A Formal Framework for Program Anomaly Detection

Xiaokui Shu, Danfeng (Daphne) Yao, and Barbara G. Ryder

Department of Computer Science, Virginia Tech
Blacksburg, VA 24060, USA
{subx, danfeng, ryder}@cs.vt.edu

Abstract. Program anomaly detection analyzes normal program behaviors and discovers aberrant executions caused by attacks, misconfigurations, program bugs, and unusual usage patterns. The merit of program anomaly detection is its independence from attack signatures, which enables proactive defense against new and unknown attacks. In this paper, we formalize the general program anomaly detection problem and point out two of its key properties. We present a unified framework to present any program anomaly detection method in terms of its detection capability. We prove the theoretical accuracy limit for program anomaly detection with an abstract detection machine. We show how existing solutions are positioned in our framework and illustrate the gap between state-of-the-art methods and the theoretical accuracy limit. We also point out some potential modeling features for future program anomaly detection evolution.

Keywords: Program Anomaly Detection; Unified Framework; Automata Theory; Detection Accuracy; Theoretical Accuracy Limit

1 Introduction

Security problems in program executions – caused by program bugs, inappropriate program logic, or insecure system designs – were first recognized by the Air Force, the Advanced Research Projects Agency (ARPA), and IBM in early 1970s. In 1972, Anderson pointed out the threat of subverting or exploiting a piece of software by a malicious user [2]. This threat was developed to a multitude of real-world attacks in the late 1980s and 1990s including buffer overflow, return-into-libc, denial of service (DoS), etc.

Defenses have been proposed against categories of attacks from the perspectives of hardware (e.g., NX bit), operating system (e.g., address space layout randomization), compiler (e.g., canaries) and software architecture (e.g., sandbox) [56]. Although these defenses create barriers to exploiting a program, they can be circumvented. For example, new program attacks are developed leveraging unattended/uninspected execution elements, such as return-oriented programming [51], jump-oriented programming [5,10], and non-control data attacks [11].

Denning proposed an intrusion detection expert system (IDES) in 1987 [15], which learns how a system should behave (normal profiles) instead of how it

should not (e.g., an attack). In this paper, we formalize one area of intrusion detection, namely *program anomaly detection* or *host-based intrusion detection* [52]. The area focuses on intrusion detection in the context of program executions. It was pioneered by Forrest et al., whose work was inspired by the analogy between intrusion detection for programs and the immune mechanism in biology [22].

Two major program anomaly detection approaches have been established and advanced: *n-gram-based dynamic normal program behavior modeling* and *automaton-based normal program behavior analysis*. The former was pioneered by Forrest [23], and the latter was formally introduced by Sekar et al. [50] and Wagner and Dean [59]. Other notable approaches include probabilistic modeling methods pioneered by Lee and Stolfo [40] and dynamically built state machine first proposed by Kosoresow and Hofmeyr [36]. Later work explored more fine-grained models [4, 28, 30] and combined static and dynamic analysis [24].

Evaluating the detection capability of program anomaly detection methods is always challenging [59]. Individual attacks do not cover all anomalous cases that a program anomaly detection system detects. Control-flow based metrics, such as average branching factor, are developed for evaluating specific groups of program anomaly detection methods [59]. However, none of the existing metrics is general for evaluating an arbitrary program anomaly detection system.

Several surveys summarized program anomaly detection methods from different perspectives and pointed out relations among several methods. Forrest et al. summarized existing methods from the perspective of system call monitoring [21]. Feng et al. formalized automaton based methods in [19]. Chandola et al. described program anomaly detection as a sequence analysis problem [8]. Chandola et al. provided a survey of machine learning approaches in [9]. The connection between an n-gram method and its automaton representation is first stated by Wagner [60]. Sharif et al. proved that any system call sequence based method can be simulated by a control-flow based method [52].

However, several critical questions about program anomaly detection have not been answered by existing studies and summaries.

