to subvert programs and systems. Intuitively, LESE is powerful enough to express the effect of varying features of the input, such as number of fields or their lengths (which, in turn, affect the loop iteration counts), on program variables in a single step. This allows new vulnerabilities to be discovered using many fewer iterations than single-path symbolic execution. In addition, for a known vulnerability, our techniques are useful to diagnose a set of general conditions under which it may be exploited. These conditions are useful for understanding the vulnerability, testing for it, fixing it, and blocking attacks targeting it [7, 14, 15, 18, 20, 26, 45–47].

Because symbolic execution is often used in security-related applications such as this one, it is important that it works well for binary programs for which source code is not available. Our algorithms are designed with this constraint in mind, and overcome some of the challenges inherent in targeting binaries—such as recovering program structure like the boundaries of loops, which appear trivially in the original source.

We have built a full implementation of this technique, using a dynamic tool to collect program traces and an off-the-shelf decision procedure to simplify and solve constraints. Our tool discovers and diagnoses vulnerabilities in both a standard benchmark suite and three real-world programs on Windows and Linux. Our results show that LESE is practically effective, and confirms that the behavior of loops in real programs is often very regular.

In summary, this paper makes the following contributions:

- We introduce loop-extended symbolic execution, a new, more powerful approach to symbolic execution that incorporates the semantics of loops.

- We give algorithms and heuristics to implement LESE that are simple enough to implement at scale, but effective in practice.

- We show an application of LESE to the important security challenge of buffer overflow vulnerabilities, including a realistic implementation that does not require source code.

- We evaluate the implementation, showing that it is effective at finding and diagnosing vulnerabilities in both standard benchmarks and vulnerable real-world programs.

The rest of the paper is organized as follows: Section 2 motivates loop-extended symbolic execution with an example and provides a detailed overview. Section 3 describes the two key algorithms used in LESE. Section 4 introduces a primitive for condition analysis and how to apply it to security vulnerabilities. Section 5 provides an experimental evaluation of our technique on public benchmarks and real-world vulnerabilities. Finally, Section 6 surveys related work, and Section 7 concludes.

2. OVERVIEW

In this section, we first motivate our approach with an example showing the limitation of single-path symbolic execution, then give an overview of our technique of loop-extended symbolic execution.

2.1 Motivation and Challenges

Using symbolic execution to generalize over observed program behavior is a powerful technique because it combines the strengths

```
1   #define URI_DELIMITER ' '
2   #define VERSION_DELIMITER '\n'
3
4   void process_request(char * input)
5   {
6     char URI[80], version[80], msgbuf[100];
7     int ptr=0, uri_len=0, ver_len=0, i, j;
8
9     if (strncmp input, "GET ", 4) != 0)
10        fatal("Unsupported request");
11    ptr = 4;
12    while (input[ptr] != URI_DELIMITER) {
13        if (uri_len < 80)
14          URI[uri_len] = input[ptr];
15        uri_len++;    ptr++;
16    }
17    ptr ++;
18    while (input[ptr] != VERSION_DELIMITER) {
19        if (ver_len < 80)
20          version[ver_len] = input[ptr];
21        ver_len++;    ptr++;
22    }
23    if (ver_len < 8 || version[5] != '1')
24        fatal("Unsupported protocol version");
25
26    for (i=0,ptr=0; i < uri_len; i++, ptr++)
27        msgbuf[ptr] = URI[i];
28    msgbuf[ptr++] = ',';
29    for (j = 0; j < ver_len; j++, ptr++)
30        msgbuf[ptr] = version[j];
31    msgbuf[ptr++] = '\0';
32    LogRequest(msgbuf);
33  }
```

Figure 1: A simplified example from an HTTP server that handles requests of the form: `"GET␣"` *URI* `"␣"` *Version* `"\n"`

of dynamic and static analysis. It starts with a fully correct and detailed concrete program trace, and then generalizes that trace to predict the behavior of software on other inputs. For instance, this approach can be used to find bugs [13, 25, 41] or vulnerabilities [26] in software, to understand the conditions under which a program path can occur [7], and even to automatically exploit a security vulnerability [8]. However, the core single-path symbolic execution technique corresponds to an analysis of just one control-flow path in a program, which is a significant limitation in programs that contain loops. Next, we show this limitation with a specific example.

Consider a simplified example of a function in an HTTP server, shown in Figure 1, that processes HTTP GET requests. The program first checks that the request's method field has the value GET on line 9, and then proceeds to parse the URI and version fields into separate buffers on lines 12–16 and 18–22 respectively. It rejects this request if the version number is unsupported. Finally, it records the URI requested by the client and the version number in a comma separated string denoted by msgbuf on lines 26-30, which it subsequently logs by invoking LogRequest on line 32.

Readers may have already noticed that this code is vulnerable to a buffer overflow, but suppose we were attempting to check for such vulnerabilities using a single-path symbolic execution technique. For instance, in the course of its exploration, such an iterative test generation tool might consider the program input GET x y. It will trace the execution of the program with this input, which causes the program to reach the error condition on line 24. In order to explore the rest of the function, the exploration tool needs to find a program input that passes the checks on line 23. However, a single path does not contain enough information to reason about the length check, because the ver_len variable is not directly dependent on any byte of the input: single-path symbolic execution would not mark it as symbolic. At this point, testing tools based on
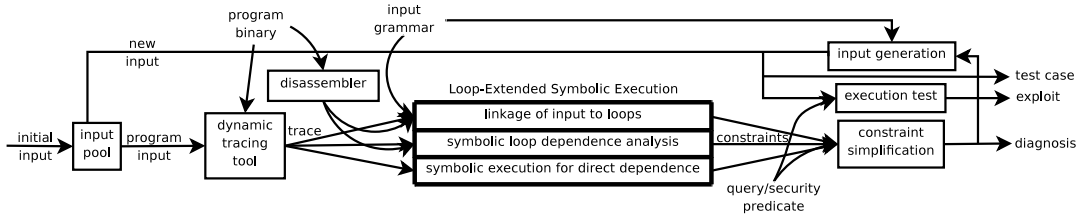
Figure 2: Overview of our loop-extended symbolic execution tool and accessory components. LESE, our main contribution, enhances symbolic execution for directly input-dependent data values, as in single-path symbolic execution, with symbolic analysis of the affects of loops (Section 3.1) and an analysis that links loops to the input fields they process (Section 3.2). Additional components, described in Sections 4 and 5.1, support LESE and particular applications such as detecting and diagnosing security bugs.

symbolic execution will usually attempt to explore other program paths, but without information from the first path to guide them, they can only choose further paths in an undirected fashion, such as by trying to take a different direction at one of the branches that occurred on the observed path. (Such tools treat the execution of a loop simply as a sequence of branches, one for each time the loop end test is executed.) For instance, a tool might determine that changing the last character of the input from a newline to z would cause the loop at line 18 to run for one additional iteration. A series of many such changes would be required before the version field was long enough to pass the check.

