Bit-vector Support in Z3-str2 Solver and Automated Exploit Synthesis

by

Sanu Edayath Subramanian

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2015

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Improper string manipulations are an important cause of software defects, which make them a target for program analysis by hackers and developers alike. Symbolic execution based program analysis techniques that systematically explore paths through string-intensive programs require reasoning about string and bit-vector constraints cohesively. The current state of the art symbolic execution engines for programs written in C/C++ languages track constraints on a bit-level and use bit-vector solver to reason about the collected path constraints. However, string functions incur high-performance penalties and lead to path explosion in the symbolic execution engine. The current state of the art string solvers are written primarily for the analysis of web applications with underlying support for the theory of strings and integers, which limits their use in the analysis of low-level programs. Therefore, we designed a decision procedure for the theory of strings and bit-vectors in Z3-str2, a decision procedure for strings and integers, to efficiently solve word equations and length functions over bit-vectors. The new theory combination has a significant role in the detection of integer overflows and memory corruption vulnerabilities associated with string operations. In addition, we introduced a new search space pruning technique for string lengths based on a binary search approach, which enabled our decision procedure to solve constraints involving large strings. We evaluated our decision procedure on a set of real security vulnerabilities collected from Common Vulnerabilities and Exposures (CVE) database and compared the result against the Z3-str2 string-integer solver. The experiments show that our decision procedure is orders of magnitude faster than Z3-str2 string-integer. The techniques we developed have the potential to dramatically improve the efficiency of symbolic execution of string-intensive programs.

In addition to designing and implementing a string bit-vector solver, we also addressed the problem of automated remote exploit construction. In this context, we introduce a practical approach for automating remote exploitation using information leakage vulnerability and show that current protection schemes against control-flow hijack attacks are not always very effective. To demonstrate the efficacy of our technique, we performed an over-the-network format string exploitation followed by a return-to-libc attack against a pre-forking concurrent server to gain remote access to a shell. Our attack managed to defeat various protections including ASLR, DEP, PIE, stack canary and RELRO.

Acknowledgements

I would like to express my sincere gratitude to my supervisor Professor Vijay Ganesh for his continuous guidance and support, without which this thesis would not have been possible. Also, I am very thankful to Yunhui Zheng, research staff member at IBM T.J Watson Research Center, for helping me to understand the internals of the Z3-str2 solver, troubleshooting and debugging the code base whenever I was stuck. Many thanks to Professor Derek Rayside and Professor Lin Tan for reading my thesis and part of my thesis committee. Many thanks to Omer Tripp, research staff member and technical lead at IBM T.J Watson Research Center, for his valuable feedback and suggestions on my thesis. I thank my fellow colleagues Riyad Parvez, Murphy Berzish for their valuable suggestions and feedback with my thesis. I would like to thank Abiesh Jose for his advice, guidance and motivation in my career and study. Finally and most importantly I thank my family - my parents E.R Subramanian and Omana Subramanian for supporting and making me what I am today, my wife Mily M. Raju and my brother Sabin Subramanian.

Dedication

This thesis is dedicated to my parents and my wife.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Decision procedures or constraint solvers have recently gained considerable importance both in research and industry as essential tools in addressing hardware and software verification problems [24, 27, 69, 42, 65]. Our ability to solve logical assertions efficiently is crucial to the success of program analysis, verification, and automated testing tools [18, 19, 59, 60, 30]. The reason is that these tools are typically designed to generate mathematical logic formulas that characterize behaviors of the program-under-test, and then use off-the-shelf constraint solvers to solve them. In order to be effective, these solvers must be efficient and expressive enough to capture program behavior. For example, typical theories supported by solvers include the quantifier-free first-order theory of bit-vectors that can effectively model machine arithmetic, the quantifier-free theory of arrays that can model memory, and the theory of uninterpreted functions and integers to model abstractions of program state.

In recent years, there is significant interest in reasoning about string-manipulating programs written in C/C++/Java/JavaScript due to security issues associated with improper handling of untrusted string values. Researchers have come up with various powerful string solvers such as Z3-str2 [69], CVC4 [42], S3 [65] etc. to tackle these problems. These tools solve the satisfiability problem over the quantifier-free theory of string equations, regular expression (RE) membership predicates, and linear arithmetic over the length function. These tools gained much popularity in analyzing security vulnerabilities in web applica-

tions because they handle strings as a primitive data type and provide a tight integration of string length with the integer theory. However, a fundamental problem associated with these string solvers is that it is not clear at present, whether the satisfiability problem for the quantifier-free theory of word equations, regular-expression membership predicate and length function is decidable. Therefore, all current practical string solvers suffer from incompleteness and non-termination. Even so these solvers have proven to be very useful in the analysis of string-manipulating programs.

Lack of bit-vector [27] support is the major deficiency in current string solvers that limit their application in the analysis of low-level system codes. We are motivated by the above problem to build a solver for the combination of bit-vectors and strings. The reason for this particular combination is two-fold (1) bit-vectors can efficiently model the behavior of C/C++ programs (2) string solver can efficiently reason about the string operations by interpreting the string as a primitive data type. An array of bit-vectors is the best choice to represent the memory of a program. Also, it is well known that the quantifier-free fragment of bit-vectors abbreviated as QF_BV is very useful for reasoning low-level system descriptions in languages such as C and Verilog which uses finite precision integer arithmetic and bit-wise operations on bit-vectors. Moreover, the arithmetic used by digital computers is bounded, and consequently it is often more efficient or appropriate for a variety of applications to capture program behavior in bounded or bit-vector arithmetic. Symbolic execution engines for C/C++ programs, such as KLEE [19] and S2E [21] rely on the above criteria to model the state and memory of programs. These applications often symbolically analyze code, generate constraints for the decision procedure to solve and use the results of the decision procedure to guide further analysis to generate new test cases.

The above-mentioned problems motivated us to design a decision procedure for the theory of strings and bit vectors atop of Z3-str2 with an efficient integration of the underlying theories. To the best of our knowledge, there is no other native string solver that reason about strings and bit-vectors at the same time. We designed a binary search based heuristic for efficiently pruning the search space of string lengths while solving constraints on large string operations. Also, we integrated this heuristic into our base solver Z3-str2, a decision procedure for strings and integers, and our experiments reported that the binary search based approach is about 229 times faster than naive approach implemented in the

prior version of the Z3-str2 solver.

In addition to the above-mentioned contributions, we also developed an automatic exploit generation technique. The motivation for developing this technique is to showcase the power of modern program analysis in automating many aspects of the hacking process, and effectively evading widely deployed protection mechanisms. The major protection schemes deployed in modern machines are Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR). DEP allows the processor to mark writable memory locations such as stack and heap not to contain executable code. Thus, it prevents the code injection attack by making all writable memory segments as non-executable. ASLR limits the capabilities of a resource-bounded attacker by randomizing the base address of the stack, heap, code, and the memory mapped segments of an executable. A compile-time randomization technique called Position Independent Executable(PIE) strengthen ASLR by enabling the binary to be loaded and executed at any memory address without modifying it. Aside from these techniques, many compile-time attack defenses protect programs from control-flow hijack attacks, one of which include, placing canary values between a function's local variables and its return address. These canaries cannot prevent buffer overflows, but they can detect them retroactively and terminate the program before an attacker can influence control flow.

Even though there are many attack defenses and these safeguards have raised the bar significantly, attackers propose innovative attack models and enumerate different techniques to bypass these defenses. Among them, information leakage vulnerabilities play a significant role in revealing the internals of secured systems, where an attacker can leverage subtle information from program memory to augment the state of the art exploitation techniques. Therefore, we also addressed the problem of automated remote exploit construction. In this context, we introduced a practical approach for automating remote exploitation using information leakage and memory related vulnerabilities and showed that current protection schemes against control-flow hijack attacks are not always very effective.

Contributions

To summarize, this thesis makes the following principal contributions:

- **Decision procedure for a theory of strings and bit-vectors:** We designed a decision procedure for the theory of strings and bit vectors atop of Z3-str2 with an efficient integration of the underlying theories. To the best of our knowledge, there is no other native string solver that reason about strings and bit-vectors at the same time.

- **Binary search based heuristics:** We designed a binary search based heuristic for efficiently pruning the search space of string lengths while solving constraints on large string operations. Also, we integrated this heuristic into our base solver Z3-str2, a decision procedure for strings and integers. The binary search based approach is about 229 times faster than naive approach implemented in the prior version of the Z3-str2 solver.

- **Experimental study:** We evaluated the decision procedure on a set of real vulnerabilities collected from Common Vulnerabilities and Exposures (CVE) database. Also, we evaluated the binary search based heuristics on a set of benchmarks and compared to other state of the art string solvers.

- **Automated Exploit Synthesis:** We performed an automated remote exploitation breaking major attack defenses such as ASLR, DEP, PIE, RELRO, and Stack Canaries in a modern Linux machine by making use of information leakage and buffer overflow vulnerabilities.

Thesis overview

We organize the thesis into two parts. In the first part, we discuss the decision procedure for the logic of strings and bit-vectors and how the state of the art logic combination could be used in the vulnerability detection when the traditional solvers fail. In Chapter 1, we discuss the new decision procedure for the theory of strings and bit-vectors, its constraint language, and basic solving technique. Chapter 2 gives a detailed evaluation of the solver in real vulnerabilities and other benchmarks. In the second part, we discuss our work on automated exploit synthesis. Chapter 3 describes the techniques and automated system for vulnerability detection, information leakage, and control flow hijack attacks. Also, we

explain an automated remote exploitation attack that bypasses all known attack defenses in the latest Linux machine.

Chapter 2

Bit-vector Support in Z3-str2 Solver

In this chapter, we present a decision procedure for the combined theory of string and bit-vectors. This decision procedure takes an input formula over the quantifier-free theory of string equations, bit-vector arithmetic, and length function that takes input as a string and outputs a bit-vector. If the input formula is satisfiable, the decision procedure outputs a satisfying assignment, else declares that the input is unsatisfiable. Symbolic execution based program analysis techniques are extensively used in the detection of security vulnerabilities in low-level programs written in C/C++. However, these techniques are not efficiently employing the power of string solvers due to the limitation of its underlying theories. We were motivated to design and implement this decision procedure for the program analysis of low-level string manipulating C/C++ programs.

Chapter Overview

This chapter is divided into nine sections. In section 1, we provide a brief background of SMT solvers, various theories of interest, solving of string equations and its application in software engineering. In section 2, we introduce the problem we are solving. In section 3, we motivate the importance of string plus bit-vector combinations comparing to other, state of the art theory combination for hunting security vulnerabilities. In section 4, we describe the constraint syntax and semantics of our logic of bit-vectors and strings. Then

we introduce various APIs for integrating this logic to the powerful Z3-str2 solver. Section 5 explains the design, the core solving algorithm and implementation of the solver. Section 6 gives a detailed discussion of the motivating example. Section 7 presents our experimental method followed by our various results and comparison. We present the previous attempts and other related work for solving string equations in section 8. Finally, our future work plans are depicted in section 9.

2.1 Background

This section gives a brief background about SMT solvers, various supporting theories, solving string equations and its application in program analysis and automated testing.

2.1.1 SMT Solvers

Decision procedures gained much interest both in research and industry for hardware and software verification problems. The ability to solve logical assertions is essential in several tools that perform program analysis, verification, and automated testing. In computer science and mathematical logic, the satisfiability modulo theories (SMT) problem [8] is a decision problem for logical formulas with respect to combinations of background theories expressed in classical first-order logic with equality.

2.1.2 First Order Theories

Full first-order logic is not decidable, and many applications only require satisfiability over a syntactically restricted subset of full FOL, thus the SMT solvers consider the satisfiability of formulas with respect to some of these background theories. A couple of relevant theories of interest are discussed as below.

- **Bit-vectors:** Bit-vectors are extremely useful data structures used in symbolically representing hardware and software constructs. The world of bit-vectors is finite,

and it is not possible to represent arbitrarily large numbers by bit-vectors. Each term of bit-vector sort is associated with a fixed width that indicates the number of bits used to represent the value of the term. The function and predicate symbols in these theories may include extraction, concatenation, bit-wise Boolean operations, and arithmetic operations.

- **Linear Arithmetic:** Linear arithmetic is a restricted theory of arithmetic where only addition and subtraction can be used; multiplication of arbitrary terms is not allowed.. These functions can be applied to either numerical constants or variables. The relations between equality and inequalities $(=, <)$ are used for forming atomic predicates. Presburger [6] showed that the general satisfiability problem for the theory of *linear integer arithmetic* $\mathcal{T}_{LIA}$ is decidable, but its complexity is triply-exponential where the quantifier-free satisfiability problem is NP-complete. However, non-linear arithmetic is undecidable even for the quantifier-free case. In the case of *linear real arithmetic* the satisfiability problem for $\mathcal{T}_{LRA}$ is decidable, but its complexity is doubly exponential.

- **Uninterpreted functions:** In pure first-order logic, function and constant symbols are uninterpreted or free, in other words, there is no a priori interpretation attached functions and constant symbols. However, this is in contrast to functions belonging to the signature of theories, such as arithmetic where the function + has a fixed standard interpretation. They allow any interpretation that is consistent with the constraints on the function or constant. Given a conjunction of equalities between terms using free functions, a *congruence closure* [46] can be used for representing the smallest set of implied equalities.

- **Arrays:** Theories of arrays are commonly used to model actual array data structures in programs. They are also often used as an abstraction for memory. The advantage of modeling memory with arrays is that the size of the model depends on the number of accesses to memory rather than the size of the memory being modeled.

2.1.3 Solving String Equations

Makanin [44] was the first to show that quantifier-free theory of word equations is decidable in 1977, considered a theoretical breakthrough. Since then many mathematicians have improved Makanin's result [58, 35, 52, 53, 34]. Plandowski [53] showed the complexity of this problem was in PSPACE [53] in 2006. Despite decades of effort the status of the satisfiability problem for the theory of word equations, length functions and regular expression membership predicates(T_{wlr}) is still open, i.e., it is not known whether it is decidable [45]. The resolution of this question would be a breakthrough, given its connections to Hilbert's Tenth problem [45]. Hence, all currently available solvers for the theory T_{wlr} are in fact semi-decision procedures.

2.1.4 Solvers in Software Security

Constraint solvers are of tremendous value in automated testing and software security. The last decade has seen some success in deploying program verification tools to industrial software. The main techniques for program verification include theorem proving, model checking, and symbolic execution. These techniques use powerful constraint solvers due to the following reasons. Machine arithmetic is bounded and efficiently modeled by the theory of bit-vectors, a theory of arrays can represent memory locations such as stack and heap, and a theory of uninterpreted functions and integers can be used to abstract program state. Thus, solver-based techniques and tools for precise security analysis helps to reason about various corner cases that could violate the desired security policy. Furthermore, solver-based analysis tools are often more robust and easier to build than others.