1. How to formalize the detection capability of any detection method?
2. What is the theoretical accuracy limit of program anomaly detection?
3. How far are existing methods from the limit?
4. How can existing methods be improved towards the limit?

We answer all these questions in this paper. We unify *any existing or future* program anomaly detection method through its detection capability in a formal framework. We prove the theoretical accuracy limit of program anomaly detection methods and illustrate it in our framework. Instead of presenting every proposed method in the literature, we select and explain existing milestone detection methods that indicate the evolution of program anomaly detection. Our analysis helps understand the most critical steps in the evolution and points out the unsolved challenges and research problems.

The contributions of this paper are summarized as follows.

1. We formalize the general security model for program anomaly detection. We prove that the detection capability of a method is determined by the expressiveness of its corresponding language (Section 2).
2. We point out two independent properties of program anomaly detection: *precision* and *the scope of the norm*. We explain the relation between precision and deterministic/probabilistic detection methods (Section 2).
3. We present the theoretical accuracy limit of program anomaly detection with an abstract machine $\tilde{M}$. We prove that $\tilde{M}$ can characterize traces as precise as the executing program (Section 3).
4. We develop a hierarchal framework unifying any program anomaly detection method according to its detection capability. We mark the positions of existing methods in our framework and point out the gap between the state-of-the-art methods and the theoretical accuracy limit (Section 5).
5. We explain the evolution of program anomaly detection solutions. We envision future program anomaly detection systems with features such as full path sensitivity and higher-order relation description (Section 6).
6. We compare program anomaly detection with control-flow enforcement. We point our their similarities in techniques/results and explain their different perspectives approaching program/process security (Section 7).

2 Formal Definitions for Program Anomaly Detection

We formally define the problem of program anomaly detection and present the security model for detection systems. Then we discuss the two independent properties of a program anomaly detection method: the detection capability and the scope of the norm. Last, we give an overview of our unified framework.

2.1 Security Model

Considering both transactional (terminating after a transaction/computation) and continuous (constantly running) program executions, we define a **precise program trace** based on an *autonomous portion of a program execution*, which is a consistent and relatively independent execution segment that can be isolated from the remaining execution, e.g., an routine, an event handling procedure (for event-driven programs), a complete execution of a program, etc.

Definition 1. *A precise program trace* $\mathbf{T}$ *is the sequence of all instructions executed in an autonomous execution portion of a program.*

$\mathbf{T}$ is usually recorded as the sequence of all executed *instruction addresses*[1] and *instruction arguments*. In real-world executions, addresses of *basic block*s can be used to record $\mathbf{T}$ without loss of generality since instructions within a basic block are executed in a sequence.

We formalize the problem of program anomaly detection in Definition 2.

[1] Instruction addresses are unique identifiers of specific instructions.

Definition 2. *Program anomaly detection is a decision problem whether a precise program trace* $\mathbf{T}$ *is accepted by a language* L*.* L *presents the set of all normal precise program traces in either a deterministic means (*$L = \{\mathbf{T} \mid \mathbf{T}$ *is normal*$\}$*) or a probabilistic means (*$L = \{\mathbf{T} \mid P(\mathbf{T}) > \eta\}$*).*

In Definition 2, η is a probabilistic threshold for selecting normal traces from arbitrary traces that consist of instruction addresses. Either parametric and non-parametric probabilistic methods can construct probabilistic detection models.

In reality, no program anomaly detection system uses $\mathbf{T}$ to describe program executions due to the significant tracing overhead. Instead, a **practical program trace** is commonly used in real-world systems.

Definition 3. *A practical program trace* $\ddot{\mathbf{T}}$ *is a subsequence of a precise program trace* $\mathbf{T}$*. The subsequence is formed based on alphabet* Σ*, a selected/traced subset of all instructions, e.g., system calls.*

We list three categories of commonly used practical traces in real-world program anomaly detection systems. The traces result in *black-box*, *gray-box*, and *white-box* detection approaches with an increasing level of modeling granularity.

