DESIGN, IMPLEMENTATION AND EVALUATION OF A

SYMBOLIC N-VARIANT SIMULATOR

A Thesis in
Information Sciences and Technology
by
Pinyao Guo

© 2016 Pinyao Guo

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

May 2016
The thesis of Pinyao Guo was reviewed and approved* by the following:

Peng Liu
Professor of Information Sciences and Technology
Thesis Advisor

Anna Squicciarini
Assistant Professor of Information Sciences and Technology

Dinghao Wu
Assistant Professor of Information Sciences and Technology

Carleen Maitland
Chair of Graduate Program

*Signatures are on file in the Graduate School.
Abstract

Artificial diversity is an approach aims to increase the cost for attackers to compromise software systems by randomizing implementation properties of software. However, its protection depends heavily on maintaining secrecy. N-variant system addresses this problem by executing a set of diversified software variants on the same input, and monitors their behaviors to detect divergence. It exhibits appealing security properties of high assurance detection for certain class of attacks without relying on secrets. However, requiring large amount of extra computing resources and extra time hampers its practicality in real world scenarios.

In this thesis\(^1\), we focus on reducing the cost for deploying N-variant system, yet still maintaining its security advantages. As a popular program analysis and testing technique, symbolic execution characterizes program input and the part of program which the input causes to execute. We observe that such characterization can be used to simulate actual program execution efficiently. Based on this observation, we propose a symbolic N-variant system framework as a new approach to build N-variant system. After introducing artificial divergences to software, we symbolic execute each application variants to generate program summaries. During runtime, instead of executing input in the real program, we simulate the execution in a symbolic N-variant simulator, and detect behavior divergences.

We build a prototype based on the proposed system framework. This thesis emphasizes on the design, implementation and evaluation of the symbolic N-variant simulator, as part of the whole system. We also evaluate the prototype in terms of run-time performance and resource consumption, and the effectiveness of detecting attacks of certain class.

\(^1\)The thesis project is collaborated with Jun Xu (jxx13@psu.edu). Jun Xu is mainly responsible for the design and implementation of SAV Generator (Section 3.2.2) and security effectiveness evaluation (Section 6.2).
Chapter 5
Simulation Performance Optimization 26
5.1 Caching Parsing Result 27
5.1.1 Intra-Simulation Cache 27
5.1.2 Inter-Simulation Cache 28
5.2 Input Request Filtering 29
5.3 Symbolic Path Search Tree Pruning 32

Chapter 6
Prototype Evaluation 34
6.1 Efficiency Evaluation 34
6.1.1 Simulation versus Execution Performance 35
6.1.2 Performance for Serving Multiple Applications 36
6.1.3 Performance for Different Tree Size 37
6.1.4 Memory Consumption 37
6.1.5 Performance Improvement from Speedup Heuristics 39
6.2 Security Effectiveness Evaluation 40

Chapter 7
Discussion and Future Work 42
7.1 Incomplete Symbolic Paths 42
7.2 Scalability 43
7.3 Future Work 43

Chapter 8
Related Work 44
8.1 Artificial Diversity 44
8.2 Multi-Variant system 45

Chapter 9
Conclusion 46

Bibliography 47
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Cox et al. N-Variant System Framework</td>
</tr>
<tr>
<td>3.1</td>
<td>Symbolic N-Variant System Architecture.</td>
</tr>
<tr>
<td>3.2</td>
<td>An example of search tree building.</td>
</tr>
<tr>
<td>4.1</td>
<td>SNV Simulator design.</td>
</tr>
<tr>
<td>4.2</td>
<td>V-simulator structure.</td>
</tr>
<tr>
<td>5.1</td>
<td>Example symbolic path search tree with repeated predicate cause by looping.</td>
</tr>
<tr>
<td>5.2</td>
<td>Flowchart for processing input request with filtering system.</td>
</tr>
<tr>
<td>5.3</td>
<td>Identifying benign symbolic paths by simulating test cases for each symbolic path.</td>
</tr>
<tr>
<td>6.1</td>
<td>Performance comparison under different concurrency level between lighpptd and symbolic N-variant system with different number of v-simulators.</td>
</tr>
<tr>
<td>6.2</td>
<td>Performance of symbolic N-variant system serving multiple applications under concurrency level of 48 (saturated).</td>
</tr>
<tr>
<td>6.3</td>
<td>Simulator performance comparison between different search tree size under concurrency level of 48 (saturated).</td>
</tr>
<tr>
<td>6.4</td>
<td>Memory consumption for different number of variants and different tree size under concurrency level of 48 (saturated).</td>
</tr>
<tr>
<td>6.5</td>
<td>Performance improvement after deploying two caching table in our system.</td>
</tr>
<tr>
<td>6.6</td>
<td>Statistical percentages for processing input request of the whole system with filter.</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Example of a symbolic application. ................................. 12

4.1 Statistics of path exploration of lighttpd (Size: symbolic input size; Path #: average number of symbolic path explored; Time: average time elapsed for path exploration.) ................................. 17

5.1 Example of intra-simulation caching table. ....................... 28
5.2 Example of inter-simulation caching table. ....................... 29

6.1 Comparison between number of parsing during simulation with or without caching. ................................................. 39
6.2 Effectiveness of symbolic N-Variant system against attack cases .. 41
Chapter 1

Introduction

The commoditization of commercial computer systems brought homogeneity with respect to computer software, which simplifies the logistics of software distribution and maintenance, and provides software users with consistent behaviors [1]. However, the homogeneity provides convenience for adversaries around cyberspace as well. Using an identical exploit, an attacker can probe a vulnerability in one software copy, and target on millions of computers that run the same distribution. For instance, recently disclosed Heartbleed vulnerability in 2014 affects around 17% of SSL web servers which use OpenSSL cryptography library [2], including major websites such as Amazon, Github, etc.

The risks of monoculture have been widely recognized in secure computing community [3]. The resilience nature provided by software diversity is gaining more attention in security research community. Recent research studied a set of promising strategies by introducing artificial diversity to software. Artificial diversity increases the difficulty and cost for attackers by introducing uncertainty into aspects of software implementation. Without the knowledge of specific implementation, time and energy required for attackers to breach the software increase significantly. Moreover, attackers are forced to target on specific software distribution individually, and thus, raise the bar for mass scale exploitation [1].

In the past decade, several techniques for intentionally introducing software diversity have been developed, including address space layout randomization (ASLR)
[4, 5], instruction set randomization [6, 7], data structure layout randomization (DSLR) [8], data randomization [9], etc.

However, the effectiveness of artificial diversity relies heavily on keeping diversity method (how the diversity of the running execution of the particular software is generated) as secret to attackers. Otherwise, attackers can craft customized malicious input for the software variant. Recent discovered location inference side channels [10, 11] show that the secret can be compromised by ROP attacks.

To mitigate this problem, Cox et al. [12] propose an N-variant system framework as shown in Figure 1.1. In the N-variant system, a polygrapher replicates external program input into different artificially diversified application variants. A monitor collects behaviors from each variant and detects divergences which reveal attacks. The assumption behind this framework is that: there must exist an execution pathway in one variant that exploits a vulnerability without producing anomalous behaviors in other variants. That is, as long as no single malicious input can simultaneously compromise all the variants, attacks could be detected, although every variant might be vulnerable. This framework shows appealing security advantages:

- The framework requires no secret keeping. Even if an attacker has complete knowledge on the diversity key used in each software variant, attacks could be detected as long as no pathway as described above exists.

- By using some diversification techniques, the framework provides deterministic detection of attacks of certain class.

However, despite the security merits it offers, a dominant barrier for wide de-
ployment of N-Variant systems is its inherent costly nature:

- The extra computing resources required for N-Variant system is \( N \) times greater than the original software system to be protected.
- The performance for malicious input detection must be slower than an execution in the original software system, since it requires a synchronization and synopsis phase to detect divergences.

In this thesis, we focus on addressing the cost problem for N-Variant system using symbolic execution. As a program analysis and testing technique, symbolic execution can be used to characterize program input and the part of program which the input causes to execute. We make an observation that such characterization can serve as a program summary and be used to simulate actual program execution efficiently. Based on this observation, we propose a symbolic N-variant system, as a new approach to build N-variant system.

