Comparing Different Moving Target Defense Techniques

Jun Xu
Pennsylvania State University
jxx13@ist.psu.edu

Pinyao Guo
Pennsylvania State University
pug132@ist.psu.edu

Mingyi Zhao
Pennsylvania State University
muz127@ist.psu.edu

Robert F. Erbacher
Army Research Laboratory
robert.f.erbacher.civ@mail.mil

Minghui Zhu
Pennsylvania State University
muz16@psu.edu

Peng Liu
Pennsylvania State University
pliu@ist.psu.edu

ABSTRACT

Moving Target Defense techniques have been proposed to increase uncertainty and apparent complexity for attackers. When more than one Moving Target Defense techniques are effective to limit opportunities of an attack, it is required to compare these techniques and select the best defense choice. In this paper, we propose a three-layer model to evaluate and compare effectiveness of different Moving Target Defenses. This model is designed as an attempt to fill a gap among existing evaluation methods and works as a systematic framework for Moving Target Defense comparison.

Categories and Subject Descriptors

I.6 [**Computing Methodologies**]: Simulation And Modelling

General Terms

Defense; Model; Evaluation

Keywords

Moving Target Defense Techniques; Performance Evaluation; Three Layer Comparing Model

1. INTRODUCTION

Moving Target Defense (MTD) techniques are defense mechanisms via changing aspects of a system (e.g. an enterprise network supporting a business) to prevent attacks. The basic idea is to hide properties of a system that are required by attackers to leverage an exploit [12]. Plenty of MTD techniques [22, 25, 23, 6, 3, 11, 7, 13, 1, 26, 2, 31] have been developed to protect a system against various attacks.

In Table.1, we abstractly present effectiveness of different MTD instances against different attack instances. An MTD instance refers to the set of MTDs deployed in a system. In an MTD instance, each MTD is deployed to protect one or multiple programs in the system. For example, {*Address Space Layout Randomization, Code Sequence Randomization, Instruction Set Randomization*} is an MTD instance to mitigate buffer overflow attacks. Likewise, an attack instance refers to a set of attack actions launched against a system. These actions in an attack instance may form a multiple-step attack, or work independently to exploit the same vulnerability in a system. For example, {*Code Injection, Return-to-libc Attack, Return Oriented Programming Attack* } is an attack instance to exploit a buffer overflow vulnerability. Effectiveness represents in which aspects an MTD instance can mitigate an attack instance. For instance, *Instruction Set Randomization* can prevent *Code Injection* in a buffer overflow exploitation.

As shown in Table.1, an attack instance could be mitigated by multiple MTD instances. In such cases, it is necessary to compare applicable MTD instances and select the best one. In this paper, we study the problem of comparing MTD instances in a mission-centered way. From the point view of a system, a *mission* refers to one or multiple tasks to be completed for a certain purpose. For example, decrypting an encrypted file is one mission for a crypto system. Specifically, our problem could be stated as:

When an attack instance is generated against a system for a mission, we want to understand how the mission will be impacted when different MTD instances are deployed to neutralize the attack instance.

Defenses \ Attacks	$\{a_1, a_2, ...a_{n1}\}$	$\{a_2, a_4, ...a_{n2}\}$	...	$\{a_3, a_6, ...a_{np}\}$
$\{d_1, d_2, ..., d_{t1}\}$	$\{eff_1, eff_2, eff_3\}$	$\emptyset$	...	$\emptyset$
$\{d_3, d_6, ..., d_{t2}\}$	$\emptyset$	$\{eff_4, eff_5, eff_6\}$	...	$\emptyset$
...	...	...	...	...
$\{d_5, d_{10}, ..., d_{tq}\}$	$\{eff_1, eff_3\}$	$\emptyset$	...	$\{eff_1, eff_2, eff_3\}$

Table 1: MTD v.s. Attack: Each cell represents effectiveness of an MTD instance against an attack instance. If an MTD instance produces no effects on an attack instance, the corresponding cell is filled with an empty set.

Existing methods to evaluate the effectiveness of MTD techniques could be classified into two categories: low-level methods [28, 30] and high-level methods [32, 19].

Low-level methods evaluate MTDs through attack-based experiments. In the experiments, attacks are launched to compromise a program secured by MTDs. Low-level contexts are captured to reflect how attacks affect the program. Such effects are interpreted to evaluate the deployed MTDs. However, low-level methods express evaluation results as program status, which require extra interpretations before

MTD'14, November 3, 2014, Scottsdale, Arizona, USA.

http://dx.doi.org/10.1145/2663474.2663486.

being used as comparison metrics. Also, low-level methods reflect how attacks impact individual programs separately, without considering interactions between programs. Consequently, low-level methods are unable to capture effects propagation between programs. This makes low-level methods unable to evaluate effects of attacks on a whole system that contains multiple interconnected programs.

High-level methods evaluate MTDs based on simulation experiments, probability models, or a combination thereof. High-level methods require little knowledge about low-level contexts. Applicable scope of a high-level method varies from individual programs to large systems with multiple programs. Besides, high-level methods present evaluation results in an explicit way, which could be directly used for MTDs comparison. However, high-level methods evaluate MTD techniques based on abstracted information, rather than contexts of programs. Consequently, results from high-level methods are abstractions or approximations of reality, which makes it less convincing.

A method for MTD comparison requires merit both from low-level methods and high-level methods. However, the advantages of low-level methods are deficiencies of high-level methods, and vice versa. A method to incorporate merits of both is missing, and we regard such as a gap between low-level methods and high-level methods. The motivation of our work is to design an evaluation method to fill the gap, and use our method for MTD comparison.

In this paper, we propose a three-layer model as an attempt to fill the gap. The first layer is expected to capture the low-level contexts information of individual programs in a system; the second layer models interactions between different programs and enables our model to apply in a system scale; the third layer is designed as a user interface, which abstracts impacts of different attacks and defenses on a mission and provides an uniform and explicit channel for MTD techniques comparison. We design a case study to demonstrate the feasibility of such a model. A comprehensive evaluation on this model would be our future work.

2. REVIEW OF MTD TECHNIQUES AND EVALUATION METHODOLOGIES

2.1 MTD Techniques

2.1.1 Software-based Diversification

Existing works achieve Software-based Diversification via directly manipulating software or relying on compilers to produce diversification.

Software Manipulation The technique of software manipulation enhances security via changing software functionality to eliminate vulnerabilities or limit their exposure. Accretion, excision, and replacement of functionality are all leveraged as manipulation mechanisms. Input rectification is an example of accretion mechanism. Via inserting functions to manipulate inputs in software, input rectification converts each input to stay in a secure zone. For instance, the Pine email rectifier is developed to force messages into a specific constrained form in [24]. In order to simplify software behavior and possibly avoid vulnerabilities, functionality excision removes functional but non-critical sections of a program, which may introduce result noise, but normally will not cause failure [22, 25]. Functionality replacement switches between different implementations of the same function in a program, and thus generates system variation to neutralize attacks [23].

Compiler-generated Diversity Jackson et. al [11] presented two types of compiler-based diversity techniques: Malt-Variant Execution Environment (MVEE) and Massive-Scale Software Diversity (MSSD). Both of these techniques rely on automated tools, especially the compiler, to produce functionally equivalent, but internally different program variants.