- *Black-box level traces:* only the communications between the process and the operating system kernel, i.e., system calls, are monitored. This level of practical traces has the smallest size of Σ among the three. It is the coarsest trace level while obtaining the trace incurs the smallest tracing overhead.
- *White-box level traces:* all (or a part of) kernel-space and user-space activities of a process are monitored. An extremely precise white-box level trace $\ddot{\mathbf{T}}$ is exactly a precise trace $\mathbf{T}$ where all instructions are monitored. However, real-world white-box level traces usually define Σ as the set of function calls to expose the call stack activity.
- *Gray-box level traces:* a limited white-box level without the complete static program analysis information [24], e.g., all control-flow graphs. Σ of a gray-box level trace only contains symbols (function calls, system calls, etc.) that appear in dynamic traces.

We describe the general security model of a real-world program anomaly detection system in Definition 4. The security model derives from Definition 2 but measures program executions using $\ddot{\mathbf{T}}$ instead of $\mathbf{T}$.

Definition 4. *A real-world program anomaly detection system* Λ *defines a language* L_Λ *(a deterministic or probabilistic set of normal practical program traces) and establishes an attestation procedure* G_Λ *to test whether a practical program trace* $\ddot{\mathbf{T}}$ *is accepted by* L_Λ*.*

A program anomaly detection system Λ usually consist of two phases: *training* and *detection*. Training is the procedure forming L_Λ and building G_Λ from known normal traces $\{\ddot{\mathbf{T}} \mid \ddot{\mathbf{T}}$ is normal$\}$. Detection is the runtime procedure testing incoming traces against L_Λ using G_Λ. Traces that cannot be accepted by L_Λ in the detection phase are logged or aggregated for alarm generation.

2.2 Detection Capability

The detection capability of a program anomaly detection method Λ is its ability to detect attacks or anomalous program behaviors. Detection capability of a detection system Λ is characterized by the precision of Λ. We define *precision* of Λ as the ability of Λ to distinguish different precise program traces in Definition 5. This concept is independent of whether the scope of the norm is deterministically or probabilistically established (discussed in Section 2.3).

Definition 5. *Given a program anomaly detection method Λ and any practical program trace $\ddot{\mathbf{T}}$ that Λ accepts, the precision of Λ is the average number of precise program traces $\mathbf{T}$ that share an identical subsequence $\ddot{\mathbf{T}}$.*

Our definition of program anomaly detection system precision is a generalization of *average branching factor* (using regular grammar to approximate the description of precise program traces) [59] and *average reachability measure* (using context-free grammar to approximate the description of precise program traces) [28]. The generation is achieved through the using of $\mathbf{T}$, the most precise description of a program execution. `average` in Definition 5 can be replaced by other aggregation function for customized evaluation.

We formalize the relation between the expressive power of L_Λ (defined by detection method Λ) and the detection capability of Λ in Theorem 1.

Theorem 1. *The detection capability of a program anomaly detection method Λ is determined by the expressive power of the language L_Λ corresponding to Λ.*

Proof. Consider two detection methods Λ_1 (L_{Λ_1}) and Λ_2 (L_{Λ_2}) where Λ_1 is more precise than Λ_2, one can always find two precise program traces $\mathbf{T}_1/\mathbf{T}_2$, so that $\mathbf{T}_1/\mathbf{T}_2$ are expressed by L_{Λ_1} in two different practical traces $\ddot{\mathbf{T}}_{1\Lambda_1}/\ddot{\mathbf{T}}_{2\Lambda_1}$, but they can only be expressed by L_{Λ_2} as an identical $\ddot{\mathbf{T}}_{\Lambda_2}$. Because the definition of the norm is subjective to the need of a detection system, in theory, one can set $\mathbf{T}_1/\mathbf{T}_2$ to be normal/anomalous, respectively. In summary, Λ_1 with a more expressive L_{Λ_1} can detect the attack $\mathbf{T}_2$ via practical trace $\ddot{\mathbf{T}}_{2\Lambda_1}$, but Λ_2 cannot.