Similarly, consider the execution of the program on the normal program input GET /index.html HTTP/1.1. For this simple function, a single input already exercises a large proportion of the code (for instance, it executes all of the lines of non-error code in the figure). However, examining this single path is not enough to elucidate the relationship between the variable ptr and the input, because that relationship involves control dependencies.

2.2 Technique Overview

We propose a new type of symbolic execution, *loop-extended symbolic execution* or LESE, which captures the effects of more related program executions than just a single path (as in single-path symbolic execution), by modeling the effects of loops.

Broadly, the goal of loop-extended symbolic execution is to extend the symbolic expressions computed from a single execution by incorporating additional information reflecting the effects of loops that were executed. In single-path symbolic execution, the values of variables are either concrete (i.e., constant, representing a value that does not directly depend on the symbolic input) or are represented by a symbolic expression (for instance, the sum of an input byte and a concrete value). But some of the values considered concrete by single-path symbolic execution are in fact indirectly dependent on the input because of loops. In loop-extended symbolic execution these values can also be represented symbolically, and variables whose values were already symbolic because of a direct input dependency can have a more general abstract value.

To make loop-extended symbolic execution more tractable, we split the task into two parts by introducing a new class of symbolic variables, which we call *trip counts*. Each loop in the program has a trip count variable that represents the number of times the loop has executed at any moment. Then to obtain the relationship between a symbolic values and the program input, we separately obtain first the relationships between the symbolic values and one or more trip counts (in addition to their direct relationships with the input, as in single-path symbolic execution), and then the relationships between the program's trip counts and the program input:

- **Step 1: Symbolic analysis of loop dependencies.** To determine dependencies on loop trip counts, we use a program analysis that maintains the trip counts as symbolic variables that are implicitly incremented for each new loop iteration, and then looks for relationships between those variables and others in the program. (This is done at the same time as the analysis tracking direct dependencies as in SPSE, and the results combined in single symbolic expressions.) Specifically, we have found that looking for linear functions of the trip counts covers the most important loop dependent variables without excessive analysis cost.

- **Step 2: Constraints linking the input grammar to loops.** Loops are often used when fields of the input are of variable length, such as character strings and sequences of data of the same type. Our approach takes advantage of this connection by using a grammar that specifies the inputs to the program, and matching loops with the parts of the input over which they operate. In particular, the approach introduces *auxiliary* input variables to capture features of the grammar such as lengths and repetition counts.

A summary of the components of our system is shown in Figure 2; the center box, LESE, represents the primary contribution of this research.

To summarize our approach, we now return to the example of Figure 1 and explain how loop-enhanced symbolic execution is more helpful to our vulnerability testing application.

1. In the first step, the symbolic loop dependence analysis expresses various program values in terms of four trip count symbolic variables TC_i, one for each loop i in the program. For instance, the value of the variable ptr at the end of execution is abstracted by the expression $TC_3 + TC_4 + 2$, and similarly uri_len $= TC_1$, ver_len $= TC_2$, i $= TC_3$, and j $= TC_4$. The path predicate is also maintained (as in single-path symbolic execution). In this example, for instance, i < uri_len inside the third loop, while the negation holds after the loop has completed, and similarly for j and ver_len.

2. In the second step, we link the trip counts to auxiliary variables representing features of the input. In the running example, the execution counts of the first two loops are equal to the lengths of input fields: $TC_1 = \text{Length}(URI)$ and $TC_2 = \text{Length}(Version)$.

In the case of vulnerability checking, we would combine these symbolic constraints describing a class of program executions with

the condition for a violation of the security policy. In this case, for instance, the array access on line 30 will fail if `ptr` ≥ 100. Then in the same way as in a single-path symbolic execution approach, we can pass these conditions to a decision procedure to determine whether an exploit is possible, and if so, determine specific values for input variables that will trigger it. In this case, the decision procedure will report that an overflow is possible, specifically on an input for which $\mathrm{Length}(URI) + \mathrm{Length}(Version) \geq 99$.

Applying the approach to binaries. Because we wish to use these analysis techniques for security applications, it is an important practical consideration that they work on binary programs for which source code is not available. This adds further challenges for our approach: for instance, purely static analysis is more difficult on binaries because much of the structure that existed in the source code has been lost. (And of course, the real constraints we generate do not contain variable names, which we added in the example for readability.) It is in part for this reason that the symbolic execution approach is valuable in the first place, so we choose algorithms to retain these benefits in our extension. For instance, even though the technique we use to infer linear relationships between variables is closely related to a sound static analysis approach, we do not limit it to finding relationships that could hold on all possible inputs. Instead, our goal is to combine static and dynamic analysis to produce results that cover as large as possible a range of inputs for which we can still produce useful results.

Use of an input grammar. Information that constrains the space of valid inputs to a program, in the form of a grammar or otherwise, is key to scaling input space exploration beyond the limits of brute-force exhaustive search. Previous research using symbolic execution [10, 24, 33] demonstrates the benefit of using an input grammar for this purpose. In the application domains we target, suitable grammars are easily available, so we simply use them. However, for domains in which grammars are not already available, previous research shows how a grammar can be inferred [11, 31, 44]; such a system could easily be combined with ours.

3. ALGORITHMS

In this section, we discuss the algorithmic details of the key steps in loop-extended symbolic execution introduced in Section 2. Section 3.1 describes the analysis that identifies relationships between values of variables and numbers of loop iterations (step 1). Section 3.2 outlines techniques to capture the relationships between loops and the input, using auxiliary variables in the external specification of the input grammar (step 2).

The steps described below require accessory components to extract control flow graphs from binaries, make irreducible CFGs reducible, extract sizes of allocated objects, and parse input grammars. The details of these components, which form the preparation phase for steps outlined here, are given later in Section 5.1.

3.1 Symbolic Analysis of Loop Dependencies

In order to generalize its description of computations that involve loops, our tool must determine the relationship between loop-dependent variables and the loops in which they are modified. Potentially, this could be done by enhancing the basic single-path symbolic execution approach with any data-flow-style value analysis. Since linear dependencies on loop counts are very common, we choose to use a linear relationship analysis.