Within MVEE, multiple semantically equivalent variants of a program are executed, and their behaviors are compared at synchronization points. Diverging behavior detected during synchronization indicates potential attacks. As early as the 1970s, Chen and Avizenis [6] developed a technique known as N-Version Programming, to generate $N \geq 2$ functionally equivalent programs. The mechanism required to achieve N-Version Programming could be completed by a compiler. The SQL interpreter known as SQLrand, prevents SQL injection by randomizing the Standard Query Language, and reflects the idea of MVEE [3].

MSSD refers to the idea of releasing diversified software variants to each user, and thus hides the internal structure of the software variants from attackers. Code Sequence Randomization (CSR) is a MSSD technique that uses instruction scheduling, call inlining, code hoisting, loop distribution, partial redundancy elimination, and many other compiler transformations to generate variants of machine codes. CSR could mitigate attacks that rely on knowledge about instruction at certain locations, such as return oriented programming attack [27, 5]. Equivalent Instructions (EI) is another MSSD technique. In software, EI stands for a set of instruction sequences that have identical effects and can be substituted for one another. Variants of machine code could be produced by replacing instructions through EI [11] (e.g., loop unrolling).

2.1.2 Runtime-based Diversifitcation

Many defenses techniques mitigate attacks via introducing diversification in runtime environments. Address Space Layout Randomization (ASLR) [28] is a widely known example of Runtime-based Diversification techniques. ASLR randomly arranges starting positions of key code and data areas in a process's address space, including the base memory address of the executable, the stack, the heap, and the libraries, to prevent an attacker from correctly jumping to a particular exploited function in memory. Kc et al. present an Instruction Set Randomization (ISR) algorithm based MVEE technique. With ISR, operators in machine instructions are encoded with a randomization key during compilation. Immediately before execution, operators in randomized instructions are required to be correctly decoded. In a program instance with ISR, code injected by attackers will not properly encoded but still go through the decoding process. This leads to illegal CPU instructions and exceptions, or at least different execution results from the program instance without ISR [14]. System Call Number Randomization (SCNR) is another Runtime-based Diversification technique related to ISR. Instead of encoding instructions, SCNR encodes system calls in a program. With SCNR, execution of injected code that contains incorrectly encoded system call leads to an error, or a different result from the normal program [7, 13].

2.1.3 Communication Diversification

Techniques of communication diversification protect systems against network related attacks, in a way to hide internal information or communication protocols. Christodorescu et al. propose to change the internals of a system, aiming at defending attacks that rely on internal knowledge of the system. For instance, periodically altering the schema of the database of a website, such as changing table titles, could mitigate SQL injections but without causing service failures [8]. The Mutable Networks (MUTE) [1] defense architecture proposed by Al-Shaer enables systems to automatically change communication configurations, and thus reduce adversary's capabilities in target scanning or exploiting. With IP randomization, a network would change its IP address frequently, and only synchronize new IP addresses with authorized users. Hence, attackers are segregated from the network. Another technique called Random Finger Printing in MUTE enables a system to interpret and modify its response to outside requests. Fake information of a system exposed to the public, such as OS type and application identity, and such information could hinder remote exploitation.

2.1.4 Dynamic Platform Techniques

Dynamic platform techniques (DPT) change platform properties to stop attacking processes [18]. Temporal changes (e.g. virtual machine rotation) and diversity (e.g. multiple variants execution) are widely used as DPT techniques for system protection. DPT techniques at compiler-level have been discussed above, so we focus on machine-level/OS-level DPT techniques, which includes migration-based techniques, server-switching techniques, and self-cleansing techniques. Migration-based techniques, such as Talent [17], leverage OS-level virtualization to create a virtual execution environment for migrating a running application across different platforms and preserving the state of the application. DPT could also be achieved by switching between different implementations of servers. Saidane et al. [26] propose the deployment of redundant servers with diverse software (e.g. Apache and IIS) in a web system. Redundancy improves system availability, and diversification increases the difficulty of vulnerability exploitation. Self-cleansing techniques change current platform instance without diversification, such as reimaging the OS. SCIT [2] achieves self-cleansing by decommissioning a server that has been exposed to the Internet for a predetermined period of time, and launching a pristine copy of the server. Ideas of DPT are also used in works on assessing software reliability. For instance, Zhang and Liu [31] designed and deployed diverse-drivers based replicas of virtual machines (Heter-device), to compare run-time behaviors of drivers from different vendors. Replicas in a Heter-device load heterogeneous drivers for identical virtual devices. The behavior of these drivers are compared to capture inconsistencies as evidence of un-trustworthiness.

2.2 MTD Evaluation Methodologies

2.2.1 Attack-based Experiments

Attack-based experiments evaluate an MTD by measuring the amount of effort to compromise a program guarded by the defense. Shacham et al. explore the effectiveness of ASLR on Linux against a return-to-libc attack [10]. In their study, PAX ASLR was deployed to randomize Apache's user address space, which leads to an unpredictable offset of libc in virtual memory. However, they correctly guessed the base address of libc through brute force, and compromised the Apache server with a subsequent return-to-libc attack. On average 216 seconds were required for a successful guess (under PAX with ≤ 24 bits randomization space) [28].

Inspired by Schacham et al's work, Sovarel et al. developed a technique to attack ISR-protected servers. They incrementally broke the keys used by ISR to encode instructions, through a chosen-ciphertext attack. Experimental results demonstrated that on average 283.6 seconds were required to crack the 32 bytes keys used by ISR, with a success rate as high as 98% [30]. Attack-based experiments quantify security provided by MTD against a motivated adversary and bring insight as to how to improve MTD techniques (e.g. keys with longer length are required in ISR). During experiments, contexts of the program, including user level contexts (heap, stack, code, and data) and OS related contexts (e.g. opened files), are tracked and analyzed for evaluation. Attack-based experiments reflect what happens to a program when attacks occur. However, this evaluation method only reflects how attacks impact individual programs separately, without considering impacts propagation between programs. Further, the experiment results are represented by context statuses, which might need further interpretation to be used as comparison metrics.

2.2.2 Probability Model

Probability models are frequently used to analyze MTDs. First, probability models are used to analyze attacking success when MTDs are deployed. Zhang et al.'s work [32] provides a model to reflect the relationship between diversification frequency of MTD and successful probability of attacks. Second, probability models are proposed to understand the effort required to crack the randomization keys used by some MTDs [28, 30]. Third, Okhravi et al. [19] deduced a probability model to predict the expected time required for an attack to compromise a system that is protected by MTDs. In a probability model, specifics of systems, attacks and defenses are abstracted as probability parameters. For instance, in [30], the length of the randomization key in ISR is represented as a numeric variable related to the probability of a successful attack. A probability model deals with effects of attacks and defenses in a concise way, which are often represented as the success rate of an attack. However, in a probability model, high excision and abstraction of contexts at low-level might lead to missing of key information and mismatch between the model and the real scenario.

