Technical Report No. CE-J07-001, Dept. of Electrical Engineering, Princeton University, Dec. 12, 2007

Virtualization-assisted Framework for Prevention of Software Vulnerability Based Security Attacks

Najwa Aaraj†, Anand Raghunathan‡, and Niraj K. Jha†
† Department of Electrical Engineering, Princeton University, Princeton, NJ 08544
‡ NEC Laboratories America, Princeton, NJ 08540
†{*naaraj*, *jha*}*@princeton.edu* ‡*anand@nec-labs.com*

Abstract

Virtualization is a useful technology for addressing security concerns since it allows for the creation of isolated software execution environments, *e.g.*, for separation of the sensitive parts of a system from the complex, untrusted parts. In this paper, we describe a tool for dynamically detecting and preventing software vulnerabilities that exploits the availability of virtualized (isolated) execution environments. A program that is not itself malicious, but could have vulnerabilities, is first safely executed within a virtualized *Testing* environment, wherein its execution is traced using dynamic binary instrumentation and checked against extensive security policies that express the behavioral patterns associated with various vulnerability exploits. The program's execution trace is represented using a hybrid model that consists of regular expressions along with data invariants in order to capture both control and data flow. The execution trace and security policies are then used to construct an abstract behavioral model of the program's exercised execution paths. The program and its behavioral model are then migrated to a *Real* execution environment, wherein the behavioral model is used for efficient run-time monitoring (in place of the security policies), along with a continuous learning process, in order to prevent non-permissible program behavior.

We implemented the proposed framework using the `PIN` dynamic binary instrumentation tool from `Intel` and the `Xen` virtual machine monitor, and evaluated the utility and performance of the proposed methodology using several benchmarks and attack data sets. Our evaluation demonstrates almost 100% coverage for the considered software vulnerabilities, including buffer overflows, race conditions, link attacks, memory vulnerabilities, and NULL-pointer referencing. Execution time overheads of 38.42X and 1.46X are imposed on program execution in the *Testing* and *Real* environments, respectively, indicating the potential of the proposed approach in preventing vulnerability exploits with acceptable overheads.

1 Introduction

Software vulnerabilities are at the root of a large number of security compromises and attacks on computer systems. During the past decade, the number of software vulnerabilities discovered and exploited has increased drastically, as testified to by the dislosures from various organizations (*e.g.*, SECUNIA [1], CERT [2], and CVE [3]). From an attacker's perspective, exploiting bugs and flaws in software programs is often the easiest path to overcoming the security perimeter of a platform and consequently escalating his privileges to access the platform resources. Therefore, the development of technologies to address software vulnerabilities is a significant concern in information security.

Software vulnerabilities are classified into different categories [4], including buffer overflows, race conditions, cross-site scripting, *etc.*, and vary in severity and frequency of occurrence. In fact, leading IT companies have been trying to establish a Common Vulnerability Scoring System (CVSS) [5] in order to provide a common way to describe the seriousness of computer security vulnerabilities. Hence, it would be interesting to establish a common technique that encompasses the detection/prevention of multiple security vulnerabilities and scales its resources according to the prevalence and seriousness of the addressed vulnerability.

From research and commercial view points, the large number of security breaches caused by software vulnerabilities has instigated a lot of effort in the field of intrusion detection and prevention. Approaches to addressing software vulnerabilities can be classified into three main categories: static detection [6, 7, 8, 9], dynamic detection [10, 11, 12, 13, 14, 15] and hybrid approaches [16, 17] (*i.e.*, a combination of both static and dynamic techniques). The technique proposed in this paper lies in the category of dynamic/run-time detection techniques.

Recent advances in virtualization allow for the widespread use of isolated execution environments with minimal overheads, with several applications to information security. For example, virtualization has been used to enable the implementation of honeypots [18] and execution environments for security-critical functions such as trusted computing, antivirus tools, *etc.* [19]. In this work, we exploit the fact that virtualized environments provide the ability to safely test untrusted or vulnerable code without the danger of corrupting a "live" execution environment. In order to preserve a system's integrity, an isolated compartment, duplicating a system's configuration and state, can be built and used for vulnerability detection and monitoring purposes. This allows us to overcome the limitations of most current techniques for detecting software vulnerabilities, which are either incomplete, conservative, or require too much run-time overhead in order to be accurate.

In this paper, we address the issue of detecting program vulnerabilities and preventing their exploits. We focus on the specific problem of observing the execution of a program whose source code is not available, modeling safe/unsafe behavior with respect to specified security policies, and ensuring that the program does not deviate from safe behavior. We utilize the concept of isolated execution tackle this problem, by defining two virtual execution environments, namely a *Testing* environment and a *Real* environment. We describe in detail the architecture and flow of a dynamic vulnerability detection and prevention tool, whose behavior is summarized as follows:

- Statically generate safe and specially-crafted input test vectors using an automatic and policy-dependent input generation technique. The input vectors are necessary for a high code coverage of the application executed in the *Testing* environment and a robust checking of its paths against designed security policies.
- Execute the potentially vulnerable program in a the *Testing* environment using the statically-generate input vectors. As the program executes, use dynamic binary instrumentation to collect specific information in the form of execution traces.
- Analyze the execution traces to construct a hybrid model that represents the program's dynamic control and data flow in terms of

regular expressions and data invariants. The regular expression R_U, has an input alphabet $\Sigma = \{BB_0, ..., BB_n\}$, where BB_i is an execution trace basic block. For each basic block, several properties are defined that are relevant to the detection of vulnerability exploits.

- Given security policies that represent program behavior under vulnerability exploits, express them using the same hybrid model as the program execution traces. Take the intersection of the two models to reveal security vulnerabilities (*i.e.*, security policy violations).
- Extract appropriate checkpoints based on program properties and data invariants, and derive a monitoring model *M*, which can be migrated to the *Real* environment.
- Observe program execution in the *Real* environment and ensure that its execution conforms with the extracted model *M*. Checks are done at the granularity of extracted checkpoints. If a new execution path is encountered, restrictive security policies are enforced at run-time, thus, preventing the exploit of any vulnerability, and a new execution path is learned and added to *M*.

We implemented the proposed framework using the `PIN` dynamic binary instrumentation tool from `Intel` and the `Xen` virtual machine monitor, and evaluated the utility and performance of the proposed methodology using several benchmarks and attack data sets. The attack data sets covered multiple security vulnerabilities, namely, buffer overflows, race conditions, and memory vulnerabilities. Results show that our tool enables near-complete detection and prevention of all the considered attacks. It is important to note that our approach is not limited to detecting the aforementioned attacks; it promises coverage of a wider range of software vulnerabilities and malicious software behavior, provided that appropriate security policies are designed.

The rest of the paper is organized as follows. A survey of field-relevant past work is presented in Section 2. A high-level overview of our approach is given in Section 3. The details of the proposed framework in the *Testing* environment are given in Section 4, and in the *Real* environment in Section 5. The experimental methodology and results are presented in Section 6, and conclusions in Section 7.

2 Related Work

Various techniques have been proposed for the detection and prevention of software vulnerabilities. Most of the approaches fall into three different categories: static techniques, dynamic techniques, and a hybrid combination of both techniques.

Static analysis techniques examine the source code of a program and detect possible vulnerabilities. FindBugs [6] and Flawfinder [7] are such tools, which analyze Java and C code, respectively, looking for security flaws. Other tools, such as, CodeSurfer [8] are based on data dependency and inter-procedural analysis for detecting software vulnerabilities. In [9], the proposed work disassembles a program's machine code and statically identifies malicious behavior in the program. Static techniques can be a great resource for security logs and auditing. However, due to inherent limitations in static analysis, which applies overly conservative assumptions, they result in a high rate of false positives and negatives. Moreover, accurate static analysis is often not possible when the program source code is not available.

Dynamic analysis techniques used for detection of software vulnerabilities rely on real-time execution of a program to test for vulnerabilities. Dynamic analysis has the advantage of observing the application's run-time behavior and its interaction with other system components while it executes. The main drawback of dynamic analysis is that it only examines a single execution path at a time. Multiple vulnerability detection tools fall in the category of dynamic analysis. In [10], a dynamic taint analysis technique is introduced, which marks as tainted any data originating or generated from untrusted inputs. Tainted data are checked as they propagate in order to perform automated detection of attacks. In [11], a technique for installing network filters at the host level is proposed once vulnerabilities are discovered. The exploit-generic filter acts as an intermediate agent, by examining incoming and outgoing network traffic from the vulnerable system and correcting the traffic based on manually-generated vulnerability signatures. In [12], security checks are automatically embedded in a program's execution at run-time. Program shepherding [13] monitors the control flow of a program and applies a continuum of security policies, ranging from highly-restrictive to unrestrictive policies. These security policies restrict program execution based on code origins and control transfers, such as intra-segments calls or jumps.

A third category of analysis techniques follow a hybrid approach (also known as glass-box testing) consisting of both static and dynamic analysis. Examples of techniques integrating static and dynamic analysis can be found in [16, 17]. Provided that the source code is available, augmenting dynamic analysis with source information promises to produce better accuracy and efficiency.

Our approach falls into the category of dynamic analysis. We tackle binary programs whose source code is not available. However, our work is differentiated from previous work along the following dimensions:

- It does not require a program to halt execution when a vulnerability exploit is suspected to have occurred. Since the program is executed in an isolated environment, we let it run, irrespective of its code origin, data destination, and control transfers. Execution traces are subsequently analyzed against security policies for detection of vulnerability exploits. This is especially attractive when both benign execution and an execution with a vulnerability exploit display similar or near-identical behavior.
- We virtually partition the task of detection and prevention of software vulnerability attacks into two environments. After extensive execution in a *Testing* environment, which duplicates the configuration of the user's workspace environment, we extract a vulnerability-specific behavioral model as a guideline for permissible behavior in the *Real* environment.
- The proposed approach is not specific to any single vulnerability, and can be adpated using appropriately designed security policies

Based on the above factors, we believe that the proposed approach is suitable for the detection and prevention of a wide range of software vulnerabilities, and consequently the associated security attacks.