Instead of executing actual software variants with input, we execute the execution in symbolic variants. After introducing artificial divergences to software variants, we symbolic execute each program variants. The symbolic execution generates program summaries which consist of constraint predicates on input, and program behaviors along corresponding program part. With the program summaries of a variant, we build an innovative symbolic path search tree for this variant. During run-time, inputs are duplicated into different symbolic variants for simulation. Monitor collects behavior from each variant and detect divergences.

Our key contributions are described as follows.

- We develop a novel symbolic execution based approach to build N-variant systems. To the best of our knowledge, this work is the firstly designed applicable N-variant system.
- We implement a symbolic N-variant system prototype which consists of three parts: a server system, a symbolic application variants generator, and a symbolic N-variant simulator. The prototype is based on a commercial web server \texttt{lighttpd} and we target on a real vulnerability inside the program.
- We evaluate the prototype in terms of its efficiency and security effectiveness. The experiment results show that the average performance of our system is
still much faster than the real server when 10 symbolic variants are concurrently working under saturated workload. Also, our prototype can detect attacks of certain class without raising false positive or negative.

This thesis is inspired by a collaborative research project. We mainly focus on describe the author’s contribution on the design, implementation and evaluation of a symbolic N-variant simulator. The rest of the thesis is organized as follows. In Chapter 2, we provide a more detailed description of N-variant system, formalize the problem definition, and describe the threat model used in this thesis. In Chapter 3, we present an overview for the system model and architecture of our symbolic N-variant system framework. In Chapter 4, we present the design and implementation of a prototype symbolic N-variant simulator based on a popular application. We discuss some heuristics deployed in our design to speed up the simulation performance in Chapter 5. In Chapter 6, we evaluate the efficiency and security effectiveness of our prototype. We discuss some limitations and future work in Chapter 7, and offer some related work in Chapter 8. Chapter 9 concludes.
Problem Overview

2.1 N-Variant System

Contrasting to run an individual process of a diversified software system, N-Variant System contains multiple variants that are expected to have disjoint *exploitation sets*. Exploitation set is a collection of objects or context inside a program that attacker can leverage to exploit a vulnerability. The detection goal of Cox et al.’s N-variant system is under premise that for all attacks of a particular class, if one variant is compromised by a given exploit, another one or more variants must exhibit divergent behavior detectable by the monitor [12].

The execution model for N-variant system and two indispensable variant properties provide detection guarantee for certain class of attacks. As shown in Figure 1.1, a polygrapher duplicates external inputs into different diversified program variants \( \{V_0, V_1, ..., V_{N-1}\} \). A monitor captures all behaviors generated from each variant, and raise alert if variants behave inconsistently.

The execution of an input is viewed as a possibly infinite sequence of states: \([S_0, S_1, ...]\), and the state of a N-variant system can be represented by a tuple of the states of all variants. Hence, an execution of a N-variant system can be represented as a sequence: \( [< S_{0,0}, S_{0,1}, ..., S_{0,N-1} >, < S_{1,0}, S_{1,1}, ..., S_{1,N-1} >, ...]\), where \( S_{t,v} \) represents the state of variant \( v \) at step \( t \). Each variant \( v \) has a *canonicalization function*, \( C_v \), and a *transition function*, \( T_v \). \( C_v \) maps each state of \( v \) to a canonical state that matches the corresponding state for the original process; \( T_v \) transfers a state of \( v \) to the next state upon receiving an input. The states are partitioned
into three categories: normal, alarm, and compromised. When a variant behaves as intended, it is in normal state; when a variant is intruded by an attack, it is in a compromised state; when a variant behaves anomalously and is detected by the monitor, it is in an alarm state.

Two key properties are maintained by a N-Variant System as an attack detection system: normal equivalence and detection.

**Normal equivalence:**

\[ \forall t \geq 0, 0 \leq v < N, S_t \in \text{Normal} : C_v(S_{v,t}) = S_t. \]  

(2.1)

This property guarantees consistency and correctness of a N-Variant System when no attack happens. Upon receiving normal input, all variants should be in normal states and their states must correspond to the same canonical state. In addition, the canonical state of each variant should be identical with the state of the original process.

**Detection:**

\[ \forall S \in \text{Normal}, 0 \leq v < N, t \geq 0, p \in \text{Inputs}, \forall S_{v,t} \text{ where } C_v(S_{v,t}) = S : \]

\[ T_v(S_{v,t}, p) \in \text{Compromised} \]

\[ \exists w, T_w(S_{w,t}, p) \in \text{Alarm} \text{ and } C_w(S_{w,t}) = S \]  

(2.2)

Assuming that the normal equivalence property is satisfied, the detection property guarantees all attacks of a certain type are to be detected. Specifically, the case that one attack transfers all variants from normal states to compromised states can never happen. This property imposes a requirement that variants have exclusive exploitation sets for attacks.

### 2.2 Threat Model

Under our threat model, the defender runs a server system which consists of an application \( P \). The attacker have the knowledge that a zero-day vulnerability exists inside the application. One day, the attacker launched an attack against the
server system by exploiting the vulnerability in $P$ using malicious input $A$, and compromised the server system. At the end of that day, security administrator noticed the attack and blocked input $A$ for any copies of $P$. However, analysis team did not finish vulnerability analysis yet, and thus application patch is not available. During this period of time, the attacker crafts attack mutants of $A$ exploiting the same vulnerability, and the defender constructs a N-variant system to secure the server system from $A$ mutants.

2.3 Problem Definition

The defender faces two major problems concerning deploying a N-variant system, as follows.

**Resource Overhead**: Suppose the computational resource required by application $P$ is $R$, then the resources required by N-variant system is $(N + c) \times R$, in which $c$ is a constant for synchronization cost between different variants and divergence detection. Hence, to deploy a N-variant system, the defender need more than $N - 1$ times extra computational resources.

**Performance Overhead**: The overall performance of the N-variant system depends on the slowest variant, and the algorithm used for behavior divergence detection. Evaluation results in [12] show that the performance overhead of Cox et al. N-variant system ranges from 15% to 30%.

The key insight for this thesis is to develop a new approach to help the defender to construct N-variant system in a more efficient and practical way. We have two expectations to our new approach in order to gain adoption by the defender:

- **Defense Expectation**: We expect the new approach maintains equivalent security advantages provided by N-variant system. Given any benign input, our new approach should follow the property of normal equivalence. Given any attack mutants of an attack, the new approach also satisfies detection property.

- **Cost Expectation**: We expect our new approach to significantly reduce the resource overhead and performance overhead introduced by the original N-variant system.
Symbolic N-Variant System
Overview

In this chapter, we provide an overview over the system model and architecture of our symbolic N-Variant system.

3.1 System Model

Symbolic N-variant system shares a similar framework and detection model of a N-variant system proposed by Cox et al., however, some fundamental aspects of the system vary. For illustration purpose, we use the following notations to describe the system model of our new system.

- $App_1, App_2, ..., App_N$: diversified application variants of the original application.
- $Pro_1, Pro_2, ..., Pro_N$: processes that executes $App_i$.
- $Sap_1, Sap_2, ..., Sap_N$: symbolic application variants in the symbolic N-variant system.
- $Smp_1, Smp_2, ..., Smp_N$: symbolic processes in the symbolic N-variant system.

In symbolic N-variant system, variant inside the system is no longer process $Pro_i$ executing diversified application $App_i$. Instead, we use symbolic variant $Smp_i$.
as process to simulate the execution of $Pro_i$. We call $Smp_i$ as symbolic variant. Given an input, $Smp_i$ predicts the execution path to be followed in $Pro_i$ in a real execution, and gathers the behaviors to be triggered along the execution path. We call such process as a simulation over input. Symbolic application $Sap_i$, built from $App_i$, enables $Smp_i$ to accomplish the simulation. Essentially, $Sap_i$ is a table containing two attributes: symbolic path and symbolic behavior. Each symbolic path corresponds to an execution path in $Smp_i$, and indexes a set of symbolic behaviors. Upon input arrival, $Smp_i$ matches the input to a symbolic path in $Sap_i$ and outputs the indexed symbolic behaviors. The output symbolic behaviors are expected to be consistent with real behaviors triggered in $Pro_i$ with the input.