2.2.3 Simulation-based Evaluation

Zhang et al. [32] propose a simulation-based approach to study the effectiveness of MTD techniques. In their study, a simulating network based on NeSSi2 [4] is constructed, where Communication Diversification and machine-level Dynamic Platform are deployed as MTD techniques. Periodic attacks are conceptually launched based on an attack graph to compromise a mission. The success of an attack is determined by a pre-defined probability model and a specific MTD implementation. Results of the simulation quantify the relationship between successful attacks and different MTD settings (e.g. frequency of diversification). In their settings, for instance, the ratio of successful attacks is 50% when MTD is switched off, which shrinks to 16.2% if fre-

quency of diversification via MTD reaches 20 per time unit. Thus, such a method explicitly evaluates the effectiveness of MTD techniques, and enables defenders to implement MTD properly under different security requirements.

With a simulation-based approach, a whole system can be abstracted as numeric parameter (in the simulating network), and there is no restriction on the specifics of attacks and defenses. Thus, a simulation-based method provides people with a uniform approach to evaluate MTD techniques with the least effort. Nevertheless, no real system is involved for evaluation. Therefore, lessons learned from a simulation-based approach are not well-supported and might deviate from the real world.

2.2.4 Hybrid Approaches

Researchers also study MTD techniques based on hybrid approaches. Okhravi et al.'s work [19] explores the effectiveness of MTD techniques, especially DPT , using a combination of the three approaches explained above. They simulated a system with multiple servers based on Talent [9], and deployed machine-level DPTs as MTDs. On several machines in the system, two vulnerabilities CVE-2010-4165 and CVE-2010-4249 were set up. Attacks targeting at the two vulnerabilities were conducted periodically, and the success rate of the attacks were recorded. Okhravi et al. evaluated the effectiveness of an MTD by measuring the length of time when the MTD neutralizes the attacks. Different results are presented when different features are provided by defenses. Separate probability models are constructed to illustrate the effectiveness of different features. Inspired by the similarities between these probability models, Okhravi et al. generalize a probability model to predict the effectiveness of DPTs. The generalized model involves primary awareness of system specifics (e.g. the frequency of platform diversification), and estimates the effectiveness of an MTD in a concise way (e.g. success time of an attack).

Though experimental results are used in a hybrid method, evaluation from such a method could still be insufficient to reflect the reality. First, results from the experiments are based on information of coarse granularity, which can not demonstrate what exactly happens inside a system. For instance, all information of a *divide-by-zero error* is abstracted out as merely being a successful attack and how this attack would affect the system is neglected. Second, experimental results are abstracted out when being used for deducing the specific probability models. Third, the process of constructing a general model also involves abstraction, which could result in further loss of evaluation precision.

3. THERE IS A GAP

The above four methodologies evaluate MTD techniques based on different information and describe evaluation results in different ways. Further, these methodologies have different application scopes.

3.1 Low-level Methods

Attack-based experiments look into a program and focus on the contexts of the program when an attack happens. For instance, in the experimental attack against ASLR [28], multiple context objects are collected for evaluation: (1) location of the buffer to be overwritten; (2) base-address of libc in memory; (3) functions in libc to be attacked; (4) actions for compromising the program. We term such a method a low-level method, as it collects low-level context information for evaluation. As explained, a low-level method involves information with fine granularity, thus producing evaluation results with high accuracy.

However, it is difficult to compare MTDs directly based on evaluation through attack-based experiments. First, attack-based experiments present results through program contexts, which cannot be directly used as comparable metrics. It is necessary to interpret those contexts into attack status. For instance, in the attack experiment against ASLR [28], the status of the *instruction pointer register* could be interpreted as whether a return-to-libc is successful. Second, it is necessary to translate the attack status into explicit expression of effects on a mission. For example, arbitrary code-execution via a return-to-libc attack should be interpreted as indicating that the mission is under control by attackers.

Further, a low-level method works at the scope of an individual program. When used in a system with multiple interconnected programs, a low-level method has limited applicability. Such limitations are caused by the absence of a model for interaction between different programs. Attack effects that propagate through program interactions could not be captured by a low-level method. For instance, if $Program1$ interacts with $Program2$ via a common database, damage caused to the database by $Program1$ could propagate to $Program2$. Such propagated damage might lead to further effects on $Program2$, but they can not be identified by evaluation on $Program2$ by low-level methods.

3.2 High-level Methods

The other three methodologies (including simulation-based evaluation, probability model, and hybrid methods) evaluate MTD techniques from the scope of a whole system, which we call high-level methods. Contrary to low-level methods, high-level methods use information with coarser granularity for evaluation. First, system specifics are abstracted in high-level methods. In some cases, the only required information of a system is whether it contains certain vulnerabilities. Second, specifics of a defense are represented as general information: whether a defense is deployed; what attacks can be mitigated; the frequency of diversification if the defense involves diversification. Third, in a high-level method, attacks are often abstracted as two variables: success rate and effects. Success rate of an attack is either determined by a probability model or defense specifics. Effects of an attack are expressed explicitly (e.g. time duration when the attack succeeds), which could be directly used for comparison.

High-level methods also differ from low-level methods in term of application scope. Low-level methods focus on evaluating MTD in a single program, while high-level methods evaluate MTD in a whole system.

However, high-level methods possess the above two advantages via regarding a system as a black box. Low-level contexts are largely abstracted or neglected in high-level methods, so it is difficult for high-level methods to reflect what really happens in a system.

3.3 The Gap

We regard the differences between low-level methods and high-level methods as a gap. To the best of our knowledge there is no existing works on linking the two types of methods.

4. A NEW EVALUATION METHOD

In this paper, we propose a new evaluation method that tries to fill the gap between low-level methods and high-level methods. First, we would like to discuss the evaluation settings of our method, and we will present a concrete example of a practical system with such settings in the next section for a case study.

4.1 The Evaluation Settings

We evaluate MTD techniques against attacks in a system that consists of multiple interconnected programs. This system is designed to support a mission, and a set of attacks are launched to cause the mission to fail. The evaluation aims to understand how well MTD techniques protect the mission when attacks occur. And it is assumed that we have required knowledge of the system, the programs in the system, the mission supported by the system, the set of attacks, and the available MTD techniques.

The system Sys contains $\mathcal{N}$ programs $\{P_1, P_2, ..., P_\mathcal{N}\}$. Each program provides resources for Sys, and these resources refer to low-level contexts of a program. We denote contexts for P_i as a set of objects $\{c_{i1}, c_{i2}, ..., c_{it_i}\}$. We focus on five types of contexts: (1) data objects in the stack; (2) data objects in the heap; (3) code segments; (4) data objects in the data segment; (5) OS related contexts (e.g. files and sockets). To complete the mission Mis, it requires $\{c_{im_1}, c_{im_2}, ..., c_{im_{qi}}\}$ from P_i. The set of attacks to fail the mission are known as $Att = \{a_1, a_2, ..., a_{na}\}$. Each attack in Att aims at damaging resources from programs. We denote the resource objects from P_j targeted by a_k as $\{c_{ja_1}, c_{ja_2}, ..., c_{ja_{rk}}\}$. The available set of MTD techniques are $MTD = \{d_1, d_2, ..., d_{nd}\}$. In contrast with attacks, each MTD expects to secure resource of some programs. We denote the resource objects from P_g protected by d_h as $\{c_{gd_1}, c_{gd_2}, ..., c_{gd_{vh}}\}$.