Specifically, our tool searches for variables whose value is a linear function of *trip count* variables representing the number of times one or more loops execute. Unlike the syntactic "induction variable" analysis commonly performed in compilers [1], we wish to extend dependencies on loop execution counts after the loop it-

$$[c_1] \circ [c_2] \rightarrow [c_1 \circ c_2] \text{ for any operator } \circ$$
$$[c_1 + a_1 \cdot TC_1] + [c_2 + a_2 \cdot TC_1] \rightarrow [(c_1 + c_2) + (a_1 + a_2) \cdot TC_1]$$
$$[c_1] \cdot [c_2 + a_2 \cdot TC_1] \rightarrow [(c_1 \cdot c_2) + (c_1 \cdot a_2) \cdot TC_1]$$
$$\top \circ E \rightarrow \top \qquad\qquad E \circ \top \rightarrow \top$$
$$[c_1 + a_1 \cdot TC_1] \circ [c_2 + a_2 \cdot TC_1] \rightarrow \top \text{ otherwise}$$
$$raise(E\langle 0 \rangle) \rightarrow E\langle 1 \rangle \qquad raise(E\langle * \rangle) \rightarrow E\langle * + 1 \rangle$$
$$[a\langle 0 \rangle] \sqcup [(a + b)\langle 1 \rangle] \rightarrow [a + b \cdot TC_1\langle * \rangle]$$
$$[a + b \cdot TC_1\langle * \rangle] \sqcup [(a + b) + b \cdot TC_1\langle * + 1 \rangle] \rightarrow [a + b \cdot TC_1\langle * \rangle]$$

Figure 3: Key rules for linear relationship analysis. Square brackets delimit abstract values and lowercase characters represent constants.

self has finished, and combine dependencies on separate loops, so we implement the linear relationship analysis in style of symbolic execution. Our approach is intermediate between purely syntactic induction variable analysis, and a general analysis for linear equalities among arbitrary program variables, which would be significantly more expensive. We will first describe the abstract interpretation in general terms, in which form it can also be applied purely statically, and then discuss how to modify it to produce more useful results in our mixed static/dynamic context.

Analysis algorithm. For each loop i in the program, we introduce a symbolic trip count variable TC_i, which represents the number of times the loop (specifically, its back edge) has executed. The core of an abstract value in our analysis is a symbolic linear combination whose terms are trip counts or other symbolic variables, with integer scaling factors and an integer constant term. For instance, the abstract value $10 + 4 \cdot TC_1 + 2 \cdot TC_2$ would correspond to a variable initialized as 10, then incremented by 4 on each iteration of the first loop and by 2 on each iteration of the second loop.

In order to link these abstract values with the loops and understand how to combine them between loop iterations, each abstract value also specifies the domain for each trip count variable it applies to. We refer to the four possible domains as 0, 1, $*$, and $* + 1$. Intuitively, 0 represents points before the loop has finished its first iteration, 1 represents later points before the end of the second iteration, and $*$ and $* + 1$ both represent abstract values applicable to all iterations, before and after the trip count in question is incremented. We write the domains in angle brackets after an expression, in order (first TC_1, then TC_2, etc.); domains not listed are assumed to be 0. Finally, to represent values that cannot be represented as a linear combination of trip counts, we have a distinguished element $\top$

The key rules for operations on these abstract values are shown in Figure 3. The analysis builds an abstract store that associates an abstract value as described above with abstract variables corresponding to distinct variables in our machine-level trace (temporaries and machine registers) and memory locations. The abstract store is updated with the side effect of each assignment, including stores to memory, and propagates forward through the program. We propagate across forward control flow graph edges in a topologically sorted order to reduce re-computation. The *raise* operator is applied to abstract values on loop back edges; the values are then joined with the abstract values representing previous iterations. It is the join operation $\sqcup$ that introduces trip count variables into a symbolic expression; $\sqcup$ also prefers $*$ to $* + 1$, which ensures that 1 and $* + 1$ domain values will not be propagated. After the first (abstract) execution of a back edge, 0 and 1 values will be joined to a $*$ value. After each subsequent abstract execution, the $*$ and $* + 1$ values will be joined into either a $*$ value if they are consistent, or to $\top$ otherwise.

For instance, consider the analysis of loop 3 on lines 26–27 of Figure 1. At the beginning of the loop, `ptr` has the abstract value

$0\langle 0, 0, 0\rangle$. At the end of the first iteration, `ptr` is incremented, and on the loop back edge the two abstract values are joined to give $0\langle 0, 0, 0\rangle \sqcup 1\langle 0, 0, 1\rangle = TC_3\langle 0, 0, *\rangle$. When `ptr` is incremented again on the next iteration, its abstract value after the back edge will be $1 + TC_3\langle 0, 0, * + 1\rangle$, which again joins to $TC_3\langle 0, 0, *\rangle \sqcup 1 + TC_3\langle 0, 0, * + 1\rangle = TC_3\langle 0, 0, *\rangle$. The effect of the increments on lines 28 and 31 and loop 4 on lines 30–31 are analyzed in a similar way, giving a final abstract value for `ptr` of $1 + TC_3 + TC_4\langle 0, 0, *, *\rangle$.

Adapting to dynamic traces. Though as previously described, the linear dependence analysis could be applied in a completely static context, some additional improvements are possible when operating as our tool does on a single execution trace.

An important simplification is that analysis of a trace does not require a conservative alias analysis, which is often a source of scalability challenges and/or imprecision in static analysis. Instead, our analysis can distinguish memory regions using the concrete addresses observed on the trace. When a symbolic value is used as a memory address (e.g., indexing an array), we use the concrete address value, as is common in single-path symbolic execution.

A second difference relates to our coverage goals. A purely static analysis attempts to give an answer that holds for the entire space of program inputs; but sometimes, no informative answer can be given, such as if the true relationship is too complex for the abstract domain. Other things equal, a result that covers a larger class of executions is most useful, but results that represent no constraint at all are useless. In mixed concrete and symbolic execution the particular set of executions to which our results apply can be flexible, so we aim for the largest set of executions for which the analysis gives an informative result.

To achieve this, we also allow our tool to lower uninformative $\top$ abstract values back to the constant value representing the value the variable had in the concrete trace at that point. This is similar in effect to removing from consideration all the executions on which that variable had any other value, though less drastic because those executions can still contribute to the generality of other abstract values. Given that there is a limit to the amount of generality our abstract values can represent, this lowering reflects a judgment that it is more valuable for them to abstract over variation that occurs close to the point where they are queried. For instance, if the combined effect of two nested loops is nonlinear, our analysis will retain the dependence on the inner loop's trip count.

Theoretically, it is not clear when the best points to lower an abstract value in this way would be: for instance, delaying a lowering at one program point might remove the need to lower another value later. However, we have had good results by performing the lowering eagerly just before a $\top$ value would otherwise propagate.

3.2 Linking Loops to Input

After Step 1 (symbolic analysis of loop dependencies), the symbolic expressions for program state variables our tool computes depend on two types of symbolic variables: the symbolic variables representing the data values of each byte in the input and the trip count variables. Thus, to obtain the relationship between the program state variables and the input, we need to obtain the relationship between the trip count variables and the input. In general, such relationships might be very complicated. However, we leverage the observation that most such trip count variables relate to certain features of the structure of the input such as the length of a variable-length field (such as a string) or the number of records of the same type (called *iterative fields*).

To precisely capture these repetitive features of program inputs, which are missing from descriptions like context-free grammars,

we introduce the concept of *auxiliary* attributes. For instance, we introduce *length* attributes to represent the size of fields that might vary in length, and *count* attributes to represent the number of times iterative fields are repeated. Auxiliary attributes are associated with grammatical units at any level (e.g., terminals and non-terminals in a context-free grammar), such as Length(*URI*) for the length of a URI field in the HTTP grammar. They can also be systematically added to an existing parser as an attribute grammar (as in `yacc` [29]); for instance, the length for a non-terminal in a rule can be computed as the sum of the lengths on the right-hand side of the rule. In some cases, the value of an auxiliary attribute is provided in another field of the input. Our technique can take advantage of auxiliary attributes that appear in the input in this way, but it also uses them in ways that do not require them to appear in the input.