We model the execution of $Smp_i$ as a sequence of states. Upon input arrival, symbolic behaviors output by $Smp_i$ are mapped into one of four states: normal, compromised, alarm and null. A variant goes to null state when simulation returns no symbolic behaviors. For a Symbolic N-Variant System, the state transition is determined by the input, which makes the system stateless.

For a Symbolic N-Variant System, we also maintain the properties of normal equivalence and detection. Both properties are similar to the corresponding two equations (see Equation 2.1 2.2) for N-variant system. We regard the null state as an alarm state in the equation.

### 3.2 System Architecture

The architecture of our system is presented in Figure 3.1. It consists of four major components: Firewall, Server System, Symbolic Application Variant Generator, and Symbolic N-Variant Simulator. In our system, the Firewall is used to replicate input request and feed data into the Server System and the Symbolic N-Variant Simulator. We will explain other components in the following subsections.

#### 3.2.1 Server System

Server System consists of the platform that runs the real user-space applications we protect. We assume that: 1) the programs are written in C/C++, and source code is available; 2) no artificial diversification is already introduced into the platform and
the user-space applications; 3) the applications contain zero-day vulnerabilities.

### 3.2.2 Symbolic Application Variant Generator

Symbolic Application Variant Generator is designed to generate application variant $A_{pi}$ and symbolic application $S_{pi}$. It consists of an LLVM based Application Variant Generator (AV Generator), a Symbolic Application Explorer (SA Explorer), and a running platform identical to the Server System.

#### 3.2.2.1 AV Generator

AV Generator introduces artificial diversity into application, and generates application variant $A_{pi}$. Different diversification techniques can be leveraged to meet certain security requirement, and determines the attack class that our system can detect. Techniques that generate variants with disjoint exploitation set provide deterministic detection of certain class of attacks. For instance, address space partitioning technique partitions the absolute address space used by each variant into disjoint sets. Hence, any attacks involve referencing an absolute address would trigger at least one variant to exit normal state. However, for some attack classes such as heap corruption, when disjointedness is infeasible, probabilistic techniques such as heap randomization could be a favorable choice.

After source code level randomization, we use LLVM to compile each variant
into LLVM IR for further processing.

3.2.2.2 SA Explorer

SA Explorer is the unit for symbolic application \( S_a p_i \) generation. It consists of customized KLEE [13] as symbolic execution engine that explores execution paths for each application variant and captures behavior information along each path.

KLEE Overview: KLEE is designed for automatic exploration of execution paths in complex applications. It executes program on symbolic value instead of concrete value. The exploration process is assisted by an interpreter. During exploration, the interpreter follows a particular path. Along each path, KLEE keeps an execution state, which includes a global table for recording global variables and stack for managing local variables and function calls.

When the interpreter encounters a branch, it queries the constraint solver (STP [14]), to determine if the branch condition is either provably true or provably false in the current execution state. If the condition is proven to be true or false, the interpreter will directly follow the determined branch. Otherwise, both branches would be considered in future execution by following bellowing steps:

1. fork the current execution path,
2. clone the current execution state for each forked path,
3. add constraints to each branch to be followed into corresponding path.

During one execution, the interpreter follows a single path. When branching fork occurs, the interpreter relies on a path selector to determine which path to follow. When one execution ends, the interpreter terminates and starts another execution if any unexplored path remains. However, the number of forked paths increases exponentially with the size of application, so does the time and resource cost for building a complete symbolic application \( S_a p_i \). This problem is called the path explosion problem.

To reduce the amount of paths to be explored, KLEE enables to limit the exploration scope by assigning symbolic input size. Specifically, user can assign standard data fields inside a symbolic input, and only allows variance within a limited set. For instance, if we have the knowledge that an attack exploit is
in form of "GET.* HTTP/1.0" (.* means any character with any length), then we can assign "GET.S[len] HTTP/1.0" (S[len] as the symbolic input field with limited size of len) as its symbolic input. Larger symbolic input size leads to more possibilities during symbolic execution, and thus generates more paths.

**Symbolic Application Generation:** We name a path explored by KLEE as a *symbolic path*, which is a set of logical constraint predicates on symbolic inputs in STP language. For instance, "assert(sym_in[0x00] = 0)" is a predicate that requires the first byte of input sym_in be equal to 0. Along with each symbolic path, we extract *symbolic behaviors* during symbolic execution. Different aspects of a program can be extracted as symbolic behavior depending on the class of attacks to be detected. For instance, we can extract system calls along a path with their names and arguments as symbolic behavior to detect attacks that leads to different system call invokes in the original program. We can also extract file operations to detect attacks related to sensitive file reading.

After the symbolic execution, symbolic paths and corresponding symbolic behaviors of an application variant constitute a symbolic application variant $S_{ap_i}$. Table 3.1 shows a simple example of a symbolic application variant. In this example, we extract system calls as symbolic behaviors of each path.

<table>
<thead>
<tr>
<th>Symbolic Paths</th>
<th>Symbolic Behaviors</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASSERT(NOT(0x0D = Sym[0x02])) &amp;&amp; open: arg1, arg2, ...</td>
<td>...</td>
</tr>
<tr>
<td>ASSERT(NOT(0x00 = Sym[0x02])) &amp;&amp; ioctl: arg1, arg2, ...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Table 3.1. Example of a symbolic application.**

### 3.2.3 Symbolic N-Variant Simulator

The Symbolic N-Variant Simulator (SNV Simulator) accomplishes the functionality of detection. It resembles the structure of the Cox et al. N-Variant System. For clarity, we call each symbolic variant process $S_{mp_i}$ in SNV Simulator a *v*-simulator. At a high level, the polygrapher takes input replicated from the firewall and distributes them to v-simulators for simulation. The monitor collects behavior results and detects divergences to reveal attack.
Simulation Process: Simulation is a process of matching an input to a symbolic path and retrieving its corresponding symbolic behaviors in a symbolic application. A v-simulator achieves this following several steps.

First, we construct a symbolic path search tree for each v-simulator from each symbolic application $S_{ap_i}$ before we deploy the whole system (off-line). Figure 3.2 shows an example of how a search tree is built. The example contains a symbolic application with three symbolic paths: $P_1$, $P_2$, and $P_3$. After sorting the predicate indexes, we found logical predicate 1 is shared by all paths, and 2 is shared by $P_1$ and $P_2$. We build a search tree by merging the shared logical predicates together as share nodes, and append path indexes at leaf nodes. Since predicates along each path are ordered as they appear on an actual execution path, we expect many symbolic paths share logical predicates, and thus the structure of search tree would be similar with real program execution tree.

Second, after building search tree for each v-simulator, the whole system is ready for deployment. During run time, each v-simulator waits for input replicate from the polygrapher. Upon input arrival, v-simulator conducts a Depth First Search (DFS) through the tree to locate a symbolic path or exit if nothing retrieved. Upon arriving at a node, v-simulator parses the input against the corresponding predicate, and decides which direction to go. For instance, for a logical predicate: "assert(sym$in[0][00] = 0)"”, the parser checks whether the first value of the input is 0x00. If so, it goes through the true branch. Otherwise, it returns and tries other branches of its parent node. Once the v-simulator arrives at a leaf node, a symbolic path index and corresponding symbolic behaviors would be retrieved.

Finally, once all v-simulators finished simulation for an input, symbolic behaviors from each v-simulator would be gathered by the monitor, and be further analyzed to detect divergence.
**Number of v-simulators**: The number of v-simulators required to detect a certain class of attacks is determined by the diversification technique used in AV Generator as described in Section 3.2.2.1. For instance, Cox et al. implemented 2 variants for using address space partitioning or instruction set tagging as the diversification techniques [12]. Berger and Zorn [15] implemented 6 replicas for using a probabilistic heap layout randomization.
Chapter 4

Design and Implementation of SNV Simulator

We build a prototype Symbolic N-Variant System based on a popular light weight commercial web server lighttpd, which consists of 30,000+ C code. The Server System in our prototype runs lighttpd application as a web server. To demonstrate the effectiveness of our symbolic N-variant system, we craft attacks exploiting a public vulnerability (CVE-2007-3949 [16]) in lighttpd to mimic the situation of zero-day attacks. In this chapter, we focus on describing the design and implementation of symbolic N-variant simulator. Firstly, we introduce the vulnerability that we choose.