Theorem 1 enables the comparison between detection capabilities of different detection systems through their corresponding languages. It lays the foundation of our unified framework. The more expressive L_Λ describes a normal precise trace $\mathbf{T}$ through a practical trace $\ddot{\mathbf{T}}$, the less likely an attacker can construct an attack trace $\mathbf{T}'$ mimicking $\mathbf{T}$ without being detected by Λ.

2.3 Scope of the Norm

Not all anomaly detection systems agree on whether a specific program behavior (a precise program trace $\mathbf{T}$) is normal or not. Even given the set of all practical program traces Σ^* with respect to a specific alphabet Σ (e.g., all system calls), two detection systems Λ_1 and Λ_2 may disagree on whether a specific $\ddot{\mathbf{T}} \in \Sigma^*$ is normal or not. Σ^* denotes the set of all possible strings/traces over Σ.

Definition 6. *The scope of the norm S_Λ (of a program anomaly detection system Λ) is the selection of practical traces to be accepted by L_Λ.*

While L_Λ is the set of all normal practical traces, S_Λ emphasizes on the selection process to build L_Λ, but not the expressive power (detection capability) of L_Λ. S_Λ does not influence the detection capability of Λ.

For instance, VPStatic [19] (denoted as Λ_s) utilizes a pushdown automaton (PDA) to describe practical program traces. Therefore, its precision is determined by the expressiveness of context-free languages[2]. S_{Λ_s} is all *legal control flows* specified in the binary of the program. VtPath [18] (denoted as Λ_v) is another PDA approach, but S_{Λ_v} is defined based on dynamic traces. Since dynamic traces commonly forms a subset of all feasible execution paths, there exists $\ddot{\mathbf{T}}$ not in the training set of Λ_2. Thus, $\ddot{\mathbf{T}}$ will be recognized as anomalous by Λ_2 yet normal by Λ_1. Because the precisions of Λ_1 and Λ_2 are the same, Λ_2 can be made to detect $\ddot{\mathbf{T}}$ as normal by including $\ddot{\mathbf{T}}$ in its training set (changing S_{Λ_v}).

There are two types of scopes of the norm:

- **Deterministic scope of the norm** is achieved through a deterministic language $L_\Lambda = \{\ddot{\mathbf{T}} \mid \ddot{\mathbf{T}} \text{ is normal}\}$. Program anomaly detection systems based on finite state automata (FSA), PDA, etc. belong to this category.
- **Probabilistic scope of the norm** is achieved through a stochastic language $L_\Lambda = \{\ddot{\mathbf{T}} \mid P(\ddot{\mathbf{T}}) > \eta\}$. Different probability threshold η results in different S_Λ and different L_Λ/Λ. Program anomaly detection systems based on hidden Markov model, one-class SVM, etc. belong to this category.

2.4 Overview of Our Unified Framework

We develop a unified framework presenting any program anomaly detection method Λ. Our framework unifies Λ by the expressive power of L_Λ.

We illustrate our unified framework in Fig. 1 showing its hierarchical structure[3]. In Fig. 1, L-1 to L-4 indicate four major precision levels with decreasing detection capabilities according to the expressive power of L_Λ. The order of precision levels marks the potential of approaches within these levels, but not necessarily the practical detection capability of a specific method[4]. Our design is based on both the well-defined levels in Chomsky hierarchy and the existing milestones in the evolution of program anomaly detection.

L-1: context-sensitive language level (most powerful level)
L-2: context-free language level
L-3: regular language level
L-4: restricted regular language level (least powerful level)

[2] Context-sensitive languages correspond to pushdown automata.
[3] The hierarchy is reasoned via Chomsky hierarchy [12], which presents the hierarchical relation among formal grammars/languages.
[4] For example, one detection approach Λ_a in L-2 without argument analysis could be less capable of detecting attacks than an approach Λ_b in L-3 with argument analysis.