3 Overview of Proposed Approach

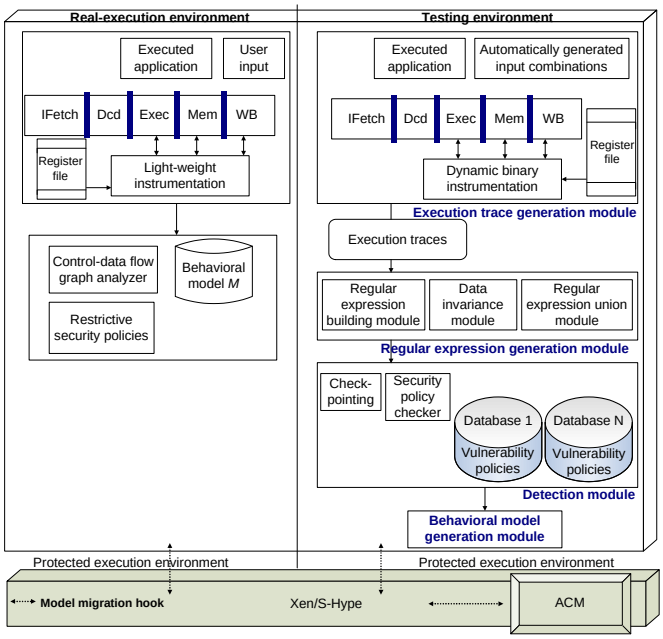

Figure 1: Overview of the proposed vulnerability detection and prevention approach

Figure 1 presents an overview of the proposed tool. The tool consists of two isolated virtual machines (execution environments) on top of the

Xen [20] virtual machine monitor. In each virtual machine, dynamic binary instrumentation based on the PIN [21] framework is used to analyze the program's behavior for detection and prevention of vulnerability exploits. In the *Testing* environment, the dynamic binary instrumentation tool, built using PIN's API, instruments executables without the need for re-compiling or re-linking. Program execution traces are obtained at the instruction level (Section 4.1). A regular expression generator combines each sequence of instructions that terminates in a control-flow transfer instruction into a basic-block (*BB*). Consequently, the execution trace is expressed as a regular expression R defined over the alphabet $\Sigma = \{BB_0, ..., BB_n\}$, which represents the set of all basic blocks in the program. The union of all generated regular expressions (each representing a separate program execution) is performed to combine the program execution paths into a single regular expression R_U. Meanwhile, regular expressions are passed through a data invariance detector, which formulates invariants obeyed by the data associated with each basic block (Section 4.2). The application's execution in the *Testing* environment is halted once program coverage is deemed to be sufficient, or when no new execution paths are activated by new input combinations. Thereafter, R_U is subjected (through a regular expression intersection operation) to security policy checks in order to detect any exploit of the application's code (Section 4.3). The intersection procedure detects any violation of the specified security policies. It also allows checkpointing trace segments whose monitoring is critical in order to prevent any vulnerability exploits. Checkpoints and invariants are consequently used by a behavioral-model generator, which filters out basic blocks that are significant for the tool's purposes and defines a fixed set of basic blocks, which capture properties indicative of permissible program behavior. The behavioral model, M, also maintains a record of the application's branching points (*i.e.*, the points with conditional control transfer) in order to keep track of the executed paths (Section 4.4). M is then migrated by a model migration hook managed by Xen's Access Control Module (ACM) to the *Real* execution environment. In the *Real* execution environment, the running application is subjected to instrumentation at the basic-block granularity and to run-time monitoring, wherein only checkpointed basic-blocks are monitored (Section 5.1). Run-time monitoring also involves the application of restrictive security policies in the case where new execution paths are encountered (Section 5.2).

4 Details: *Testing* Environment

This section describes the operation of the proposed tool in the *Testing* environment. Section 4.1 describes the execution trace generation module, Section 4.2 describes the regular expression generation module, Section 4.3 describes the detection module, and Section 4.4 describes the behavioral-model generation module.

4.1 Execution Trace Generation Module

Before getting into the details of this step, an introduction to the PIN infrastructure is appropriate. Instrumentation is a technique for inserting extra code into an application for behavioral observation. PIN performs such an instrumentation transparently at run-time. It provides an API that enables the development of new tools with the capability of inspecting the processor state as the program executes.

The tool presented in this work is built on top of PIN. It executes the instruction intercepted and re-generated by PIN and generates information, which specifies vulnerability-specific properties that will be checked against violation of the security policies. Information that is observed for the particular vulnerability examples considered in this work is specified in Table 1.

Upon an application's execution, instrumentation monitors its control flow and that of dynamically-linked libraries mapped into its address space. Dynamically-linked libraries are considered unsafe and subjected to the same inspection procedures as the application itself. In order to test the application, we run it on a large set of safe and specially-crafted inputs that are automatically generated using the technique described next.

Table 1: Tracked program state information

Vulnerability	**Architectural process state captured in each execution trace**
Buffer overflow	Static/Automatic/Dynamic memory (S/A/D) allocation (Starting address, Size)
	(S/A/D) memory de-allocation (Starting address)
	Writes to allocated memory and adjacent locations
	"execve" and C-library "system" function
	function pointer address values
	setjmp - longjmp buffer environment values
	global offset table (GOT) entries and invocations
	user input strings
	loop variables and corresponding dependent-variables
	stack frame boundaries
	control transfer instructions and addresses
Race conditions	file-system related system calls
Memory vulnerabilities	S/A/D memory allocation
	S/A/D memory de-allocation
	Writes to allocated memory and adjacent locations

4.1.1 Automatic Input Generation Technique

The effectiveness of our detection/prevention tool considerably depends on code coverage, *i.e.*, the number of paths executed in the *Testing* environment and the number of triggered vulnerable paths.

In this section, we describe a system (Figure 2) that allows automatically generating input sequences exercising a high percentage of a program's paths and triggering security violations in vulnerable paths. This system is based on static binary analysis and symbolic propagation. Despite the conservative nature of static analysis, we adopt this technique for the following reasons:

- Static analysis is able to leverage a complete knowledge of a program's structure and properties that are relevant to specified security vulnerabilities.
- Static analysis is able to resolve first-level dependencies between a program input and its control flow and between inputs and instructions whose execution might contribute to a security violation.
- Static analysis might result in some imprecise dependencies in a program flow. However, such inaccuracies will be accounted for in the *Testing* and *Real* environments. *E.g.*, analysis in the *Testing* environment might not detect any vulnerabilities in a path otherwise suspected vulnerable when executed under the test vector generated by the input generation system. Similarly, analysis in the *Real* environment halts a suspicious path execution otherwise deemed safe and input-independent by the input generation system.
- Using dynamic binary analysis with a random initial input vector does not guarantee the identification of all input-dependent paths. Dynamic analysis alone would be unable to explore all path possibilities.

While we adopt a static-analysis based input generation technique for testing the efficiency of our detection/prevention tool, future work involves input generation based on a more efficient hybrid approach combining static analysis refined by dynamic analysis results as performed in the *Testing* environment.

a) Implementation details

Our system operates on a disassembled binary. It proceeds by using symbolic propagation analysis to automatically identify different input-dependent paths and input-dependent instructions that are susceptible to result in specific security vulnerabilities. It is basically composed of three parts: (i) binary output pre-processing, (ii) path predicates generation, and (iii) solver.

a.1) Pre-processing binary output: Prior to static analysis, disassembled binaries are pre-processed in order to resolve function calls and loop unrolling.

Loop analysis: To avoid infinite iterations, we limit the number of loop iterations to τ, where τ iterations are sufficient for approximating a loop behavior and the dependency of the loop condition on input variables.

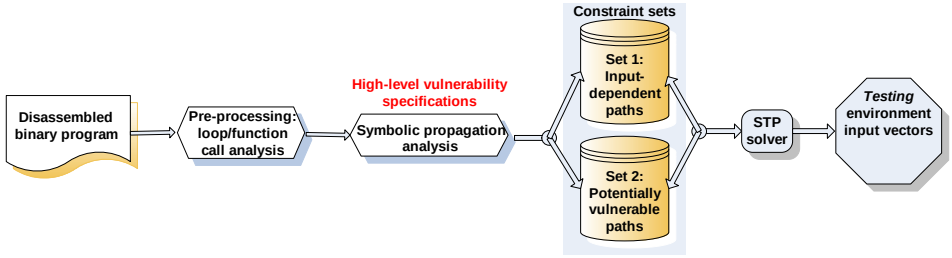

Figure 2: Automatic input generation technique

Function call analysis: Unless functions are linked dynamically, they are replaced by their corresponding code in the disassembled binary, thus allowing a more accurate propagation of symbolic inputs.

a.2) Generating path predicates: Our system operates by generating the following formulas:

1. Symbolic formulas representing input constraints as dictated by input-dependent instructions.
2. Symbolic formulas representing input constraints as dictated by assembly instructions that might result in specific security vulnerabilities.
3. Two symbolic formulas representing input constraints as dictated by input-dependent control transfer instructions.
4. Concrete input-independent formulas as dictated by input-independent instructions.

Given each instruction, our system checks whether it depends on the symbolic input, in which case we perform a direct translation of the instruction to an STP [22] conditional formula. The translation is based on techniques presented in [23] and [24]. The example in Table 2 demonstrates such a translation.

1. For instructions that perform binary, unary, and assignment operations (in addition to load and store to concrete memory addresses), we generate an STP LET expression, which binds a generated variable name to the expression computed by the assembly instruction. For each operation, we use the corresponding STP bitwise and arithmetic functions.
2. For conditional control transfers, we generate two formulas, thus resulting in two sets of path predicates, one path where the current path continues with the *True* branch and one path where the current path continues with the *False* branch. Each control transfer instruction is translated to two symbolic formulas using the corresponding STP predicates.
3. For assembly instructions, which do not depend on the program inputs, their operands are set or abstracted to concrete values using the STP ASSERT function.
4. For instructions that contain properties embedded in the basic blocks of our designed security policies, our system generates conditional input-dependent formulas. For example, in Table 2, we see that instruction *I*1 contains a call to function *open* and instruction *I*2 contains a call to function *chmod*, thus, resulting in a vulnerability window that might be exploited by an external attacker. If the vulnerability window is input-dependent, we might be able to increase its length by generating appropriate input formulas. In the example of Table 2, conditional formulas (V_1–V_5) are generated *n* times. At the n^{th} iteration, formulas are generated to exit the loop executing between the pair of system calls. Increasing the input-dependent vulnerability window enhances the probability of successfully attacking a vulnerable application by an attack code running in parallel to it.

a.3) The solver: The solver that we use is STP, which is an efficient decision procedure for the validity of a set of formulas. It has its own built-in language with specific functions and predicates. For all generated path conditions, the solver checks if the conditions are satisfiable. In case the path conditions are satisfiable, the solver generates an input test vector that triggers the execution of the path in question. If no input combination leads to path execution, STP returns unsatisfiable. STP may also return segmentation faults (excessive memory usage) in case it cannot solve the path predicates in a reasonable time period.

b) Experimental results

We have implemented the approach above and tested it on various benchmarks that contain real vulnerabilities and are fairly complex to analyze manually. Experiments (Table 3) show that our technique is automatically and efficiently able to generate (offline) suitable input vectors used in the *Testing* environment. Column 1 presents the tested application, column 2 the number of input-dependent paths (ID paths), column 3 the STP time per execution (t/exec) of the tested application, column 4 the number of satisfiable path conditions (SP), column 5 the number of derived path constraints (Constr.), and column 6 the total time (Tt) required for generating the set of testing vectors for the application.

Table 3: Automatic input generation results

Benchmark	# ID paths	STP (t/exec-s.)	# SP	# Constr.	(Tt-min.)
bzip2	64	7.60	56	8195	7.14
gzip	32	9.65	19	2075	2.47
unzip	28	7.59	27	2104	3.42
gpm	0	n/a	n/a	n/a	n/a
sdiff	236	9.63	231	16396	37.07
OpenSSH	6	1.87	6	1420	0.19
mpg123	51	2.29	51	24642	1.95
ghttp	7	0.47	7	1055	0.05
flex	21	5.32	21	45114	1.86
gawk	61	0.84	60	1436	0.837
nasm	19	1.87	19	1211	0.50
zlib	11	1.86	11	1052	0.34
zoo	61	1.88	57	2099	1.78

4.2 Regular Expression Module

This section is organized as follows. Section 4.2.1 describes the regular-expression generation module, Section 4.2.2 describes the regular-expression union module, and Section 4.2.3 describes the data invariance module.