We provide evidence here to indicate that the settings are reasonable. First, we assume that we know what programs are included in a system and whether they are interconnected. We believe this knowledge can be identified from system descriptions. Second, we assume that we have knowledge about contexts of a program and we summarized five types of contexts in the previous paragraph. In fact, such knowledge has been used by low-level methods in their evaluation settings [28, 30]. Thus, the second assumption is also supported. Third, we assume that we have the knowledge about the required set of resources to complete Mis. Though it is hard to precisely determine such a set, we can at least identify a superset of the required resources. For example, the set that includes all resources from each program in Sys must be such a superset. However, how to reduce redundancy in the superset depends on the mission specifics. Fourth, we assume that we have knowledge about what resources are targeted by an attack or protected by a defense. We believe such an assumption is well founded, because low-level methods [28, 30] evaluate MTD based on such knowledge.

4.2 Three-layer Model

Within the evaluation settings, we propose a three-layer model as an attempt to fill the gap. As shown in Figure.1, the first layer captures low-level contexts information from individual programs. For each program, a state machine is created to present how attacks cause damage to the program. The second layer connects state machines from the first layer, and models damage propagation between different programs. In this layer, a state machine is constructed to demonstrate how attacks damage a system. The third layer explicitly expresses evaluation results to work as a user interface. Information from the second layer is abstracted and included in the third layer.

Each layer is modeled as a set of state machines. First, state machine is a widely used and accepted computational model in computer science [15]. Second, our model is expected to describe change happens in a system when attacks and MTDs occur. So the ability of sate machine to deal with change/transition satisfies our requirements.

4.2.1 The First Layer: Program State Machine

In the first layer, a state machine (abbr. PSM) is created for each program. We anticipate a PSM to represent how an attack instance and an MTD instance would affect resources in the program. Before discussing details of a PSM, we would like to introduce the basics of a general state machine. A state machine could be presented as $\{Q, \sum, \delta, q_0, F\}$, in which, Q is a set of states; $\sum$ is a set for the input alphabet; $\delta : Q \times \sum \rightarrow Q$ is the state transition function; $q_0 \in Q$ is the start state, and $F \subseteq Q$ is the set of accept states [29].

Likewise, a PSM could be described as a tuple with 5 elements $\{Q\_P, \sum\_P, \delta\_P, q_0\_P, F\_P\}$. For simplicity, we discuss PSM_i for P_i as an example:

Q_P_i: States of PSM_i refer to damage on $\{c_{i1}, c_{i2}, ..., c_{it^i}\}$. For each $c_{ij}(1 \leq j \leq t_i)$, we present its status as a set of logic statements, and such statements are interpreted as damage to c_{ij}. And a set of applicable logic statements for c_{ij} is pre-defined. For instance, given a buffer Buf with nb bytes allocated space, there are two different statements about its length when Buf is accessed: (1) $len(Buf) \leq nb$ (2) $len(Buf) > nb$. Damage to Buf is null, if $len(Buf) \leq nb$; while damage to Buf is *Buffer Overflow* or *Buffer Overread*, if $len(Buf) > nb$. Three steps are necessary to determine the state of a program. First, for each $c_{ij}(1 \leq j \leq t_i)$, applicable logic statements are selected from the set of available statements. Second, each selected statement of c_{ij} is translated to damage on c_{ij}. The translation is completed according to a mapping function, and we denote it as $Dam(c_{ij})$. The function Dam depends on specifics of available logic statements. Third, the vector $\{Dam(c_{i1}), Dam(c_{i2}), ..., Dam(c_{it_i})\}$ is constructed as a state of PSM_i.

$\sum\_P_i$: Three types of input to P_i are regarded as input events to PSM_i: data from network traffics (I_DN); data from local devices (I_DF) (e.g. local disk and keyboard); data through Inter-process Communication (IPC) (I_DI). I_DN and I_DF could be affected by users, which are regarded as external inputs. I_DI are transparent to users, which are regarded as internal inputs. However, we do not model some special inputs, if either they are controlled by the operating system (e.g. SIGXCPU for CPU time limit excess), or it is too complicated to be handled (e.g. signal communication between threads).

$\delta\_P_i(Q\_P \times \sum\_P_i \rightarrow Q\_P_i)$: In this model, the transition function of a PSM is defined by the program itself, and each transition is caused by one execution of the program. Given context status and an input to a program, the period during which the program processes the input is regarded as one execution. Such a processing period starts when a program