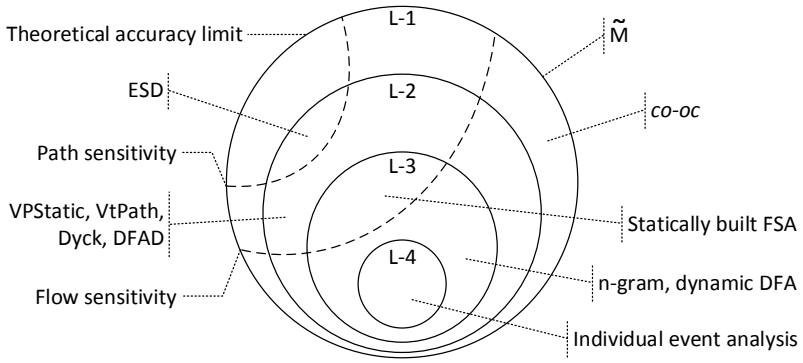

Fig. 1. The hierarchy of our program anomaly detection framework. L-1 to L-4 are four major precision levels with decreasing detection capabilities.

The restricted regular language corresponding to L-4 does not enforce specific adjacent elements for any element in a string (program trace). Two optional properties within L-1, L-2 and L-3 are *path sensitivity* and *flow sensitivity* (Section 5.2). We prove the theoretical accuracy limit (the outmost circle in Fig. 1) in Section 3 with an abstract detection machine $\tilde{M}$. We abstract existing methods in Section 4 and identify their positions in our unified framework in Section 5. We present details of our framework and point out the connection between levels in our framework and grammars in Chomsky hierarchy in Section 5. We describe the evolution from L-4 methods to L-1 methods in Section 6.2.

3 Accuracy Limit of Program Anomaly Detection

We describe an abstract detection machine, $\tilde{M}$, to differentiate between any two precise program traces. Thus, $\tilde{M}$ detects any anomalous program traces given a scope of the norm. A practical program trace $\ddot{\mathbf{T}}$ that $\tilde{M}$ consumes is a precise program trace $\mathbf{T}$. We prove that $\tilde{M}$ has the identical capability of differentiating between traces (execution paths) as the program itself. Therefore, $\tilde{M}$ is the accuracy limit of program anomaly detection models.

3.1 The Ultimate Detection Machine

The abstract machine $\tilde{M}$ is a 9-tuple $\tilde{M} = (Q, \Sigma, \Gamma, A, \Omega, \delta, s_0, Z, F)$ where the symbols are described in Table 1. $\tilde{M}$ operates from s_0. If an input string/trace $\ddot{\mathbf{T}}$ reaches a final state in F, then $\ddot{\mathbf{T}}$ is a normal trace.

$\tilde{M}$ consists of three components: *a finite state machine*, *a stack* Π, and *a random-access register* Υ. In $\tilde{M}$, both Π and Υ are of finite sizes. Indirect addressing, i.e., the value of a register can be dereferenced as an address of another register, is supported by Υ and $A \subset \Omega$. Because a random-access register can simulate a stack, Π can be omitted in $\tilde{M}$ without any computation power loss.

Table 1. Descriptions of symbols in $\tilde{M}$. All sets are of finite sizes.

	Name	Description
Q	States	Set of states
Σ	Input alphabet	Set of input symbols
Γ	Stack alphabet	Set of symbols on the stack
A	Register addresses	Set of addresses of all registers
Ω	Register alphabet	Set of symbols stored in registers
δ	Transition relation	Subset of $Q \times (\Sigma \cup \{\varepsilon\}) \times \ \Gamma \times \Omega^* \times Q \times \Gamma^* \times \Omega^*$
s_0	Initial state	State to start, $s_0 \in Q$
Z	Initial stack symbol	Initial symbol on the stack, $Z \subseteq Q$
F	Final states	Set of states where $\ddot{\mathbf{T}}$ is accepted, $F \subseteq Q$

ε denotes an empty string.
Ω^* or Γ^* denotes a string over alphabet Ω or Γ, respectively.