2.1.5 Symbolic Execution

Symbolic execution [38] is a program analysis technique that gathered great recognition in the last few years and are widely implemented in several tools in research and industry for automated test case generation and vulnerability detection in complex software applications. The key idea behind this approach is to run the program on a symbolic input

to explore paths systematically through a program by reasoning about the feasibility of explored paths using a constraint solver. When the program execution encounters a branch that is directly or indirectly controlled by the symbolic input, appropriate constraints are added on each side of the branch, and the execution is conceptually forked to follow both sides if feasible. Finally, whenever a path terminates or hits an error, the constraints gathered on that path are solved to produce a concrete input that exercises the path.

Modern symbolic execution techniques have the power of mixing concrete and symbolic execution. In one of the approaches named Directed Automated Random Testing (DART) [30], or Concolic testing [60] symbolic execution is performed dynamically while the program is executed on some concrete input values. Another class of approach is based on execution generated testing, where both symbolic and concrete executions are mixed by dynamically checking before every operation if the values involved are all concrete. A recent technique called selective symbolic execution [21] uses a bidirectional symbolicconcrete state conversion that help the execution to seamlessly and correctly weave back and forth between symbolic and concrete mode.

Significant advances of constraint solving techniques in the past few years are the primary reason for the success of symbolic execution techniques. Bit-vector solvers are used in the analysis of low-level programs written in C/C++ whereas, string solvers are widely used for detecting security vulnerabilities in web application domains. KLEE [19] and S2E are two widely used symbolic execution engines for programs written in unsafe languages like C/C++, and Kudzu [57], Jalangi [59] and SymJS [39] [40] are well known symbolic execution engines using string solvers for JavaScript programs.

2.1.6 Integer Overflow Vulnerabilities

Integer overflow vulnerabilities are the result of an operation on an integer value that causes it to exceed the maximum possible value or decrease below its minimum possible value. Because of this, the number wraps and resulting a very large number to become very small or vice versa. These vulnerabilities could only be revealed using bit-vector solvers as bounded arithmetic is the primary reason for these overflows. When these overflow bugs

occur on calculating some buffer size, it results in severe security vulnerabilities leading to stack or heap overflows. There are three types of integer overflow vulnerabilities

- **widthness overflows** [1]**:** occur when the code tries to store a value in a variable that is too small (in the number of bits) to handle it: a typical situation is when a variable of a given type is cast into another one whose type is smaller than the original one.
- **arithmetic overflows:** occur when a calculation produces a result that is greater in magnitude than that which a given target type can accommodate.
- **signedness bugs** [1]**:** occur when an application fails to differentiate between both signed and unsigned integers when measuring the lengths of buffers, and confuses the signed type with unsigned one at some point. Therefore, the signed value is interpreted as its unsigned equivalent, meaning that a negative number becomes a large positive number.

2.2 Problem Statement

Symbolic execution based program analysis techniques are extensively used in the detection of security vulnerabilities in low-level programs written in C/C++. However, these techniques are not efficiently employing the power of string solvers due to the limitation of its underlying theories. Design a decision procedure for the theory of strings and bit-vectors to improve the efficiency of symbolic execution engines and to expose vulnerabilities that remain hidden when using traditional string solvers.

2.3 Motivation

Our primary motivation behind the decision procedure for the logic of strings and bit-vectors is to provide significant performance improvement in the symbolic execution of programs written in C/C++. Improper string manipulations are an important cause of

software defects, which make them a target for program analysis. A study [54] on the concolic testing tools and their limitations shows that there are significant portions of string operations in system level codes and which add additional overhead to current symbolic execution engines. In current symbolic execution techniques, there is a semantic gap between the high-level notion of strings and low-level representation of program states and memory. Bit-vector solver is an unavoidable part of low-level program analysis as it need to capture constraints with bit-level precision to efficiently reason about arithmetic overflows, bitwise operations, and pointer casting. Also, symbolic memory is also modeled as an array of bit-vectors and binary instructions as operations in bit-vector theory. Symbolic execution engines like KLEE and S2E collect constraints as bit-vectors by symbolically executing each branch in program statements and solve it using powerful constraint solvers like STP[27] and Z3[24]. However, these engines perform poorly on programs containing string functions as it fails to capture the high-level semantics of string data type in accordance with the low-level bit-vector representation of all program data.

Current symbolic execution engines that track constraints on a bit-level cause the path explosion problem when string functions are iterated character by character through their inputs. Typically string functions, such as *strlen* or *strcmp*, mainly consist of loops that iterates character by character through one or multiple input strings until a particular condition is met, for instance until the current character is the null terminator, marking the end of the string. Given that each character in the string is checked for the terminating condition, the symbolic execution engine will fork one new state for each symbolic character, leading to path explosion. For example, an invocation of the string library function *strlen* on a symbolic string s of size N will generate a total of $N + 1$ paths, one for each possible value of the length, between 0 and N, regardless of the usage of this return value throughout the rest of the program. In essence, uninteresting part of code regions are explored by enumerating each branch without efficiently pruning the search space. Prior works from the S2E [21] group explained these performance bottlenecks with the strings in their symbolic analysis. These limitations forced us to apply string solvers that reason string as a primitive data type similar to integers and bit-vectors, in the context of symbolic execution. However, to the best of our knowledge there are no other native string solver that reason about strings and bit-vectors at the same time.

Moreover, it is tough to detect certain classes of security vulnerabilities arising from certain overflow and underflow errors with the state of the art string solvers. As the theory of integer is unbounded, the existing string solvers are not very efficient to reason about arithmetic overflow and underflow errors. Therefore, heap memory corruption vulnerabilities originated from integer overflows mostly remain undetected by the state of the art program analysis techniques that use traditional string solvers. This kind of security vulnerabilities inspired us to propose a new theory combination for the underlying solver to efficiently reason about different corner cases leading to overflows and memory corruptions when used with existing program analysis techniques. For instance, if we analyze the history of integer related overflows in CVE database, we can see that operations on the large value of strings are one of the primary sources of integer overflow vulnerabilities. These bugs remain undetected by most of the analysis engines if we use traditional string solvers, that focus the analysis of string functions in scripting languages and web applications. Existing string solvers primarily support the theory of strings and integers and they do not need to reason about the low-level memory when identifying vulnerabilities in web applications. However, integer related vulnerabilities typically arise in low-level applications written in C/C++, when the developer fails to take account of the upper bound defined for the data type.

We will explain the limitations of existing string solvers in the context of low-level program analysis and the importance of a latest theory combination using a motivating example. Consider the *check_login()* function shown in Figure 2.1. Here, the program calculates the length of the user controlled input value *username*, adds 1 to accommodate the trailing null character. A new buffer is allocated for the resulting size and copies the *username* into it using the *strcpy* function. This code behaves as intended for the normal-sized input. However, an integer overflow occurs if the user submits a *username* consisting 65,535 characters. The variable *len* is declared as unsigned short in which the size of the variable is 16 bit long and can hold any value between 0 and 65,535. When a string of length 65,535 is submitted as *username*, the result of *strlen(username)*+1 wraps to become 0 causing integer overflow. Also, the integer overflow causes a zero size buffer to be allocated in the heap due to malloc(), and the long *username* is copied into it, causing a heap overflow. We encode the assertions in SMT-LIB format by hand for the

```
bool check_login(char* username, char* password){
    unsigned short len = strlen(username)+1;
    if(len > 32){
        invalid_login_attempt();
        exit(-1);
    }
    char* _username= (char*) malloc(len);
    strcpy(_username, username);
    ...
}
```

Figure 2.1: Overflow vulnerabilities in login module

vulnerable path of the program. Using the same set of constraints solvers like Z3-str2 which has native support for the theory of strings and integer is very inefficient to solve the constraints. However, the decision procedure for the theory of strings and bit-vectors finds the vulnerability and generates a satisfying model.

2.4 Constraint Syntax and Semantics

The syntax of word equations and the lengths are defined using the following notions. We fix a disjoint two-sorted set of variables $var = var_{str} \cup var_{bv}$; var_{str} containing string variables, denoted $X, Y, S, \ldots$ and var_{bv} consists of bit-vector variables, denoted $m, n, \ldots$. We also define a two-sorted set of constants $Con = Con_{str} \cup Con_{bv}$. Moreover, $Con_{str} \subset \Sigma^*$ for some finite alphabet, Σ, whose elements are denoted $f, g, \ldots$. Elements of Con_{str} will be referred to as *string constants* or *strings*. Elements of Con_{bv} are constant sized bit-vectors. Terms may be string terms or bit-vector terms. A string term is either an element of var_{str}, an element of Con_{str}, or a concatenation of string terms (represented by the function *concat* or interchangeably by $\cdot$ operation). The *strlen_bv()* is used to represent the length function of string terms. The empty string is represented by ϵ, and its length is a bit-vector value of zero. A bit-vector term is an element of var_{bv}, an element of Con_{bv}, the *strlen_bv()* function

Term:bool	::=	*Var:bool*
	|	true
	|	false
	|	(Contains *Term:string* *Term:string*)
	|	(StartsWith *Term:string* *Term:string*)
	|	(EndsWith *Term:string* *Term:string*)
	|	(RegexIn *Term:string* *Term:regex*)
Term:int	::=	*Var:int*
	|	*Number*
	|	$(\{+, -, \times, \div\}$ *Term:int* *Term:int*)
	|	(Length *Term:string*)
	|	(IndexOf *Term:string* *Term:string*)
	|	(IndexOf2 *Term:string* *Term:string* *Term:int*)
	|	(LastIndexOf *Term:string* *Term:string*)
$Term{:}BitVec_n$	::=	$Var{:}BitVec_n$
	|	$(BitVecConst_n)$
	|	$(\{+, -, \times, \div\}$ $Term{:}BitVec_n$ $Term{:}BitVec_n$) $\rightarrow \mathrm{BitVec}_n$
	|	(strlen_bv *Term:string*) $\rightarrow \mathrm{BitVec}_n$
	|	(CharAt_BV *Term:string* $Term{:}BitVec_n$) $\rightarrow \mathrm{BitVec}_8$
Term:string	::=	*Var:string*
	|	*ConstStr*
	|	(Concat *Term:string* *Term:string*)
	|	(Substring *Term:string* *Term:int* *Term:int*)
	|	(Substring_BV *Term:string* $Term{:}BitVec_n$ $Term{:}BitVec_n$)
	|	(Bv2Str $Term{:}BitVec_8$)
	|	(BvArray2String $Term{:}ArrayBitVec_n$ *Term:* $BitVec_n$)
	|	(Replace *Term:string* *Term:string* *Term:string*)
	|	(CharAt *Term:string* *Term:int*)
Term:regex	::=	(Str2Regex *ConstStr:string*)
	|	(RegexStar *Term:regex*)
	|	*(Term:regex)+*
	|	*(Term:regex)?*
	|	(RegexConcat *Term:regex* *Term:regex*)
	|	(RegexUnion *Term:regex* *Term:regex*)
Expr:bool	::=	*Term:bool*
	|	(= *Term:bool* *Term:bool*)
	|	(not *Expr:bool*)
	|	(and *Expr:bool* *Expr:bool*)
	|	(or *Expr:bool* *Expr:bool*)
	|	(ite *Expr:bool* *Expr:bool* *Expr:bool*)
	|	(implies *Expr:bool* *Expr:bool*)
	|	$(\{<, \leq, =, \geq, >\}$ *Term:int* *Term:int*)
	|	(= *Term:string* *Term:string*)
Assertion	::=	(**assert** *Expr:bool*)

Figure 2.2: The input language of Z3-str2

applied to a string term, a constant bit-vector multiplied by a bit-vector term, or a sum of bv terms. The theory contains two types of atomic formulas, namely, word equations and length constraints. This decision procedure supports a list of common string-related operators such as *CharAt*, *Contains*, *Endswith*, *Indexof*, *Lastindexof*, *Replace*, *Substring* and etc,. which are original to the base solver along with *strlen_bv*, *CharAt_BV*, *Substrting_BV*, *Bv2str*, *BvArray2String* etc,. Formulas are defined inductively over atomic formulas and are quantifier-free. The constraint syntax for the decision procedure is presented Figure 2.2.

2.5 Design and Implementation

In this section, we explain the design overview of the constraint solver Z3-str2. Further, we explain how word equations are solved in Z3-str2 followed by the solving algorithm. Later we explain the new search space pruning technique based on the binary search in the context of underlying solver modes.

2.5.1 Design Overview

The decision procedure for the theory of bit-vectors and string is built atop of Z3-str2 [69]. The base solver Z3-str2 is essentially a string plug-in built into the Z3 SMT Solver [24], with an efficient integration between the string plug-in and Z3's integer solver. The architectural schematic of the Z3-str2 string solver is given in Figure. 2.3 in which the word equations are solved with respect to the underlying integer solver.

The solver in the string plus bit-vector mode purifies input into bit-vector and string constraints. The string constraints are solved using string plug-in and bit-vector constraints through Z3's bit-vector solver. The plug-in may consult the Z3 core to detect equivalent terms. The word equations are solved using an algorithm described in detail in the section 2.5.3 below. The length constraints are converted into a system of pure integer linear arithmetic inequations and solved using Z3's bit-vector solver. The interaction between the bit-vector and string theory is explained in the section 2.5.4

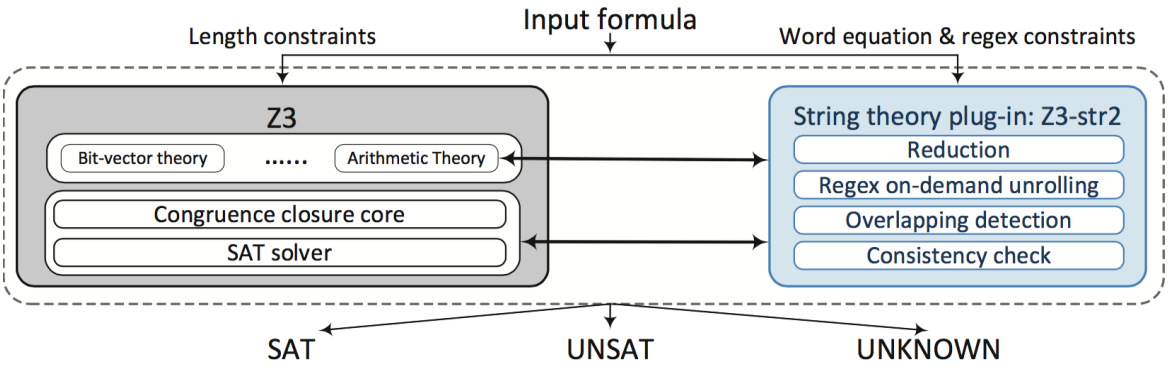

Figure 2.3: Architecture of Z3-str2 solver [69]

2.5.2 Solving Word Equations

The word equation solving component of our decision procedure for the theory of strings and integers is inherited from the base solver Z3-str2 [69]. Starting with the work of Makanin [44], many decision procedures [53, 58, 34] have been proposed. While most procedures are not accompanied by practical implementations, they are a rich source of ideas for all the solvers that have recently been implemented. For example, the Z3-str2 solver follows ideas, namely, *boundary labels, generalized word equations, and arrangements* that have their roots in the very first decision procedure for word equations by Makanin.