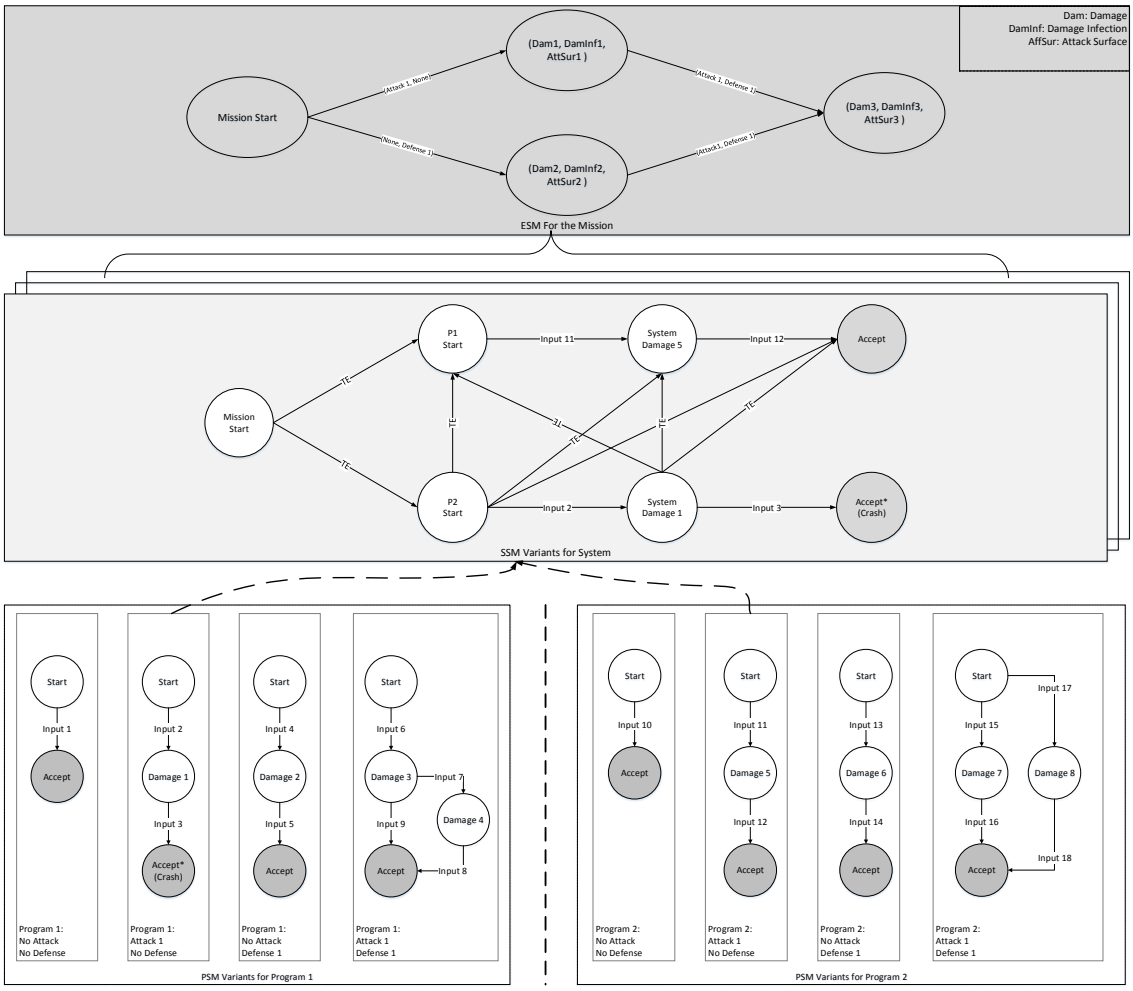


Figure 1: Three-layer model for MTD evaluation (The bottom part presents PSM variants for two programs in the first layer; the middle part presents SSM variants for the system consist of Program 1 and Program 2, as the second layer; the top part presents the ESM constructed based on the middle part, as the third layer. Only one defense and one attack are considered in this figure).

receives the input, and ends when status of the program remains unchanged without subsequent inputs. During one execution, if contexts changes and different damage is caused to resource objects, a transition occurs.

$q_0\_P_i$: When a program finishes initialization and receives no inputs, its state is regarded as the start state.

F_P_i: There are three different accept states. First, when no further damage can be caused to $\{c_{i1}, c_{i2}, ..., c_{it^i}\}$ by any input, P_i arrives at an accept state. Second, when P_i crashes and can receive further no inputs, P_i arrives at an accept state. Third, when a object in $\{c_{im_1}, c_{im_2}, ..., c_{im_{qi}}\}$ is so damaged that becomes unavailable to P_i and Mis can never be completed without the object, the mission arrives at an accept state. For instance, if the file for decryption keys is deleted by an attack, the mission to decrypt an encrypted file can never be completed.

PSM_i as above is defined when a program is under a certain pair of attack instance and MTD instance. However, when different pairs of attack instance and defense instance occur, PSM_i would be transformed in two aspects. First, new states could appear. For instance, a newly identified zero-day attack could lead to new damage on $\{c_{i1}, c_{i2}, ..., c_{it^i}\}$, which would introduce a new state. Second, new transitions could appear. For example, when new states appear, new transition must appear to connect these new states with existing states. When P_i is under no attacks and defenses, we define the PSM_i to be clear. When P_i is under a pair of attack instance and defense instance, we call the corresponding PSM_i a variant of the clear PSM_i and we label the PSM_i variant with the pair.

4.2.2 The Second Layer: System State Machine

The first layer models each program in Sys separately, without considering interactions between different programs. Consequently, the first layer could not tell how damage propagates through programs, and is unable to evaluate damage to the whole system.

In our model, the second layer is designed upon the first layer. This layer is expected to model interaction between programs, and present damage caused to Sys. Similar to the first layer, the second layer is designed as a state machine $\{Q\_S, \sum\_S, \delta\_S, q_0\_S, F\_S\}$, and we name it a System State

Machine (SSM). Algorithm.1 in Appendix.A demonstrates how an SSM is generated.

Two procedures are included in Algorithm.1. The first procedure *INTERACT(PSMs)* conservatively connects different PSMs to model interactions between different programs. Line 2 to line 6 initializes SSM. *Def*($q0_S$) defines q_0_S as the start state, which is a virtual state; F_S and δ_S are initialized as empty sets; TE stands for the input event when transition occurs between programs. Specifically, TE could be normal input event or null. Line 7 to line 12 expands SSM iteratively. All states, input events, and transitions in each PSM are included in SSM; a transition between the start state of SSM and the start state of each PSM is included to δ_S, and input event to trigger this transition is TE. Line 13 to line 24 connects different PSMs, to present interactions between different programs. Specifically, a transition is added between any two states from two different PSMs, with TE as the input event.

When completed, *INTERACT(PSMs)* returns a primitive SSM. According to such an SSM, any two programs in any states could interact with each other. We model interactions between programs in such a way, because it is hard to exactly determine how and when interactions would happen. For instance, if two programs access a common database, interactions between them could happen at any time. However, many interactions modeled by *INTERACT(PSMs)* never happen in practice, and thus, transitions to present those interactions are redundant. We designed the second procedure $PRUNE(SSM)$ to prune redundant transitions in SSM.

The procedure $PRUNE(SSM)$ enforces two pruning principles from line 28 to line 50 in Algorithm.1. First, if P_i can not actively communicate with P_j, any transitions from states in PSM_i to states in PSM_j are removed. If information could flow from P_i to P_j, we say that P_i can actively communicate with P_j, and the function $InterAct(P_i, P_j)$ returns $true$. Otherwise, $InterAct(P_i, P_j)$ returns $false$. Second, any transition from a state in F_S to another state in Q_S are removed. We only propose the two principles, and it is sure that all transitions removed based on such principles are actual redundancies.

In Algorithm.1, the SSM is constructed when the system is under a specific pair of attack instance and defense instance. However, as explained previously, different pairs of attack instance and MTD instance lead to PSM variants, and subsequently lead to variants of SSM.

4.2.3 The Third Layer: Evaluation State Machine

An SSM enables our model to analyze MTD techniques in a system scope. However, different states in an SSM represent damage to different components in the system. To understand what can happen to the whole system, it is necessary to scan the state space of an SSM and collect damage information. Also, damage information in an SSM is expressed as status of low-level contexts, which requires further interpretations for being understood by general users (e.g. the mission manager). In addition, to evaluate the effectiveness of different MTD techniques, it is necessary to use different SSM variants and switch between them.

To handle the above problems, we design the third layer, to work as a user interface. We expect this layer to support our model in three aspects: (1) it describes damage to the whole system in a centralized way; (2) damage is explicitly expressed, which requires little interpretation to be understood; (3) a uniform mechanism to evaluate different defenses against different attacks is provided. This layer is designed as a state machine $\{Q\_E, \sum\_E, \delta\_E, q_0\_E, F\_E\}$, and we call it an Evaluation State Machine (ESM). An ESM is built upon variants of SSM for Sys: $\{SSM_1, SSM_2, ..., SSM_{NS}\}$. Since each SSM variant is caused by a pair of attack instance and defense instance, we label different SSM variants with $\{(Att_1, Def_1), (Att_2, Def_2), ..., (Att_{NS}, Def_{NS})\}$. For each i, we call (Att_i, Def_i) the label of SSM_i. Now we discuss details of an ESM:

Q_E: For each i, SSM_i is abstracted as impacts of (Att_i, Def_i) on Mis, and such impacts are regarded as a state of an ESM. In this layer, the impacts are interpreted as three measurements: damage (Dam_i); damage infection ($DamInf_i$); and attack surface ($AttSur_i$). First, Dam_i represents damage to the mission caused by (Att_i, Def_i). In this layer, Dam_i is explicitly presented as damage to resource objects that are required by Mis. If in a state of SSM_i, the damage to a resource object is not null, then the damage would be assigned to this object. If multiple damages are assigned to a object, the largest damage or damages would be kept as damage to this object. So damage could be expressed as a vector $Dam = \{(c_{m1} : dam1), (c_{m2} : dam_2), ..., (c_{mt} : dam_y)\}$. How to interpret such a damage vector as damage to Mis depends on specifics of the mission, and we will give an example in the case study section. Further, any methods to quantify such a damage vector could be applied to get a quantitative metric of damage. Second, $DamInf_i$ represents how damage propagate through Sys to affect Mis. Given directly damaged objects in Dam_i, the remaining objects in Dam_i are regarded as infected ones. When an attack occurs, if an object directly interacts with external inputs and is damaged, this object is regarded as a directly damaged object. $DamInf_i$ is measured by the infected objects. Third, $AttSur_i$ measures the attack surface in Sys that can be used to damage Mis. We do not define attack surfaces in this paper, but our model is compatible with many attack surface definitions. Given a model of attack surface, $AttSur_i$ could be measured by the objects in Dam_i that are defined as attack surfaces by the model. For instance, Manadhata et al. [16] define attack surface as *subset of the system's resources potentially used in attacks on the system*, which can help us to measure $AttSur$.

$\sum\_E$: We regard different pairs of attack instance and defense instance (e.g. (Att_i, Def_i)) as inputs to ESM. Under different pairs, different SSM variants for Sys could appear and thus lead to different states in ESM.

δ_E: Assuming $AD = \{a_1, a_2, ..., a_{na}\} \cup \{d_1, d_2, ..., d_{nd}\}$ and $\mathcal{P}(AD)$ is the power set of AD, then $\subset$ is a partial ordering relation on $\mathcal{P}(AD)$. Thus we can create a Hasse diagram (HG) for $< \mathcal{P}(AD), \subset >$. Each node in HG is a set of attacks and defenses, and we divide the set into a pair of an attack instance and a defense instance. By representing each node with a state that is labelled by the pair, we get a new undirected graph EG. Each edge in EG represents a transition in ESM, and the transition direction is accordant with the direction of $\subset$. For instance, if node $a \subset$ node b in HG, then in EG, the transition starts with the state labelled by node a and ends with the state labelled by node b.

q_0_E: When Sys is under no attacks and no defenses, we think the system is clear and corresponding SSM is also clear. Start state is abstracted from the clear SSM.

F_E: Accept State of an ESM is a virtual state, and it is involved to maintain completeness of an ESM. If a state in Q_E can not transit to any other state, a transition between this state and the Accepts State would be created.

4.3 Towards Filling the Gap

The three-layer model is expected to fill the gap between low-level methods and high-level methods. The first layer identifies required contexts from individual programs as raw information, and interprets them as damage to Mis. In such a way, critical low-level contexts are captured for evaluation, and contexts are translated to reduce pre-processing on contexts in the upper layers. The second layer interconnects PSMs for different programs, and tries to model interactions between different programs. This layer enables our model to work in a system scale. The third layer is designed as an interface to interact with users (e.g. security manager). In this layer, impacts caused by attacks and defenses to Sys are explicitly abstracted as universal measurements, which can be directly used as comparison metrics.

We try to link the three layers as a whole, rather than roughly stacking them together to fill the gap. Three hooks are designed to consolidate the whole model. First, in the first layer, contexts are slightly abstracted to damage to resource objects. In this process, low-level contexts are translated into damage information, and the translation maintains sufficient low-level information for upper layers. Second, PSMs from the first layer are directly incorporated into the second layer, and mechanisms are designed to remove redundancy (or mistakes) during the incorporation. Hence, we think that the first layer and second layer are hooked. Third, an SSM variant from the second layer, as a whole, is interpreted as a state in the third layer. The interpretation is completed based on all states in the SSM, and required information for evaluation are reserved during this process. In this way, the second layer and third layer are correlated.

5. A CASE STUDY

In this section, we present a case study to elaborate our evaluation model.

5.1 Mission and System

The mission is to extract information from an encrypted and then compressed file. The compression is based on Huffman encoding and the encryption is based on AES algorithm.

Two steps are required for the mission. First, data in the file should be decompressed. Second, the decompressed data are required to be decrypted. Though in real world, a file should be firstly compressed and then decrypted, this reality does not affect how we describe the case study.

A system is designed to support the mission. The system consists of two open source programs: Huffman decoder [21] for decompression and OpenAES [20] for decryption. The two programs interact through a common directory. Whenever Huffman decoder gets a new file, it will decompress the file and move it into the common directory. OpenAES periodically scans the common directory and whenever a new file is found, OpenAES will decrypt the file.

```
1 char KeyBuf[128];
2 char DataBuf[128];
3 ...
4 fseek(DataFile, 0, SEEK_END)
5 file_size = ftell(Datafile);
6 // if (file_size > 128) exit(ERROR_SIZE);
7 fread(DataBuf, 1, file_size, DataFile);
8 ...
```

Listing 1: OpenAES code snippet

5.2 Attack and Defense

We insert a vulnerability into *OpenAES* to launch a buffer overflow attack. As shown in Listing.1, buffer `KeyBuf` and `DataBuf` are allocated adjacently on the stack. Buffer `DataBuf` and `KeyBuf` are used to store the key and encrypted data, and `KeyBuf` is in lower address space than `DataBuf` on the stack. We comment out the codes to check the size of the input file, as showed in line 6. Once more than 128 bytes of data are read into `DataBuf` from the input file, `KeyBuf` buffer would be overflowed.

During the mission, we suppose an adversary lunches a buffer overflow attack targeted on the revised *OpenAES* program. First, when the `KeyBuf` is partially overflowed, namely the length of data read from `DataFile` exceeds 128 bytes but less than 256 bytes, *OpenAES* cannot successfully decrypt the data since decryption key is polluted. Second, an attacker can carefully design a `DataFile` with 256 bytes. The first 128 bytes in the `DataFile` are padded with fake data that are encrypted with self-decided keys, and the remaining 128 bytes are padded with the self-decided keys. Then the `KeyBuf` will be completely overflowed by $129 \sim 256$ bytes in the `DataFile`, and *OpenAES* would decrypt the first 128 bytes with the modified keys and return the fake date. Third, *OpenAES* might get data more than 256 bytes from `DataFile`, which means it could pollute the `KeyBuf` or even other important data on the stack. In this case, the whole program might crash.

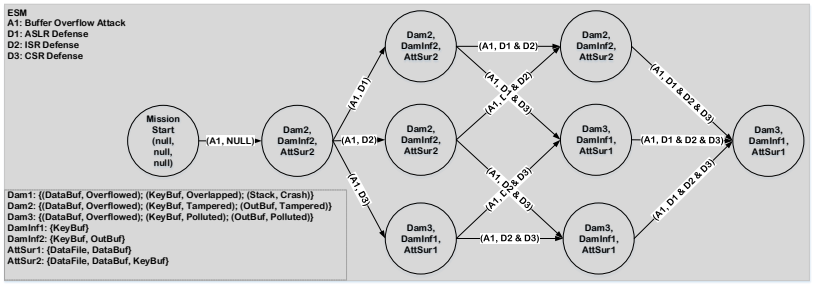
We deploy ASLR, ISR, and CSR as MTD techniques in the system to mitigate the buffer overflow attack.