The goal for the linking step is to identify loop-computed values in the program that represent auxiliary attributes; for instance, if a loop is used to compute the length of a field. Previous work [11] shows that automatic inference of variables that iterate over multiple variable-length fields is feasible, and more recently Caballero *et al.* show how to relate certain program variables to features of an input grammar [10]. We use similar techniques based on the same intuition; we determine that a loop's iteration count is the length of a field if its exit condition checks either a delimiter for the field or a value derived from a length or count auxiliary attribute of the field. In more detail, we use the following steps:

1. *Relate data-dependent bytes to fields.* As in single-path symbolic execution, our tool determines for each variable in the trace which input byte(s) (identified by offset) it directly depends on. Our tool also parses the input according to the known grammar, and so determines which protocol field contains each input byte. Therefore, one simple way of matching variables with one or more input fields is to combine these two mappings. For instance, in the example of Figure 1, the buffer `URI` contains the contents of the field *URI*.

2. *Identify variable length fields, counts, and delimiters.* The input grammar also identifies which fields correspond to the lengths or iteration counts of other fields, and our tool maps this information through direct dependencies to determine program variables that represent lengths and counts. Also, we use the grammar to determine which values are used as delimiters to signal the end of a variable-length field. For instance, in the HTTP grammar, the field *URI* is delimited by a space character.

3. *Identify variables used in loop exit conditions.* By analyzing loops as described in Section 5.1, our tool determines which variables are used in the conditions that determine when to exit a loop. For instance, the loop on lines 26–27 of Figure 1 is guarded by a condition on the variables `i` and `uri_len`.

4. *Recognize loops over delimited fields.* If the exit condition of a loop compares bytes of a field to a value that is the delimiter of the field, then we link the iteration count of the loop to the length of the field. For instance, in Figure 1, the loop on lines 12-16 compares each byte of the URI field to a space, which is known from the grammar to be the delimiter of the URI, so the execution count of that loop is the length of the field ($TC_1 = $ Length(*URI*)). In other situations, a loop may process several bytes on each iteration, which gives a relation with a scale factor. For instance, if each iteration processes a 4-byte word, the field length is equal to 4 times the loop trip count.

5. *Recognize loops over counted fields.* If the exit condition of a loop compares a variable to a value that is identified in the grammar as the length of a field or the counter for a repeated field, then we link the iteration count of the loop to that length or count field. As in the case of a delimited field, the scale factor between the field and the trip count may not be 1, for instance if a loop process several items in each iteration.

While these techniques are not enough to recognize every loop that might be written, they represent the most common patterns, and we have found them to be sufficient to capture the relationships for both length and count attributes in practice.

4. APPLYING LESE

Loop-extended symbolic execution can be used to get better results from mixed concrete and symbolic execution whenever it is used with programs in which loops occur. In this section we describe how to apply it to test generation and in problems about security bugs in software. First, we describe the primitive operation of using LESE to determine how a given predicate might be satisfied during program execution: on a single program path, but perhaps involving different numbers of loop iterations. We then show how to use this primitive for improving coverage in test generation, discovering previously unknown security bugs, and diagnosing the cause of a bug given only an execution that exercises it.

4.1 Loop-extended Condition Analysis

A basic use of single-path symbolic execution is to determine the conditions under which a predicate at a program location can be true. For instance, the predicate might be a branch condition, a programmer-provided assertion, or an array bounds check. We start with the predicate (which we will call the *query predicate*), associated with a program point, and an execution that reaches that point, but does not satisfy the predicate. Then the task is to determine the conditions on an input to the program that could cause execution to follow the same path, but cause the query predicate to be true. Using loop-extended symbolic execution, we enhance this condition analysis by taking into account other program executions that are similar to the observed one, but might involve different numbers of loop executions. Once the predicate has been chosen, this loop-extended condition analysis takes the following 3 steps:

1. *Derive symbolic expressions in terms of inputs.* Given the original execution trace, our tool first performs loop-extended symbolic execution on the trace as described in previous sections. The result of this step gives a symbolic expression for each program state variable that depends on the inputs, including both data dependencies and control dependencies introduced by loops.

2. *Instantiate query predicate.* Our tool instantiates the query predicate by using the symbolic expression computed for each variable that appears in the predicate.

3. *Solve constraints.* The query predicate can be satisfied if there exist inputs to the program that simultaneously cause it to reach the location of the predicate, and satisfy the predicate. So our tool conjoins a path condition with the query predicate, and passes this formula to a decision procedure to determine if it is satisfiable. Constraints in the path condition that arise from loop exit conditions are removed, since they are superseded by loop-dependent symbolic expressions. Our implementation uses STP [21], an SMT solver that represents machine values precisely as bounded bit vectors. If

the formula is solvable, STP returns a satisfying assignment to its free variables, which represent particular input bytes and auxiliary attributes. A grammar-based input generation tool [5, 24] can then be used to produce a version of the initial input, modified according to the satisfying assignment, which is a candidate to satisfy the predicate.

4.2 Uses for Loop-enhanced Conditions

Loop-extended condition analysis has many applications. In this section, we describe three: improving the coverage of test generation based on mixed concrete and symbolic execution, discovering violations of security properties, and diagnosing the exploit conditions of a security flaw.

4.2.1 Improving Test Generation

Test generation is the task of discovering inputs to a program that cause it to explore a variety of execution paths. Single-path symbolic execution can be used in an iterative search process to find such inputs [12,25,41], but it does not cope well with program branches that involve loop-dependent values; using LESE instead allows test generation to achieve higher coverage.

The basic operation in such an iterative search is to take an execution path and a branch along that path, and *reverse* the branch: find an input that causes execution to reach that branch, but then take the opposite direction. Reversing a branch is just an application of the primitive of Section 4.1, where the query predicate is a branch condition or its negation. The benefit of using loop-extended symbolic execution instead of single-path symbolic execution in test generation can be seen in two aspects: First, an LESE-based exploration is able to reverse branches whose conditions involve loop-dependent values; in a tool based on SPSE, by contrast, loop-dependent values are not considered symbolic. Second, an iterative search performed with LESE is more directed, since the conditions it reasons about capture the effect on values computed in loops. For instance, if a subsequent branch depends on a loop-derived value, LESE-based search requires only one iteration to determine a number of iterations of the loop to reverse the condition. The length check on line 23 in the example of Figure 1 shows this benefit: an LESE-based generation tool can immediately construct an input with a long-enough version field, because the length is a symbolic variable, while an SPSE-based tool could only stumble on such an input by trial and error.