The key technique used by Z3-str2 [69] to solve a word equation W is to recursively convert W equisatisfiably into the disjunction of conjunctions of simpler equations we call arrangements. These arrangements are computed by aligning the concatenation function on the LHS and RHS of a given equation such that an occurrence of concatenation function in the LHS (resp. RHS) may "split" or "cut" variables on the RHS (resp. LHS). There are many different alignments of variable boundaries in the LHS (resp. RHS) that can split variables in the RHS (resp. LHS). We call every such alignment an *arrangement*. The crucial fact about word equations is that every equation can be equisatisfiably rewritten into a finite set of arrangements, where each arrangement is a finite set of word equations obtained from the splitting procedure. The Z3-str2 solver exploits this fact and solves word equations by converting them into finite sets of arrangements and inspecting each

one individually to see if they are satisfiable. The input word equation is SAT if and only if at least one arrangement is SAT. This, in a nutshell, is how the Z3-str2 solver solves the word equations, i.e., by recursively converting equations into a disjunction of arrangements (where each arrangement is a simpler set of equations) until a set of arrangements is derived where the satisfiability is determined purely via inspection. While simple, elegant and efficient for typical equations obtained from program analysis, the word equation solver described here may fall into infinite loops when the word equation contains overlapping variables. However, Z3-str2 solver has support for detecting overlapping variables and the technique for the detection of overlapping variables is well presented in [69].

Label Arrangements: We leverage boundary labels to reason about the relative positions of the subparts in words, such that we can reduce the original equations to a set of smaller equations for the corresponding subparts until the equations become so fine-grained that the solution can be directly inferred. The set of input equations is UNSAT if none of the possible breakdowns leads to valid solutions. In this subsection, we explain how to split equations into smaller ones based on the arrangements and how to determine if equations are in *solvable form*. A formula (i.e. a conjunction of equations) is in solvable form if each equation is either an equivalence between a variable and a character, or equivalence between two variables.

Formula Transformation: Now we discuss how to generate arrangements from equations and generate arrangements for variables from equation arrangements. Once a variable arrangement is selected, the constraints for its sub-parts (i.e. their alignments with other variables and characters) are determined. Therefore, we discuss how to split equations to represent the constraints on sub-parts.

The process consists of two steps.

- In the first step, a variable is split to a set of new variables according to the variable arrangement. Each equation is rewritten by replacing each variable with the split variables.
- In the second step, each equation is divided into a set of new equations, each constraining a sub-part of the original equation. This process is guided by determining

the common labels between the label sets of the LHS and RHS words of the equations generated in the first step.

A formula defined as the conjunction of equations is said to be in solvable form if each equation is either an equivalence between a variable and a character, or equivalence between two variables. For variables that are directly or indirectly (i.e. through other variables) equivalent to a character, their solution is the character. The solving process, namely the consistency condition in arrangement production ensures that the same variable is not equivalent to different characters. Variables that are not equivalent to any character, directly or indirectly, are free variables such that we can assign any characters to them. Overlapping arrangement detection brings the following significant benefits. If the input formula is UNSAT, and it may have overlapping arrangements, a decision procedure without detecting overlaps will inevitably lead to an infinite loop in formula reduction. Overlap detection also allows the procedure to avoid exploring the overlapping arrangements and quickly find the non-overlapping solutions. If the input formula is SAT and the solution does not contain any overlapping arrangement, our procedure will be able to reach the solvable form.

2.5.3 Algorithm

The decision procedure for the theory of bit-vector and strings inherits all the major features of its predecessor Z3-str2 for solving word equations such as boundary labels, word equation splits, label arrangements and detection of overlapping variables. However, it differs from the Z3-str2 in reasoning about the length constraints derived from the word equations and in search space pruning strategy for reaching consistent lengths. The procedure is summarized in Algorithm 1. It takes a formula F, produces SAT, UNSAT or UNKNOWN results. UNKNOWN means that the algorithm has encountered overlapping arrangements and pruned those arrangements, even though it did not find any SAT solution.

The algorithm consists of three steps. In step one, it generates the set of possible arrangements for each equation and selects one from the set to proceed. In the next step,

Algorithm 1 High-level description of the word equation solving

Input: $\mathcal{Q}_w$: word equations, $\mathcal{Q}_l$: bit-vector constraints over length function
Output: SAT / UNSAT / UNKNOWN

procedure solveStringConstraint($\mathcal{Q}_w$, $\mathcal{Q}_l$)
  if all equations in $\mathcal{Q}_w$ are in solved form **then**
    if $\mathcal{Q}_w$ is UNSAT **or** $\mathcal{Q}_l$ is UNSAT **then**
      return UNSAT
    end if
    if $\mathcal{Q}_w$ and $\mathcal{Q}_l$ are consistently SAT **then**
      return SAT
    end if
  end if
  $\mathbb{H}$=StringBvIntegration($\mathcal{Q}_w$)
  convert $\mathcal{Q}_w$ equisatifiably into DNF formula $\mathcal{Q}_a$
  for each disjunct D in $\mathcal{Q}_a$ **do**
    $\mathbb{A}$= all possible arrangements of equation in D
    for each arrangement A in $\mathbb{A}$ **do**
      l_A=extract the length constraint implied by A from $\mathbb{H}$
      if l_A is inconsistent with bit-vector theory **then**
        Remove arrangement A from $\mathbb{A}$
      end if
    end for
    for each arrangement combinations **do**
      split each variable to sub-variables based on the selected arrangement
      convert $\mathcal{Q}_w$ equisatifiably to $\mathcal{Q}_w^{'}$ of simpler equations
      $\mathcal{Q}_l^{'}$ be the new set of length constraints
      r=solveStringConstraint($\mathcal{Q}_w^{'}$,$\mathcal{Q}_l^{'}$)
      if r=SAT **then**
        return SAT
      end if
    end for
  end for
  if overlapping variables have ever been detected **then**
    return UNKNOWN
  else
    return UNSAT
  end if
end procedure

variable arrangements for each variable are computed based on the previously selected equation arrangements where each variable may have multiple arrangements. In the third step, the formula is split based on the selected variable arrangements. The algorithm checks an overlapping arrangement has ever detected and pruned. If so, the procedure returns UNKNOWN; otherwise, it returns UNSAT. The essence of the algorithm is to discover all the boundaries that are correlated and search for a total order of them. This decision procedure also detects infinite loops in formula reduction by identifying overlapping arrangements. Also, the procedure can always find the solution arrangement if it is non-overlapping.

2.5.4 String and Bit-vector Theory Integration

During the solving process, the string plug-in may generate length constraints that are incrementally added on demand to Z3's bit-vector solver, that are regularly checked for consistency with both the input length constraints and previously added ones. On any well-formed input as described in section 2.4, the decision procedure may return SAT, UNSAT or UNKNOWN. On inputs containing bit-vector and string constraints, if either Z3's bit-vector solver or the string plug-in determines that their respective purified inputs be UNSAT, decision procedure reports UNSAT. When both the bit-vector and string solver returns SAT and establishes a consistency between their results, the decision procedure returns SAT. If the string plug-in detects that the input equations have complicated overlaps that its heuristics cannot handle, it reports UNKNOWN. This is a source of incompleteness in the implementation of the base solver Z3-str2. Note that Z3-str2, like other competing solvers such as CVC4, is sound but not complete.

Basic Length Rules

The basic length assertion on strings X, Y and Z can be expressed as follows. In the constraint language l_X, l_Y and l_Z are represented by $strlen\_bv(X, n)$, $strlen\_bv(Y, n)$ and $strlen\_bv(Z, n)$ respectively where n is the bit-vector sort. Null string is represented by ϵ.

1. $l_X \geq 0$

2. $l_X = 0 \iff X = \epsilon$

3. $X = Y \implies l_X = l_Y$

4. $X \cdot Y \cdot Z \cdots \implies l_X + l_Y + l_Z + \cdots$

Theory Interaction

When solving the word equations, the solver for the strings and bit-vectors generates new assertions in the domain of the other theory and vice versa. Inside the string theory, the set of arrangements that is explored is constrained by the assertions on string lengths, which are provided by the bit-vector theory. On the other hand, the string theory will derive new length assertions when it makes progress in exploring new arrangements. These assertions are provided to the bit-vector theory so that the search space is pruned. The procedure for string and bit-vector theory integration is summarized in Algorithm 2.

Consider the word equation $X \cdot Y = M \cdot N$ where X, Y, M and N are non empty string variables. It has three possible arrangements [69] as shown below where T_1 and T_2 are temporary string variables.

1. $(X = M \cdot T_1) \wedge (N = T_1 \cdot Y)$

2. $(X = M) \wedge (N = Y)$

3. $(M = X \cdot T_2) \wedge (Y = T_2 \cdot N)$

The corresponding length assertions derived from the above three arrangements are as follows

1. $(l_X = l_M + l_{T_1}) \wedge (l_N = l_{T_1} + l_Y)$

2. $(l_X = l_M) \wedge (l_N = l_Y)$

3. $(l_M = l_X + l_{T_2}) \wedge (l_Y = l_{T_2} + l_N)$

These length assertions are added to the Z3 core and then processed using the bit-vector theory. We use a binary search based heuristics to search the value of each length variable in the bit-vector theory. The search space pruning technique is explained in the section 2.5.5. For instance, if the bit-vector theory infers that $l_X > l_M$ and $l_N > l_Y$ the string theory only needs to explore the first arrangement, and the procedure can converge faster.

Algorithm 2 Integration between theory of strings and bit-vectors

Input: Word equations($\mathcal{Q}_w$)
Output: Hash table of length assignments

```
1:  procedure StringBvIntegration(Q_w)
2:      H= Hash table of length(L_str) and bit-vector value
3:      for each equation E in Q_w do
4:          A= all possible arrangements of E
5:          for each arrangement A in A do
6:              L= all implied length constraints
7:              for each length L in L do
8:                  L_bv be the length implied by bit-vector theory
9:                  low=LOWER_BOUND
10:                 high=UPPER_BOUND
11:                 while low ≤ high do
12:                     mid=low+(high-low)/2
13:                     if L_bv=mid then
14:                         Add (L,mid) into H
15:                         break
16:                     else if L_bv > mid then
17:                         low=mid+1
18:                     else
19:                         high=mid-1
20:                     end if
21:                 end while
22:             end for
23:         end for
24:     end for
25:     return H
26: end procedure
```

2.5.5 Pruning the Search Space via Binary Search

Even though the tight interaction between these theories improved the solving time to a great extent; the decision procedure performs poorly on inputs containing large values of length constraints. In Z3-str2, the string theory and integer theory works in parallel pruning search space to find consistent string lengths using naive linear search based techniques. Once the consistency is fixed between string variables and their lengths, value arrangements are performed for further solving. However, the linear search consumes a significant amount of solving time for choosing consistent length value when dealing with large length constraints that further impose additional performance overhead for the decision procedure. Therefore, we designed a new search space pruning technique based on the binary search with heuristics support for Z3 core's backtracking functionality. We implemented this technique in the base solver along with the new decision procedures and found that the solving time decreased dramatically as presented in the section 2.7.2.

String - bit-vector solver: In the theory of bit-vectors, we can always restrict variables within a lower and upper bound. If we select a bit-vector variable of sort n, the value of the variable will always be between 0 and $2^n - 1$. We implemented a binary search based heuristic for guiding the search, relying on its fixed upper and lower bounds. Therefore, the heuristic for adding length assertions to the bit-vector theory follows a binary search pattern and can solve constraints involving large strings very efficiently.

The string plug-in continuously queries the underlying theory to get a suitable length for each variable through Z3 core. For instance, consider the constraint $l_X \geq 52000$ where X is a string variable and l_X represents the length operation. The prior version of the Z3-str2 solver, linearly samples the search space and adds assertions for each value of the sample space. The Z3 core pick one assertion from the given set and verifies it with the underlying theory, and backtracks to try another assertion if it is not consistent. The assertions added in the first iterations are $(l_X = 0) \vee (l_X = 1) \vee (l_X = 2) \vee (l_X \geq 2)$, and in the second iteration $(l_X = 3) \vee (l_X = 4) \vee (l_X = 5) \vee (l_X \geq 5)$ is added and so on. Say initially Z3's core picks the assertion $l_X = 2$ but it is not consistent with our length constraint. Therefore, the core backtracks and try another assertion, for instance, $l_X \geq 2$ to further continue the search procedure. In essence, the string plug-in needs to

add the assertion in a linear order till it finds a valid one. The same process has to be done for all string variables to find the consistent length that significantly reduces the solver performance.

In binary search based procedure, for instance with a bit-vector sort of 16, we have the lower bound as 0 and the upper limit as 65535. In the first iteration, string solver adds assertion $(l_X < 32767) \vee (l_X = 32767) \vee (l_X > 32767)$ to the Z3's core and in the second iteration the assertion becomes $(l_X < 49151) \vee (l_X = 49151) \vee (l_X > 49151)$, and so on. Following the binary search pattern, the solver quickly finds a valid length assignment for l_X in few iterations.

String - integer solver: Z3-str2 in the string plus integer mode also suffer the same performance issue when solving large string lengths; as a remedy to this, we ported our technique into the base solver as well. However, the major challenge was that we could not fix an upper bound for integer theory, and the traditional binary search based technique option ruled out in the first place. This motivated us to design new heuristics, driven by binary search and backtracking, to guide the search. In this approach, we chose a window with concrete lower and upper bound in which the size of the window varies dynamically in each iteration depending on the search criteria. We perform the binary search within the window while sliding the window from lower to higher values to find an upper bound. In other words, the lower and upper end of the window is modified when the window slides and the window size expands or shrinks by the order of two. The backtracking ability of the heuristics helps to guide the search and choose previous windows when the Z3's core backtracks. The heuristics for the theory of strings and integers is summarized in Algorithm 3.