5.3 Evaluation with Three-layer Model

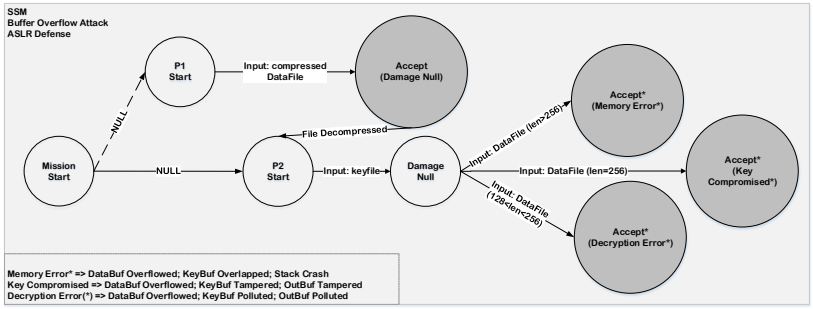
In this part, we explain how to evaluate above MTDs based on our three-layer model.

5.3.1 First Layer

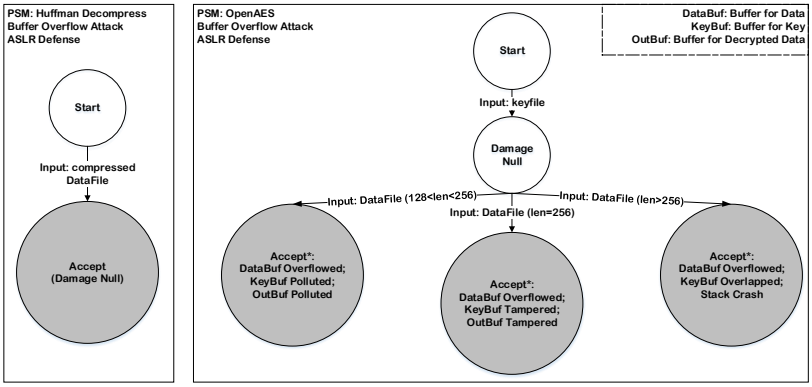
As mentioned, a buffer overflow attack is launched to fail the mission. We design a tool to track selected context objects in Huffman decoder and OpenAES. In Huffman decoder, context objects we tracked are: (1) the file that stores compressed data (`datafile`); (2) the memory buffer to load data from `datafile` (`databuf`); (3) the memory buffer for decompressed data (`outbuf`); (4) the file to store data in `outbuf` (`outfile`). In OpenAES, context objects we tracked are: (1) the file that stores encrypted data (`DataFile`); (2) the memory buffer to load data from `DataFile` (`DataBuf`); (3) the buffer to load decryption keys (`KeyBuf`); (4) the memory buffer to store decrypted data (`OutBuf`); (5) the file to store decrypted data (`OutFile`). As explained before, status of tracked contexts are interpreted as damage, and a state machine is created to illustrate transitions between damage. Figure.2-(c) shows how damage transits in Huffman decoder and OpenAES when the attack happens and ASLR is deployed. When different MTDs are deployed, such state machines would transform to variants.



(a)Layer 3: ESM for the mission (when different MTDs are deployed)



(b)Layer 2: SSM variant for the system (when ASLR is deployed)



(c)Layer 1: PSM variants for Huffman decoder and OpenAES (when ASLR is deployed)

Figure 2: A Case Study (How the buffer overflow attack affect the mission when different MTDs are deployed).

5.3.2 *Second Layer*

*PSM*s for Huffman decoder and OpenAES are merged in this layer. As explained, Huffman decoder and OpenAES interact with each other via a common directory. If and only if a file that contains attack payloads has been decompressed by Huffman decoder, the file can be loaded by OpenAES and compromise OpenAES. This is the only way can damage transits from Huffman decoder to OpenAES. As shown in Figure.2-(b), the only connection between Huffman decoder and OpenAES is the edge connecting accept state of Huffman decoder and start state of OpenAES.

This state machine presents how the attack damages the system when ASLR is deployed. When different MTDs are deployed, the whole state machine would change to different variants.

5.3.3 *Third Layer*

The third layer explicitly expresses how different MTD combinations could mitigate the launched attack. Impacts of the attack are presented in Figure.2-(a) when different MTD instances are deployed. *Dam* and *DamInf* on resource objects are described as explained previously. It is possible to further interpret these descriptions: $Dam1$ can be interpreted as a system crash; $Dam2$ means that the decryption key is under the control of attackers and thus, the decryption process in the mission is controlled by attackers; and $Dam3$ can be viewed as a decryption error, and which can be identified by a mission manager. The attack surfaces are determined based on measurements proposed by Manadhata et al. [16]. If CSR is deployed, damage caused by the attack would be identified by the mission manager, be-

cause the attack would lead to a system crash. Otherwise, the attackers control the decryption process, which could be regarded as a failure of the mission, and such a failure could not be identified. Also, deploying CSR along with other MTDs can not improve the defense. So in this case, a good choice of MTD could be CSR.

6. CONCLUSION

In this paper, we present a three-layer model to evaluate effectiveness of MTDs. This model is proposed as an attempt to fill the gap between low-level methods and high-level methods. Each layer in this model is designed to integrate merits of the two types of methods. The first layer captures low-level contexts in separate programs; the second layer models damage propagation between different programs; the third layer works as a user interface to explicitly expresses evaluation results. We made a case study to demonstrate how our model works and the feasibility of this model to fill the gap. In our future works, we will evaluate this model in a more comprehensive way.

7. ACKNOWLEDGMENTS

This work was supported by ARO W911NF-09-1-0525 (MURI), NSF CNS-1223710, NSF CNS-1422594, ARO W911NF-13-1-0421 (MURI), and AFOSR W911NF1210055.

8. REFERENCES

[1] E. Al-Shaer. Toward network configuration randomization for moving target defense. In *Moving Target Defense*, pages 153–159. Springer, 2011.

[2] A. K. Bangalore and A. K. Sood. Securing web servers using self cleansing intrusion tolerance (scit). In *Dependability, 2009. DEPEND'09. Second International Conference on*, pages 60–65. IEEE, 2009.

[3] S. W. Boyd and A. D. Keromytis. Sqlrand: Preventing sql injection attacks. In *Applied Cryptography and Network Security*, pages 292–302. Springer, 2004.

[4] R. Bye, S. Schmidt, K. Luther, and S. Albayrak. Application-level simulation for network security. In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, page 33. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008.

[5] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 559–572. ACM, 2010.

[6] L. Chen and A. Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8)*, pages 3–9, 1978.

[7] M. Chew and D. Song. Mitigating buffer overflows by operating system randomization. 2002.

[8] M. Christodorescu, M. Fredrikson, S. Jha, and J. Giffin. End-to-end software diversification of internet services. In *Moving Target Defense*, pages 117–130. Springer, 2011.

[9] M. Crouse and E. W. Fulp. A moving target environment for computer configurations using genetic algorithms. In *Configuration Analytics and Automation (SAFECONFIG), 2011 4th Symposium on*, pages 1–7. IEEE, 2011.

[10] S. Designer. return-to-libc attack. *Bugtraq, Aug*, 1997.