4.1 Vulnerability Description

Our prototype is based on a vulnerability (CVE-2007-3949 [16]) in lighttpd-4.15. This vulnerability ignores trailing ‘/’ (slash) characters in request url, which allows remote attackers to bypass file access control.

When lighttpd receives an file request, it checks the name extension of the requested file. If the extension is forbidden by file access rules, lighttpd denies the file request. Listing 1 shows an abstract of the checking process. Specifically, function process_uri invokes ext_control to check whether a uri ends with an access control file extension (line 19 and 30). However, when there is a trailing slash in the request uri, memset in line 25 would set the length of memory up to
4 bytes larger than member uri_access.dir. Since offset member trails dir in uri_check struct, it would be over written with 0. Hence, during access checking process, the result always returns permission grant. Therefore, the checking process would be bypassed.

```c
struct uri {
    char    dir[MAXR];
    int     offset;
    int     index;
...
}uri_check;
...
int ext_control(char *dir, int offset) {
    if (strncmp(dir + offset - extaccess.size, extaccess.rule, extaccess.size)) 
        return FORBIDDEN;
    } else {
        return GOON;
    }
}
...
int process_uri(char *request_uri) {
    uri_check uri_access;
    char  *slash = NULL;
    if (ext_control(request_uri, strlen(request_uri)) == FORBIDDEN) 
        return FORBIDDEN;
    ...
    while ((slash = strrchr(request_uri, '/')) != NULL) {
        uri_access.offset = slash - request_uri;
        ...
        memset(uri_access.dir, 0, len);
        // len is up to 4 bytes larger than MAXR
        ...
        strncpy(uri_access.dir, request_uri, slash - request_uri);
        ...
        if (ext_control(uri_access.dir, uri_access.offset) == FORBIDDEN) 
            return FORBIDDEN;
        ...
        request_uri = uri_access.dir;
    }
    ...
    access(request_uri);
}
```

Listing 1: Simplified access control code in lighttpd

Our prototype is built and deployed under the following scenario: A commercial corporation sets up lighttpd-4.15 (the latest version up to deployment) as corporation web server, and a security team monitors and maintains its daily operation. The server is configured to deny access to source code files with extension .c, and thus requests such as "GET./a.c HTTP/1.0" should be denied. However, an at-
tacker discovered a zero-day vulnerability (CVE-2007-3949) inside lighttpd, and by which he can bypass the checking process and access the forbidden files (*.c). Specifically, the attacker crafted malicious request "GET./a.c/**#/!// HTTP/1.0", and downloaded a forbidden file a.c. The security team quickly noticed the abnormal behavior of source file read. They shutdown the server first, and start to build a symbolic N-variant system based on the exploit sample to detect attack mutants.

4.2 SNV Simulator

From the SAV Generator, we introduce DSLR to lighttpd by reordering data fields and insert garbage fields into struct definition, and generate 10 application variants. By assigning up to 7 bytes of symbolic input size, we symbolic execute the 10 application variants, and generate symbolic application variants. During which, we extract symbolic behavior as symbolic behaviors by logging system call names and arguments appear in each path. Table 4.1 shows statistics of path exploration for lighttpd.

<table>
<thead>
<tr>
<th>Size</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Path #</td>
<td>10</td>
<td>43</td>
<td>161</td>
<td>604</td>
<td>2291</td>
<td>8839</td>
<td>34667</td>
</tr>
<tr>
<td>Time (s)</td>
<td>15</td>
<td>13</td>
<td>14</td>
<td>49</td>
<td>146</td>
<td>649</td>
<td>4089</td>
</tr>
</tbody>
</table>

Table 4.1. Statistics of path exploration of lighttpd (Size: symbolic input size; Path #: average number of symbolic path explored; Time: average time elapsed for path exploration.)

Listing 2 shows an sample symbolic path test011800. In the path, array variable readsym_1_0x594ae70 represents the input request, and each ASSERT statement is a constraint predicate in STP language. STP is a efficient language used to depict decision procedure for formula satisfiability. It includes functions for byte operations such as concatenation, extraction, left/right shift, sign-extension, unary minus, addition, bitwise Boolean operations, etc. For instance, line 1 specifies that the 11th byte of the input request should not equal to ASCII code 0x0D. Line 7 specifies that the result of concatenating 0x000000 and the 11th byte of the input request should be greater than value 0x00000020. By collecting all predicates
together, each symbolic path captures constraints on the input request to follow an execution path in the original application.

```c
1  ASSERT(0x0D = readsym_1_0x594ae70[0x0000000B]);
2  ASSERT(NOT(0x0D = readsym_1_0x594ae70[0x0000000C]));
3  ASSERT(NOT(0x0D = readsym_1_0x594ae70[0x0000000E]));
4  ASSERT(NOT(0x20 = readsym_1_0x594ae70[0x0000000B]));
5  ASSERT(NOT(0x20 = readsym_1_0x594ae70[0x0000000C]));
6  ASSERT(NOT(0x20 = readsym_1_0x594ae70[0x0000000E]));
7  ASSERT(SBVGT((0x000000 @ readsym_1_0x594ae70[0x0000000B]),
136  (0x000000 @ BVPLUS(32, 0xFFFFFFD0, (0x000000 @
136  readsym_1_0x594ae70[0x0000000B])[7:0]), 0x00000009)));
8  ...  
9  ASSERT((NOT(SBVGT((0x000000 @ BVPLUS(32, 0xFFFFFFD0, (0x000000 @
136  readsym_1_0x594ae70[0x0000000B])[7:0]), 0x00000009))));
10 ASSERT(SBVGT((0x000000 @ BVPLUS(32, 0xFFFFFFD0, (0x000000 @
136  readsym_1_0x594ae70[0x0000000C])[7:0]), 0x00000009));
11 ASSERT((NOT(SBVGT((0x000000 @ BVPLUS(32, 0xFFFFFFFAA, (0x000000 @
136  (0x000000020 @ BVPLUS(32, 0x0000002F, (0x000000 @ BVPLUS(32,
136  0xFFFFFFD0, (0x000000 @
136  readsym_1_0x594ae70[0x0000000C])[7:0])][7:0])][7:0])][7:0]),
136  0x0000000F))));
12 ...  
```

Listing 2: Example symbolic path generated by SA Explorer.

The SNV Simulator is designed for simulating input requests and comparing simulation results to detect divergence. At a high level, SNV Simulator is designed as a process which contains a polygrapher, a collection of v-simulators running in different threads, and a monitor (see Figure 4.1).

### 4.2.1 Polygrapher and Monitor

The polygrapher in SNV Simulator is built for receiving and distributing external input requests to each v-simulator inside. We build the polygrapher using the framework of a web server system. TCP port 8099 is registered to listen to socket connections from firewall. In the polygrapher, request handling function is customized to distribute requests, and all parameters are configured to be identical as those in lighttpd for performance evaluation purpose.
When input request is received from socket, the polygrapher adds each request into an input queue and informs each v-simulator to fetch and simulate. To speedup the simulation process, we fork $N$ threads in the main process, and each v-simulator run simulation in parallel. Once all v-simulators finished simulation, the monitor collects results from each v-simulator and analyze the results to detect divergence.

The synchronization between polygrapher, v-simulators, and monitor resembles the classic producer–consumer problem [17] in multi-process synchronization. However, instead of one producer serving one customer, in our model, the producer (polygrapher) serves each of his product (input request) to $N$ customers (v-simulators), and collects feedback (simulation result) from each customer at the counter (monitor). To handle this problem, we use two sets of semaphores to do synchronization.

Listing 3 shows a simplified flow of how the SNV Simulator works. We create $N$ threads in the main process, and each thread is assigned with a unique identifiable id and a signature which indicates different symbolic variants (we will describe the structure of v-simulator in detail in Section 4.2.2). For each v-simulator, we assign unique semaphore $\text{sem\_arrives}[i]$ as a signal to indicate new input request. We also create a shared semaphore between v-simulators $\text{sem\_done}$ as a signal to indicate whether simulation process is done.