4.2.2 Vulnerability Discovery

Many classes of security vulnerabilities can occur when a *security predicate* is violated during program execution. For instance, given a program that writes to an array, a buffer overflow occurs if the index of a write to an array is outside of the correct bounds. In a program that uses machine integers to compute the length of a data structure, an integer overflow vulnerability occurs if a computation gives the wrong result when truncated to word size. To check whether program logic is sufficient to prevent such failures, the problem of vulnerability discovery, or "fuzzing," asks whether there is a program input that could violate the security predicate. Vulnerability discovery is similar to test case generation; the only difference is the additional checking of a security predicate at each dangerous operation. Thus, like test generation, it can be performed using our loop-extended condition analysis: the query predicate is just the negation of the security predicate.

Loop-extended symbolic execution is a particularly good match for discovering vulnerabilities related to input processing, because the data structure size values that are misused in buffer overflow and integer overflow vulnerabilities are often processed using loops.

The buffer overflow in Figure 1 is typical in this way. Depending on the security property, some preprocessing might be needed to precisely define the security predicate describing how an operation might be unsafe: for instance, when checking for a buffer overflow, to determine the length of the vulnerable buffer. We will discuss some practical aspects of such preprocessing in Section 5.1.

4.2.3 Vulnerability Diagnosis

If a vulnerability has already been exploited by an attacker, another important application is diagnosing it: extracting a set of *vulnerability conditions* (general constraints on the values of inputs that exploit the vulnerability). Diagnosis is an important problem in security because vulnerability conditions are useful for automatically generating signatures to search for or filter attacks, or to help a security analyst understand a vulnerability.

Vulnerability diagnosis is again based on the loop-extended condition analysis primitive of Section 4.1: in fact, the combination of a path predicate and a negated security predicate gives a vulnerability condition. However, symbolic execution typically generates thousands of constraints, so our tool performs several optimizations to simplify them into a smaller set, as discussed in Section 5.1. Such simplification is particularly important for applications involving manual analysis, but a compact condition is also more efficient for use by later automated tools.

Some forms of vulnerability diagnosis could be performed using SPSE, but an SPSE-based diagnosis would be too narrow for many applications, including most buffer overflows. For instance, an SPSE-based diagnosis of the web server in Figure 1 could capture some generality in the contents of the input fields, but it would restrict their lengths to the particular values seen in the sample exploit. A filter based on such a diagnosis could be easily bypassed by an attack that used a different length URI. By contrast, LESE finds more general conditions; for instance, in the example of Figure 1, it finds that `msgbuf` can be overflowed by inputs of arbitrary size, as long as the sum of the lengths of two fields is at least 99.

5. EXPERIMENTAL EVALUATION

We evaluated the effectiveness of loop-extended symbolic execution by implementing an infrastructure based on the proposed techniques and applying it to discovery and subsequent diagnosis of buffer overflow vulnerabilities. We selected two kinds of subject programs for this evaluation. For comparison with other implementations, which require source code and/or run only on Linux, we use standard benchmark suites containing known overflows. To test the practical utility of our tool, we use real-world Windows and Linux applications with historic vulnerabilities. Our tool discovers all the benchmark overflows, as well as those in real-world applications, by generating just a few candidate inputs.

5.1 Implementation

We have implemented the core loop-extended symbolic execution component described earlier in OCaml, and the protocol format linkage in OCaml combined with C and Python code to integrate with off-the-shelf parsers. We utilized our existing binary analysis infrastructure [4, 42] for taking an execution trace and getting the semantics of x86 instructions.

In this section, we outline several additional components we developed to realize our proposed primitives, and heuristics that make this approach practical when working with binaries.

Memory layout extraction. To check for overflows in pointer accesses, we need a representation of the memory allocations made by the program at different points in its execution. To deal with dynamic allocation, our existing infrastructure records the arguments to memory allocation functions. For stack-based memory accesses, we implemented an existing technique called stack analysis [40], though more detailed techniques [2, 3] could alternatively be used.

Loop information extraction. Our infrastructure uses the IDA Pro tool [28] to disassemble binaries and we reused standard loop detection analysis algorithms existing in our infrastructure [10]. There are two notable additional caveats which were useful for obtaining results for our case studies.

1. *Addition of dynamic edges.* The presence of indirect call and jump instructions hinders static CFG extraction: an analysis may completely miss code blocks that are reachable only through indirect jumps. Our static control flow graph extraction is supplemented with indirect jump targets observed in the trace, which allow many more loops to be discovered. For instance, such loops were critical to obtaining accurate results in the SQL Server case study of Section 5.3.

2. *Irreducible loops.* Unlike in high-level languages, loops in binaries are often irreducible. We dealt with this by employing standard transformation techniques to make loops reducible.

Protocol Grammar. Our existing infrastructure interfaces with Wireshark [43], an off-the-shelf IDS/IPS, to obtain protocol grammars of network protocols we study.

Input Generation. We find that a relatively simple input generation approach works well with our LESE implementation: when a constraint requires that a length or count be larger, we repeat elements from the initial input until the result is long enough. In more general examples where the field being extended is subject to more additional constraints, one could also leverage grammar-based input generation approach [5, 24].

Constraint simplification. Our tool performs live-variable analysis to remove irrelevant constraints. It then performs constant folding on the remaining constraints, and simplifies them using the algebraic simplification routines built-in with the STP constraint solver [21].

5.2 Benchmarks Comparison

As benchmarks, we used a set of 14 samples extracted from vulnerabilities in open-source network servers (BIND, Sendmail, and WuFTP) by researchers at the MIT Lincoln Laboratories [47], which range between 200 and 800 lines of code each. (These are the same benchmark programs used by Xu *et al.* [46].)

Replacing SPSE with LESE would be beneficial throughout input space exploration in vulnerability discovery, since symbolic expressions for loop-dependent values allow more branches to be reversed, as discussed in Section 4.2.1. However, it can be difficult to fairly compare symbolic execution tools on an end-to-end basis, because of differences in input assumptions and search heuristics. Therefore, we confine our evaluation to the last stage of vulnerability search by starting both our tool and an SPSE tool with a program input that reaches the line of code where a vulnerability occurs, but does not exploit it. These inputs are short and/or close to usual program inputs, so they could be found relatively easily by either an SPSE-based or an LESE-based approach (though the time required would still be highly dependent on the initial input and search heuristics used). Therefore, the results on these inputs provide a bound on the performance of an end-to-end system: if a tool is unable to find a vulnerability given the hint of a nearby input, it would also be unable to find it starting from a completely unrelated input.