4.2.1 Regular Expression Extraction Module

This module transforms each execution trace into a regular expression. Constructed regular expressions are not only a text-parsing tool, they present a codified method that allows parsing and isolation of specific properties within a body of the execution trace. Each regular expression is defined over alphabet $\Sigma = \{BB_0, \ldots, BB_n\}$. At the highest level of granularity, each literal, *BB*, is a basic block derived as a sequence of instructions terminating at a control-flow transfer instruction (*i.e.*, any computed or indirect jump instruction or call instruction).

Each basic block encapsulates various properties and components, presented in data-structure ***BB*** (Table 4), which should be accurately defined since they are indicators of a software exploit in the application. Each property captured in a basic block is coded into a concise representation for memory and execution time scalability purposes in large programs. Each *BB* is identified by a unique or standardized address **unique_block_address** and by its **block_number**. If *BB*'s entry point corresponds to a static executable address *A*, then, **unique_block_address** = *A*. On the other hand, if this entry point corresponds to a dynamic link entry, we calculate address offset *O* in an execution trace with respect to a pre-defined value A' that we specify. Given

Table 2: Translation to STP constraints/Augmenting vulnerability window between paired system calls

```
(I1)  80483DB    call <fopen@plt>
                                        —> LET V_i1 = INPUT_i
                                        IN
      80483E0    subl $0x1, (%eax)          —> LET V_i2 = (BVSUB(32,V_1,0hex00000001))
                                        IN
      80483E3    lea 0xfffffffc(%ebp), (%eax)    —> LET V_i3 = V_i2
                                        IN
      80483E6    cmpl $0x0, 0xfffffffc(%ebp)
      80483E9    jg 80483e0                 —> LET V_i5 = (BVGE(V_i2, 0hex00000000))
                                        —> LET INPUT_(i+1) = V_i2
      80483EB    movl $0x0, 0x4(%esp)
(I2)  80483EE    call <chmod@plt>
                                        —> LET V_n1 = INPUT_1
                                        IN
                                        —> LET V_n2 = (BVSUB(32,V_1,0hex00000001))
                                        IN
                                        —> LET V_n3 = V_n2
                                        IN
                                        —> LET V_n5 = (BVLT(V_n2, 0hex00000000))
```

that the address of the first dynamically-linked address that follows an application's entry point virtual address is equal to D_A, $O = D_A - A'$. Thereafter, the address of each such *BB* is adjusted by this offset in order to allow for efficient and correct comparison by the regular expression union module, thus yielding the value of **unique_block_address**. Another constituent of ***BB*** is the **execution_id** field = $b_{63}...b_0$, where b_i = 1, if *BB* is executed along the i^{th} execution of the program. **num_child** is the number of BB_j that can be executed following BB_i's execution. **num_child** > 1, if BB_i is a bifurcation point, and **child** is an array that stores the index of BB_i's children. The remaining fields are set to *NULL* unless BB_i, respectively, contains: (i) a call instruction, (ii) a return instruction, (iii) a function pointer call, (iv) a setjmp function, (v) a longjmp function, (vi) a GOT entry call, (vii) an instruction operating on the user input, (viii) a loop whose condition depends on the value of an array of variables (VAR), (ix) memory allocation, (x) memory de-allocation, (xi) a system call, (xii) a conditional control transfer (in which case *BB* is referred to as *BB_jump*), (xiii) a routine or function definition, (xiv) a condition's exit point, (xv) a conditional control transfer with multiple paths associated with it, or (xvi) any data invariant forms (Section 4.2.3). Details of each constituent field are given in Table 4. Constituents of each BB_i in regular expression R_k are extracted from the k^{th} execution trace and R_k is generated accordingly.

4.2.2 Regular Expression Union Module

A logical succession to the regular expression generation is to subject it to a check for security exploit violation. However, a more scalable method is to find the union R_U of the regular expressions, and subject it to the security policies. The union module operates on the control flow of the application and is complemented by a data invariance module, which operates on the data flow of the application (Section 4.2.3). The union operation is presented in Procedure ***Union***. It recursively combines single-trace regular expressions, R_k, into R_U, *i.e.*, when R_k is generated, the following operation is performed: $R_U = R_U \bigcup R_k$. As a starting point, $R_U = \lambda$ (empty regular expression). When combining R_U and R_k, if a path in R_k already exists in R_U, the execution_id of the *BB*s in the corresponding path are updated, as in line 7. When a *BB_jump* is encountered, if a new execution path is simulated, the latter is handled by adding it as a new child of *BB_jump* (lines 55-78). Otherwise, if no new execution path is simulated, ***Union*** is recursively called on the repeated execution path (line 52). ***Union***'s recursive call exits when a join-point (denoted by *BB_join_pt*) is encountered. A *BB_join_pt* is defined as the point where different execution paths converge, *i.e.*, the exit point of the corresponding *BB_jump*. (Join-points are captured by running Procedure ***Join-Pt*** on R_U and R_k before calling Procedure ***Union***). Due to space limitations, we limit ourselves to a brief explanation of ***Join-Pt*** instead of writing the full algorithm details. In ***Join-Pt***, both R_U and R_k are parsed in a top-down manner until a common *BB_jump* is encountered, such that, *BB_jump* has an an execution path in R_k that is not included in R_U. At that point, both regular expressions are traversed until a common *BB* is executed. If all call transfers in both R_U and R_k, that have been executed between *BB_jump* and *BB*, have returned, then *BB* is the *BB_join_pt* (*i.e.*, *BB*.join_pt = 1) basic block corresponding to *BB_jump* and *BB_jump*.stamp = 1.

4.2.3 Data Invariance Module

This module's intent is to formulate invariants obeyed by data at specific basic blocks of R_U. It gives insight into the properties of the program data flow that might be useful in identifying malicious exploits. Invariants have been previously proposed with various intent, including the identification of behavioral program bugs [25], and understanding a program's behavior [26]. The on-line data invariance module maintains invariants at various program points, *i.e.*, specific locations in the execution trace, which we specify depending on the addressed software vulnerabilities.

For the program vulnerabilities addressed in this paper, namely, buffer overflows, race conditions, link attacks, memory, and NULL-pointer dereferencing vulnerabilities, the data invariance module operates on the following data elements of a regular expression basic blocks:

- Data path flow, which consists of pairs (*BB.block_number*, *BB.unique_block_address*).
- Memory allocation elements, specifically, allocated memory size and address space.
- Addresses of function pointers.
- Environmental values at longjmp and setjmp function calls.
- Addresses/Entry-points of GOT entries.
- Original and modified values of user inputs.
- Loop constraints variables and corresponding dependencies.
- Program routine addresses.
- Arguments of "execve" system call.
- Arguments of file system calls.
- Pushed return addresses at call sites, except for dynamically-linked address values.

The data invariance module keeps track of data invariants within R_U basic blocks. The invariants inferred over the above variables are:

- Acceptable or unacceptable constant values, indicating a value that a variable should or should not assume, respectively.
- Acceptable or unacceptable range limits, indicating the minimum and maximum values that a variable can assume or should not assume, respectively.
- Acceptable or unacceptable value sets, indicating the set of values that a variable can assume or should not assume, respectively.
- Acceptable or unacceptable functional invariants, indicating a relationship within a number of variables that should be satisfied or should not be satisfied, respectively.

Table 4: Basic block (*BB*) data structure

Data structure *BB*: Basic block data structure	
Field	**Comment**
UINT unique_block_address	standardized address of *BB*
INT block_number	index number of *BB*
INT num_child	number of *BB*'s children
INT* child	index array of *BB*'s children
UINT64 execution_id	execution_id = $b_{63}...b_1b_0$, where each b_i is set in case *BB* is executed in the *i*-th execution of the program
CALL_STR call_site	call_site is built if *BB* contains a call instruction. CALL_STR data structure contains: (i) called site address, (ii) pushed base-pointer value, and (iii) pushed return address value
RET_STR ret_site	ret_site is built if *BB* contains a return instruction. RET_STR data structure contains: (i) popped return address, and (ii) popped base-pointer value
UINT function_ptr[2]	function_ptr is built if *BB* contains a function pointer. The array contains: (i) backtracked (check Table 7 for explanation) address of invoked function, and (ii) function address at invocation time
UINT* setjmp_buf	setjmp_buf is built if *BB* contains a setjmp function. The array contains environment values stored when setjmp is called
UINT* longjmp_buf	longjmp_buf is built if *BB* contains a longjmp function. The array contains environment values restored when longjmp is called
FUNCTION GOT_fct	GOT_fct is built if *BB* contains a call to a function in the dynamic relocation ELF section
Char * user_input	user_input is built if *BB* contains an instruction operating on the user input
LOOP_VAR* VAR	VAR is built if *BB* contains a loop whose condition depends on the value of variable var_1. VAR is an array containing var_1 and other variables in *BB* upon which var_1 depends or which depend on var_1
MALLOC_STR malloc	malloc is built if *BB* contains a malloc function. MALLOC_STR data structure contains: (i) size of allocated memory, and (ii) starting address of allocated memory
FREE_STR free	FREE_STR data structure contains the starting address of freed memory
SYS_CALL system_call	system_call is built if *BB* contains a system call. SYS_CALL data structure contains: (i) name of system call, and (ii) arguments of the system call
BOOL condition	True, if *BB* contains a conditional call transfer
RTN Routine	name and address of a routine if basic block contains one
BOOL join_pt	True, if *BB* is a condition's exit point
BOOL stamp	True, if *BB* contains a conditional instruction and has multiple paths associated with it
INV* inv	array inv is built if data invariants of any form are associated with *BB*

When combining regular expressions R_k and R_U, each potential data invariant is checked. When a data point stemming from R_k does not satisfy a data invariant type stored at R_U, this invariant is not checked subsequently and is removed from within the basic block. Deduced data invariants are used by the security violation detection module and the behavioral model generation module (Sections 4.3 and 4.4, respectively).

4.3 Detection Module

This section describes the detection module of Figure 1. First, Section 4.3.1 describes the different software vulnerabilities we tackle as a proof of concept of the efficiency of our approach. Section 4.3.2 details the security policies to be applied to the application's R_U. Section 4.3.3 describes the methodology of applying these policies.

4.3.1 Software Vulnerabilities and Potential Exploits

Prior to getting into the details of describing security policies and the methodology of applying them, an overview of the vulnerabilities we address in this paper is due.