2.6 Discussion of Motivating Example

In section 2.3 we discussed a motivating example shown in Figure 2.1. In this section, we give a detailed analysis of our example, using the latest decision procedure for strings and bit-vectors against the original version of Z3-str2. Here, the program calculates the length of the user controlled input value *username*, adds 1 to accommodate the trailing null

Algorithm 3 Binary search based heuristics for string-integer solver

Input: l_X
Output: Integer value
procedure GETCONSISTENTLENGTH(l_X)
    L_{int} be the length implied by integer theory
    low=0
    high=2
    upper_bound_fixed=false
    while low $\leq$ high **do**
        mid=low+(high-low)/2
        if L_{int}= mid **then**
            return mid
        else if L_{int} > mid **then**
            low=mid+1
            if not upper_bound_fixed **then**
                high=2*high
            end if
        else if L_{int} < mid **then**
            upper_bound_fixed=true
            high=mid-1
        end if
    end while
end procedure

character. A new buffer is allocated for the resulting size and copies the *username* into it using the *strcpy* function. Even though the program tries to exit when $len > 32$, the attack can be triggered using specially crafted input. The variable *len* is declared as unsigned short in which the size of the variable is 16 bit long and can hold any value between 0 and 65,535. When a string of length 65,535 is submitted as *username*, the result of *username*+1 wraps to become 0 causing integer overflow. Also, the integer overflow causes a zero size buffer to be allocated in the heap due to malloc(), and the long *username* is copied into it, causing a heap overflow.

Figure 2.4 represents the constraints encoded in the SMT-LIB format for the Z3-str2 solver. In the above set of constraints *username*, *_username* are modelled as string vari-

```
(declare-variable username  String)
(declare-variable _username String)
(declare-variable len Int)

; len=strlen(username)+1
(assert (= len (mod (+ (Length username) 1 ) 65535)))

; len<32
(assert (< len 32 ))
(assert (= (Length _username) len ))
(assert (> (Length username) (Length _username) )); overflow condition
```

Figure 2.4: Path constraints for string - integer solver of Z3-str2

ables, and *len* as an integer variable. The path constraints to trigger the vulnerability is also encoded in the form of assertions. The solver in the string plus integer theory took more than twenty hours and timed out because of expensive modular arithmetic constraints.

It is shown in the same set of constraints in the language of strings and bit-vectors as presented in Figure 2.5. In the program, *len* is of type unsigned short and is encoded as a bit-vector variable of width 16. The vulnerable path conditions are exactly same as that of the one presented for the theory of strings and integers but are expressed in the language of strings and bit-vectors. The new decision procedure solved the constraints as SAT and produced a model in 0.27 seconds. The solver returned an empty string for *_username*, 0 for *len* and a 65535 character string for *username*.

2.7 Experimental Results and Evaluation

This section presents the experiments and evaluation of the results to compare our techniques against the state of the art solvers.

```
(declare-variable username  String)
(declare-variable _username String)
(declare-variable len (_ BitVec 16))

;len=strlen(username)+1
(assert (= len (bvadd (strlen_bv username 16) (_ bv1 16) )))

;len<32
(assert (bvult len (_ bv32 16) ))
(assert (= (strlen_bv _username 16) len))

;strlen(username)>strlen(_username)
(assert (bvugt (strlen_bv username 16)  (strlen_bv _username 16)))
```

Figure 2.5: Path constraints for the solver of strings and bit-vectors

2.7.1 Evaluating the Solver for Strings and Bit-vectors

We did an analysis of heap overflow vulnerabilities identified in string-manipulating programs in the common vulnerabilities and exposure (CVE)[2] database. We found that the overflow or underflow in arithmetic computations on buffer size was the primary reason for these bugs, leading to severe heap overflows and memory corruptions. Therefore, we are interested in capturing such vulnerabilities using the decision procedure for strings and bit-vectors. To demonstrate the efficacy of the theory combination of bit-vectors and strings, we selected seven real vulnerabilities from the CVE database. For the benchmarks discussed below, we analyzed the vulnerable part of the code regions manually and expressed path constraints in the SMT-LIB [12] format. The handcrafted constraints are solved using the decision procedure for the theory of strings and bit-vectors and compared the result against Z3-str2 solver with native string and integer theory support.

Google Stagefright Vulnerabilities

Stagefright is the name given to a potential exploit that lives fairly deep inside the Android operating system. Mobile security firm Zimperium [9] announced the exploit as part of the BlackHat conference, and believed to be the worst Android vulnerability ever discovered. StageFright is a system service for Android implemented in native C++ to handle multiple media formats. Many integer overflows and underflows leading to code injection attacks, present in the libstagefright media library is the main reason for stagefright vulnerability. All devices running the Android versions Froyo 2.2 to Lollipop 5.1.1 are affected which estimates approximately 95% of all Android devices and cover around 1 billion peoples. Applications using the stagefright library run under the media permission. However, if the attack succeeds, then the attacker can view corresponding files in the media library and control the device through privilege escalation attack.

We briefly explain the exploitation nature of the vulnerability and proceed with the analysis in the below sections. To trigger the vulnerability, an attacker sends a multimedia message (MMS) containing malware to any messenger apps that can process the specific media. Stagefright library is not only used for playing media files but also for generating thumbnails automatically by extracting metadata like length, height, width, frame frequency, channels and other information from video and audio files. Consequently, when users view the thumbnails included in the malicious MMS, this vulnerability would be triggered. The most alarming about it is that the user does not even have to open the message or watch the video to activate the attack. The built-in applications like Hangouts automatically process videos and pictures from MMS messages to have them ready in the phone's gallery app. We chose two vulnerabilities from this package to see if the decision procedure for the theory of bit-vectors and strings can solve path constraints leading to the overflow bug.

CVE-2015-3824: Google Stagefright 'tx3g' MP4 Atom Integer Overflow Remote Code Execution

This vulnerability is associated with the 'tx3g' atom that refers to the text metadata constitutes the first benchmark. An MPEG-4 is composed of several units called atoms or

```
status_t MPEG4Source::parseChunk(off64_t *offset) {
  ...
  uint64_t chunk_size = ntohl(hdr[0]);
  uint32_t chunk_type = ntohl(hdr[1]);
  off64_t data_offset = *offset + 8;
  if (chunk_size == 1) {
    if (mDataSource->readAt(*offset + 8, &chunk_size, 8) < 8) {
      return ERROR_IO;
    }
  chunk_size = ntoh64(chunk_size);
  ...
  case FOURCC('t', 'x', '3', 'g'):
  {
    uint32_t type;
    const void *data;
    size_t size = 0;
    if (!mLastTrack->meta->findData(
            kKeyTextFormatData, &type, &data, &size)) {
      size = 0;
    }
    uint8_t *buffer = new (std::nothrow) uint8_t[size + chunk_size];
    if (buffer == NULL) {
      return ERROR_MALFORMED;
    }
    if (size > 0) {
      memcpy(buffer, data, size);
    }
    ...
}
```

Figure 2.6: Stagefright tx3g MP4 atom integer overflow

boxes. These atoms begin with a header, size, and a box type. The box types contain a four character code such as 'covr', 'esds', 'tx3g' etc,. The Android's media server encounters vulnerabilities while reading those boxes. The buggy part of the code is presented in Figure 2.6. Here, a buffer is created dynamically in which the size computed by adding variables, and the values of the user controlled variables can be selected to cause an integer overflow that further leads to a heap overflow. Analyzing the erroneous code fragment we can see that the size is guaranteed to be greater than 0 when *mLastTrack*→ *meta*→ *findData* returns true. The *new* operator creates a buffer using the sum of size and chunk_size as its parameter. The chunk_size is read from a file, and it is a uint64_t variable. However, the parameter of the new operator is defined as size_t and the size of which varies according to the underlying architecture leading two different scenarios. In a 32 bit architecture, size_t is 32 bit long, and the result get truncated causing an integer overflow whereas in 64-bit architecture, the size_t is 64 bit, and an enormous value of chunk_size can lead to an arithmetic overflow. Thus, the integer overflow leads to the creation of a small sized buffer on the heap which further results in a heap overflow with the memcpy() function call. We abstracted the program state, manually encoded the path constraints into a set of constraints in string plus bit-vector and string plus integer each in their SMT-LIB format and solved it using Z3-str2 in different modes. The string plus bit-vector solver successfully produced a satisfying model while native solver with integer theory failed to solve the constraints.

CVE-2015-3826: Google Stagefright 3GPP metadata buffer overread

The second vulnerability is the "Google Stagefright 3GPP metadata buffer overread" associated 'covr' box type that deals with the album cover artworks. The vulnerable part of the code is presented in Figure 2.7. If the 'chunk_data_size' value is SIZE_MAX, an integer overflow will occur and cause a small buffer to be allocated, and the following call to readAt will overwrite memory locations. We expressed the constraints for the path triggering the vulnerability in SMT-LIB format and solved using Z3-str2. The string plus bit-vector solver produced a model for the input constraints while string-integer solver failed.

```
status_t MPEG4Source::parseChunk(off64_t *offset) {
  ...
  off64_t chunk_data_size = *offset + chunk_size - data_offset;
  ...
  switch(chunk_type) {
  ...
  case FOURCC('c', 'o', 'v', 'r'):
  {
    *offset += chunk_size;
    if (mFileMetaData != NULL) {
      ALOGV("chunk_data_size = %lld and data_offset = %lld",
              chunk_data_size, data_offset);
      sp<ABuffer> buffer = new ABuffer(chunk_data_size + 1);
      if (mDataSource->readAt(data_offset, buffer->data(),
                    chunk_data_size) != (ssize_t)chunk_data_size) {
        return ERROR_IO;
      }
      ...
   ...
}
```

Figure 2.7: Stagefright 3GPP metadata buffer overread

CVE-2009-0585: libsoup Integer Overflow

libsoup is a library that provides HTTP client/server routines for GNOME. Integer overflow in the *soup_base64_encode* function in soup-misc.c in libsoup 2.x.x before 2.2.98 and 2.x before 2.24, allows context-dependent attackers to perform code injection attack via a long string that is converted to a base64 representation. The vulnerable part of the source code is presented in Figure 2.8, where the function *soup_base64_encode* encodes a sequence of binary data into its Base-64 stringified representation and returns the Base-64 encoded string representing text. Heap memory is allocated through g_malloc() using a length

calculated by a user supplied, platform specific value. In the above code integer *len* at the place where the allocation occurs can wrap around if it contains an enormous value. Thus, if a large untrusted input is passed to the function, an insufficient amount of memory is allocated, followed by a heap-based buffer overflow with the Base64 encoded data. We verified the vulnerable part of the code using the decision procedure for the theory of bit-vectors and strings by generating a satisfying model that can trigger the vulnerability while string plus integer solver failed to solve constraints.

```
char *
soup_base64_encode (const char *text, int len)
{
    unsigned char *out;
    int state = 0, outlen,  save = 0;

    out = g_malloc (len * 4 / 3 + 5);
    outlen = soup_base64_encode_close ((const guchar *)text,
                                       len,
                                       FALSE,
                                       out,
                                       &state,
                                       &save);
    out[outlen] = '';
    return (char *) out;
}
```

Figure 2.8: libsoup integer overflow

FreeBSD wpa_supplicant(8) Base64 Integer Overflow

This bug [4] was reported to the FreeBSD, and it affects 7.2 stable releases of the FreeBSD. The *wpa_supplicant* utility is an implementation of the WPA Supplicant component. It

```
unsigned char * base64_encode(const unsigned char *src,
                              size_t len,
                              size_t *out_len){
    unsigned char *out, *pos;
    const unsigned char *end, *in;
    size_t olen;
    int line_len;
    olen = len * 4 / 3 + 4; /* 3-byte blocks to 4-byte */
    olen += olen / 72; /* line feeds */
    olen++; /* nul termination */
    out = os_malloc(olen);
    if (out == NULL)
        return NULL;
    ...
}
```

Figure 2.9: FreeBSD wpa_supplicant(8) Base64 Integer Overflow

implements WPA key negotiation with a WPA authenticator with an authentication server. Figure 2.9 shows is the buggy code as seen in src/contrib/wpa_supplicant/base64.c from FreeBSDs CVS. During the first arithmetic operation, *len* is multiplied with constant values. However, the type of *len* and *olen* is size_t therefore the size varies according to the underlying architecture. The *len* taints the *olen* variable, which is used as the argument of os_malloc() for allocating memory in heap. If *len* is a large value, it leads to an integer overflow and a heap memory corruption in the subsequent call to os_malloc(). We abstracted the program behavior, encoded constraints for the vulnerable path into SMT-LIB and compared the results of Z3-str2 with different modes of theory combination. The decision procedure for the theory of string and bit-vectors solved the constraints generating a satisfying model when the string plus integer solver failed.

CVE-2009-2463: Mozilla Firefox/Thunderbird Base64 integer overflow

This bug [7] was disclosed by Mozilla in 2009. The issue affects the Base64 routines in Mozilla Firefox before the 3.0.12 release. The vulnerability consists of an integer overflow followed by a heap overflow in the PL_Base64Encode function in some 32-bit architectures that help remote attackers to cause a denial of service or code injection attacks. The vulnerable part of the source code is explained in the Figure 2.10. In the PL_Base64Encode function, the variable *destLen* is of type PRUint32 where the value is calculated from an arithmetic expression containing *srclen*. As per the Mozilla documentation, the PRUint32 data type is defined as an unsigned int or an unsigned long depending on the platform. The *srclen* is the length of *src*, a user controlled string. Therefore the value of *dataLen* is tainted by user input and can cause integer arithmetic overflow by providing a large *src*. Also, *destLen* is used as the parameter of PR_MALLOC() for allocating memory in heap space leading to a heap memory corruption. We abstracted the program state and expressed the path constraints in the SMT-LIB format for triggering the heap overflow vulnerabilities and solved it using Z3-str2. The Z3-str2 solver failed to solve in the native mode supporting strings and integers. However, the solver returned a returned a satisfying model in the bit-vector plus string mode.