When new request arrives, the function $\text{handle\_request}$ is invoked, and set $N$
sem_arrives[i] semaphores (line 31) to inform each v-simulator for new request simulation. When a v-simulator receives signal, it fetches the new request, and starts simulation (line 10). When simulation finishes, it sets sem_done semaphore (line 12) to inform the main thread that simulation is done, and waits for new simulation request. The main process waits for sem_done for N times (line 23). When the for loop finishes, it indicates that simulations for an input request are done, and the results are ready for behavior divergence detection.

In the monitor, we firstly collect results from all N v-simulators. The result generated from v-simulator are symbolic behaviors correspond to a particular path, or null if no symbolic path is matched. As described in previous section, we extract system calls along the execution trace as symbolic behaviors, including system call names and parameters. However, for the vulnerability that we chosen for our prototype, we can differentiate malicious request only from system call names, since only successful exploit would invoke file opening and reading related system calls. Hence, we compare the system call names of all simulation results with each other, and report attack when divergence occurs.

4.2.2 V-Simulator

V-simulator is the module where simulation takes place. It retrieves program behavior for input request by simulating actual execution, and achieves performance efficiency. Each v-simulator consists of four major parts as shown in Figure 4.2: a predicate table, a behavior table, a search tree, and a predicate parser.

4.2.2.1 Predicate Parser

To determine the symbolic path that an input request follows, the simulation process parses through all constraint predicates in a path against the input request, and see whether all predicates are parsed to be true. If so, we accept the path and retrieve corresponding symbolic behaviors as the simulation result. If not, we search other possible paths until one is found, otherwise, return null. We build an efficient STP predicate parser in v-simulator to achieve this functionality.

Firstly, we build a automatic syntax translator from STP to C/C++ style language for efficiency purpose. As we see from Listing 2, the language syntax for
STP predicate cannot be properly handled by C/C++ compiler, and running a STP parser on-line could be very costly. To mitigate this problem, we translate each STP predicate into a C/C++ style function. For instance, an sample STP predicate generated by SAV Generator:

\[
\text{ASSERT}((\text{NOT}(\text{SBVGT}(\text{BVPLUS}(32, 0xFFFFFFFFD0, \text{READSYM}_1_0x594ae70[0x0000000B][7:0]), 0x00000009)));
\]

can be translated into a function:

```c
bool lighttpd_sym7_v1_pre_136() {
    struct item cnst0 = {0, 24};
    struct item cnst1 = {32, 8};
    struct item cnst2 = {4294967248, 32};
    struct item cnst3 = {0, 24};
    struct item cnst4 = {11, 32};
    struct item cnst5 = {7, 8};
    struct item cnst6 = {0, 8};
    struct item cnst7 = {9, 32};
    return solve(int2(not(sbvgt(concat(cnst0, extract(bvplus(cnst1, cnst2, concat(cnst0, readsym_1_0x594ae70[0x0000000B][7:0]), 0x00000009]))));
}
```
After the translation, each predicate function was indexed by a function pointer array. We can invoke corresponding function for each predicate by accessing the function pointer array with its predicate index to accomplish predicate parsing.

Secondly, we implement each function provided by STP syntax (e.g. `not`, `concat`, `extract` in the above function) including concatenation, extraction, shifting, sign-extension, unary minus, bitwise boolean operations, etc.

### 4.2.2.2 Symbolic Path Search Tree

Predicate records the constraint specified on input request to suffice to follow a particular execution path. According to the execution trace of a program, two execution paths that diverge at a certain branch shares identical execution path before the branch. Hence, we expect that many symbolic paths share same predicates.

Based on this observation, we propose to build a symbolic path search tree to reduce the amount of predicates we need to parse through, and thus increase the performance for simulation. Specifically, we merge the paths that share the same predicates together into one branch, and fork new branch when predicate starts to differ. Firstly, we index all predicates that appear in all paths to build a predicate table (See Figure 4.2). Then, we represent each symbolic path as a sequence of predicate indexes. Secondly, we go through predicate indexes of all paths in order from the first to the end. We merge paths that share identical predicate index together and create one shared tree node, and fork different tree branches when it diverges. Finally, when reaching the end of a symbolic path, we append the corresponding path index at the leaf node, which indexes the path behavior information in behavior table. Algorithm 4.2.2.2 shows the pseudocode for building symbolic path search tree.
For each symbolic application variant generated from SAV Generator, we create a corresponding v-simulator off-line, and assign each v-simulator with a unique signature. When The SNV Simulator is started on-line, it creates different v-simulators by different signatures and thread id. During simulation process, we accomplish the simulation following several steps:

1. Starting from the root tree node, upon arriving a tree node, we query the predicate table for the predicate and receive a function pointer. By calling the corresponding function, the parser returns true or false indicating whether the predicate is satisfied against the input request.

2. Search through the symbolic path search tree following depth first search.
algorithm.

3. When arriving a leaf node, we query the behavior table by path index and retrieve the symbolic behaviors as simulation result. Otherwise, if no path is found during the search, return null as simulation result.

4.3 Implementation

For SAV Generator, we separately implemented a AV Generator and an SA Explorer. The AV Generator consists of two tools to introduce DSLR into LLVM IR of C/C++ program. The first tool is implemented as a Clang LibTooling plug-in to randomize struct types at source code level. This tool contains about 120 lines of C++ codes. The second tool is a LLVM pass to randomize local variables of each function in IR. This tool is also written with about 120 lines of C++ codes. The SA Explorer is a revised version of KLEE. We implemented a socket environment model for KLEE, following standards specified by POSIX. We built this model into KLEE with more than 1000 lines of C++ codes. We also customized KLEE to speed up the path exploration process, such as implementing the BFS searching algorithm. About 1000 lines of C++ codes are inserted to customize KLEE. The SNV Simulator is built from scratch with over 1500 lines of C++ codes.
sem_t sem_arrives[N];
sem_t sem_done;

void* simulator(void *th) {
    // create v-simulator with unique thread number and signature
    Simulator sim = Simulator(th->thread_id, th->signature);
    while (true) {
        // wait for input request semaphore from main thread
        sem_wait(&sem_arrives[((struct thread_param
        th)->thread_id)];
        result = sim.simulate(request, request_length);
        // set semaphore when simulation finished
        sem_post(&sem_done);
    }
    ...}
}

void handle_request(char* request, int request_length) {
    for (int i = 0; i < N; i++)
        // inform each v-simulator for new request
        sem_post(&sem_arrives[i]);
    for (int i = 0; i < N; i++)
        // wait for all simulators to finish simulation
        sem_wait(&sem_done);
    ...}

int main () {
pthread_t simulator_thread[N];

    for (int i = 0; i < N; i++)
        sem_init(&sem_arrives[i], 0, 0);
    sem_init(&sem_done, 0, 0);

    for (int i = 0; i < N; i++)
        pthread_create(&(simulator_thread[i]), 0, &simulator,
        &((th[i]));

    ...}

Listing 3: Synchronization between polygrapher, v-simulators, and monitor using semaphore.
Chapter 5

Simulation Performance Optimization

One key expectation for the symbolic N-variant system is performance superiority over the Cox et al. N-variant system. The advantages brought by performance superiority is in twofolds, according to the way we expect to deploy and use our symbolic N-variant system.

Firstly, obtaining simulation results for input request fast could bring precious time for security countermeasures against potential attack. When a malicious input request was received and sent to the Server System and the SNV Simulator, the Server System starts executing the input request in the original application, and the SNV Simulator starts to simulate the input request for behavior divergence detection simultaneously. Suppose we can generate the simulation results before the execution finishes, we may be able to take necessary security actions to prevent malicious action from happening by terminate the execution or other measures. For instance, in our prototype design, if the SNV Simulator detects behavior divergence, several remedies could be leveraged to block the information leakage to attacker: 1) immediately terminate the execution of \texttt{lighttpd}; 2) temporarily reclaim the reading access of \texttt{*.c}; 3) inform firewall of the system to drop network packets to the attacker machine. In general, we believe that the more time savage we can achieve between simulation and real execution, the more options we can provide for the defender.