Results and New Bugs. The upper half of Table 1 shows the results of our tool on the Lincoln Labs overflow benchmarks. The first

Program	Input Format	Initial Input	Exploit Input	Bug / Candidate	Time (s)	Loop-Dep. Conditions
BIND 1	DNS QUERY	104 bytes, RDLen=48	RDLen=16	1/5	2511	16
BIND 2	DNS QUERY	114 bytes, RDLen=46	RDLen=30	1/4	2155	12
BIND 3	DNS IQUERY	39 bytes, RDLen=4	RDLen=516	1/2	586	13
BIND 4	DOMAINNAME	"web.foo.mit.edu"	"web.foo.mit.edu" (64 times)	1/1	4464	52
Sendmail 1	Byte Array	"<><><>"	"<>" (89 times)	4/5	672	1
Sendmail 2	struct passwd (Linux)	("","root",0,0,"root","","")	("","root",0,0,"rootroo","","")	1/1	526	38
Sendmail 3	$[String]^N$	$[\text{"a=\textbackslash n"}]^2$	$[\text{"a=\textbackslash n"}]^{59}$	1/4	626	18
Sendmail 4	Byte Array	"aaa"	"a" (69 times)	1/1	633	2
Sendmail 5	Byte Array	"\\\"	"\" (148 times)	3/3	18080	6
Sendmail 6	OPTION∘' '∘ARG	"-d aaaaaaaaaa-2"	"-d 4222222222-2"	1/1	676	11
Sendmail 7	DNS Response Fmt	TXT Record : "aaa"	Record : "a" (32 times)	1/1	237	16
WuFTP 1	String	"aaa"	"a" (9 times)	2/2	483	5
WuFTP 2	PATH	"aaa"	"a" (10 times)	1/1	197	29
WuFTP 3	PATH	"aaa"	"a" (47 times)	1/1	109	7
GHttpd	Method∘URI∘Version	"GET /index.html HTTP/1.1"	"GET "+188 bytes + " HTTP/1.1"	2/2	1562	41
SQL Server	Command∘DBName	x04 x61 x61 x61	x04 x61(194 bytes)	1/3	205	1
GDI	(Not required)	1014 bytes, INP[19:18]=0x0182	INP[19:18]=0x4003	1/1	353	2

Table 1: Discovery Results for benchmarks and real-world programs. A circle (∘) represents concatenation. In $[X]^k$, k denotes the auxiliary count attribute specifying the number of times element X repeats.

column identifies each benchmark, and the second column summarizes the input grammar our tool uses. The third and fourth columns give the initial input our tool started with, and the exploit input it found. The fifth column gives the number of candidate inputs our tool generates (after the slash), and the number of those that in fact cause an overflow (before the slash). The sixth column gives the total runtime of our tool, starting with the initial input trace and including all the discovered overflows. (The seventh column will be discussed in Section 5.4.) All experiments were performed on a 3GHz Intel Core 2 Duo with 4GB of RAM.

Our LESE tool discovers most of the bugs in just a few minutes, requiring only a few candidate inputs each. In each case, we supplied a small benign input, and the tool automatically found that a longer input could cause an overflow. Our tool also discovered an apparently new bug in one of the Lincoln Labs benchmarks: in addition to the known overflows (marked with `/* BAD */` comments in the benchmark code) our tool finds a new overflow on line 340 of the function `parse_dns_reply` in Sendmail benchmark 7. (In the other cases where our tool reports multiple overflowing inputs, they were a set of related errors marked in the benchmark.)

Comparison with Splat. Xu *et al.* [46] suggest a different approach to making SPSE work better for certain buffer overflows, by abstracting over the length of string buffers. Specifically, their length abstraction technique requires programmer-supplied source code annotation to mark a chosen prefix of the relevant buffer's contents as symbolic. In contrast, our technique automatically extracts memory buffers and their dependency on the input fields using a combination of static and dynamic analysis. More importantly, LESE does not need any information about string-manipulating functions and uses no programmer specifications for summarizing common string operations—our key enhancement to handle loop dependencies is practically sufficient to reason about the implementations of the string functions for our applications. As a result, LESE can reason about vulnerabilities present in custom operations on array inputs that may not use any common string operations (examples of these are available in our studied benchmarks).

Though the Lincoln Labs benchmarks were also studied by Xu *et al.* [46], a head-to-head empirical comparison was not possible. Unfortunately, because of the way the original benchmarks are designed to be self-contained, it was unclear which buffers (and which parts) were annotated as program inputs in their work. For instance, the BIND 2 benchmark exercises code from BIND that parses a DNS packet, and also includes code to generate an appro-

priate packet. In Xu *et al.* [46], it was unclear which value in the packet generation process was treated as the input. As shown in Table 1, we considered the whole packet itself to be the input, so that only an input that is a mostly syntactically correct packet will cause an overflow. We believe our choice makes for a more realistic evaluation, but it implies that a direct comparison of the tools' execution times would not be meaningful.

Our tool was able to find exploits for the two benchmarks (Sendmail 1 and 5) on which Splat times out. (In the case of Sendmail 5, the total running time of our tool to evaluate 3 candidate inputs is longer than the two hour timeout used with Splat, but our tool reports its first vulnerability before two hours have elapsed.) On the remaining benchmarks, our tool reproduces Splat's positive results on the complete programs.

Accuracy of candidate inputs. In the fifth column, Table 1 shows the number of candidate test inputs our tool generated in the process of finding each exploit. The fact that only a few tests were required (on average, 62.5% of the candidates our tool generates are real exploits) demonstrates the targeted nature of LESE-based search: the tool efficiently chooses appropriate loop iteration counts and prunes buffer operations that are safe, concentrating on the most likely vulnerability candidates. Of course, since the candidates are concrete inputs that can be automatically tested, failed candidates are not reported: the tool gives no false positive results.

5.3 Evaluation on Real-World Programs

As full-scale case studies, we took 3 real-world Windows and Linux programs which are known to have buffer overflow vulnerabilities. These include the program targeted by the infamous Slammer worm in 2003 [37], the one affected by a recent GDI vulnerability in 2007 [34], and an HTTP server [22]. Table 1 summarizes the vulnerabilities in these programs and the input grammars our tool used. We gave benign initial inputs to these programs that are representative of normal inputs that they would receive in practice.

Starting with a benign input, our tool uses just one iteration of LESE to discover buffer overflows in all 3 real world programs. The bugs found in the GDI and SQL cases are the same reported earlier in these programs, as we manually confirmed. For ghttpd, our tool discovers two buffer overflow vulnerabilities in the `Log` function in `util.c`. One of these is described in previous research using this subject program [14]. The new overflow involves a separate buffer and would need a separate fix. These results are shown in Table 1; next we explain each vulnerability in more detail.

GHttpd vulnerability. GHttpd is a Linux web server; we use version 1.4.3. We send an initial benign input, `GET /index.html HTTP/1.1`, to the running web service, and it responds normally. Given a trace of this execution and the HTTP grammar, our tool discovers 2 potential buffers to overflow and generates candidate exploits for each. These inputs are the same as the initial input except that their URI fields have lengths of 188 and 140 bytes respectively. Testing confirms that both candidates indeed cause overflows: the shorter request overflows one buffer, and the longer one overflows both that buffer and a subsequent one.

SQL Server vulnerability. This vulnerability is a stack-based overflow in Microsoft's SQL Server Resolution Service (SSRS), which listens for UDP requests on port 1434. Based on its specification [35], one valid message format contains 2 fields: a header byte of value 4, followed by a string giving a database name. We send the SSRS service a benign request that consists of the header byte and a string "aaa", to which the service responds correctly. Given the trace and the input grammar, our tool finds 3 potential buffers to overflow and generates one candidate inputs for each. Our automated testing reports that one candidate, which is 195 bytes long, overflows a buffer that is the same one exploited by the SQL Slammer worm. (The other two candidate inputs are longer than the maximum-length UDP packet, so they are discarded during testing and not reported.) The fact that such large inputs could be generated in a single step, rather than via a long iteration process, shows the power of LESE.