Buffer overflows: Buffer overflows are a major form of security vulnerabilities in today's software. They have dominated the software vulnerabilities reported by CERT advisory [2] and are rated as the second most common vulnerability (2156 incidents) in the past five years by the Common Vulnerabilities and Exposure (CVE) list by Mitre [3]. In Figure 3, we illustrate how a simple vulnerability, namely, a lack of in-

Procedure *Union*: Union of R_U with a single-trace regular expression R_2

Inputs: single-trace regular expression R_2 (generated for the program's k^{th} execution), R_1, start_index_R_2, and start_index_R_1. Note that $R_1 = R_U$ when ***Union*** is first called.
Output: regular expression R_U

```
0.  static index_joint, index for R1, index2 for R2, and new_index for RU
1.  if first recursion
2.    R1 = RU
3.  endif
4.  while TRUE
5.        copy(RU.BB_new_index , R1.BB_index )
6.        RU.BB_new_index.block_number = new_index
7.           RU.BB_new_index.execution_id = RU.BB_new_index.execution_id Or (64-k)'b{0}.1'b{1}.(k-1)'b{0}
8.        Find_invariant(RU.BB_new_index.invariant, R1.BB_index, R2.BB_index2)
9.        if R1.BB_index.condition = 1 and R1.BB_index.stamp = 1
10.           break
11.       endif
12.       index = R1.BB_index.child[0]
13.       increase(index2)
14.       if R1.BB_index equal BB_join_pt
15.           return
16.       else
17.           RU.BB_new_index.child[0] = new_index + 1
18.           RU.BB_new_index.child_num = R1.BB_index.child_num
19.       endif
20.       increase(new_index)
21. end of while
22. CHILD_NUM = R1.BB_index.num_child
23. for i = 1 ... CHILD_NUM
24.       if R2.BB_index2+1 is equal to R1.BB_R1.BB[index].children[i]
25.           Found = 1
26.           COMMON_CHILD = index
27.           break
28.       endif
29. end of for
30. temp_index = new_index
31. if Found equal 1
32.       RU.BB_new_index.child_num = Cn = R1.BB_index.child_num
33.       increase(new_index)
34.       j = 0
35.       for i = 1 ... CHILD_NUM
36.           child = R1.BB_index.child[i]
37.           if child not equal to COMMON_CHILD
38.               RU.BB_temp_index.child[j++] = new_index
39.               while R1.BB_index.join_pt = 0
40.                   copy(RU.BB_new_index,R1.BB_index)
41.                   RU.BB_new_index.block_number = new_index
42.                   increase(new_index)
43.                   index = R1.BB_index.child[0]
44.                   BB_join_pt = R1.BB_index
45.               end of while
46.           endif
47.       end of for
48.       copy(RU.BB_new_index, BB_join_pt)
49.       index_joint = RU.BB_new_index.block_number = new_index
50.       increase(new_index)
51.       RU.BB_temp_index.child[Cn-1] = new_index
52.       Union(R2, R1, index2, COMMON_CHILD)
53.       RU.BB_new_index.child[0] = index_joint
54. else
55.       RU.BB_new_index.child_num = R1.BB_index.child_num + 1
56.       increase(new_index)
57.       j = 0
58.       for i = 1 ... CHILD_NUM
59.           RU.BB_temp_index.child[j++] = new_index
60.           while R1.BB_index not equal to BB_join_pt
61.               copy(RU.BB_new_index, R1.BB_index)
62.               RU.BB_new_index.block_number = new_index
63.               increase(new_index)
64.               index = R1.BB_index.child[0]
65.               BB_join_pt = R1.BB_index
66.           end of while
67.       end of for
68.       copy(RU.BB_new_index, BB_join_pt)
69.       index_joint = RU.BB_new_index.block_number = new_index
70.       increase(new_index)
71.       RU.BB_temp_index.child[Cn] = new_index
72.       while R2.BB_index2 not equal to BB_join_pt
73.           copy(RU.BB_new_index, R2.BB_index2)
74.           RU.BB_new_index.block_number = new_index
75.           RU.BB_new_index.execution_id = (64-k)'b{0}.1'b{1}.(k-1)'b{0}
76.           increase(new_index)
77.           increase(index2)
78.       end of while
79.       RU.BB_new_index.child[0] = index_joint
80. endif
```

put validation and boundary checks is exploited using specially-crafted input sequences in order to execute arbitrary malicious code. The vulnerable code reads a user input into a local **buffer** without any validation of the user-provided input. The block storage segment (BSS) contains the value of a function pointer, which is to be called after filling up **buffer**. In order to launch his exploit, an attacker creates a maliciously-crafted user input of length $> |P_2 - P_1|$. When the program executes, it causes the function pointer value to be overwritten and directed towards the malicious code. While the vulnerability in Figure 3 was exploited for attacking a function pointer allocated on the BSS, it was proved in [27] that various attack targets (return address, old base pointer, function pointers, and longjmp buffers) can be exploited by buffer overflows to change the control flow of a program. Those targets can be placed under attacks in two different locations (stack or heap/block storage/data segment) and using two different techniques (overflowing a buffer all the way to the attack target or overflowing a buffer to redirect a pointer to the target).

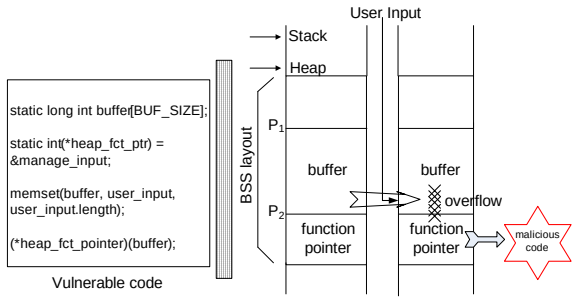

Figure 3: Example of an attack overflowing a function pointer value

Considering all combinations of techniques, locations, and target attacks, in [27] a testbed of 20 different attack forms is considered and the effectiveness of publicly-available tools for dynamic buffer overflow prevention is investigated. The best tool is only 50% effective in detecting/preventing the attacks and there are six attack forms which none of the tools can handle.

Another form of buffer overflow opportunities that is often under scrutiny by attackers and that we consider in our work is to overwrite memory locations, which are often to be executed through calls to execve or to the standard libc "system" function. Buffer overflow attacks are also implemented as non-control-data attacks, which, unlike control-data attacks that alter a program's control data (*e.g.*, return values, function pointers, etc.) to execute malicious code, operate by overflowing the value of non-control application data, such as system call arguments, decision making data, user identity data, GOT entries, configuration data, etc. ([28, 29, 30]).

We ran our tool, with appropriately-designed security policies, over the 20 attack benchmarks in [27] and we obtained a coverage of 100%. We also obtained a 100% detection rate of buffer overflow vulnerabilities in open-source packages, such as gzip and ncompress (Section 6.2.3).

We also demonstrate the efficiency of our approach in detecting (a 100% detection rate) non-control-data attacks by running it on synthetic and real-life commercial benchmarks that contain vulnerabilities allowing attacks overflowing the values of GOT entries, system call arguments (execve system call), user inputs, and decision making data.

Race conditions and link attacks: Race conditions, specifically file access race conditions, are known as time-of-check, time-of-use (TOCTOU) race conditions. They are a common form of software vulnerabilities (534 incidents in the past five years, as reported by CVE [3]), and the exploit of security holes related to them enables an attacker to gain read, write, and execute privileges that interfere with the file system. Race conditions occur when two events follow each other and the second depends upon the first. If a malicious action occurs during the time interval (known as vulnerability window) between the two events, the outcome of the second event will not be as intended. Attackers have numerous techniques for expanding the length of the vulnerability window. Moreover, an exploit attempt can be continuously repeated until it succeeds [31]. An example of such an attack is presented in Table 5.

Table 5: Race condition example: Open-chown attack

Action	File system calls	Description
file = fopen ("**FILE**", "w")	open(file, O_WRONLY| O_SYNC| O_CREAT| O_TRUNC)	user opens a write-file **FILE** with root privileges
link -s file /etc/pwd		The attacker exploits the window between open and chown32 and creates a symbolic link from **FILE** to the passwords file /etc/pwd
chown (file, user)	chown32(file, user, NULL)	user is assigned root privileges over **FILE**. The symbolic link is resolved and the user now has access privileges over the passwords file

An attacker can simply exploit the vulnerability interval between `open` and `chown` system calls, and create a symbolic link from **FILE** to `etc/pwd`, causing the program to have root privileges over the system passwords file, thus, causing actions, such as overwriting or deleting of the etc/pwd file. When a symbolic link joins two events, the race condition is then called "link attacks." For the purpose of this paper, we classify race conditions and link attacks into three categories:

- (sys_call_1 - sys_call_2) attack, in which the attacker exploits a time window between two system calls operating on a particular file F. Most frequently, such an attack is accomplished through creating a symbolic or hard link from F to a privileged file. Most common (sys_call_1 - sys_call_2) pairs are listed in Table 6.
- Directory redirection attacks, in which the attacker redirects the directory, in which an event was being performed to another directory. Therefore, after such an attack, a program would be operating in a directory different than the intended one.
- Temporary-file race condition attack, in which a program creates temporary files with easily predictable names. An attacker exploits this vulnerability by launching link attacks on guessed file names, potentially gaining elevated access privileges to the system.

Table 6: System calls with an exploitable vulnerability window

sys_call_1	**sys_call_2**
access	open
open	open
open	chmod/fchmod
open	chown/lchown/fchown/lchown32/fchown32/ chown32
rename	chown/lchown/fchown/lchown32/fchown32/ chown32
fstat/lstat/lstat64/fstat64	open
chdir	rmdir
mkdir	chmod

We have run our tool on an attack data set of nine different race condition benchmarks and several open-source packages with reported race condition vulnerabilities, and we obtained a 100% detection rate of the attacks (Section 6.2.3).

Memory vulnerabilities and null-pointer dereferencing: These include memory leaks, double-free vulnerabilities, boundary-check vulnerabilities (usually classified as a buffer overflow attack), uninitialized memory, etc. Unsafe memory operations are not very complicated. However, most are spread across a large number of code lines or files, a fact that leaves programmers susceptible to committing such operations. This also means that the detection of such memory vulnerabilities is quite hard to track manually or statically. They are easily exploited and their exploit can trigger buffer overflow, sensitive information leak, DoS attacks, etc. A variety of tools have been designed to detect memory management problems [32].