Secondly, performance superiority over real execution enables Symbolic N-

Variant System to serve multiple applications in parallel, which reduces the amount of resources required for each system. If the input requests could be simulated faster than real execution, we can make full use of the computing resource of Symbolic N-Variant System by running SNV Simulators for multiple applications.

For above reasons, we improve the performance of simulation using several heuristics, including caching parsing results, filtering out benign input requests, and symbolic path search tree pruning.

5.1 Caching Parsing Result

Predicate parsing consumes major time consumption during input request simulation. Based on our observation on the structure of symbolic path search tree, and the constraints specified on input request by a predicate, we propose two caching algorithm for simulation.

5.1.1 Intra-Simulation Cache

In symbolic path search tree, we observe that many repeated predicates index exist in different paths. This phenomenon results from how symbolic execution handles looping and conditional statements inside an application. When symbolic execution encounters looping statements such as `while` or `for`, the loop would be unwinded as nested conditional statements. For instance, Figure 5.1 shows a code snippet including a `while` loop, the corresponding execution sequence, and the symbolic path search tree. When path explorer arrives at the `while` statement, two path branches could be followed: one indicates the looping condition is satisfied and enters the looping body; one indicates otherwise and continues to the following instructions after the `while` loop. Symbolic execution calculates corresponding constraint predicate for each branch (0 and 1 in the example). For the path that goes into the looping body, second round condition checking also generates two predicates (3 and 4). This process goes on until all possibilities for the input requests are considered, or reaches a specific bound (KLEE provide two bound options for handling looping: bound the number of loops or bound the time consumption).
While (sym_in[0] > 0) {
  sym_in[0]--;
  sym_in[1]--;
  if (sym_in[1] > 0)
    ...
  else ...
}

Figure 5.1. Example symbolic path search tree with repeated predicate cause by looping.

For both looping statements and conditional statements, when constructing the search tree, it is inevitable to have repeated predicates in different paths as shown in Figure 5.1. During one simulation, each predicate is parsed against the same input request. It is redundant to parse through the same predicate for several times. Therefore, we cache the result of each predicate by recording its predicate index and parsing result into a cache table (see Table 5.1). Before parsing a predicate, we first query the cache table by predicate index. Only for those predicate that has not been parsed would invoke parsing function.

<table>
<thead>
<tr>
<th>Predicate Index</th>
<th>Parsing Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>True</td>
</tr>
<tr>
<td>4</td>
<td>False</td>
</tr>
<tr>
<td>10</td>
<td>True</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 5.1. Example of intra-simulation caching table.

5.1.2 Inter-Simulation Cache

In constraint predicates generated from symbolic execution, we observe that almost all predicates specify constraint on less than 2 bytes of input request. For instance, all predicates in Listing 2 only involves one byte of input request. We also observe that many input requests share the same data bytes. For instance, a legitimate
lighttpd request should be in form of `GET/*_HTTP/1.0` or `POST/*_HTTP/1.0`. Although individual input request might be different with each other, parsing results for a particular predicate could still be the same as long as the corresponding data byte involved in the predicate is the same.

Therefore, we cache the parsing result into a cache table indexed by predicate index, input request byte index, and input hash value (See Table 5.2). In pre-

<table>
<thead>
<tr>
<th>Predicate Index</th>
<th>Input Byte Index</th>
<th>Input Hash Value</th>
<th>Parsing Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[2], [4]</td>
<td>2654436024</td>
<td>True</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8293476324</td>
<td>False</td>
</tr>
<tr>
<td>3</td>
<td>[0], [1]</td>
<td>9480198356</td>
<td>False</td>
</tr>
<tr>
<td>10</td>
<td>[2]</td>
<td>3489756847</td>
<td>True</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 5.2. Example of inter-simulation caching table.

processing phase, we analyze each predicate and extract the related byte index for symbolic input. For instance, the following predicate:

```plaintext
ASSERT( (0x30 = ((0x000000 @ BVPLUS(32, 0xFFFFFFFFD0, (0x000000 @ readsym_1_0x594ae70[0x0000000A]))[7:0]) | ((0x000000 @ BVPLUS(32, 0xFFFFFFFFD0, (0x000000 @ readsym_1_0x594ae70[0x00000009]))[7:0])[27:0] @ 0x0)) [7:0]) );
```

puts constraint on two bytes of the symbolic input request: 0x0A and 0x09. We record both index for this predicate. During simulation, when arrives at a search tree node with this predicate, we only need to check whether the 9th and 10th byte of the input request is the same as a certain previously encountered input request. To achieve this, we use a hash function in Algorithm 5.1.2 to calculate a hash value for each input request.

### 5.2 Input Request Filtering

In our SNV Simulator design, parsing input request through the symbolic path search tree consumes majority of time for simulation. However, in most real world scenario, only a small portion of requests are malicious inputs exploiting a specific
vulnerability (except some flooding attacks or certain denial of service attacks). Simulating benign input requests consumes large amount of computing resources and time. Hence, filtering out as many benign input requests as possible would provide performance increase for our simulation, and enable the SNV Simulator to provide service for more applications.

For the vulnerability we choose in our prototype design, malicious requests that bypass access checking rules follows the form of "GET\textcolor{red}{\textbullet\textbullet\textbullet}.c\textcolor{red}{\textbullet\textbullet\textbullet}HTTP/1.0" (\textcolor{red}{\textbullet\textbullet\textbullet} means any character with any length). We can expect that any requests using POST command, or requesting other types of files do not exploit the same vulnerability, and thus simulation results from different v-simulators would not show divergence. Hence, such input requests should not be simulated for performance purpose.

We design a filtering system that intends to filter out benign requests from our SNV Simulator. The following requirements must be satisfied for the filtering system:

- The filtering system should filter out majority of benign requests from entering SNV Simulator (low false positive), but should never filter out malicious requests (no false negative).
- Filtering system should be quick so that simulation would not be delayed.
- Based on out threat model, the filtering system should be automatically generated based on attack exploit samples. It should require no manual effort.

The functionality of our filtering system resembles that of intrusion detection or prevention system (IDS/IPS). Basically, both the filtering system and IDS/IPS

---

**Algorithm 2:** Calculating predicate hash value for each input request.

<table>
<thead>
<tr>
<th>Data: Input request array $sym_in$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Result:</strong> Hash value $hash$</td>
</tr>
</tbody>
</table>

$hash \leftarrow 0$;

forall the index $i$ appear in predicate do

$| hash \leftarrow sym\_in[i]+0x9e3779b9+(hashvalue \ll 6)+(hashvalue \gg 2)$;

end
monitors system or network activities for malicious actions or policy violation. Most commercial or open source IDSes/IPSes, such as Cisco and Snort [18], use *regular expression* based signature in its matching engine [19]. The advantage for using regular expression for matching is that regular expression provides enough accuracy, and recent research improved its high speed with little memory occupation [20, 21, 22].

We choose regular expression as signature for matching potential malicious input request exploiting the vulnerability in application, and filter out those that are irrelevant. To achieve this, we implement an algorithm to automatically extract predicates in each path, and convert them into regular expression signatures. However, one challenge is that the expressive power of regular expression is weaker than a predicate written in STP language. That is, some predicates involved with complicated syntax cannot be converted to regular expression [23]. For instance, predicate that involves relation between two or more bytes such as `sym_in[i]@sym_in[j]>0x0100`, predicate with arithmetic operation or bitwise operation such as `sym_in[i]+1<0x11`, etc. To mitigate this problem, we only extract predicates that only involves `"="` or `"!="` in each symbolic path. For the path in Listing 2, we extract only the first 6 predicates:

```
ASSERT(0x0D = readsym_1_0x594ae70[0x0000000B]);
ASSERT(NOT(0x0D = readsym_1_0x594ae70[0x0000000C]));
ASSERT(NOT(0x0D = readsym_1_0x594ae70[0x0000000E]));
ASSERT(NOT(0x20 = readsym_1_0x594ae70[0x0000000B]));
ASSERT(NOT(0x20 = readsym_1_0x594ae70[0x0000000C]));
ASSERT(NOT(0x20 = readsym_1_0x594ae70[0x0000000E]));
```

and construct a regular expression as the signature of this symbolic path.