GDI vulnerability. This vulnerability in the Microsoft Windows Graphic Rendering Engine was patched in 2007. We created a benign and properly formatted WMF image file using Microsoft PowerPoint, containing only the text "aa"; the file is 1014 bytes long. We attempt to open the file using a sample application and record the program execution. Without using an input grammar, our tool discovers a potential buffer read overflow and creates an exploit input, which crashes the sample application. The only differences between the exploit and the benign input are the values in bytes 18 and 19 (shown in Table 1). Comparing with a grammar for the WMF format, these bytes correspond to the size of the image field.

5.4 Further Applications

Improving test coverage. Though our evaluation does not focus on the exploration phase of vulnerability detection, our experiments do demonstrate a feature of loop-extended symbolic execution that makes it more effective in obtaining input space coverage. As described in Section 4.2.1, LESE improves on SPSE by finding symbolic expressions for more branch conditions that depend on the number of times loops execute, making it possible for a coverage tool to reverse them. To measure this effect, we give in the last column of Table 1 the number of branches for which our tool found a loop-dependent condition but no directly input-dependent condition, so that an LESE-based tool would be able to reverse them but an SPSE-based tool would not. The count is a number of unique program-counter locations (i.e., static and context-insensitive), and excludes loop exit conditions. For instance, one of the 29 loop-dependent conditions in WuFTP 2 is a length check (on line 464) intended to prevent the buffer overflow. Because the check is faulty, it is false on both our benign and exploit inputs, but exploring both sides would be critical for an exploration task, such as verifying the lack of overflows in a fixed version. The condition is immediately apparent to our tool, but would not be considered symbolic under standard SPSE.

Vulnerability diagnosis. Our tool can also be used for vulnerability diagnosis: to find a general set of conditions under which an exploit occurs. Diagnosis is most useful when a vulnerability is

Program	Buffer size (bytes)	Condition for overflow	Constraint generation time (s)
GHttpd (1)	220	`URI.len > 172`	420 + 23
GHttpd (2)	208	`URI.len > 133`	420 + 140
SQL Server	128	`DBName.len > 64`	192
GDI	4096	`(2·INP[19:18])»2 < 0`	200

Table 2: Diagnosis results on real-world software. Generation time for GHttpd consists of the pre-processing time (420 s) and the post-processing time (23 s and 140 s) for each condition.

already being used by attackers, and it is important to understand and defend against attacks quickly: vulnerability conditions can accelerate or replace manual analysis of an exploit, and be used to generate filters to detect or block attacks. But to be useful, such conditions must be broad enough to cover a large class of attacks.

We used our tool to perform diagnosis on the same real-world programs described in Section 5.3. Either a publicly available exploit, or the exploits generated by our discovery tool, could be used and produce the same results.

Our tool's diagnoses, summarized in Table 2, are more accurate and usable than those given in previous work [18]. For instance, for the Microsoft SQL Server vulnerability, the condition our tool generates states that the vulnerable field's length must be greater than 64 bytes, whereas the buffer overrun vulnerability condition generated in previous work states that the length must be at least 97 bytes [18]. This difference turns out to be significant. Because we have no access to source code, we validated our results experimentally by supplying inputs of various sizes to the server. We found that when the vulnerable field has a size larger than 64 bytes, the overflow overwrites pointers with invalid values, causing an exception when these values are dereferenced.

Also note that most diagnoses of buffer overflows, including the GHttpd and SQL Server examples shown in Table 2, could not be produced by a standard SPSE tool, which lacks even a notation to refer to the length of an input field.

6. RELATED WORK

This section discusses two classes of related research: first, other work on analysis approaches similar to our loop-extended symbolic execution; then, work that also addresses the problem of discovering and/or diagnosing buffer-overflow attacks.

6.1 Analysis Approaches

Single-path symbolic execution. The technique we refer to as single-path symbolic execution has been proposed by a number of researchers, though the same core idea has been given several different names. It is also called "directed testing" [25], "execution-generated test cases" [12], "concolic testing" [41], and "whitebox fuzzing" [26]. It was first proposed as a test-generation technique to produce program inputs that cover new program paths, and therefore uncover bugs, including security vulnerabilities. In addition to generating new inputs, the symbolic conditions derived from an execution path also have a number of other applications, such as building signatures to filter network attacks [7] or searching for differences between implementations [6].

Extensions to single-path symbolic execution. Several previous approaches have extended single-path symbolic execution with additional information about the program or its possible inputs. Previous grammar-based approaches [10, 23, 24, 33] have taken advantage of knowledge of which program inputs are legal to reduce the size of the search space when generating new inputs. By com-

parison, our use of an input grammar in Section 3.2 is focused on extracting more information from a single execution. The Splat tool of Xu *et al.* [46] also targets the problem of buffer-overflow diagnosis, but they do not explicitly model loop constructs as in loop-extended symbolic execution. An empirical and analytical comparison to their approach is presented in Section 5.2. Pre- and post-conditions can summarize the behavior of a function so that it need not be reanalyzed [23], similar to how our approach avoids the need to reanalyze with different numbers of loop iterations. If repeated constraints are generated, they can also be later removed by optimizations such as constraint subsumption [26].

Static analysis. Determining linear (technically, "affine") relationships among the values of variables, as our analysis in Section 3.1 does, is a classic problem of static program analysis, pioneered by Karr [30]. Like many properties that involve more than one variable, it can potentially become expensive. For instance the polyhedron technique [16] requires costly conversion operations on a multi-dimensional abstract representation. More recent research has considered restricted abstract domains that allow for more efficient computation, such as "octagons" [36] and "pentagons" [32]. The techniques of Müller-Olm and Seidl [38] have the advantage of giving precise results even with respect to overflow, but their runtime is a high power of the number of variables in a program (k^7 for the interprocedural case). Random analysis [27] can also be used to determine linear relationships, with a small probability of error. For the simpler case we consider, it is sufficient to take a more efficient non-relational approach: we express the values of program variables not in terms of each other but in terms of a small set of auxiliary trip-count variables.

6.2 Discovering and Diagnosing Buffer Overflows

Buffer-overflow vulnerabilities are a critical security challenge, and many approaches target them. Sound static analysis holds the possibility of eliminating false negatives, but in practice buffer overflow checking is difficult enough that sound analysis is possible only for small programs with extensive user annotation [19]. More comparable to our approach are scalable bug-finding tools [20,45]. However, pure static analysis approaches suffer from false positives, which tool users must examine by hand. For instance, one comparison [47] using the same benchmarks we use in Section 5.2 found that many tools produced so many false positives they did only slightly better than chance. Dynamic analysis techniques, on the other hand, avoid false positives by examining programs as they execute [15, 17, 39]. However, the requirement of running on all executions means that the overhead of dynamic analysis tools can limit their applicability. Symbolic execution combines static and dynamic techniques to generalize from observed executions to similar unobserved ones, and loop-extended symbolic execution extends this generalization to include loops.