CVE-2002-0639: Integer and heap overflows in OpenSSH 3.3

Integer overflow in sshd [3] in OpenSSH 2.9.9 through 3.3 allows remote attackers to execute arbitrary code during challenge response authentication when OpenSSH is using SKEY or BSD_AUTH authentication. Figure 2.11 shows the code excerpt from OpenSSH 3.3 leading to a classic integer overflow. In the vulnerable code, the variable *nresp* stores an integer value read from the client using packet_get_int() function, and packet_get_string() method returns a pointer to a buffer that resides on heap containing a string read from the client. The value of *nresp* can be set to an extremely long value to cause integer arithmetic overflow in the argument of xmalloc() function and thereby to allocate zero sized buffer. In such a case, the subsequent loop iterations cause heap buffer to overflow. We encoded the path constraints for triggering heap overflow into the SMT-LIB format and solved it using two different combinations of theories in Z3-str2. The string plus bit-vector solver generated

```
PR_IMPLEMENT(char *)
PL_Base64Encode
(
    const char *src,
    PRUint32    srclen,
    char       *dest
)
{
    if( 0 == srclen )
    {
        srclen = PL_strlen(src);
    }
    if( (char *)0 == dest )
    {
        PRUint32 destlen = ((srclen + 2)/3) * 4;
        dest = (char *)PR_MALLOC(destlen + 1);
        if( (char *)0 == dest )
        {
            return (char *)0;
        }
        dest[ destlen ] = (char)0; /* null terminate */
    }

    encode((const unsigned char *)src, srclen, (unsigned char *)dest);
    return dest;
}
```

Figure 2.10: Mozilla Firefox/Thunderbird Base64 integer overflow

a satisfying model in this benchmark as well while the string plus integer solver failed to solve the constraints.

```
...
nresp = packet_get_int();
if (nresp > 0) {
 response = xmalloc(nresp*sizeof(char*));
 for (i = 0; i < nresp; i++)
  response[i] = packet_get_string(NULL);
}
...
```

Figure 2.11: Integer and heap overflows in OpenSSH 3.3

CVE-2005-0180: Linux Kernel SCSI IOCTL Integer Overflow

The primary reason for the vulnerability is the lack of sufficient sanitization performed on user-controlled integer values before employed as the size argument of a user space to kernel memory copy operation. The vulnerable part of the code is presented in the Figure 2.12. The sg_scsi_ioctl() function contains two signed integers textitin_len and textitout_len, read from user space. In the definition of copy_from_user() function, the third parameter is of type unsigned long. However, the function takes in_len as its third parameter when calling from sg_scsi_ioctl(). Therefore when a negative value in the in_len variable is type cast into unsigned long and result in very large value and cause memory overflow. The decision procedure for the theory of strings and bit-vectors solved the manually generated path constraints expressed in SMT-LIB format and produced a model for triggering the vulnerability. However, the solver with native support for the theory of strings and integers failed to solve the constraints.

2.7.2 Evaluation of Search Space Pruning Technique

The binary search based approach with backtracking functionality helps to prune the search space efficiently when the input contains operations on large values of strings. The technique was initially designed for the decision procedure of bit-vectors and strings and later ported into the base solver for strings and integers because of its superior performance in

```
static int sg_scsi_ioctl(struct file *file, request_queue_t *q, ...){
    char *buffer = NULL;
    int bytes;
    int in_len, out_len;                    /* two integers */
    ...
    if (get_user(in_len, &sic->inlen))  /* read from user space */
            return -EFAULT;
    if (get_user(out_len, &sic->outlen))
            return -EFAULT;
    if (in_len > PAGE_SIZE || out_len > PAGE_SIZE)
            return -EINVAL;
    ...
    bytes = max(in_len, out_len);
    if (bytes) {
            buffer = kmalloc(bytes, q->bounce_gfp | GFP_USER);
            if (!buffer)
                    return -ENOMEM;
            memset(buffer, 0, bytes);
    }
    ...
    if (copy_from_user(buffer, sic->data + cmdlen, in_len))
            goto error;
    ...
}
```

Figure 2.12: Linux kernel SCSI IOCTL integer overflow

solving equations containing large values of strings. We used the string-integer solver for the comparison and performance evaluation of the binary search approach. We performed two set of experiments for this evaluation and discussed in the below subsections.

Table 2.1: Performance of Z3-str2 over string lengths

Benchmark	**Length (l)**	**Result**	**Binary Search (seconds)**	**Naive Search (seconds)**
test1	10	SAT	0.042	0.040
test2	50	SAT	0.045	0.041
test3	100	SAT	0.039	0.051
test4	500	SAT	0.041	0.254
test5	1000	SAT	0.055	0.928
test6	5000	SAT	0.056	46.699
test7	10000	SAT	0.088	Timeout
test8	50000	SAT	0.323	Timeout
test9	100000	SAT	0.842	Timeout
test10	500000	SAT	21.084	Timeout
test11	1000000	SAT	105.636	Timeout

Performance Evaluation of String Lengths in Z3-str2

The first set of experiments is to demonstrate the inefficiency of the naive search in the prior version of Z3-str2. For this purpose, we chose a tiny constraint set containing only the length function. The constraints we chose are of the form $length_{str} = l$, where l varies in the order presented in the Table 2.1. There is a total of 11 test inputs. We used 200 seconds as the timeout per test case. The stable release of the Z3-str2 solver in the naive search mode solved constraints of length up to 5000 and timed out afterward. The solver took more than 12 hours to terminate when the length is the order of 10^5. However, the solver in the binary search mode performed very efficiently and solved the constraints of length up to 1 million. The comparison of the solving time for the two modes in Z3-str2 is shown in Table 2.1 and the cactus plots in Figure 2.13.

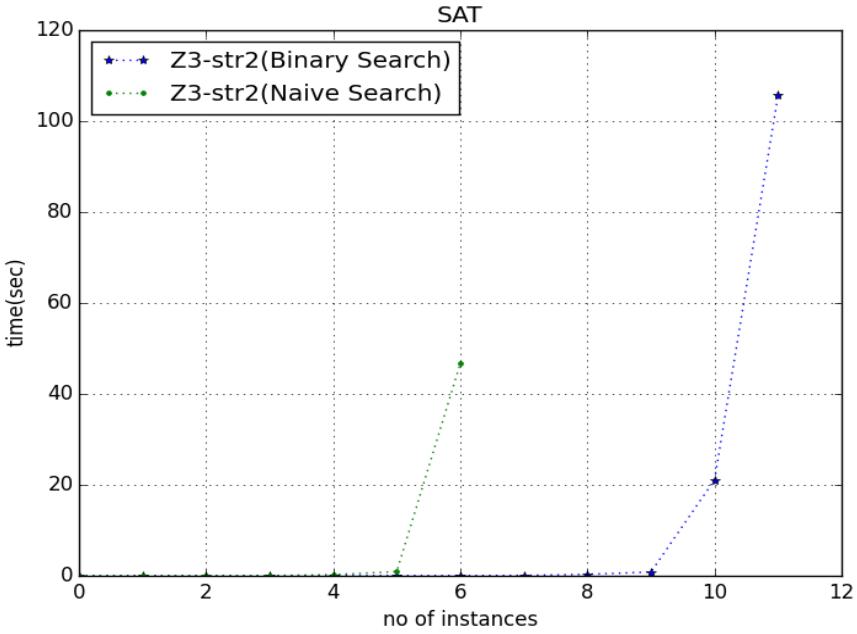

Figure 2.13: Cactus plots for the length test

Evaluation on benchmark suite

To measure the efficacy of our approach we designed a benchmark suite containing 205 test cases, containing handcrafted constraints involving operations on large strings. The benchmark consists of constraints on various string operators supported by Z3-str2 such as *Length, Concat, Indexof, Substring, EndsWith, StartsWith, Replace* etc,. Furthermore, we translated the constraints into the language of CVC4 and compared the performance of the benchmarks against CVC4 of version 1.5. We performed all the experiments on a workstation running Ubuntu 12.04 with an i7-3770 CPU and 8GB of RAM memory. We used 200 seconds as the timeout per benchmark.

The results of the comparison are presented in the Table 2.2, where *Tool reports error* counts the number of inputs on which the solver reports an error. *Crash* instead, refers to run-time errors such as segfaults. The *SAT* and *UNSAT* denotes the number of sat and unsat results respectively. According to the Table 2.2, Z3-str2 with binary search solves

Table 2.2: Performance on benchmark suite

	Z3-str2 (Binary Search)	**Z3-str2 (Naive Search)**	**CVC4**
SAT	169	138	126
UNSAT	34	34	19
UNKNOWN	2	2	0
Timeout	0	31	60
Tool reports error	0	0	0
Crash	0	0	0
Total no. of benchmarks	205	205	205
Total time (sec)	41.697 (**1x**)	9569.639(**229x**)	12014.893(**264x**)

† 'unknown' indicates Z3-str2 detected and avoided overlapping arrangements.

all test instances very quickly comparing to the others. Z3-str2 with the naive search timed out with 200 seconds on 31 test cases where CVC4 timed out in 60 test cases. The Z3-str2 solver detects the overlapping arrangements in both search modes and produces "UNKNOWN" result. Neither of these solvers crashed or reported errors in any of the test instances. The cactus plots for the sat and unsat results are presented in the Figure 2.14 and Figure 2.15 respectively. Comparing the overall time taken by the solvers Z3-str2 with the binary search approach beats the others on the run. The overall time of the binary search based approach is 41.697 seconds for these 205 test cases, where that of naive approach is 9569.639 seconds and that of CVC4 is 12014.893 seconds. The binary search based approach is about 229 times faster than naive approach of Z3-str2 and 288 times faster than CVC4.

2.8 Related Work

Practical methods for solving string equations can be roughly categorized into bounded and unbounded methods. HAMPI [37] is a well-known solver for string constraints over fixed-size string variables. The constraints in HAMPI express the membership in regular

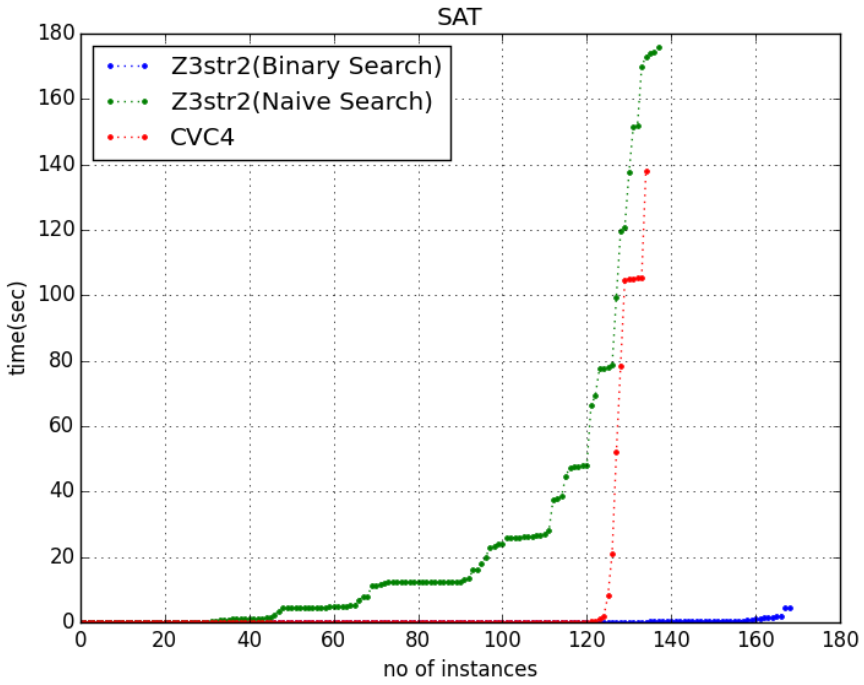

Figure 2.14: Cactus plots for the benchmarks: SAT instances

and fixed-size context-free languages. The main limitation with HAMPI is that, it requires the user to provide an upper bound on string lengths. The solver was initially designed for detection of SQL injection vulnerabilities where input problems are reduced first to bit-vector problems and then solved using STP [27]. Also, HAMPI does not support direct string comparison and other string operations such as indexof, substring, etc,.

Kaluza, the core of a JavaScript symbolic execution framework named of Kudzu [57], is another popular solver that supports both string and non-string operations. This solver extends HAMPI's input language to multiple string variables of bounded length. Kaluza constraints contain word equations over string variables, membership in regular languages, and inequality formulas over string length. However, one major drawback of Kaluza is that it requires the lengths of string variables to be known before being able to encode them and query the underlying SMT solvers. In particular, before solving for string constraints,

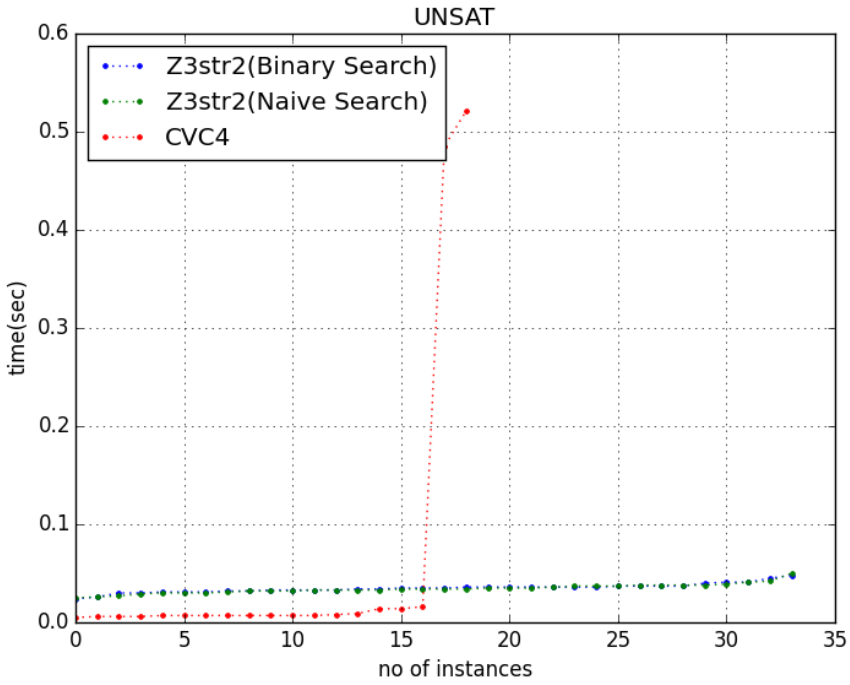

Figure 2.15: Cactus plots for the benchmarks: UNSAT instances

Kaluza finds a set of satisfying solutions for each string length. Then for each possible length, it encodes string variables as an array of bits and then queries the underlying bit-vector solver.

CVC4 [42] is an open-source automatic theorem prover for Satisfiability Modulo Theory (SMT) problems. The string solver in CVC4 uses a set of algebraic techniques for solving constraints over the theory of unbounded strings, natively supporting the length and regular language membership, without reduction to other problems. It uses an off-the-shelf solver for integer linear arithmetic and a string solver for string and regular expression constraints. The string solver contains theory specific derivation rules that assert additional string and RL constraints to the congruence closure module. The combination between the string solver and the arithmetic solver is achieved using Nelson-Oppen rule, by exchanging equalities over shared terms. The Kleene star operator for regular language constraints is

processed by unrolling the operator and makes the solver non-terminating in general over such constraints. Even though the solver is incomplete and non-terminating in general, it is sound and supports a large set of operators that can be used to solve string constraints arising from verification and security applications efficiently. Also, CVC4 has extensive supports SMT-LIB format.