We keep Π in $\tilde{M}$ to mimic the execution of a real-world program. It helps extend $\tilde{M}$ for multi-threading (Section 3.3) and unify $\tilde{M}$ in our framework (Section 5.1).

A transition in $\tilde{M}$ is defined by δ, which is a mapping from $(\Sigma \cup \{\varepsilon\}) \times Q \times \Gamma \times \Omega^*$ to $Q \times \Gamma^* \times \Omega^*$. Given an input symbol $\sigma \in \Sigma \cup \{\varepsilon\}$, the current state $q \in Q$, the stack symbol $\gamma \in \Gamma$ (stack top), and all symbols in the register $\{\omega_i \mid \omega_i \in \Omega,\ 0 \leq i \leq |A|\}$, the rules in δ chooses a new state $q' \in Q$, pops γ, pushes zero or more stack symbols $\gamma_0\gamma_1\gamma_2\ldots$ onto the stack, and update $\{\omega_i\}$.

3.2 The Equivalent Abstract Machine of An Executing Program

We state the precision of the abstract detection machine $\tilde{M}$ in Theorem 2 and interpreter both *sufficiency* and *necessity* aspects of the theorem.

Theorem 2. *$\tilde{M}$ is as precise as the target program; $\tilde{M}$ can detect any anomalous traces if the scope of the norm is specified and $\tilde{M}$ is constructed.*

Sufficiency: $\tilde{M}$ has the same computation power as any real-world executing program so that $L_{\tilde{M}}$ can differentiate any two precise program traces.

Necessity: detection machines that are less powerful than $\tilde{M}$ cannot differentiate any two arbitrary precise program traces of the target program.

Although a Turing machine is commonly used to model a real-world program in execution, an executing program actually has limited resources (the tape length, the random access memory size or the address symbol count) different from a Turing machine. This restricted Turing machine is abstracted as linear bounded automaton [34]. We prove Theorem 2 by Lemma 1 and Lemma 2.

Lemma 1. *A program that is executing on a real-world machine is equivalent to a linear bounded automaton (LBA).*

Lemma 2. *$\tilde{M}$ is equivalent to a linear bounded automaton.*

Proof. We prove that $\tilde{M}$ is equivalent to an abstract machine $\ddot{M}$ and $\ddot{M}$ is equivalent to an LBA, so $\tilde{M}$ is equivalent to an LBA.

$\ddot{M}$ is an abstract machine similar to $\tilde{M}$ except that Υ (the register) in $\tilde{M}$ is replaced by two stacks Π_0 and Π_1. $size(\Upsilon) = size(\Pi_0) + size(\Pi_1)$.

We prove that $\tilde{M}$ and $\ddot{M}$ can simulate each other below.

- One random-access register can simulate one stack with simple access rules (i.e., last in, first out) enforced. Thus, Υ can be split into *two non-overlapping register sections* to simulate Π_0 and Π_1.
- Π_0 and Π_1 together can simulate Υ by filling Π_0 with initial stack symbol Z to its maximum height and leaving Π_1 empty. All the elements in Π_0 are counterparts of all the units in Υ. The depth of an element in Π_0 maps to the address of a unit in Υ. To access an arbitrary element e in Π_0, one pops all elements higher than e in Π_0 and pushes them into Π_1 until e is retrieved. After the access to e, elements in Π_1 are popped and pushed back into Π_0.

$\ddot{M}$ is equivalent to an LBA: $\ddot{M}$ consists of a finite state machine and three stacks, Π (same as Π in $\tilde{M}$), Π_0, Π_1 (the two-stack replacement of Υ in $\tilde{M}$). $\ddot{M}$ with three stacks is equivalent to an abstract machine with two stacks [48]. Two stacks is equivalent to a finite tape when concatenating them head to head. Thus, $\ddot{M}$ is equivalent to an abstract machine consisting of a finite state machine and a finite tape, which is a linear bounded automaton.