Figure 5.2 shows a flowchart for processing an input request with the filtering system. During on-line execution, each input request is firstly sent into the filter. Only those requests that can be matched to the regular expression signature remains for further simulation and divergence detection in monitor. All malicious input requests that can be matched into a symbolic path will remain after the filter, since the regular expression signature specifies a superset of input of what the predicates specifies. Therefore, false negative for the filter will be none. After
the filter, input requests are fed into v-simulators for simulation. Since we only captures a subset of symbolic paths for the application (Symbolic paths captured by search tree are incomplete for two major reasons. One is the flaw of symbolic execution, which we will explain in Section 7.1. The other one is because of search tree pruning described in Section 5.3), some benign paths would fail to reach a leaf tree node. For those input requests that successfully arrive at leaf nodes, we retrieve symbolic behaviors and detect divergence in monitor. Only malicious input requests would be reported and enable security administrator to take further actions.

We will evaluate false positive of the filter in the evaluation section.

5.3 Symbolic Path Search Tree Pruning

SA Explorer symbolic execute the whole application, and generate symbolic paths as more as possible. We make an observation that large portion of symbolic paths are benign paths. Since benign paths in different symbolic applications share identical symbolic behaviors, simulating such paths is a waste of time. We can trim off these benign paths in each symbolic application, and maintain only potential malicious paths in the search tree.

By pruning the symbolic path search tree, our system can have the following advantages:

- Trimming off benign paths reduces the size of search tree, which saves memory consumption for each v-simulator. With less tree nodes, we expect time consumption for simulation would also drop.
Figure 5.3. Identifying benign symbolic paths by simulating test cases for each symbolic path.

- Since the number of symbolic paths decreases, the number of signatures in the filtering system also drops. Hence, time consumption for filtering decreases, and less benign input requests would go through the filter for simulation.

One challenge for pruning the search tree is how to identify benign symbolic paths. To mitigate this challenge, we use a predicate solver provided by KLEE that can generate test case for each symbolic path. Each test case is a sample input request that satisfies all predicates inside corresponding symbolic path. We simulate each test case, and see whether the monitor detects divergence (see Figure 5.3). If no, the corresponding path should be considered as a benign path, and should be trimmed off the search tree. We cross-check all paths in each symbolic application variant to make sure that no potential malicious path is deleted.
Prototype Evaluation

In this Chapter, we evaluate our prototype. The prototype is deployed on a 40-Core 2.3GHz Xeon E5 machine with 64GB RAM running Ubuntu 12.04LTS. We evaluate our prototype in terms of its efficiency including computing resources and time consumption, and the security effectiveness of detecting attacks exploiting the specific vulnerability.

6.1 Efficiency Evaluation

We evaluate the efficiency of our prototype by measuring its performance and memory consumption. Evaluations are done with ApacheBench v2.3 [24] to send HTTP requests. ApacheBench is customized to support sending different requests in one experiment session. For all experiments to evaluate the efficiency of our system, the request data are randomly selected from test cases generated by KLEE, and we also randomly replace some requests to download a file with size of 20KB. According to statistics in 2014, average web page size has increased to 1600KB [25], hence, we believe that our experiments are able to indicate optimal performance of state-of-the-art web servers.

We introduce some key parameters of our evaluation experiments as follows:

- request volume: number of input requests sent by ApacheBench in one experiment session.

- concurrency level: number of concurrent requesting clients in ApacheBench
Figure 6.1. Performance comparison under different concurrency level between lighttpd and symbolic N-variant system with different number of v-simulators.

in one experiment session.

- **search tree size**: average number of nodes in symbolic path search tree of different v-simulators.
- **variant number**: number of v-simulator variants running in the SNV Simulator.

### 6.1.1 Simulation versus Execution Performance

Firstly, we compare performance of our simulation system and real execution in lighttpd server. We leverage the benchmark program to send request and measure response time. Different combinations of evaluation parameters are recursively tested.

Figure 6.1.1 shows the performance comparison between lighttpd and our prototype with a fixed search tree size (34109 nodes). The results illustrate that our simulator is relatively more efficient than the lighttpd server, under both unsaturated workload and saturated workload. Performance for our system drops
when the number of variants increases, since more variants requires more computing and synchronization expenses. However, even when our simulator is configured with 10 variants, it runs almost 100% faster than lighttpd. Such results indicate a performance superiority for our SNV Simulator. We believe there are three main reasons for the speedup:

- The simulation process requires no interaction with the environment. Essentially, the simulation only involves predicate parsing in the symbolic path search tree. Therefore, our system does not have I/O operations, or library calls. On the contrary, real execution in web server is I/O intensive, and call other libraries.

- The simulation process maintains less data structures compared with real execution.

6.1.2 Performance for Serving Multiple Applications

As described in Section 5, in real world scenario, a symbolic N-variant system can be deployed to serve multiple applications in parallel due to efficiency advantage. We evaluate the performance of our system while serving different numbers of applications. For demonstration purpose, we use the same version of SNV Simulator for lighttpd as different applications.

Figure 6.1.2 shows the performance trend when symbolic N-variant system serves multiple applications. The number of variants in each application ranges from 2 to 10. We believe it reflects the needs for both deterministic and probabilistic diversification techniques (see Section 3.2.3). From the figure, we observe that the performance of our prototype system decreases when the number of applications or the number of variants in each application increases. Even when each application holds 10 variants, our system can still serve 6 same applications with better performance than the real execution. When the number of variants is no more than 4, our system still outruns real execution serving more than 15 applications. The results demonstrate the scalability of our system.
6.1.3 Performance for Different Tree Size

We measure the performance of our simulator when symbolic path search tree size varies. KLEE generates more and possibly longer paths, when symbolic input size is configured larger, which also results in bigger search tree (see Table 4.1 for the statistics). Figure 6.1.3 shows that simulation is still quicker than real execution when search tree size varies from 1.8k nodes to 147K nodes (symbolic input size ranges from 4 to 7), even configured with 10 variants in the simulator. Nevertheless, we observe that performance drops as search tree size grows.

6.1.4 Memory Consumption

We measure the memory consumption of our SNV Simulator and lighttpd server. Results are shown in Figure 6.1.4. During saturated workload, lighttpd is stable in terms of memory consumption. While memory consumed by the simulator keeps growing when variant number or search tree size increases. This phenomenon is consistent with the in-memory computation nature of the system design: trading space with time. Simulation loads all relevant data into memory and completes all
Figure 6.3. Simulator performance comparison between different search tree size under concurrency level of 48 (saturated).

Figure 6.4. Memory consumption for different number of variants and different tree size under concurrency level of 48 (saturated).

operations in memory. No time-consuming interactions are needed with the outer environment.
6.1.5 Performance Improvement from Speedup Heuristics

In this section, we present the performance improvement provided by the speedup heuristics we implemented in our system.

**Caching:** Table 6.1 shows the number of predicates parsed during simulation before and after our intra-simulation and inter-simulation cache are deployed for random symbolic paths. We observe that the caching reduced 30x-100x of the number of predicates to be parsed during execution. Figure 6.1.5 shows the performance improvement after two caching techniques are deployed in our system. We observe that performance increases from 10%-70%. Since our prototype is based on a relatively simple application, the performance increase is not quite significant. The reason is that predicates generated for our application is so quick that querying caching table does not show speed superiority. However, we expect higher performance improvement when application is more complicated and predicate becoming more sophisticated to parse.

<table>
<thead>
<tr>
<th>Symbolic Path</th>
<th>test000002</th>
<th>test000033</th>
<th>test0000433</th>
<th>test021433</th>
<th>test000289</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Parses without Caching</td>
<td>11899</td>
<td>4840</td>
<td>2316</td>
<td>6719</td>
<td>17750</td>
</tr>
<tr>
<td># of Parses with Caching</td>
<td>114</td>
<td>100</td>
<td>70</td>
<td>87</td>
<td>101</td>
</tr>
</tbody>
</table>

**Table 6.1.** Comparison between number of parsing during simulation with or without caching.

Filtering: We measure the percentage of benign and malicious input requests that goes through the filter. We randomly generate HTTP requests for lighttpd, and mix some malicious requests that reads a.c file. We record the percentage of benign and malicious requests that go through each module of our system. Figure 6.1.5 shows several percentages that reflect some statistics of the whole system including the filter. By running through the filtering system, about 90% of benign requests are filtered out from the SNV Simulator, which saves a huge amount of simulation effort. In the simulator, 0.2% of benign requests cannot be matched to any symbolic paths. After the simulation, we collect symbolic behavior information for each request, and by comparing the results, we rule out all remaining 9.06% benign requests. All malicious requests are detected during the process.