[11] T. Jackson, B. Salamat, A. Homescu, K. Manivannan, G. Wagner, A. Gal, S. Brunthaler, C. Wimmer, and M. Franz. Compiler-generated software diversity. In *Moving Target Defense*, pages 77–98. Springer, 2011.

[12] S. Jajodia, A. K. Ghosh, V. Swarup, C. Wang, and X. S. Wang. *Moving Target Defense*. Springer, 2011.

[13] X. Jiang, H. J. Wang, D. Xu, and Y.-M. Wang. Randsys: Thwarting code injection attacks with system service interface randomization. In *Reliable Distributed Systems, 2007. SRDS 2007. 26th IEEE International Symposium on*, pages 209–218. IEEE, 2007.

[14] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 272–280. ACM, 2003.

[15] L. Lamport. Computation and state machines. *Unpublished note*, 2008.

[16] P. K. Manadhata and J. M. Wing. An attack surface metric. *Software Engineering, IEEE Transactions on*, 37(3):371–386, 2011.

[17] H. Okhravi, A. Comella, E. Robinson, and J. Haines. Creating a cyber moving target for critical infrastructure applications using platform diversity. *International Journal of Critical Infrastructure Protection*, 5(1):30–39, 2012.

[18] H. Okhravi, T. Hobson, D. Bigelow, and W. Streilein. Finding focus in the blur of moving target techniques. 2013.

[19] H. Okhravi, J. F. Riordan, and K. M. Carter. Quantitative evaluation of dynamic platform techniques as a defensivemechanism ? In *Research in Attacks, Intrusions, and Defenses*. 2014.

[20] N. S. Ramli. Openaes portable c cryptographic library. https://code.google.com/p/openaes. Accessed: 2014-07-24.

[21] D. Richardson. huffman encoder/decoder. https://github.com/drichardson/huffman.git. Accessed: 2014-07-24.

[22] M. Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 324–334. ACM, 2006.

[23] M. Rinard. Manipulating program functionality to eliminate security vulnerabilities. In *Moving Target Defense*, pages 109–115. Springer, 2011.

[24] M. C. Rinard. Living in the comfort zone. *ACM SIGPLAN Notices*, 42(10):611–622, 2007.

[25] M. C. Rinard. Using early phase termination to eliminate load imbalances at barrier synchronization points. In *ACM SIGPLAN Notices*, volume 42, pages 369–386. ACM, 2007.

[26] A. Saidane, V. Nicomette, and Y. Deswarte. The design of a generic intrusion-tolerant architecture for web servers. *Dependable and Secure Computing, IEEE Transactions on*, 6(1):45–58, 2009.

[27] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561. ACM, 2007.

[28] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307. ACM, 2004.

[29] M. Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2012.

[30] A. N. Sovarel, D. Evans, and N. Paul. Where's the feeb? the effectiveness of instruction set randomization. In *Usenix Security*, 2005.

[31] S. Zhang and P. Liu. Assessing the trustworthiness of drivers. In *Research in Attacks, Intrusions, and Defenses*, pages 42–63. Springer, 2012.

[32] R. Zhuang, S. Zhang, S. A. DeLoach, X. Ou, and A. Singhal. Simulation-based approaches to studying effectiveness of moving-target network defense. In *National Symposium on Moving Target Research*, 2012.

APPENDIX

A. ALGORITHM FOR CONSTRUCTING SSM

Algorithm 1 Generation of SSM

Input: $\mathcal{N}$ PSMs:
$PSM_1 : \{Q\_P_1, \sum\_P_1, \delta\_P_1, q_0\_P_1, F\_P_1\}$,
$PSM_2 : \{Q\_P_2, \sum\_P_2, \delta\_P_2, q_0\_P_2, F\_P_2\}$,
...
$PSM_{\mathcal{N}} : \{Q\_P_{\mathcal{N}}, \sum\_P_{\mathcal{N}}, \delta\_P_{\mathcal{N}}, q_0\_P_{\mathcal{N}}, F\_P_{\mathcal{N}}\}$
Output: SSM: ▷ With interaction model
$\{Q\_S, \sum\_S, \delta\_S, q_0\_S, F\_S\}$

1: **procedure** INTERACT(PSMs) ▷ Model interactions
2: Def(q_0_S) ▷ Define Start State
3: $Q\_S := \{q_0\_S\}$ ▷ Assignment Expression
4: $F\_S := \emptyset$ ▷ Init Accept States
5: $\delta\_S := \emptyset$ ▷ Init Transition
6: $\sum\_S := \{TE\}$ ▷ Init Inputs
7: **for** i **from** 1 **to** $\mathcal{N}$ **do**
8: $Q\_S := Q\_S \cup Q\_P_i$
9: $F\_S := F\_S \cup F\_P_i$
10: $\sum\_S := \sum\_S \cup \sum\_P_i$
11: $\delta\_S := \delta\_S \cup \delta\_P_i \cup \{q_0\_S \xrightarrow{TE} q_0\_P_i\}$
12: **end for**
13: **for** j **from** 1 **to** $\mathcal{N}$ **do** ▷ PSM_j
14: **for** k **from** 1 **to** $\mathcal{N}$ **do** ▷ Other PSMs
15: **if** j==k **then** ▷ Exclude Self-interaction
16: `continue`
17: **end if**
18: **for all** $q\_j \in Q\_P_j$ **do** ▷ States in PSM_j
19: **for all** $q\_k \in Q\_P_k$ **do** ▷ Other States
20: $\delta\_S := \delta\_S \cup \{q\_j \xrightarrow{TE} q\_k\}$ ▷ [1]
21: **end for**
22: **end for**
23: **end for**
24: **end for**
25: **return** SSM ▷ SSM Initialized
26: **end procedure**
27: **procedure** PRUNE(SSM) ▷ Prune SSM
28: **for** j **from** 1 **to** $\mathcal{N}$ **do** ▷ Each PSM
29: **for** k **from** 1 **to** $\mathcal{N}$ **do** ▷ Other PSMs
30: **if** j==k **then**
31: `continue`
32: **end if**
33: **for all** $q\_j \in Q\_P_j$ **do** ▷ Single PSM States
34: **for all** $q\_k \in Q\_P_k$ **do** ▷ Other States
35: **if not** InterAct(PSM_j, PSM_k) **then**
36: $\delta\_S := \delta\_S / \{q\_j \xrightarrow{TE} q\_k\}$
37: **end if**
38: **if not** InterAct(PSM_k, PSM_j) **then**
39: $\delta\_S := \delta\_S / \{q\_k \xrightarrow{TE} q\_j\}$
40: **end if**
41: **if** $q\_j \in F\_P_j$ **then**
42: $\delta\_S := \delta\_S / \{q\_j \xrightarrow{TE} q\_k\}$
43: **end if**
44: **if** $q\_k \in F\_P_k$ **then**
45: $\delta\_S := \delta\_S / \{q\_k \xrightarrow{TE} q\_j\}$
46: **end if**
47: **end for**
48: **end for**
49: **end for**
50: **end for**
51: **return** SSM ▷ SSM constructed
52: **end procedure**

[1] An expression with format of $A \xrightarrow{B} C$ represents a transition rule that transits state A to state C when receiving input B.