S3 [65] is another word based state of the art solver for unbounded strings based on older version of Z3-str. This solver is incremental and expressive. S3 performs the incremental reduction on string variables using the try-and-backtrack procedure of Z3 core until the variables are bounded by constant strings. S3 reasons the Kleene star and other recursively defined functions by lazily unfolding its semantics in the process of incremental solving. In general, S3 can be viewed as an extension of Z3-str with regular expressions, membership predicates, and some high-level string operations that often work on regular expressions such as search, replaceAll, match, etc,.

Norn [10] is another a decision procedure for word equations over string variables of arbitrary lengths with support for length constraints and regular expressions. Norn first converts the given formula to DNF and recursively splits the equalities and membership constraints. It then extracts the length constraints from the formula and solves it using PRINCESS [15] solver. This decision procedure specifically targets model checking applications and is implemented in a prototype model checker to verify common string manipulating functions such as the Hamming and Levenshtein distances.

PISA [64] is the first solver that provides a path and index sensitive string analysis targeting static analysis of web applications. The verification is conducted by encoding relationships among strings and regular expressions of a program in Monadic Second-Order Logic(M2L). PISA then uses a theorem prover such as MONA to check the satisfiability of the generated constraints. However, in PISA expressiveness of the arithmetic operations are restricted due to the limitations of M2L, so it does not support numeric multiplications and divisions.

Regular languages (or automata), as well as context-free grammars (CFGs), can be used to represent strings and handling regex-related operations. A different approach for solving string constraints with regular expressions is to encode them into automata problems. One

of the major works in this category is Java String Analyzer (JSA) [22], in which static analysis is used to model flow graphs of Java programs to capture dependencies of string variables. Finite automata can be computed from the graph to reflect possible string values. Shannon et al. [62] used finite state machines (FSMs) to model strings. They do not support integer constraints in general even they have ad-hoc rules for integer relations. A primary challenge faced by automata-based approaches is to capture the connections between strings and other domains, e.g., integers. In short, using automata (regular) language representations potentially enables the reasoning of infinite strings and regular expressions. However, most of the existing approaches have difficulties in handling string operations related to integers such as length, substring, indexOf, etc,. JST [29] extends JSA. It asserts length constraints in each automaton and handles numeric constraints after conversion. PISA [64] encodes Java programs into M2L formulas that it discharges to the MONA solver to obtain path- and index-sensitive string approximations. PASS [41, 39] combines automata and parameterized arrays for efficient treatment of unsat cases. Stranger is a powerful extension of string automata with arithmetic automata [66, 68].

2.9 Future Work

The ongoing interaction with bit-vector and string theory is achieved using the binary search based approach as discussed before. Even though such technique gives superior performance in reasoning about string lengths, we still face inefficiency in the value arrangements of strings after we fix the string lengths. So one of our next primary goals is to optimize the performance of the solver by providing more robust techniques to value arrangements, replacing the naive approach implemented in the base solver. Also, we would like to add various APIs to extend our constraint language. We are also planning to integrate the decision procedure in symbolic execution engines like S2E and Triton.

Chapter 3

Automated Exploit Synthesis

In this chapter, we describe our technique for automated exploit generation. There are many protection schemes against control flow hijack attacks on both application and operating system level. However, attackers propose innovative attack models and enumerate different techniques to bypass these defenses. Among them, information leakage vulnerabilities play a significant role in revealing the internals of secured systems, where an attacker can leverage subtle information from program memory to augment the state of the art exploitation techniques. This motivated us to develop a technique is to showcase the power of modern program analysis in automating many aspects of the hacking process, and effectively evading widely deployed protection mechanisms.

Chapter Overview

This chapter is divided into seven sections. In section 1, we provide a brief background of major security vulnerabilities such as buffer overflows and format string vulnerabilities, then discusses the various attack defense methods deployed in modern operating systems against these threats. In section 2, we state the problem we are solving. Section 3 motivates the importance of local and remote exploit synthesis. The detailed design and implementation of our tool is presented in section 4. Section 5 presents our experimental methods and results. Prior attempts and other related work on exploit synthesis are shown

in section 6. Finally, our future work plans are listed in section 7.

3.1 Background

This section gives a brief background about various security threats such as buffer overflows, format string vulnerabilities and number of attack resistance techniques against these vulnerabilities.

3.1.1 Buffer Overflows

A buffer overflow condition exists when an application tries to put more data into a buffer than it can hold. In such cases, the data overflows into the nearby memory regions inside memory. This is a serious security vulnerability as an attacker can use malicious data to overflow the buffer and thereby hijack the control flow of the program. Buffer overflow vulnerabilities have been exploited for over 20 years and are the primary reason for most of the internet worms. These vulnerabilities only exist when developing applications using languages that do not enforce run-time bound checking such as C/C++. In other words, these vulnerabilities occur by the memory write operations on a buffer without sufficiently checking bound of a user-supplied input.

3.1.2 Format String Vulnerabilities

In format string attack, an attacker uses string formatting features of certain library functions to access the memory space of applications. Thus, format string attacks facilitates information retrieval from the process memory and helps to bypass attack defenses when used with other attacks based overflow related vulnerabilities. The format string attack occurred when the user submitted data be used as a command in library functions. An attacker can use this vulnerability, to execute arbitrary code on the machine, read values from the stack or cause the application to crash.

3.1.3 Attack defense Techniques

In this subsection, we discuss the common attack defenses employed in both operating system and application level.

Address Space Layout Randomisation (ASLR)

ASLR[51][13] aims to introduce a certain degree of randomness into the addresses used by a program at run-time. All modern operating systems adopt this countermeasure to prevent an attacker from predicting the address space of an application. This solution causes certain parts of a process's virtual address space to become different for each invocation of the process, with the effect that the associated memory addresses are not known a priori from the attackers. To successfully launch the attack, the attacker needs to retrieve certain addresses precisely from the process memory. Thus, ASLR relies on the low probability that an attacker has in order to guess where each area is located. The idea of address space randomization become stronger if some mechanism of entropy is present in the random offsets. If we model the address that has to be guessed as a random variable, we can define its entropy as a measure of the uncertainty associated with the address, and it increases by either advancing the amount of virtual memory area space or reducing the period in which the randomization occurs.

Data Execution Prevention (DEP)

In this approach, which is typically referred as NoExec, Data Execution Prevention (DEP) or $W \oplus X$, helps to prevent the exploits from succeeding by marking certain pages of the applications address space as non-executable. The general concept is to make the data segment of the applications address space as non-executable, making it inaccessible for attackers to execute the injected code.

Hardening the Binary

Compilers for a number of operating systems now include techniques that aim to prevent stack overflows being used to create exploits. Stack canaries are the most common technique to harden the stack along with other run-time and compile-time checks. A stack canary is a random value kept in the stack frame below the stored instruction and base pointers. These canaries cannot prevent buffer overflows, but they can detect them retroactively and terminate the program before an attacker influence the control flow. The Canaries are inserted into the stack just after the function prologue and are verified just before the function epilogue to see if the value has been modified by any user's input. If there is any modification in the canary value is detected, the application is aborted.

RELRO is a generic mitigation technique used to harden the data sections of an executable linkage format(ELF) binary. It supports two different modes of operations named "Full RELRO" and "Partial RELRO. When RELRO is enabled the ELF sections are reordered, and the internal data sections (.got, .dtors) precede the data sections (.data and .bss) of the program. The full mode supports all features of the partial mode and also makes the entire global offset table (GOT) as read-only, where the GOT is used for dynamically linking the executable.

Position-independent Executable (PIE)

Position-independent executable(PIE) is a specially compiled and linked executable that get loaded into random locations in memory on its execution. In other words, the machine codes are placed anywhere in the primary memory and get executed properly regardless of the absolute address. PIE is not very efficient unless ASLR is enabled. Randomizing memory allocation locations makes memory addresses harder to predict for an attacker who is attempting a memory corruption exploit. This PIE along with ASLR provides better protection against code reuse attacks.

3.2 Problem Statement

Given an application binary containing buffer overflow and information leakage vulnerability design an automated tool for control flow hijack attacks that bypass various attack defenses in place.

3.3 Motivation

Software bugs remain as an unavoidable factor in almost all software products. Bugs that result in memory corruption are very common security flaws in systems developed using unsafe programming languages like C/C++. However, these bugs become serious security vulnerabilities when an attacker leverage it to trigger the execution of malicious code. In any software systems, memory related bugs are very critical, and the one that can be exploited by attackers are typically the most serious among them. However, to decide whether the bug is useful for an attacker for malicious purposes or not quite easy. For instance currently Ubuntu bug repository contains more than hundred thousands of open bugs, finding critical security bugs among them is a tedious task and remains as an open challenge.

Programming languages like C/C++ rely on data integrity with the program and enhances the programmer's control and the effectiveness of the resulting programs. Software bugs that result in memory corruption are very common security flaws, in systems developed using unsafe programming languages such as C and C++. For instance, once a variable is allocated in memory, there are no built-in safeguards to ensure the contents of the variable always fit into the allocated memory space and lead to serious application errors or crashes if the memory overflows. The buffer is a continuous memory location that holds data, and it will cause to overflow into next memory locations when the size of the supplied data is more than the allocated size. An attacker can exploit bugs in memory operations memory allocation, de-allocation, pointer assignment, format strings, and call to library routines so that he can hijack the control flow. The major buffer overflow vulnerabilities consist of stack overflow, heap overflow, and integer overflow.

Buffer overflow vulnerabilities are common when developing software using unsafe languages such as C/C++, which do not enforce run-time bounds checking. These two languages account for more software than any other programming languages that exacerbate the security of related software systems. In the early days of stack buffer overflows, it was common for an attacker to include malicious code as part of the payload to overflow the buffer. As a result of code injection, an attacker could simply set the return address to a known location on the stack and execute the instructions that were provided in the buffer to cause control flow hijack. Since the first reported buffer overflow attack, the Morris worm in 1988 [50], system designers have been developing protection mechanisms to eradicate them. Most of these protections mechanisms involve modifying operating system parameters and adding additional compile time checks.

Even though there are many attack defenses and these safeguards have raised the bar significantly, the attackers still continue finding creative ways to defeat them. Reactive protection mechanisms cannot prevent human error. Thus, the solution may be better design and testing of software or the use of languages that enforce run-time bounds checking. This philosophy is credible but also expensive and is unlikely to be done at the cost of performance. Another unfortunate fact is that current state of the art formal verification and automated testing techniques can not fully guarantee a bug proof software system. However, various researches are going on to make dynamic and symbolic program analysis techniques scalable to a wide category of software and ensure security by automated testing.

Previous researches in this area revealed that thinking in the way of an attacker to generate an exploit for a bug would be an efficient way to provide a defense mechanism against such vulnerabilities. However, this is not always feasible, as it can be a time-consuming activity and requires low-level knowledge of file formats, assembly code, operating system internals, and CPU architecture. However, various researchers proposed automated exploit generation systems relying on standard code injection attacks. The unfortunate fact is that current research in automated exploit generation does not fully address these attack defenses because of the complexity and significant manual effort associated with systematic program analysis to mitigate the defense systems. Motivated by this, we proposed an automated remote exploit generation system that leverages the information leakage present

in a software system to generate working exploits that bypass the various attack defenses deployed in the application and operating system. Comparing to the previous works in this area our exploit generation approach is quite practical and easily extended to various other architectures and platforms. Our binary instrumentation approach for vulnerability detection is lightweight and significantly reduces the performance overhead within the automated exploitation framework.

3.4 Design and Implementation

Randomization-based defenses provide reasonable entropy against control flow hijack attacks that measure the randomness in the number of bits of the underlying architecture. However with the presence of information leakage this entropy can be reduced significantly by leaking addresses from process memory to bypass the attack defenses. Figure 3.1 represents the design overview for our technique that consist of dynamic binary instrumentation, taint analysis, vulnerability detection (information leakage and buffer overflow), retrieval of target memory addresses and finally the exploit synthesis.

Our significant contribution is the design and implementation of an end to end automated system considering all attack defenses in place. We wrote code for dynamic binary analysis using the Pin instrumentation framework that helped us to write a taint analysis routine from the scratch. We used return-to-libc attack model for control flow hijack attacks and generated the payload by automatically retrieving subtle memory details using information leakage attacks. Simply, given an application binary and the remote server running the application, our tool automatically generates payload using binary instrumentation and information leakage if a potential vulnerability exists.

3.4.1 Vulnerability Detection

As a first step, we detect various conditions leading information leakage vulnerability. Format string vulnerabilities play a significant role in the process of leaking address space of target process that can be used to mitigate the existing defenses. Further, we detect

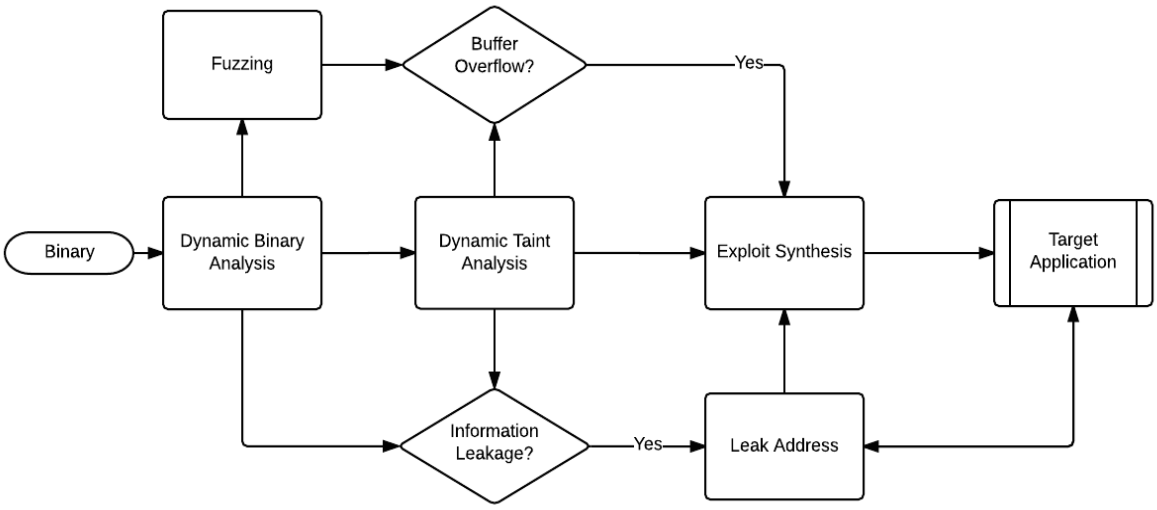

Figure 3.1: Design overview of automated exploit generator

buffer overflow vulnerabilities and synthesize attack vector to execute reverse shell payload. These vulnerabilities are detected using dynamic binary analysis featuring binary instrumentation. A dynamic taint analysis on target application helped to track the vulnerable input to generate attack vector. Format string vulnerability is detected followed by binary instrumentation stage by analyzing the presence of tainted source and format string in the stack frame created by variadic functions such as fprint, sprintf, snprintf, and vsnprintf Similarly, buffer overflow vulnerabilities are detected by fuzzing techniques followed by dynamic taint analysis.