4.3.2 Security Policy Structure

The security policies that we use in our work are a high-level language specification of a series of actions, which, if executed in a particular sequence, outline a security violation. Each security policy is translated into a regular expression R_P defined over alphabet $\sigma = \{bb_1, bb_2, ..., bb_n\}$. Policy specifications are tailored to each vulnerability semantic. Moreover, for each individual vulnerability, several policies may be

needed. Figure 4(a) shows the structure of a security policy, and Figure 4(b) demonstrates how the policy's regular expression is constructed from a high-level language specification. Tables 7 and 8 show different vulnerability-specific policies for buffer overflows, and race conditions (R.C.) and link attacks (L.A.) and memory vulnerabilities (M.V.), respectively. Column 1 (Table 8) specifies the targeted vulnerability (R.C., L.A., M.V.), Column 1 (2) in Table 7 (Table 8) gives the high-level specification of a security policy and column 2 (3), correspondingly, gives the regular-expression translation. As can be noticed from the table, the

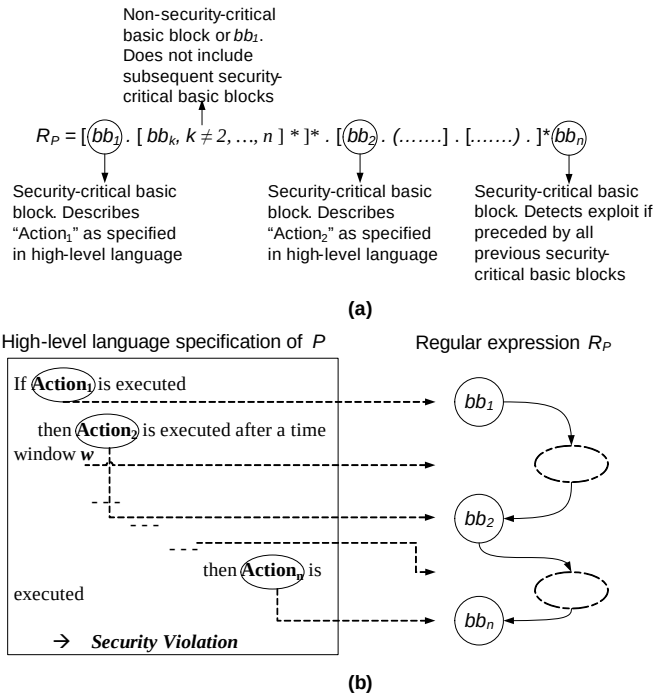

Figure 4: Security policy specification and translation into regular expression R_P

policies, through their basic blocks components, capture how a software flaw can be exploited. If a policy's behavioral model is captured in R_U, through a regular expression intersection between R_U and R_P, a security exploit must have occurred during the program's execution.

4.3.3 Application of Security Policies

A regular expression intersection engine is responsible for applying a security policy R_P against R_U. Procedure ***Intersect*** describes how vulnerability exploits are automatically detected by intersecting R_U and R_P. The ***Intersect***(R_U, R_P) routine works as follows: R_U is scanned block by block in order to find a match between its basic block properties and those of the security policy blocks (lines 3-15 and lines 23-35). When a conditional block is encountered, if the condition is stamped, *i.e.*, if the condition has multiple children paths, ***Intersect*** is called recursively over each conditional path (lines 16-20). For the K^{th} execution, a temporary output file Reg_exp_K is created in order to store a string of all matching *bb*'s in R_P. In other words, in case of a match between BB_k in R_U and $bb_{k'}$ in R_P, **Intersect** outputs a $bb_{k'}$ into file Reg_exp_K for each bit $K = 1$ in BB_k.execution_id (lines 7,8 and lines 27,28). The string in each Reg_exp_K is then matched, using a standard string matching functionality, against the string representation of R_P's regular expression (lines 9,10 and lines 29,30). If a match occurs, a security **VIOLATION** flag is asserted. Therefore, this regular expression intersection technique allows us to detect any security violation dictated by the applied policy. Moreover, due to the locality of applying R_P, the exact points, where violations have taken place, can be inferred.

4.4 Checkpointing and Behavioral Model Generation Module

The checkpointing and behavioral model generation module allows for an efficient logging of the program's behavior, and re-usability of the

Table 7: High-level specification and regular expression based specification of buffer overflow security policies

Security policy specification in high-level language	Security policy translation into a regular expression format
A return instruction, which follows a buffer overflow and does not target the instruction after its corresponding call site, should not be allowed to execute	$P_1 = [bb_1.(bb_k, k \neq 2,3,4)*]*.[bb_2.(bb_k, k \neq 3,4)*]*.[bb_3.(bb_k, k \neq 4)*]*.bb_4$ bb_1: pushed return value at call site = Ret_1 *: Kleene closure $(bb_k, k \neq 2,...,n)$: any basic block not equal to subsequent security-critical *bb* bb_2: allocated memory (D/A/S) bb_3: overflow of memory allocated in bb_2 bb_4: return value corresponding to $bb_1 \neq Ret_1$
An execve system call, following an overflow of allocated memory on stack/heap/BSS segment should not be executed	$P_2 = [bb_1.(bb_k, k \neq 2,3)*]*.[bb_2.(bb_k, k \neq 3)*]*.bb_3$ bb_1: allocated memory (D/A/S) bb_2: overflow of memory allocated in bb_1 bb_3: execve/***execve.arg***$_1$ = overflown memory
A call to longjmp, which follows an overflow of allocated memory on stack/ heap/BSS segment and its restored env. does not match the one stored at the time setjmp is invoked should not be executed	$P_3 = [bb_1.(bb_k, k \neq 2,3,4)*]*.[bb_2.(bb_k, k \neq 3,4)*]*.[bb_3.(bb_k, k \neq 4)*]*.bb_4$ bb_1: stored environment at setjmp = ENV_1 bb_2: allocated memory (D/A/S) bb_3: overflow of memory allocated in bb_2 bb_4: environment restored at longjmp $\neq ENV_1$
A return instruction, which follows a buffer overflow and does not recover the pushed value of the old base pointer, should not be allowed to execute	$P_4 = [bb_1.(bb_k, k \neq 2,3,4)*]*.[bb_2.(bb_k, k \neq 3,4)*]*.[bb_3.(bb_k, k \neq 4)*]*.bb_4$ bb_1: pushed value of old base pointer $Base\_Ptr_1$ bb_2: allocated memory (D/A/S) bb_3: overflow of memory allocated in bb_2 bb_4: restored value of old base pointer $\neq Base\_Ptr_1$
An execution path, in which memory overflows beyond recovered frame pointer should not be allowed to execute	$P_5 = [bb_1.(bb_k, k \neq 2,3)*]*.[bb_2.(bb_k, k \neq 3)*]*.bb_3$ bb_1: allocated memory (D/A/S) bb_2: overflow of memory allocated in bb_1 bb_3: recovered frame pointer with overwritten registers
A function pointer, which follows a buffer overflow, and the backtracked value of the register holding the function pointer value has a different function address should flag a security violation	$P_6 = [bb_2.(bb_k, k \neq 1,3,4)*]*.[bb_3.(bb_k, k \neq 1,4)*]*.[bb_4.(bb_k, k \neq 1)*]*.bb_1$ bb_1: backtracked[1] register containing value of function pointer = $Func\_Ptr_1$ bb_2: allocated memory (D/A/S) bb_3: overflow of memory allocated in bb_2 bb_4: invoked function pointer value $\neq Func\_Ptr_1$
A call to a function corresponding to a GOT entry, which follows a buffer overflow, and the executed address conflicts with the dynamic relocation section address should not be allowed to execute	$P_7 = [bb_1.(bb_k, k \neq 2,3)*]*.[bb_2.(bb_k, k \neq 3)*]*.bb_3$ bb_1: allocated memory (D) bb_2: overflow of memory allocated in bb_1 bb_3: call to GOT entry function with overwritten entry point
A loop, whose condition depends on a variable following a buffer overflow should not be allowed to execute	$P_8 = [bb_1.(bb_k, k \neq 2,3)*]*.[bb_2.(bb_k, k \neq 3)*]*.bb_3$ bb_1: allocated memory (D/A/S) bb_2: overflow of memory allocated in bb_1 bb_3: a loop with an overwritten constraint value
A register or memory location containing the value of a user input and following a buffer overflow should flag a security violation	$P_9 = [bb_1.(bb_k, k \neq 2,3)*]*.[bb_2.(bb_k, k \neq 3)*]*.bb_3$ bb_1: allocated memory (D/A/S) bb_2: overflow of memory allocated in bb_1 bb_3: overwritten user-input

[1] When scanning the execution trace of an application, once a function pointer is being invoked, we can infer the value of the register in which the function pointer value has been initially declared. At that point, we can backtrack (reverse-scan) the register value in the logged execution trace.

Table 8: High-level specification and regular expression based specification of race conditions and memory vulnerabilities security policies

Vulns	Security policy specification in high-level language	Security policy translation into a regular expression format
R.C. L.A.	Inode, symbolic, and hard-link discrepancy between sys_call_1 and sys_call_2 of the event-pair <sys_call_1,sys_call_2> should not be executed	$P_{10} = [bb_1.(bb_k, k \neq 2)^*]^*.bb_2$ bb_1: file-specific system call sys_call_1, with file.inode = In_1, file.links = $Link_1$ bb_2: file-specific system call sys_call_2, with file.inode $\neq In_1$, file.links $\neq Link_1$
	Execution path discrepancy between the execution of two subsequent system calls	$P_{11} = [bb_1.(bb_k, k \neq 2)^*]^*.bb_2$ bb_1: file-specific system call sys_call_1, with file.path = $Exec\_Path_1$ bb_2: file-specific system call sys_call_2, with file.path $\neq Exec\_Path_1$
	Inode, symbolic, and hard-link discrepancy between subsequent usage of a tmp/ file should not be executed	$P_{12} = [bb_1.(bb_k, k \neq 2)^*]^*.bb_2$ bb_1: usage parameters of tmp/ file = $Usage_i$ bb_2: usage parameters of tmp/ file $\neq Usage_i$
M.V.	A dynamic/automatic allocated memory, which is freed twice, should not be allowed to execute	$P_{13} = [bb_1.(bb_k, k \neq 2,3)^*]^*.$ $[bb_2.(bb_k, k \neq 3)^*]^*.bb_3$ bb_1: allocated memory (D/A/S) bb_2: de-allocation of allocated memory bb_3: re-de-allocation of freed memory
	A dynamic/automatic allocated memory, which is never freed	$P_{14} = [bb_1.(bb_k, k \neq 2,3)^*]^*.$ $[bb_2.(bb_k, k \neq 3)^*]^*.bb_3$ bb_1: allocated memory (D/A/S) bb_2: de-allocation of allocated memory bb_3: end of execution trace and num(bb_2) $\neq$ num(bb_1)
	A dereferenced NULL-pointer	$P_{15} = bb_1$ bb_1: dereferenced NULL-pointer

tested program's path properties without the additional overhead imposed by scanning redundant and irrelevant information. While intersecting R_U with R_P, basic blocks of R_U matching any of R_P's security-critical basic blocks are marked as checkpoints.

Invariants at checkpoints are manipulated as follows:

- If the regular expression intersection results in a **VIOLATION** match, invariants at involved basic blocks BB_i, which contributed to the unacceptable program behavior, are formulated as unacceptable invariants (ref. Section 4.2.3).
- If the regular expression intersection did not result in a match, invariants at involved basic blocks BB_i are deemed potentially safe and formulated as acceptable invariants (ref. Section 4.2.3).
- If invariants at a checkpointed basic block do not reveal a security violation or do not confirm a permissible program behavior, they are removed from within the basic block.