Our vulnerability diagnosis using loop-extended symbolic execution extends previous diagnosis approaches based on single-path symbolic execution [7,9,14]. Bouncer [14] employs source-code-based static alias analysis along with SPSE.

ShieldGen [18] uses a protocol-specification-based exploration of the input space to diagnose a precise vulnerability condition. However, in contrast to our work, it treats the program as a black-box, ignoring the implementation. In addition, it does not capture complex relationships between fields that may be necessary to exploit a vulnerability. For instance, as its authors point out, Shield-Gen cannot capture the condition that the combined length of two fields must exceed a buffer size for exploit (as in the example of Section 2), which our techniques can.

7. CONCLUSION

We propose loop-extended symbolic execution, a new type of symbolic execution that gains power by modeling the effects of loops. It introduces trip count variables with a symbolic analysis of linear loop dependencies, and links them to features in a known input grammar. We apply this approach to the problem of detecting and diagnosing buffer overflow vulnerabilities, in a tool that operates on unmodified Windows and Linux binaries. Rather than trying a large number of inputs in an undirected way, our approach often discovers an overflow on the first candidate it tries. Our tool finds all the vulnerabilities in the Lincoln Labs benchmark suite and gives accurate symbolic conditions describing real vulnerabilities. These results suggest that loop-extended symbolic execution has the potential to make many kinds of program analysis, including important security applications, faster and more effective.

8. REFERENCES

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, second edition, 2006.

[2] G. Balakrishnan and T. W. Reps. Analyzing memory accesses in x86 executables. In *Compiler Construction*, Apr. 2004.

[3] G. Balakrishnan and T. W. Reps. DIVINE: DIscovering Variables IN Executables. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, Jan. 2007.

[4] BitBlaze: Binary analysis for computer security. http://bitblaze.cs.berkeley.edu/.

[5] N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, Mar. 2009.

[6] D. Brumley, J. Caballero, Z. Liang, J. Newsome, and D. Song. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *USENIX Security Symposium*, Aug. 2007.

[7] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *IEEE Symposium on Security and Privacy*, May 2006.

[8] D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *IEEE Symposium on Security and Privacy*, May 2008.

[9] D. Brumley, H. Wang, S. Jha, and D. Song. Creating vulnerability signatures using weakest pre-conditions. In *Computer Security Foundations*, July 2007.

[10] J. Caballero, Z. Liang, P. Poosankam, and D. Song. Towards generating high coverage vulnerability-based signatures with protocol-level constraint-guided exploration. Technical Report CMU-CyLab-08-009, Cylab, Carnegie Mellon University, June 2008.

[11] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Computer and Communications Security*, Oct. 2007.

[12] C. Cadar and D. Engler. Execution generated test cases: How to make systems code crash itself. In *Model Checking Software, 12th SPIN Workshop*, Aug. 2005.

[13] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *Computer and Communications Security*, Nov. 2006.

[14] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: Securing software by blocking bad input. In *Symposium on Operating Systems Principles*, Oct. 2007.

[15] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. In *Symposium on Operating Systems Principles*, Oct. 2005.

[16] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Principles of Programming Languages*, Jan. 1978.

[17] J. R. Crandall, S. F. Wu, and F. T. Chong. Minos: Architectural support for protecting control data. In *ACM Transactions on Architecture and Code Optimization*, pages 359–389, Dec. 2006.

[18] W. Cui, M. Peinado, H. J. Wang, and M. Locasto. Shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing. In *IEEE Symposium on Security and Privacy*, May 2007.

[19] N. Dor, M. Rodeh, and M. Sagiv. CSSV: towards a realistic tool for statically detecting all buffer overflows in C. In *Programming Language Design and Implementation*, June 2003.

[20] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek. Buffer overrun detection using linear programming and static analysis. In *Computer and Communications Security*, Oct. 2003.

[21] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification*, July 2007.

[22] ghttpd. http://gaztek.sf.net/ghttpd/.

[23] P. Godefroid. Compositional dynamic test generation. In *ACM Symposium on Principles of Programming Languages*, Jan. 2007.

[24] P. Godefroid, A. Kieżun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *Programming Language Design and Implementation*, June 2008.

[25] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Programming Language Design and Implementation*, June 2005.

[26] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Network and Distributed System Security*, Feb. 2008.

[27] S. Gulwani and G. C. Necula. Discovering affine equalities using random interpretation. In *Principles of Programming Languages*, Jan. 2003.

[28] IDA Pro. http://www.hex-rays.com/idapro/.

[29] S. C. Johnson. Yacc: Yet another compiler-compiler. Technical Report (Computer Science) No. 32, Bell Laboratories, July 1975.

[30] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.

[31] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *Network and Distributed System Security*, Feb. 2008.

[32] F. Logozzo and M. Fähndrich. Pentagons: a weakly relational abstract domain for the efficient validation of array accesses. In *Symposium on Applied Computing*, Mar. 2008.

[33] R. Majumdar and R.-G. Xu. Directed test generation using symbolic grammars. In *Automated Software Engineering*, Nov. 2007.

[34] Microsoft Corporation. Microsoft security bulletin MS07-046: Vulnerability in GDI could allow remote code execution, Aug. 2007.

[35] Microsoft Corporation. *SQL Server Resolution Protocol Specification*, Jan. 2009. Revision 0.6.1.

[36] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, Mar. 2006.

[37] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the Slammer worm. *IEEE Security and Privacy*, 1(4):33–39, July-Aug. 2003.

[38] M. Müller-Olm and H. Seidl. Analysis of modular arithmetic. *ACM Transactions on Programming Languages and Systems*, 29(5), Aug. 2007.

[39] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Network and Distributed System Security*, Feb. 2005.

[40] P. Saxena, R. Sekar, and V. Puranik. Efficient fine-grained binary instrumentation with applications to taint tracking. In *Code Generation and Optimization*, Apr. 2008.

[41] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *European Software Engineering Conference held jointly with Foundations of Software Engineering*, Sept. 2005.

[42] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security*, Dec. 2008.

[43] Wireshark. http://www.wireshark.org.

[44] G. Wondracek, P. M. Comparetti, C. Kruegel, and E. Kirda. Automatic network protocol analysis. In *Network and Distributed System Security*, Feb. 2008.

[45] Y. Xie, A. Chou, and D. Engler. ARCHER: Using symbolic, path-sensitive analysis to detect memory access errors. In *Foundations of Software Engineering held jointly with European Software Engineering Conference*, Sept. 2003.

[46] R.-G. Xu, P. Godefroid, and R. Majumdar. Testing for buffer overflows with length abstraction. In *International Symposium on Software Testing and Analysis*, July 2008.

[47] M. Zitser, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *Foundations of Software Engineering*, Nov. 2004.