Dynamic Binary Analysis

The two principal analysis for the vulnerability detection are the static and dynamic analysis. However, both these approaches have their advantages and disadvantages. If we use dynamic analysis, we can not cover all the code but we will be more reliable. If we

use static analysis, we can cover the code, but we can not get the context information at runtime. We avoided static analysis technique and chose more promising dynamic analysis for vulnerability detection for a variety of reasons. We found that the dynamic analysis is more attractive and promising in the precise analysis and retrieval of run-time information of program state. Furthermore, it reduced the false positives and allowed us to reason about actual executions leading to more accurate security analysis. Thus, the outcome of dynamic binary analysis was two-fold, it helped us to gather sufficient information on data flow and subtle details of memory content, which later we used in the detection of the format string and buffer overflow vulnerabilities and finally in the exploit synthesis. For these, we wrote dynamic binary analysis routines called pintools, using the Pin binary instrumentation framework.

Dynamic Binary Instrumentation

Security researchers in both academy and industry use numerous techniques for gathering the subtle run-time information from an application binary, but each one adds some complexity and performance issues. The very common techniques involve (1) *execution trace listing*, where traces are logged into a database at run-time for the execution of a list of the instructions, modification of register values and memory locations. (2) *emulation* where the application executes inside a special environment that provide programmatic control over the execution of the emulated system to analyze the required data-flow. (3) *binary instrumentation* a technique whereby extra code is injected into the normal execution flow of a binary to collect run-time information. This method widely used in areas for performance profiling, error detection, capture, and replay. The injected code handles the arbitrary analysis of the target program. This method of gathering run-time information is provided by a number of different frameworks including Pin [43], Valgrind[48] and DynamoRIO[16].

Dynamic binary instrumentation relies on instrumenting code just before it run. We chose Pin[43] instrumentation framework for the binary analysis using its JIT(Just In Time) mode of instrumentation. In this mode Pin creates a modified copy of the application on the fly to inject the code for run-time analysis and the original code never executes. Also, Pin

binary instrumentation engine support programmable instrumentation by providing a rich set of APIs in C/C++ to write instrumentation tools called PinTools. Pin is more efficient and robust in performance, and it applies compiler optimizations on instrumentation code. The Pin framework also supports a probe mode in which instrumentation code is added directly to the original program along with inserting jumps to instrumentation code.

Dynamic Taint Analysis

Taint analysis is an iterative process whereby an initial set of memory locations and registers are marked as tainted, and then at each subsequent instruction element is added or removed from the set, depending on the semantics of the instruction being processed. The concept is defined iteratively as marking a location as tainted if it is directly derived from user input or another tainted location. This is a well-known technique adopted by various researchers in both industry and academia to reason about the control flow of user inputs in the context of precise security analysis. However, we avoided using any existing framework considering the inherent complexity associated with such frameworks, instead wrote our analysis routine together with binary instrumentation. We used taint analysis to reason about a set of memory locations and registers that are controlled by user inputs by monitoring the user input by controlling the read() system call when the application executes.

Taint analysis mainly consists of taint tracking and taint propagation according to our predefined taint policies. The purpose of dynamic taint analysis is to track information flow between sources and sinks. Any program value whose computation depends on data derived from a tainted source is treated as tainted, and all other value is considered untainted. We used shadow memory to store all the memory locations and registers tainted by user input. Taint introduction rules are implemented such that all variables, memory cells, and registers are initialized as untainted. Then all the memory locations corresponding to the input source are added to shadow memory, and all the destination registers affected by the source of entry are also marked tainted. Taint propagation rules specify the status of data derived from tainted or untainted operands. For each data movement and arithmetic instructions in the application execution path, shadow memory is modified by the addition

of new memory locations or the removal of existing memory locations depending whether data is tainted or untainted respectively.

3.4.2 Exploitation Techniques

The introduction and deployment of defense mechanisms against control flow hijacks motivated attackers to propose more sophisticated exploitation techniques. Analyzing the history of such techniques we can see that *code reuse attacks* are first introduced as response to protection against *Data Execution Prevention (DEP)* that resist *code injection* by enabling the memory region either writable or executable. These defenses give birth to code reuse exploitation techniques, unlike code injection attacks that redirect the control flow of the program to code written by the attacker, code reuse attacks redirect the control flow executable section of code chosen by the attacker. The most commonly discussed types of code reuse attacks are return-into-libc [47] attacks and return-oriented programming (ROP) [55] attacks. These code reuse attacks are categorized based on the granularity of the reused code fragments called *gadgets*. In return-into-libc attacks, the gadgets are entire functions whereas in ROP attacks a gadget is a series of machine instructions terminating in a *ret* instructions. ASLR substantially provided reasonable defenses against code reuse attacks, but it was defeated using gadgets from non-randomized segments and surgically returning into libc by overwriting the global offset table (GOT) [56]. However with the deployment of position-independent executable(PIE) and RELRO, a generic mitigation technique to harden the data sections of an ELF binary, it became difficult to break ASLR.

Exploitation technique consists of two stages (1) information leakage attack and (2) buffer overflow attack. In the first phase, we took advantage of the format string vulnerability present in the application to leak address space of the target process. In the context of format string vulnerability, we read the contents of stack memory from target process and use the precise information to synthesize buffer overflow exploit. In the second stage, we leverage the information gathered from target process's memory to generate a buffer overflow exploit that bypasses all the defenses in the target machine and spawn a reverse shell back to attacker machine.

Information Leakage Attack

One of the significant parts of the technique is the process of leaking memory addresses from the process memory of a remote server for defense mitigation. Attackers used to pop a sequence of data from the stack by passing a sequence of the format string, in this way they reason about the process memory for further levels of attacks. However, these approaches introduce some amount of inaccuracy in later stages of exploitation if not properly executed. Also, the size constraints commonly associated with the format string source also limit the capabilities of an attacker. So we avoid such blind data retrieval from the stack and use direct parameter access technique, a way to directly access a stack parameter from within the format string to point to the specific addresses in the stack. Thus, we calculate the relative offsets to the specific memory location containing stack canary, libc function and buffer address using the dynamic binary analysis. During the binary instrumentation stage, we extracted precise information regarding the stack frames and buffer at run-time from the binary. Once we detected format string vulnerability in our analysis stage, we calculated the relative offset to stack canary address, buffer controlled by user input and the memory pointing to the libc function in the stack. These details along with stack frames are used to mitigate the existing defenses and synthesize exploits in the later stages.

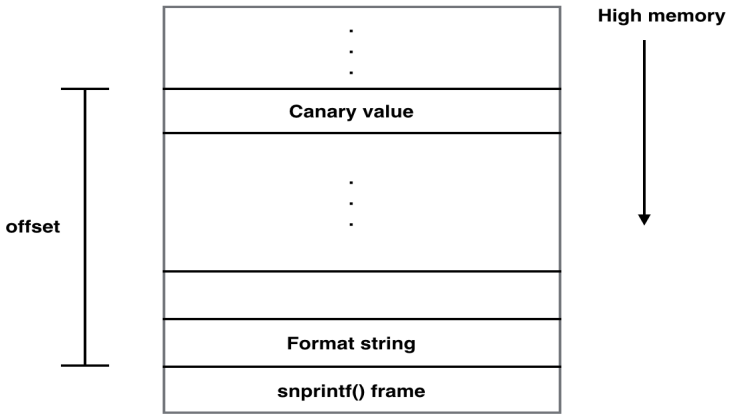

Figure 3.2: Leaking canary value

For instance, the Figure 3.2 depicts the stack canary in memory relative to the format string parameter. The value of *offset* is calculated using the information obtained from the binary run-time analysis. The corresponding format string attack vector to leak canary value for remote server's process memory is *"%offset$x"*. Our next step is to calculate the address of the *libc* function and a tainted buffer using direct parameter format string attack. When the system executes the *main* function the return address of *__libc_start_main* is pushed to the stack. To leak *libc* function address, we measured the offset to the stack address containing this value. Using these addresses retrieved from the memory, and the information read from *glibc* binary, we calculated the base address of *libc* functions. Similarly, we leaked addresses to bypass all possible defenses we introduced earlier in this paper. We also found that without the presence of format string vulnerability, we can use some specific system calls (e.g., *write()*) present in application to leak *libc* function addresses from process memory. We developed automated exploits using this technique without the presence of format string vulnerability too and bypassed ASLR, DEP, and RELRO.

3.4.3 Control Flow Hijack Attack

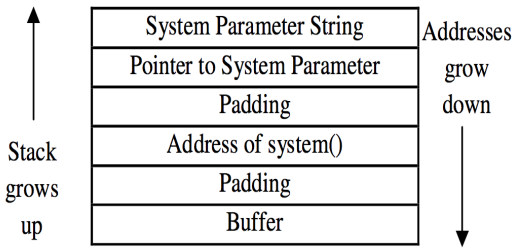

Figure 3.3: Stack layout of a return-to-libc attack

Once gathered subtle information regarding stack frame, buffer and leaked addresses to bypass defense schemes, we chose a minimal but efficient attack model to spawn a reverse shell from the target machine. We chose return-to-libc attack model for control flow hijack and leaked all the required parameters from the target memory space using

format string attack. Return-to-libc is a code reuse attack that mitigates $W \oplus X$ (aka DEP) by overwriting the return address of buffer with the *system()* address in *libc* function and passing the reverse shell payload as the argument. In this way, we mitigated defenses such as ASLR, DEP, Stack Canary, PIE, and RELRO. Figure 3.3 represent the typical stack layout of a return-to-libc attack. In the next step, we designed a more complex attack model to handle the exploit scenario when the format string vulnerability is not present. This attack model is based on return-oriented programming (ROP) using the gadgets collected from binary and *libc* functions, and thus we mitigated few defenses such as ASLR, DEP, and RELRO.

3.5 Experimental Methodology and Evaluation

In this section, we describe the experimental setup and the remote exploitation attack we performed.

3.5.1 Experimental Setup

We set up a virtual test environment using VirtualBox on a host running Mac OSX with a 2GHz Intel Core i7 CPU and 8GB RAM. The virtual network consists of a target machine running Ubuntu 13.10 with kernel 3.11.0 and an attacker machine running Ubuntu 12.04 with kernel 3.5.0. We allocated 2GB memory per each virtual machines, and VirtualBox networking adaptors are configured to create a software-based network that is visible to the selected virtual machines, but not to applications running on the host or to the outside world. For our experiment, we used a vulnerable echo server that forks a child process for each incoming connection, compiled by GCC 4.6.3. The pre-forking concurrent server operates by pooling some listening processes at startup that offers superior performance to other concurrent server designs e.g. handling requests iteratively, or spawning a child process for each new client request. The benefits make it a very popular method of handling requests for HTTP, IMAP, and SMTP servers.

3.5.2 Experimental Results

We used an echo server application for evaluating the exploit synthesis. Fig 3.4 represents the vulnerable part of source code that contain format string and buffer overflow vulnerability. The function *vulnerable()* is prone to both format string and buffer overflow vulnerabilities, where the format string vulnerability is associated with *snprintf()* and buffer overflow vulnerability with *memcpy()*. However, our vulnerability detection technique does not perform analysis on the source code level. The vulnerability detection and the exploit construction is on purely based on the analysis of application binary compiled with GCC 4.6.3 with defenses such as position independent executable (PIE), stack canary, full RELRO and non-executable stack. *ie.* The source code is compiled with the following command *gcc -pie -fpie -z relro -z now -o server server.c*. The server is setuid enabled and runs with has the privilege of super user, runs on port 4444 and executes with root privilege in Ubuntu 13.10. In this machine, ASLR is enabled and the application runs with all available protections and client machine can interact with this server using telnet or netcat.

The attacker machine is a linux machine running Ubuntu 12.04, where we perform vulnerability detection, binary analysis and launch attack to the target server. We performed dynamic binary instrumentation and tainted analysis on the target application on the attacker machine. Also, we extracted the width of all stack frames setup during the binary analysis stage. This first stage is followed by fuzzing techniques to detect buffer overflow vulnerability. Once the vulnerabilities are detected using the binary analysis guided by fuzzing, the *offset* to the various addresses of interest is calculated. These offsets form initial format string attack vector, for leaking sufficient memory information from the target process. In short, we detected vulnerabilities using fuzzing guided by dynamic binary instrumentation and extracted subtle information regarding the binary for later stages of exploitation.

The next stages of exploitation were to leak addresses from the memory of remote server process. The attacker connects to the remote server using netcat and sends format string attack vectors to collect sufficient memory information, required for breaking the defenses against control flow hijack attacks. Thus, using the format string attack, we collect the

address of *buffer*, *__libc_start_main*, and *stack canary value*. With this retrieved information, we perform a return-to-libc attack that opens a reverse connection from remote machine binding a root shell back to port number 7777 in attacker machine. The whole process starting from the binary instrumentation, vulnerability detection, format string attack and final reverse shell exploitation took in 3.6 seconds. Figure 3.5 represents the payload synthesized for spawning a reverse root shell.

```
...
void vulnerable(char *buffer,int sock){
    char local_buffer[50];
    char msg[1024];
    snprintf(msg, sizeof msg,buffer);
    write(sock,msg,strlen(msg));
    memcpy(local_buffer,buffer,strlen(buffer)+30);
    return;
}

void PerformOperations(int sock){
    int n;
    char buffer[256];
    bzero(buffer,256);
    n=read(sock,buffer,255);
    if(n<0){
        error("ERROR reading from socket");
    }
    buffer[256]='\0';
    vulnerable(buffer,sock);
}
...
```

Figure 3.4: Format string and buffer overflow vulnerability

```
00000000  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
*
00000030  90 90 00 7f f3 64 41 41  41 41 41 41 41 41 41 41  |.....dAAAAAAAAAA|
00000040  41 41 60 22 59 b7 41 41  41 41 7a 39 c5 bf 6e 63  |AA`"Y.AAAAz9..nc|
00000050  61 74 20 2d 63 20 2f 62  69 6e 2f 73 68 20 31 39  |at -c /bin/sh 19|
00000060  32 2e 31 36 38 2e 35 36  2e 31 30 20 37 37 37 37  |2.168.56.10 7777|
00000070  3b                                                |;|
00000071
```

Figure 3.5: Reverse shell payload

The absence of format string vulnerability limits an attacker capabilities, and so we designed a more subtle and complex attack model that leverages the presence of certain system calls such as *read()* in the application to leak the *libc* function address from remote server's process memory by using buffer overflow vulnerability. We managed to retrieve some *libc* function address through buffer overflow attack by overwriting the return address with the procedure linkage table (PLT) address of write(), which is common function present in many low-level system programs. When gathered one *libc* function, we could calculate the run-time *libc* base address and there by any other *libc* function addresses in the target process. The second stage of exploitation was to make a page containing data section writable by using *mprotect()* function calculated using the *libc* base address and calling *write()* system call to write payload into the memory. In the third stage, we sent *system()* to the previously chained call followed by payload string. In this way, we synthesized ROP exploit by making use of *libc* functions and a minimal number of gadgets collected from application binary. This remote exploitation without relying on format string took 5 seconds to get a reverse shell by breaking ASLR, DEP, and Full RELRO defenses.