Each BB_i, such that BB_i.condition = 1 (*i.e.*, BB_i = *BB_jump*), is also marked as a checkpoint in order to keep track of covered program execution paths, a path being the sequence of basic blocks executed following *BB_jump*. Checkpoints and invariants are then used by the behavioral model generation module, which groups checkpointed basic blocks into the behavioral model M that is to be migrated into the *Real* environment. M encapsulates, within its basic blocks, permissible (or non-permissible) real-time behavior of executing programs. This behavior is enforced by embedding suitable properties and flags within M's basic blocks. These are as follows:

1. **Flag = Ret_Value_Store**: These correspond to checkpointed call transfer basic blocks (*i.e.*, BB_i.call_site $\neq$ NULL). The flag forces the run-time monitor to store value Ret_i of the pushed return address and value $base_i$ of the pushed base pointer address. No invariants are stored at such blocks.
2. **Flag = Env_Store**: These correspond to checkpointed basic blocks, such that BB_i.setjmp_buf $\neq$ NULL. The flag forces the run-time monitor to store value F_pt_i of the frame pointer address. No invariants are stored at basic blocks with flag **Env_Store**.
3. **Flag = User_input_Store**: These correspond to checkpointed basic blocks, such that BB_i.user_input $\neq$ NULL. The flag forces the run-time monitor to store value $input_i$ of the user input value as it is being legitimitely modified (or unmodified) by the instructions within the basic block. No invariants are stored at basic blocks with flag **User_input_Store**.
4. **Flag = Sys_Call_Arg$_1$_Store**: These correspond to checkpointed basic blocks, such that BB_i.system_call.name $\in$ {sys_call_1}. The flag forces the run-time monitor to store the value of BB_i.system_call.arg$_1$, (*i.e.*, the file name F), as well as, F.directory, F.inode, and F.links. No invariants are stored at such blocks.
5. **Flag = Ret_Value_Verify**: These correspond to checkpointed BB_i, such that BB_i.ret_site $\neq$ NULL. BB_i corresponds to basic blocks with return instructions executing after a write operation to allocated memory. **Ret_Value_Verify** forces the run-time monitor to check (1) the value of popped return address Ret_i' against the fol-

Procedure *Intersect*: Intersection of regular expression R_U with security policy R_P

Inputs: regular expression R_U and security policy R_P
Output: detection of vulnerability exploit dictated by R_P

```
1.  while Index < number of basic blocks of R_U
2.    if R_U.BB_Index.condition = 1
3.        for index = 1 ... number of basic blocks of R_P
4.            if compare (R_U.BB_Index, R_P.bb_index)
5.                for index1 = 1 ... number of program executions
6.                        if R_U.BB_Index.execution_id & (one « index1)
7.                            open(concatenate("Reg_exp", index1), "app")
8.                            fprintf(concatenate("Reg_exp", index1), R_P.bb_index)
9.                            if MATCH content of concatenate("Reg_exp", index1) & R_P
10.                                   VIOLATION = 1
11.                           endif
12.                       endif
13.               end of for
14.           endif
15.       end of for
16.       if R_U.BB_Index.stamp = 1
17.           for i = 0 ... R_U.BB_Index.child_num
18.                   Interest(path[i], R_P)
19.               end of for
20.       endif
21.   endif
22.   else
23.       for index = 1 ... number of basic blocks of R_P
24.           if compare (R_U.BB_Index, R_P.bb_index)
25.               for index1 = 1 ... number of program executions
26.                       if R_U.BB_Index.execution_id & (one « index1)
27.                           open(concatenate("Reg_exp", index1), "app")
28.                           fprintf(concatenate("Reg_exp", index1), bb at index of R_P)
29.                           if MATCH content of concatenate("Reg_exp", index1) & R_P
30.                                   VIOLATION = 1
31.                           endif
32.                       endif
33.               end of for
34.           endif
35.       end of for
36.   if R_U.BB_Index.join_point = 1
37.       return
38.   endif
38.   Index = R_U.BB_Index.child[0]
39. end of while
```

lowing invariants: (i) Inv_1: $Ret_i' = C$ [C is a constant value, indicating a permissible behavior if its value is assumed by Ret_i'], (ii) Inv_2: $Ret_i' = C'$ [C' is a constant value, indicating a non-permissible behavior if its value is assumed by Ret_i'], (iii) Inv_3: $f(Ret_i')$: $Ret_i' = Ret_i$ [functional invariant indicating a permissible behavior if f holds], and (iv) Inv_4: $f(Ret_i')$: $Ret_i' \neq Ret_i$ [functional invariant indicating a non-permissible behavior if f holds], and (2) the value of popped frame pointer $base_i'$ against the following invariants: (i) Inv_1: $f(base_i')$: $base_i' = base_i$ [invariant indicating a permissible behavior if f holds], and (ii) Inv_2: $f(base_i')$: $base_i' \neq base_i$ [invariant indicating a non-permissible behavior if f holds].

6. **Flag = Fct_Ptr_Verify**: These correspond to basic blocks that invoke a function pointer call after a write operation to allocated memory. **Fct_Ptr_Verify** forces the run-time monitor to check the value of invoked function pointer Fct_Ptr against the following invariants: (i) Inv_1: $Fct\_Ptr = C$ [C is a constant value, indicating a permissible behavior if its value is assumed by Fct_Ptr], (ii) Inv_2: $Fct\_Ptr = C'$ [C' is a constant value, indicating a non-permissible behavior if its value is assumed by Fct_Ptr].
7. **Flag = GOT_entry_Verify**: These correspond to basic blocks that invoke a function call, which corresponds to a GOT entry, after a write operation to allocated memory. **GOT_entry_Verify** forces the run-time monitor to check the value of invoked function entry-point Fct_EP against the following invariants: (i) Inv_1: $Fct\_EP = C$ [C is a constant value (corresponding to the correct value of the function entry-point as retrieved from the dynamic relocation section of the executed program), indicating a permissible behavior if its value is assumed by Fct_EP], (ii) Inv_2: $Fct\_EP = C'$ [C' is a constant value, indicating a non-permissible behavior if its value is assumed by Fct_EP].
8. **Flag = LOOP_Verify**: These correspond to basic blocks that contain a loop, whose constraints depend on array *VAR*. **LOOP_Verify** forces the run-time monitor to check that the locations of the variables in *VAR* are not overwritten by an overflow of buffer *B*. The invariant stored at this basic block is (i) Inv_1: $B.len < L$ [C is a constant value (corresponding to the maximum length allowed for *B* without the risk of compromising any of the variables in *Var*), indicating a permissible behavior if its value is assumed by $B.len$].
9. **Flag = User_input_Verify**: These correspond to checkpointed BB_i, such that BB_i.user_input $\neq$ NULL and BB_i contains writing to a buffer allocated in memory and BB_i does not include a legal modification of the user input within its instruction after the memory write. This flag forces the run-time monitor to check the stored user input value $input_i'$: (i) Inv_1: $f(input_i')$: $input_i' = input_i$.
10. **Flag = Env_Verify**: These correspond to checkpointed BB_i, such that BB_i.longjmp_buf $\neq$ NULL and BB_i invokes function longjmp after a write operation to allocated memory. This flag forces the run-time monitor to check the popped value of frame pointer F_pt_i' and other environment values, such as return instructions addresses, and local variables against the following invariants: (i) Inv_1: $f(F\_pt_i')$: $F\_pt_i' = F\_pt_i$ and $\{\text{env}_i.\text{value} = C_i\}_{i=2...\#env.values}$, (ii) Inv_2: $f(F\_pt_i')$: $F\_pt_i' = F\_pt_i$ and $\{\text{env}_i.\text{value} = Range_i\}_{i=2...\#env.values}$ [$Range_i$ is a range of permissible values that can be assumed by the environment values].
11. **Flag = Sys_Call_Arg$_1$_Verify**: These are checkpointed BB_i, such that BB_i.system_call.name $\in$ {execve, sys_call$_2$}. This flag forces the run-time monitor to check BB_i.system_call.arg$_1$ against a set of functional and constant invariants, which assume one of the following values: (i) Inv_1/BB_i.system_call.name = execve: BB_i.system_call.arg$_1$ = C, (ii) Inv_2/BB_i.system_call.name = execve: BB_i.system_call.arg1 $\neq C'$, (iii) Inv_3/BB_i.system_call.name $\in$ {sys_call$_2$ with file_name F}: $f(F)$: F.directory$'$ = F.directory and F.inode$'$ = F.inode and F.links$'$ = F.links.
12. **Flag = tmp_file_Verify**: These correspond to checkpointed BB_i, such that BB_i contains an instruction that creates a temporary file, **tmp_file**. **tmp_file_Verify** forces the run-time monitor to check **tmp_file** against the invariant stored in M, namely, Inv_1: **tmp_file**.links = 0;
13. **Flag = Null_Ptr_Verify/dbl_free_Verify**: These correspond to checkpointed BB_i, in which a NULL pointer dereferencing or a double-free operation occur. Both flags, when encountered, force the run-time monitor to halt a program's execution.

Checkpointing and the extraction of the behavioral model M reduce the number of basic block invariants and the required storage space (as compared to the number of basic blocks invariants and storage space required by R_U) by an average of 73.51% and 61.64%, respectively, for the benchmark programs considered in our experiments.

5 Details: *Real* Environment

In this section, we describe the architecture and operation of our tool in the *Real* execution environment. The monitoring mechanism used in the *Real* environment is delineated into two inter-dependent components, used for exploit detection of previously tested execution paths based on behavioral model M and for restricting execution of newly encountered paths, if any, through the application of highly restrictive (conservative) security policies. Both components, embedded in the run-time monitor, prevent malicious attacks in the *Real* environment. Section 5.1 describes how run-time monitoring is performed based on behavioral model M that is migrated from the *Testing* environment. Section 5.2 discusses detection of attacks in newly encountered execution paths.

5.1 Run-time Monitoring Based on Behavioral Model M

The real-time monitoring and exploit prevention system is presented in Figure 5. It starts by loading the application's behavioral model M that is migrated from the *Testing* environment (Step S_1 in Figure 5). During program execution, the monitor automatically delineates basic block boundaries (Step S_2). When a new basic block is identified, its run-time loaded address is transformed into the standardized address format (generated as discussed in Section 4.2.1), which is then used as an input, along with the address of the so-far scanned *BB* in M (denoted BB_M), to the address checker module. This module asserts signal **Checkpointed_*BB*** in case the two standardized addresses match (step S_3). If no such signal is asserted, the instrumentation of instructions in the *BB* is suspended until a new *BB* is identified. On the other hand, if **Checkpointed_*BB*** is asserted, Step S_4 (1) checks if *BB* contains a conditional control transfer, in which case the standard-form addresses of the first *BB* in the executed path and the related join-point are stored in registers *Reg* and Reg_1, respectively, and signal **Cond_Flag** is asserted, (2) instruments the instructions within the *BB* until the end of the basic block is identified. Each instruction is checked against properties and invariants of BB_M. Once checks are cleared (*i.e.*, signal **Valid_checks** is asserted), the instruction is allowed to execute and commit, otherwise the program's execution is suspended and the *Real* environment enters an **Invalid** state. The checks performed at each instrumented instruction are based on the information stored at BB_M and are as follows:

- Store value of return address and old base pointer if BB_M is flagged as **Ret_Value_Store**.
- Store specified environment values if BB_M is flagged as **Env_Store**.
- Store user input values and corresponding legitimate modifications if BB_M is flagged as **User_input_Store**.
- Store value of specified system call arguments if BB_M is flagged as **Sys_Call_Arg$_1$_Store**.
- Map selected values against invariants stored at BB_M's flagged with **Ret_Value_Verify**, **Fct_Ptr_Verify**, **GOT_entry_Verify**, **LOOP_Verify**, **User_input_Verify**, **Env_Verify**, **Sys_Call_Arg$_1$_Verify**, **tmp_file_Verify**, and **Null_Ptr_Verify/ dbl_free_Verify**.