**Pruning:** We measure the number of nodes for symbolic path search tree before and after applying pruning heuristics. The average number of nodes decreases
Figure 6.5. Performance improvement after deploying two caching table in our system.

Figure 6.6. Statistical percentages for processing input request of the whole system with filter.

to around 51% percent of the original trees, which reduces about half of the memory consumption for each v-simulator. Less search tree nodes also leads to less predicate parsing, and thus less time consumption for simulation.

6.2 Security Effectiveness Evaluation

For the symbolic N-variant system design, we expect it maintains equivalent security merits provided by Cox et al. N-variant system. That is, our system can detect given any attack mutants of a certain class, and remain silent for benign
input requests. We use three metrics to evaluate the security effectiveness. First, we measure how many benign input requests are mistakenly detected as mutants of attack (false positives). Second, we measure how many attack mutants are missed (false negative). Third, we measure how many input requests are not matched into a symbolic path and among them, how many are attack mutants and how many are benign requests.

<table>
<thead>
<tr>
<th>Symbolic Input Size</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case #</td>
<td>6040</td>
<td>22910</td>
<td>88390</td>
<td>346670</td>
</tr>
<tr>
<td>Attack Request #</td>
<td>490</td>
<td>1460</td>
<td>19130</td>
<td>40890</td>
</tr>
<tr>
<td>False Positive #</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>False Negative #</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Average Miss #</td>
<td>7</td>
<td>628</td>
<td>423</td>
<td>1456</td>
</tr>
<tr>
<td>Average Missed Attack #</td>
<td>1</td>
<td>141</td>
<td>142</td>
<td>259</td>
</tr>
</tbody>
</table>

Table 6.2. Effectiveness of symbolic N-Variant system against attack cases

Test cases are generated from KLEE for symbolic paths of all variants. Each test case is executed in the lighttpd to check whether it is a malicious (i.e. downloads a.c) or benign request. During the experiment, we count both false positives and false negatives of detection. Table 6.2 shows all evaluation results. The results reveals that our symbolic N-variant system is accurate to detect all attacks without raising false positives or false negatives. However, it also reveals a major problem of our prototype: incomplete simulation. Around %1 benign requests and %2.8 malicious requests on average can not be matched by the simulator. We report all unmatched requests as potential malicious requests. Security administrator can take different actions to handle such requests based on different situation.
Chapter 7

Discussion and Future Work

According to the evaluation results, our prototype implementation illustrates the potential to simulate a web server system in a security-centric way and the possibility to deploy defenses in the simulator to detect malicious requests. It is an attempt to provide protection without disturbing the protected system. However, the evaluation also reveals several limitations of our design.

7.1 Incomplete Symbolic Paths

In our system design, symbolic applications generated from our customized KLEE cannot reflect complete paths in an application. Therefore, in our prototype, some input requests cannot be matched to any symbolic paths. That is, our simulator can only do partial simulation. We face three problems to achieve simulation completeness (i.e. match any input requests to a certain symbolic path and retrieve its symbolic behaviors). Firstly, an application might inherently contains numerous paths (path explosion problem). For this case, solutions using multi-resolution could be helpful. Specifically, grouping individual paths into sets of paths. Secondly, even for applications that contain limited paths, full coverage exploration is still problematic. Existing symbolic execution engines are limited by many issues, including environment interactions, unsolvable constraints, and high resource consumption. Thirdly, assuming we are able to explore all possible paths in an application, our simulator might not be scalable enough to facilitate complete simulation.
7.2 Scalability

Practically speaking, an applicable symbolic N-variant system should be scalable in terms of several aspects. Firstly, the number of symbolic paths grows when the complexity of application grows in symbolic execution. To achieve decent path coverage, larger symbolic input size is required, which also result in larger search tree. It incurs high simulation cost as shown in Figure 6.1.3. Secondly, different variants reflect on different exploitation sets for vulnerability. For some diversification techniques, more variants offer better detection, which also results in higher overhead (shown in Figure 6.1.1 and 6.1.2) and memory consumption (shown in Figure 6.1.4). Thirdly, multiple applications ask for more simulation needs (shown in Figure 6.1.2). The philosophy of our Symbolic N-Variant System is to simulate multiple applications at the same time, which requires our simulator to be scalable to serve numbers of applications in parallel.

7.3 Future Work

One of our future work is to improve the scalability of our prototype. For scalability issues related to search tree size and variant, we propose to augment it with optimized cache algorithms and better search heuristics. To maintain scalability while simulating multiple applications, we plan to leverage parallel computing techniques. We also plan to build our system on more complicated applications with various vulnerabilities.
Chapter 8

Related Work

8.1 Artificial Diversity

In the past decade, many techniques for intentionally introducing software diversity have been developed. Techniques can be grouped into two major categories: platform diversification and application diversification.

Address space layout randomization (ASLR) \cite{4, 5} randomly rearranges the position of key data sections in a process including stack, heap, libraries, etc., preventing attacker from jumping into a particular exploited function in memory \cite{4}. Giuffrida et. al. develop fine-grained address space randomization inside operating system \cite{5}, to enhance OS security. Other researchers also propose memory rearranging via modifying the platform \cite{26, 27, 28}. Chew and Song propose to diversify interfaces of an operating system, via randomizing system call mapping, library entry points, and stack placement. Such interface randomization aims at mitigating buffer overflow attacks \cite{29}.

Application Diversification increases the difficulty for attackers to exploit security vulnerabilities in user space applications. Instruction set randomization (ISR) \cite{6, 7} randomly alter the instructions used by a host machine, program, or a single execution by encoding instructions with secret keys and decodes them upon execution. It is effective to prevent code injection attacks. Data structure layout randomization (DSLR) randomizes encapsulated objects in an application via reordering nested fields or inserting garbage fields \cite{8}. It hides knowledge of data structure from attackers. Salamat et. al. propose to reverse the execution
stack of a process, as a way to detect stack tampering [30]. These techniques ei-
ther transform instructions of an application or rely on the compiler to introduce
diversification.

8.2 Multi-Variant system

Multi-variant system configures a set of artificially diversified application variants
to run identical task. Behaviors of these variants are monitored during execution
time. Divergence among behaviors triggers the alarm for potential attacks. In
1988, Joseph proposes N-version programming [31], which requires multiple in-
dependent groups to complete different implementations for the same application
[31, 32, 33]. The key insight is that some vulnerabilities might be masked in cer-
tain versions, so that attack would cause inconsistency between different versions.
Cox et. al. introduce the framework of N-variant system [12]. Instead of relying
on uncertain difference between different versions of application, they propose
to intentionally generate the variants sharing exclusive exploitation sets. In their
prototype design, they use two techniques to generate such variants: dividing the
memory space and masking instruction with different value. Some works follow
such philosophy of developing multi-variant systems, but using probabilistic based
diversification techniques to generate variants [15, 34]. Instead of providing prov-
able guarantee of detection, Berger and Zorn propose to create program variants
through heap randomization and provide probabilistic protection for attacks that
cannot be deterministically detected.
Conclusion

In this thesis, we propose a symbolic N-variant system which leverages symbolic execution to build N-variant system, and exhibits practicality in real world scenario. We design a framework for symbolic N-variant system, and build a prototype system based on a commercial web server. We emphasize on describing the symbolic N-variant simulator, and propose several speedup heuristics to boost its performance, and the evaluation indicates superiority in terms of both computing resource consumption and performance. For security effectiveness, our prototype also shows the capability to detect mutants of a certain attack class without raising false positive or false negative. Also, our system requires no instrumentation or logging, which introduces zero overhead and requires no customization to the original program.

However, our approach still suffers from several limitations such as incomplete simulation and limited scalability. In future work, we expect to handle these limitations.
Bibliography


URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-3949


URL https://www.snort.org/


[29] **Chew, M. and D. Song** (2002) “Mitigating buffer overflows by operating system randomization,”.