In summary, $\tilde{M}$ is equivalent to an LBA and Lemma 2 holds. □

3.3 Usage and Discussion

Operation of $\tilde{M}$: $\tilde{M}$ consists of a random-access register Υ and a stack Π. The design of $\tilde{M}$ follows the abstraction of an executing program. Π simulates the call stack of a process and Υ simulates the heap. The transition δ in $\tilde{M}$ is determined by the input symbol, symbols in Υ and the top of Π, which is comparable to a real-world process. Given a precise trace $\mathbf{T}$ of a program, $\tilde{M}$ can be operated by emulating all events (instructions) of $\mathbf{T}$ through $\tilde{M}$.

Multi-threading handling: although $\tilde{M}$ does not model multi-threading program executions, it can be easily extended to fulfill the job. The basic idea is to model each thread using an $\tilde{M}$. Threads creating, forking and joining can be handled by copying the *finite state machine* and *stack* of an $\tilde{M}$ to a new one or merging two $\tilde{M}$s. δ needs to be extended according to the shared register access among different $\tilde{M}$s as well as the joining operation between $\tilde{M}$s.

Challenges to realize $\tilde{M}$ in practice: $\tilde{M}$ serves as a theoretical accuracy limit. It cannot be efficiently realized in the real world because

1. The number of normal precise traces is infinite.
2. The scope of the norm requires a non-polynomial time algorithm to learn.

The first challenge is due to the fact that a trace $\ddot{\mathbf{T}}$ of a program can be of any length, e.g., a continuous (constantly running) program generates traces

in arbitrary lengths until it halts. Most existing approaches do not have the problem because they only model short segments of traces (e.g., n-grams with a small n [21], first-order automaton transition verification [19]).

Pure dynamic analysis cannot provide a complete scope of the norm. The second challenge emerges when one performs comprehensive static program analysis to build $\tilde{M}$. For example, one well-known exponential complexity task is to discover induction variables and correlate different control-flow branches.

4 Abstractions of Existing Detection Methods

In this section, we analyze existing program anomaly detection models and abstract them in five categories. We identify their precision (or detection capability) in our framework in Section 5.

Finite state automaton (FSA) methods represent the category of program anomaly detection methods that explicitly employs an FSA. Kosoresow and Hofmeyr first utilized a deterministic finite state automaton (DFA) to characterize normal program traces [36] via black-box level traces (building a DFA for system call traces). Sekar et al. improved the FSA method by adopting a limited gray-box view [50]. Sekar's method retrieves *program counter* information for every traced system call. If two system calls and program counters are the same, the same automaton state is used in the FSA construction procedure.

Abstraction: all FSA methods explicitly build an FSA for modeling normal program traces. A transition of such an FSA can be described in (1). p_i is an automaton state that is mapped to one or a set of program states. Each program state can be identified by a system call (black-box level traces) or a combination of system call and program counter (gray-box level traces). $s*$ denotes a string of one or more system calls.

$$p_i \xrightarrow{s*} p_{i+1} \tag{1}$$

***n*-gram methods** represent the category of program anomaly detection methods those utilize sequence fragments to characterize program behaviors. n-grams are n-item-long[5] substrings[6] of a long trace, and they are usually generated by sliding a window (of length n) on the trace. The assumption underlying n-gram methods is that *short trace fragments are good features differentiating normal and anomalous long system call traces* [23]. A basic n-gram method tests whether every n-gram is in the known set of normal n-grams [21].

Abstraction: a set of n-gram (of normal program behaviors) is equivalent to an FSA where each state is an n-gram [60]. A transition of such an FSA can be described in (2). The transition is recognized when there exist two normal n-grams, $(s_0, s_1, \ldots, s_{n-1})$ and $(s_1, \ldots, s_{n-1}, s_n)$, in any normal program traces.

$$(s_0, s_1, \ldots, s_{n-1}) \xrightarrow{s_n} (s_1, \ldots, s_{n-1}, s_n) \tag{2}$$

[5] n can be either a fixed value or a variable [45, 63].