5.2 Handling New Execution Paths

In the case where signal **Cond_Flag** is asserted, the path checking engine identifies whether the taken path has been previously tested, and asserts signal **New_path** accordingly. If **New_path** remains unasserted, the monitor switches to S_3 and follows the sequence outlined in Section

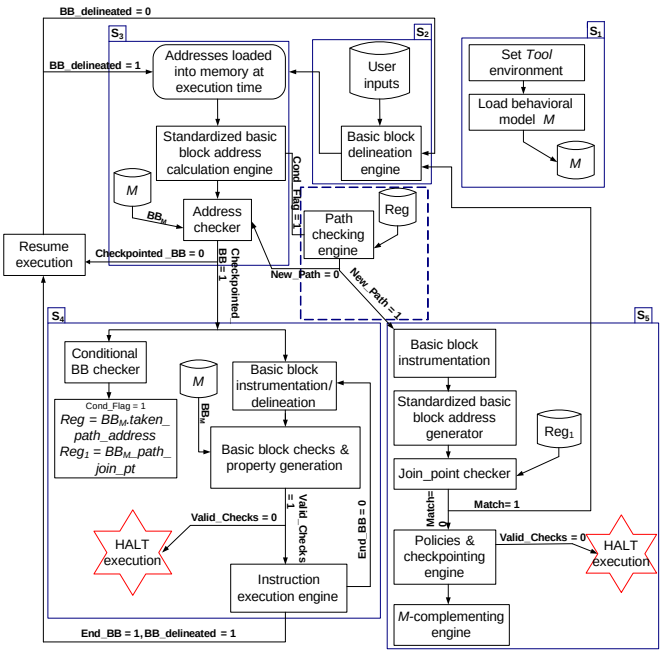

Figure 5: Monitoring and exploit prevention system in the *Real* environment

5.1. On the other hand, if **New_path** is asserted, the monitor jumps into S_5 in order to prevent any attack that might exploit vulnerabilities in the new executed path. In S_5, the new path instrumentation proceeds until the join_point checker matches the standard-form *BB* address with the content of Reg_1. At that point, signal **Match** is asserted and the monitor resumes execution in S_2. During instrumentation of the new path, a set of highly-restrictive policies (listed in Table 9) is enforced at run-time before any instruction completes execution. (This part of our work uses a similar concept to the one in [13]). While those policies would eventually result in some false positives, they keep the overhead in the *Real* environment low and prevent the execution of malicious code.

Similar to Section 4.4, basic blocks in new executed paths, to which security policies were applied, are checkpointed and added to behavioral model *M* once **Match** is asserted (M-complementing engine).

Table 9: List of restrictive policies built for preventing malicious vulnerability exploits in new executed paths

Restricted-execution instruction	**Restrictive policies**
Inst/Function returns	Not allowed to execute if corresponding call value has not been stored
	If call values have been stored, execution is allowed only if return is to after call values
Frame pointer	Not allowed to execute if corresponding pushed base pointer value has not been stored
	If pointer value has been stored, execution is allowed only if it restores the correct pointer
Function pointer call	Not allowed to execute, if following write to allocated memory
GOT entry call	Not allowed to execute, if following write to allocated memory
Loop iteration	Not allowed if constraint value follows write to allocated memory
longjmp call	Not allowed to execute, if following write to allocated memory
execve	Not allowed to execute
sys_call_2	Not allowed to execute, if checks at sys_call_1 have not been performed
	If checks at sys_call_1 have been performed, execution is allowed only if sys_call_2.args comply with sys_call_1.args
tmp_file creation	Allowed to execute only if tmp_file.links = 0

6 Experimental Evaluation

In this section, we present experimental results obtained through extensive evaluation of our approach in both the *Testing* and *Real* environments. For ease of presentation and comparison, results are reported separately for each environment. Overheads for each modular component, detection and prevention rates, and false alarms are reported in each environment. Section 6.1 reviews the experimental set-up, and Section 6.2 details the experiments performed and presents the corresponding results.

6.1 Experimental Set-up

The experimental set-up used in our experiments for evaluating the execution time overheads and the efficacy of our tool is as follows. The system used is a 3.2GHz Intel Pentium IV PC running XenLinux 2.6. Xen [20] is an open-source VMM supporting the execution of multiple OSs. Since it can securely execute multiple VMs, we use Xen to build two virtual domains, executed in isolation, in order to set up the *Testing* and *Real* execution environments. Xen's domain 0, which is a privileged management domain, is used in order to create both domains, and securely migrate information (behavioral model *M*), through Xen's S-Hype security hooks, between them. Memory is hard-partitioned between the two domains, domain 0, and Xen itself. Moreover, since the two domains do not need to be active concurrently, the deterioration in the processor speed (as compared to a domain running alone) is negligible. We developed our approach based on PIN [21], a dynamic binary instrumentation framework from Intel, which provides a rich API for building a variety of instrumentation tools. For the purpose of our experiments, we selected applications with known software vulnerabilities, reported in security advisories such as SECUNIA [1], CERT [2], *etc.* These applications were subjected to a large set of input combinations, some of which were specially-crafted in order to trigger an exploit of those flaws, many of which constitute serious security holes. On the other hand, we also used our own benchmarks and attack data sets, which develop each of the attacks identified in this paper. We executed the attacks randomly within both the *Testing* and *Real* environments, using specially crafted inputs (for buffer overflows and memory vulnerabilities) or injecting attack code in parallel with the program's execution (race condition attacks). Further details about test input vectors and evaluation results are reported in Sections 6.2.2 and 6.2.3.

6.2 Performance Measurements

This section discusses the different benchmark programs, as well as the attack data sets we have built, with a complete explanation of the vulnerabilities embedded within each one (Section 6.2.1). It then describes the different overheads and efficacy figures of our tool in both *Testing* and *Real* environments (Sections 6.2.2 and 6.2.3, respectively).

6.2.1 Benchmarks and Attack Data Sets

The benchmarks and attack data sets used for our experimental evaluations are as follows: (Vulnerabilities in the following open-source benchmarks are collected from [1, 2, 3, 33, 34]).

bzip2: Race condition vulnerability (between creating a new file and setting its permissions).

flex: Buffer overflow vulnerability (triggered using specially-crafted input, usually of length greater than 2190).

gawk: Buffer overflow vulnerability (triggered by providing long input sequences (>8204 input parameters)).

gpm: Temporary-file race condition attack (possible if an attacker creates a symbolic link to the temporary files of predictable names in the tmp/ folder).

gzip: Buffer overflow vulnerability (triggered by providing specially-crafted input file names (> 1020 input parameters)).

gzip: Race condition vulnerability (exploitable through a symbolic link of a decompressed file to a root-owned file).

nasm: Buffer overflow (triggered by a malicious assembler source file).

ncompress: Buffer overflow vulnerability (triggered by long input sequences (>1023 input parameters)).

rm: Directory redirection race condition attack in recursive calls of the rm-gnu utility.

sdiff: Temporary-file race condition vulnerability (triggered by a symbolic link from a guessed file name to a critical file such as /etc/pwd).
unzip: Race condition vulnerability (exploited by repeatedly running an attack code, which tries to create a symbolic link between the output file and the system password file).
zlib: Double-free vulnerability (No exploits of this vulnerability have been reported, however, through a backward analysis of the source code of zlib, we could craft an input data stream that triggers it).
zlib: Buffer overflow vulnerability (exploitable through specially-crafted input streams).
zoo: Buffer overflow vulnerability (triggered by providing the zoo archiver with long input file names).
mpg123: Buffer overflow vulnerability (triggered by providing specially-crafted remote input of length greater than 1040 characters). This benchmark also contains a double-free memory vulnerability.
ghttpd-1.4.3: Buffer overflow vulnerability (triggered by providing specially-crafted input of length greater than 255 characters, the attack is taken from [28]).
OpenSSH: Buffer overflow vulnerability (triggered by providing specially-crafted input of length greater than 8192 characters).
Buffer overflow benchmarks: We developed an attack data set of 24 benchmarks *BM*1-*BM*24, which cover the different forms of control-data and non-control-data buffer overflow attacks (*i.e.*, considering all locations, techniques and attacks target presented in Section 4.3.1). 20 benchmarks were provided by John Wilander [27]. Benchmarks stimulating buffer overflow attacks targeting (1) a register containing a pointer to the code executed by *execve*, (2) decision making data, (3) user input data, and (4) GOT entries values are added to the data set.
Race condition benchmarks: We developed an attack data set *BM*25-*BM*33 simulating the following attacks: <access-open>, <open, chmod>, <open-chown>, <rename-chown>, <stat-open>, <open-open>, <chdir, rmdir> and <mkdir, chmod> (directory redirection attack), and temporary-file race condition.
Memory vulnerability benchmarks: *BM*34, *BM*35 consisting of: (1) double-free vulnerability attack, and (2) NULL-pointer dereferencing attack.

6.2.2 Execution Time Overheads

A. *Testing* environment: The key parameters in the *Testing environment* that contribute to its execution time overhead are: (i) t_1, overhead due to binary instrumentation, (ii) t_2, overhead due to the regular expression module, (iii) t_3, overhead due to the detection module, and finally (iv) t_4, the overhead caused by the checkpointing and behavioral model generation module. Table 10 reports the overheads for different open-source benchmarks. Column 1 corresponds to the executed benchmark (B.M.), column 2 lists the number of basic blocks in R_U (# *BB*) extracted for each benchmark. Columns 3-6 report the total time required by each of the overhead categories listed above. Reported overheads are based on an average of 40 executions per benchmark using the automatically generated input vectors. Column 7 reports the total overhead (T.O.) induced by all modules (as compared to running the benchmarks without any instrumentation and checkings). Column 8 lists one of the policies from Table 7 or 8 that was able to detect a security violation in the benchmark, and column 9 gives the number of the basic block, at which a security violation was detected.

As can be observed from Table 10, the overhead induced in the *Testing* environment is quite significant (an average of 33.01X). However, we can argue that this overhead is acceptable since it happens transparently to the user, while subjecting the application to rigorous security checks.