3.6 Related Work

Shacham et al. [61] studied the effectiveness of address-space randomization and found that its efficacy on a 32-bit architectures is restricted by the number of bits available for address randomization. Also, they demonstrated a brute-force attack to convert any common buffer overflow vulnerabilities into an exploit that works against systems protected by address

space randomization. Their analysis suggested that run-time address space randomization is far less efficient on 32-bit architectures than commonly believed. Also, compile time randomization is more powerful than run-time randomization because the address space can be randomized at a much finer granularity at compile time than run time.

Brumley et al. in their paper Automatic Exploit Generation (AEG) [11] introduced the first fully automatic end-to-end approach for exploit generation. Also, they showed that how exploit generation for control-flow hijack attacks can be represented as a formal verification problem. In AEG, the analysis was solely based solely on source code using a technique called preconditioned symbolic execution to narrow the search space. Their significant contributions include the introduction of preconditioned symbolic execution, a path prioritization algorithm for vulnerability detection and the generation of a working exploit for the discovered vulnerable bug. However, the exploitation technique did not consider the protection mechanisms deployed in the application and operating system.

Schacham et al. [55] introduced the technique of return-oriented programming, in which an attacker induce arbitrary behavior in a program by diverting the control-flow without injecting any executable code. In return-oriented programming attacker chains together gadgets, short instruction sequences end with *return* instruction already present in a program's address space. This technique could be seen as a generalization of traditional *return-into-libc* attacks, but more generic and powerful. They constructed a Turing-complete set of building blocks using gadgets collected from standard C library to demonstrate the wide applicability of the technique. Giampaolo et al[56] presented a new attack against programs vulnerable to stack-based buffer overflows that bypassed two of the most widely adopted protection techniques, namely write xor execute only ($W \oplus X$) and Address Space Layout Randomization (ASLR). In their attack, they extracted the base address of library functions from the address space of the vulnerable process and used this information to mount a return-to-libc attack. However, these attacks are ineffective in the presence of position-independent executable (PIE) and stack canary.

Brumley et al. presented MAYHEM [20], a new system for automatically finding exploitable bugs in binary programs, which was a logical extension of author's previous work Automatic Exploit Generation to binary code. MAYHEM worked on binary code without debugging information and accompanied by a working shell exploit. This work introduced

hybrid symbolic execution for combining on-line and offline execution thus maximizing the benefits of both techniques along with index-based memory modeling that allows to efficiently reason about symbolic memory at the binary level. However, this work did not address the various defense scheme present in the application and operating system. Zeldovich et al. [63] in their work performed a systematic analysis of defenses against Return-Oriented Programming and code reuse attacks by building a formal model of attacks, their requirements, defenses and their assumptions. They also used SAT solver to perform scenario analysis on their model by analyzing the defense configurations of a real world system by reasoning the hypothetical defense bypasses.

3.7 Future Work

As future works, we would like to evaluate the automated exploit generation technique on significant real world application. Also, we are interested in exploring other ways of leaking information from process memory in the absence of format string vulnerabilities. Even though, we developed a prototype to achieve this in the presence of certain system calls and successfully bypassed ASLR, DEP, and RELRO, bypassing stack canary, and PIE remains as future work. Currently, we are using fuzzing techniques along with dynamic binary analysis to reveal the vulnerabilities. Also, we intend to integrate our string solver to symbolic execution engines by reasoning about various paths in execution tree for bug detection. We would like to augment the efficiency and reduce the complexity in exploit synthesis using SMT-based ROP compiler. We would like to reason about the application of string based solvers in the context of exploit synthesis and integrate it into an end-to-end testing system. Finally, we would like to suggest more sophisticated attack resistance that incorporates techniques from cryptography and information flow analysis against control flow hijack attacks.

Chapter 4

Conclusion

This thesis presents a decision procedure for the theory of bit-vectors and strings, as an efficient solver for the detection of security vulnerabilities in symbolic execution based analysis tools for programs written in C/C++. We designed a new search space pruning technique based on the binary search to solve constraints involving large string values. Also, we replaced existing linear search based length consistency check between the integer and string theory of Z3-str2 with binary search based heuristics, which enabled it to solve constraints containing large string values in the integer mode as well.

We built the bit-vector theory support atop of Z3-str2 string solver and evaluated the decision procedure on a set of real security vulnerabilities collected from Common Vulnerabilities and Exposures repository. Also, we evaluated the binary search based heuristic on a benchmark suite containing 205 test instances, that contains string operations on the large value of strings. We compared the performance of the technique with the prior version of the solver and CVC4. The evaluation revealed that the binary search based approach is about 229 times faster than naive approach implemented in the prior version of the Z3-str2 solver, and 288 times faster than CVC4.

The final part of the thesis include the design and implementation of an automated exploit synthesis system, for the detection of buffer overflow and information leakage vulnerabilities in low-level system codes, and thereby performing control flow hijack attacks that bypass all known attack defenses in Linux based machines.

References

[1] Basic integer overflows. http://phrack.org/issues/60/10.html. Accessed: 2015-09-10.

[2] Common vulnerabilities and exposure. https://cve.mitre.org/. Accessed: 2015-09-10.

[3] Integer overflow in openssh. http://www.cvedetails.com/cve/CVE-2002-0639. Accessed: 2015-09-10.

[4] Integer overflow in wpa_supplicant(8) base64 encoder. https://bugs.freebsd.org/bugzilla/show_bug.cgi?id=137484. Accessed: 2015-09-10.

[5] Linux kernel scsi ioctl integer overflow vulnerability. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-0180. Accessed: 2015-09-10.

[6] Mojesz presburger. https://en.wikipedia.org/wiki/Moj%C5%BCesz_Presburger. Accessed: 2015-09-10.

[7] Mozilla firefox/thunderbird base64 integer overflows. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-2463. Accessed: 2015-09-10.

[8] Satisfiability modulo theory. https://en.wikipedia.org/wiki/Satisfiability_modulo_theories. Accessed: 2015-09-10.

[9] Zimperium: Enterprise mobile security. https://www.zimperium.com/. Accessed: 2015-09-10.

[10] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukás Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. String constraints for verification. In *Proceedings of the 26th International Conference on Computer Aided Verification*, CAV'14, pages 150–166, 2014.

[11] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. Aeg: Automatic exploit generation. In *Network and Distributed System Security Symposium*, February 2011.

[12] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa, 2015. Available at `www.SMT-LIB.org`.

[13] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a board range of memory error exploits. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 8–8, Berkeley, CA, USA, 2003. USENIX Association.

[14] Nikolaj Bjørner, Vijay Ganesh, Raphaël Michel, and Margus Veanes. An smt-lib format for sequences and regular expressions. In *SMT workshop 2012*, 2012.

[15] Angelo Brillout, Daniel Kroening, Philipp Rümmer, and Thomas Wahl. An interpolating sequent calculus for quantifier-free presburger arithmetic. *J. Autom. Reason.*, 47(4):341–367, December 2011.

[16] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society.

[17] J.Richard Büchi and Steven Senger. Definability in the existential theory of concatenation and undecidable extensions of this theory. In *The Collected Works of J. Richard Büchi*, pages 671–683. Springer New York, 1990.

[18] C. Cadar, V. Ganesh, P.M. Pawlowski, D.L. Dill, and D.R. Engler. EXE: automatically generating inputs of death. In *ACM Conference on Computer and Communications Security*, pages 322–335, 2006.

[19] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[20] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *IEEE Symposium on Security and Privacy*, pages 380–394. IEEE Computer Society, 2012.

[21] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 265–278, New York, NY, USA, 2011. ACM.

[22] Aske Simon Christensen, Anders Mller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *In Proc. 10th International Static Analysis Symposium, SAS 03, volume 2694 of LNCS*, pages 1–18. Springer-Verlag, 2003.

[23] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 196–206, New York, NY, USA, 2007. ACM.

[24] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08, pages 337–340, 2008.

[25] M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *ISSTA*, pages 151–162, 2007.

[26] Hiroaki Etoh and Kunikazu Yoda. Protecting from stack-smashing attacks. Published on World-Wide Web at URL http://www.trl.ibm.com/projects/security/ssp/main.html, 2000.

[27] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification*, CAV'07, pages 519–531, 2007.

[28] Vijay Ganesh, Mia Minnes, Armando Solar-Lezama, and Martin Rinard. Word equations with length constraints: what's decidable? In *HVC'12*, 2012.

[29] Indradeep Ghosh, Nastaran Shafiei, Guodong Li, and Wei-Fan Chiang. JST: An Automatic Test Generation Tool for Industrial Java Applications with Strings. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 992–1001, 2013.

[30] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[31] Pieter Hooimeijer and Westley Weimer. A decision procedure for subset constraints over regular languages. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 188–198, 2009.

[32] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to automata theory, languages, and computation*. Pearson/Addison Wesley, 2007.

[33] Lucian Ilie and Wojciech Plandowski. Two-variable word equations. *ITA*, 34(6):467–501, 2000.

[34] Artur Jeż. Recompression: Word equations and beyond. In *Developments in Language Theory*, Lecture Notes in Computer Science, pages 12–26. 2013.

[35] Juhani Karhumäki, Filippo Mignosi, and Wojciech Plandowski. The expressibility of languages and relations by word equations. *J. ACM*, 47(3):483–505, May 2000.

[36] Chi keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa, and Reddi Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *In Programming Language Design and Implementation*, pages 190–200. ACM Press, 2005.

[37] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. Hampi: A solver for string constraints. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 105–116, 2009.

[38] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[39] Guodong Li, Esben Andreasen, and Indradeep Ghosh. SymJS: Automatic symbolic testing of javascript web applications. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 449–459, 2014.

[40] Guodong Li, Esben Andreasen, and Indradeep Ghosh. Symjs: Automatic symbolic testing of javascript web applications. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 449–459, New York, NY, USA, 2014. ACM.

[41] Guodong Li and Indradeep Ghosh. PASS: String solving with parameterized array and interval automaton. In *9th International Haifa Verification Conference*, HVC '13, pages 15–31. 2013.

[42] Tianyi Liang, Andrew Reynolds, Cesare Tinelli, Clark Barrett, and Morgan Deters. A dpll(t) theory solver for a theory of strings and regular expressions. In *Proceedings of the 26th International Conference on Computer Aided Verification*, CAV'14, pages 646–662. 2014.

[43] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of*

the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[44] G.S. Makanin. The problem of solvability of equations in a free semigroup. *Math. Sbornik*, 103:147–236, 1977. English transl. in Math USSR Sbornik 32 (1977).

[45] Yu. Matiyasevich. Word equations, fibonacci numbers, and hilbert's tenth problem. In *Workshop on Fibonacci Words*, 2007.

[46] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, April 1980.

[47] Nergal. The advanced return-into-lib(c) exploits: PaX case study. *Phrack*, 11(58), Dec 2001.

[48] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[49] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), November 1996.

[50] Hilarie Orman. The morris worm: A fifteen-year perspective. *IEEE Security and Privacy*, 1(5):35–43, September 2003.

[51] PaX Team. Pax address space layout randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt. Accessed: 2015-09-10.

[52] Wojciech Plandowski. Satisfiability of word equations with constants is in pspace. *J. ACM*, 51(3):483–496, May 2004.

[53] Wojciech Plandowski. An efficient algorithm for solving word equations. In *Proceedings of the 38th Annual ACM Symposium on Theory of Computing*, STOC '06, pages 467–476, 2006.

[54] Xiao Qu and Brian Robinson. A case study of concolic testing tools and their limitations. In *ESEM*, pages 117–126. IEEE Computer Society, 2011.

[55] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2, 2012.

[56] Giampaolo Fresi Roglia, Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. Surgically returning to randomized lib(c). In *ACSAC*, pages 60–69. IEEE Computer Society, 2009.

[57] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 513–528, 2010.

[58] K. Schulz. Makanin's algorithm for word equations-two improvements and a generalization. In K. Schulz, editor, *Word Equations and Related Topics*, volume 572 of *Lecture Notes in Computer Science*, pages 85–150. Springer Berlin / Heidelberg, 1992.

[59] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 488–498, New York, NY, USA, 2013. ACM.

[60] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[61] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *CCS04*, 2004.

[62] D. Shannon, I. Ghosh, S. Rajan, and S. Khurshid. Efficient symbolic execution of strings for validating web applications. In *Proceedings of the 2Nd International Workshop on Defects in Large Software Systems: Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009)*, DEFECTS '09, pages 22–26, New York, NY, USA, 2009. ACM.

[63] Richard Skowyra, Kelly Casteel, Hamed Okhravi, Nickolai Zeldovich, and William W. Streilein. Systematic analysis of defenses against return-oriented programming. In *RAID*, pages 82–102, 2013.

[64] Takaaki Tateishi, Marco Pistoia, and Omer Tripp. Path- and index-sensitive string analysis based on monadic second-order logic. *ACM Trans. Softw. Eng. Methodol.*, 22(4):33:1–33:33, October 2013.

[65] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1232–1243, 2014.

[66] Fang Yu, Muath Alkhalaf, and Tevfik Bultan. Stranger: an automata-based string analysis tool for php. In *Proceedings of the 16th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'10, pages 154–157, 2010.

[67] Fang Yu, Tevfik Bultan, and Oscar H. Ibarra. Symbolic string verification: Combining string analysis and size analysis. In *in Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS*, pages 322–336.

[68] Fang Yu, Tevfik Bultan, and Oscar H. Ibarra. Symbolic string verification: Combining string analysis and size analysis. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '09, pages 322–336, 2009.

[69] Yunhui Zheng, Vijay Ganesh, Sanu Subramanian, Omer Tripp, Julian Dolby, and Xiangyu Zhang. Effective search-space pruning for solvers of string equations, regular expressions and length constraints. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 235–254. Springer, 2015.

[70] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. Z3-str: A z3-based string solver for web application analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 114–124, 2013.