Note that we do not include the attack data set benchmarks (*BM*1-*BM*35) in the execution time analysis, since their machine code is modular and they are not representative of most real system applications. Therefore, their inclusion in the time analysis prevents an exact evaluation of the tool's time overhead. However, since the benchmarks represent real attack scenarios, they are included in the evaluation of the detection and prevention rates achieved by our tool (Section 6.2.3).
B. *Real* environment: Performance in the *Real* environment is evaluated while the run-time monitor is running in parallel with the program,

Table 10: Execution time overhead in the *Testing* environment

B.M.	# *BB*	t_1 (sec.)	t_2 (sec.)	t_3 (sec.)	t_4 (sec.)	T.O. (X)	*Pol.*	BB #
bzip2-1.0.2	1276	0.72	8.015	0.19	0.05	18.81	P_{10}	836
flex-2.5.31	14374	20.51	242.69	2.21	0.41	52.96	P_5	7572
gawk-3.1.2	12827	17.87	221.75	1.83	0.35	60.45	P_4	12382
gpm-1.19.3	3906	0.65	4.03	0.52	0.12	26.45	P_{10}	2080
gzip-1.2.4	2178	0.78	8.29	0.32	0.08	31.56	P_4 P_{10}	1359 1927
nasm-0.98.38	2852	0.94	11.37	0.44	0.10	42.8	P_5	1518
ncompress-4.2.4	1290	0.62	7.20	0.17	0.03	53.4	P_4	655
rm -R	12252	18.63	251.31	1.62	0.33	43.28	P_{11}	10387
sdiff-2.7	1507	0.59	5.83	0.22	0.04	23.69	P_{12}	404
unzip-5.50	2853	0.78	9.15	0.45	0.07	34.83	P_{10}	1580
zlib-1.1.3	12416	11.8	130.27	2.21	0.34	25.63	P_{13}	11360
zlib-1.2.2	12808	13.62	160.05	1.81	0.34	31.38	P_4	9743
zoo-2.10	2192	0.58	3.82	0.32	0.06	15.86	P_1	1347
mpg123	3495	1.03	5.41	0.48	0.08	17.25	P_7 P_{13}	3622 4829
ghttpd	6289	8.39	63.09	0.76	0.16	21.67	P_9	3391
OpenSSH	15191	31.72	294.81	2.92	0.43	28.11	P_8	13881

and performing various checks against behavioral model *M* and the list of restrictive security policies given in Table 9. For evaluation purposes, we divided the testing procedure into two scenarios: (1) application is executed with user inputs already tested in the *Testing* environment, (2) application is executed with user inputs not tested in the *Testing* environment and which exercise new execution paths. We divide the execution time overhead in the *Real* environment into three categories: (i) t_1, overhead due to binary instrumentation, (ii) t_2, overhead of checking against behavioral model *M*, and finally (iii) t_3, overhead of testing a new execution path and rebuilding *M*. Table 11 reports an average (over 32 executions) of the different overhead time components (t_1, t_2, and t_3) for different benchmarks (B.M. in column 1) in columns 3-5. Column 2 lists the number of basic blocks checked at run-time (*i.e.*, number of basic blocks in *M*). Column 6 reports the total overhead (T.O.) in the *Real* environment as compared to running a standalone benchmark without any security prevention measurements applied to it. Column 7 reports the overhead reduction (O.R.) between the *Testing* and *Real* environments. A 1.43X average overhead is imposed by the monitoring module in the prevention mode (maximum of 2.75X and minimum of 1.09X). This overhead increases as a function of the number of conditional basic blocks in *M*. It is quite minimal and overcomes the high slowdowns imposed in the *Testing* environment (by an average of 22.93X).

6.2.3 Effectiveness Results

In order to evaluate the effectiveness of our approach in detecting multiple forms of software vulnerability exploits in both the *Testing* and *Real* environments, three main issues have to be taken into consideration: (A) **Detection and prevention locality**, *i.e.*, the capability to point out the exact location where the security violation has taken place; (B) **Security exploit detection and false alarm rates** in the *Testing* environment; and (C) **Security exploit prevention and false alarm rates** in the *Real* environment.
A. Testing locality: In addition to detecting and preventing vulnerability exploits, a feature that our work provides is **Testing locality**, which basically means that our approach can point out the exact security violation location in the application's execution scope. This can definitely be useful for debugging and secure coding practices.

B. Detection and false alarm rates in the *Testing* environment: We

Table 11: Execution time overhead in the *Real* environment

B.M.	# *BB*	t_1 (sec.)	t_2 (msec.)	t_3 (msec.)	T.O. (X)	O.R. (X)
bzip2-1.0.2	895	0.49	27.45	n/a	1.09	17.34
flex-2.5.31	6253	13.08	752.12	n/a	2.75	19.22
gawk-3.1.2	6430	6.41	368.59	n/a	1.68	35.67
gpm-1.19.3	1860	0.27	14.6	n/a	1.41	18.69
gzip-1.2.4	990	0.33	54.87	n/a	1.27	24.61
nasm-0.98.38	1240	0.36	52.22	n/a	1.39	31.17
ncompress-4.2.4	608	0.17	25.26	n/a	1.33	41.07
rm -R	6844	10.84	872.14	88.97	1.81	23.04
sdiff-2.7	1004	0.28	46.63	n/a	1.18	20.45
unzip-5.50	2194	0.35	36.95	n/a	1.29	27.02
zlib-1.1.3	6711	6.31	459.70	n/a	1.22	21.36
zlib-1.2.2	7269	6.72	512.16	n/a	1.29	24.31
zoo-2.10	1118	0.31	16.12	15.6	1.11	14.26
mpg123	1889	0.50	63.29	n/a	1.39	12.43
ghttpd	2845	4.08	81.93	n/a	1.24	17.40
OpenSSH	7268	16.78	691.57	n/a	1.48	18.88

have evaluated the effectiveness of our tool in the *Testing* environment by running it on the benchmark programs of Section 6.2.1. Table 10 shows a 100% detection for all open-source benchmarks, across all considered vulnerabilities, and a 100% detection rate is obtained for the attack data sets we have developed (100% for buffer overflows, 100% for race conditions, and 100% for memory vulnerabilities). Since policies in the *Testing* environment are vulnerability-specific, and represent the behavioral and data flow of a definite vulnerability exploit, and since execution of a program is permitted to complete without making any assumptions about the safety of an instruction before it executes, there are no false alarm instances (positives or negatives) in the detection mode.

C. Detection and false alarm rates in the *Real* Environment: Behavioral models extracted in the detection mode, in addition to the restrictive security policies of Table 9, successfully halted the exploit of any code vulnerabilities in the *Real* environment or prevention mode. We have tested our approach in the *Real* environment using a set of input vectors that was used in the detection mode and another set of inputs that was not tested.

Experiments in the *Real* environment resulted in new execution paths only 5.8% of the time. No false positives were encountered. However, due to the nature of the restrictive policies imposed on the new execution paths, we expect a small percentage of false positives to occur if spawned by adequate user-input combinations.

7 Conclusion

In this paper, we have introduced an efficient and scalable approach for the detection and prevention of software vulnerability exploits, that is based on the use of two isolated environments and dynamic binary instrumentation. We have shown that our tool successfully detects and thwarts run-time attacks, through the design of appropriate security policies. We applied the tool to detect several software vulnerabilities as a proof of concept for the efficacy of the proposed approach. The use of an abstracted behavioral model for monitoring in the *Real* execution environment results in acceptable performance penalty incurred on the user at run-time.

We believe that this is a practical and scalable way to address a wide range of software attacks. Our proposed work currently implements different policies for preventing vulnerability exploits in code that is not malicious in intent. However, future work includes its expansion in order to be able to detect malicious programs, such as viruses, worms, *etc*.

Acknowledgment: We thank John Wilander for his assistance and providing source code of the attack benchmarks presented in [27].

References

[1] "Vulnerabilities and Virus information." [Online]. Available: http://secunia.com

[2] "Computer Emergency Readiness Team." [Online]. Available: http://www.cert.org

[3] "Common Vulnerabilities and Exposures." [Online]. Available: http://cve.mitre.org

[4] K. Petkove, "Overcoming programming flaws: Indexing of common software vulnerabilities," in *Proc. Conf. Information Security Curriculum Development*, Sep. 2005, pp. 127–134.

[5] "Common Vulnerability Scoring System." [Online]. Available: http://www.first.org/cvss/cvss-guide.html

[6] "FindBugs - Find Bugs in Java Programs." [Online]. Available: http://findbugs.sourceforge.net

[7] "Flawfinder." [Online]. Available: http://www.dwheeler.com/flawfinder

[8] "CodeSurfer." [Online]. Available: http://www.grammatech.com/products/codesurfer

[9] G. Vigna, *Static Disassembly and Code Analysis*. Springer-US, 2007.

[10] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proc. Annual Network and Distributed Systems Security Symp.*, Feb. 2005.

[11] H. J. Wang, C. Guo, D. Guo, D. Simon, and A. Zugenmaier, "Shield: Vulnerability-driven network filters for preventing known vulnerability exploits," in *Proc. Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications*, Aug. 2004, pp. 193–204.

[12] B. Horne, L. Matheson, C. Sheehan, and R. E. Tarjan, "Dynamic self-checking techniques for improved tamper resistance," in *Proc. ACM Wkshp. on Security and Privacy in Digital Rights Management*, Nov. 2001, pp. 141–159.

[13] V. Kiriansky, D. Bruening, and S. Amarasinghe, "Secure execution via program shepherding," in *Proc. USENIX Security Symposium*, Aug. 2002, pp. 191 – 206.

[14] "The Peach Fuzzer Framework." [Online]. Available: http://peachfuzz.sourceforge.net

[15] "Scapy." [Online]. Available: http://www.secdev.org/projects/scapy

[16] R. Gupta, M. L. Soffa, and J. Howard, "Hybrid slicing: Integrating dynamic information with static analysis," *ACM Trans. on Software Engineering and Methodology*, vol. 6, pp. 370–397, Oct. 1997.

[17] Z. Liu, S. M. Bridges, and R. B. Vaughn, "Combining Static Analysis and Dynamic Learning to Build Accurate Intrusion Detection Models," in *IEEE Int. Wkshp. on Information Assurance*, Mar. 2005, pp. 164–177.

[18] N. Provos and T. Holz, *Virtual Honeypots: From Botnet Tracking to Intrusion Detection*. Addison-Wesley, 2007.

[19] "Intel vPro Processor Technology." [Online]. Available: http://www.intel.com/business/vpro/

[20] "XenSource: Delivering the Power of Xen." [Online]. Available: www.xensource.com

[21] "Pin - A Dynamic Binary Instrumentation Tool." [Online]. Available: http://rogue.colorado.edu/pin

[22] "STP: A decision procedure for bitvectors and arrays." [Online]. Available: http://theory.stanford.edu/ vganesh/stp.html

[23] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: Automatically generating inputs of death," in *Proc. ACM Conf. on Computer and Communications Security*, Nov. 2006, pp. 322–335.

[24] "Automatically identifying trigger-based behavior in malware." [Online]. Available: http://bitblaze.cs.berkeley.edu/papers/botnet_book-2007.pdf

[25] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *Proc. Int. Conf. Software Engineering*, May 2002, pp. 291–301.

[26] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," in *Proc. Int. Conf. on Software Engineering*, May 1999, pp. 213–224.

[27] J. Wilander and M. Kamkar, "A comparison of publicly available tools for dynamic buffer overflow prevention," in *Proc. Network and Distributed System Security Symp.*, Feb. 2003.

[28] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *Proc. Conf. Usenix Security Symp.*, Jul. 2005.

[29] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *Proc. Operating systems design and implementation*, Jul. 2006, pp. 147–160.

[30] C. Parampalli, R. Sekar, and R. Johnson, "A practical mimicry attack against powerful system-call monitors," in *Technical Report SECLAB07-01, Secure Systems Laboratory, Stony Brook University*, 2007.

[31] "Time of check, time of use race condition." [Online]. Available: http://www.owasp.org

[32] "Valgrind." [Online]. Available: http://valgrind.org

[33] "Internet Security Systems - Research." [Online]. Available: http://xforce.iss.net

[34] "Linux Weekly News." [Online]. Available: http://lwn.net