[6] Lookahead pair methods are subsequent variants of n-gram methods [35].

Since n-gram methods are built on a membership test, various deterministic [45,62] and probabilistic [17,61] means are developed to define the scope of the norm (the set of normal n-grams) and perform the membership test. And system call arguments were added to describe system calls in more details [7, 55, 57].

Pushdown automaton (PDA) methods represent the category of program anomaly detection methods those utilize a PDA or its equivalents to model program behaviors. DPA methods are more precise than FSA methods because they can simulate user-space call stack activities [18].

An FSA connects control-flow graphs (CFGs) of all procedures into a *monomorphic* graph, which lacks the ability to describe direct or indirect recursive function calls [31, 59]. A PDA, in contrast, keeps CFGs isolated and utilizes a stack to record and verify function calls or returns [18,19,29]. Thus, it can describe recursions. However, only exposing the stack when system calls occur is not enough to construct a deterministic DPA [19]. There could be multiple potential paths transiting from one observed stack state Γ_i to the next stack state Γ_{i+1}. Giffin et al. fully exposed all stack activities in Dyck model [30] by embedding loggers for function calls and returns via binary rewriting.

Abstraction: a typical PDA method consumes white-box level traces [19] or gray-box level traces [43]. The internal (user-space) activities of the running program between system calls are simulated by the PDA. Denote a system call as s and a procedure transition as f. We describe the general PDA transition in (3) where Γ_i/Γ_{i+1} is the stack before/after the transition, respectively.

$$p_i, \Gamma_i \xrightarrow{f \text{ or } s} p_{i+1}, \Gamma_{i+1} \tag{3}$$

System call arguments can be added to describe calls in more details like they are used in previous models. In addition, Bhatkar et al. utilized data-flow analysis to provide complex system call arguments verification, e.g., unary and binary relations [4]. Giffin et al. extended system call arguments to environment values, e.g., configurations, and built an environment-sensitive method [28].

Probabilistic methods differ from deterministic program anomaly detection approaches that they use stochastic languages to define the scope of the norm (Section 2.3). Stochastic languages are probabilistic counterparts of deterministic languages (e.g., regular languages). From the automaton perspective, stochastic languages correspond to automata with probabilistic transition edges.

Abstraction: existing probabilistic program anomaly detection methods are probabilistic counterparts of FSA, because they either use n-grams or FSA with probabilistic transitions edges. Typical probabilistic detection methods include hidden Markov model (HMM) [61, 64], classification methods [16, 37, 41, 46], artificial neural network [27], data mining approaches [40], etc. Gu et al. presented a supervised statistical learning model, which uses control-flow graphs to help the training of its probabilistic model [32].

Probabilistic FSA does not maintain call stack structures[7], and it constrains existing probabilistic approaches from modeling recursions precisely. In theory, FSA and probabilistic FSA only differ in their scopes of the norm; one is deterministic the other is probabilistic. The precision or detection capability of the two are the same as explained in Section 2.3. Different thresholds in parametric probabilistic models define different scopes of the norm, but they do not directly impact the precision of a model.

N-variant methods define the scope of the norm with respect to the current execution path under detection. They are different from the majority of detection methods that define the scope of the norm as all possible normal execution paths.

In N-variant methods, a program is executed with n replicas [14]. When one of them is compromised, others – that are executed with different settings or in different environments – could remain normal.

The anomaly detection problem in N-variant methods is to tell whether one of the concurrently running replicas is behaving differently from its peers; N-variant methods calculate the behavior distance among process replicas. Gao et al. proposed a deterministic alignment model [25] and probabilistic hidden Markov model [26] to calculate the distances.

Abstraction: existing N-variant models are FSA or probabilistic FSA equivalents. The precision is limited by their program execution description based on n-grams. This description forms a deterministic/probabilistic FSA model underlying the two existing N-variant methods.

5 Unification Framework

We develop a hierarchical framework to uniformly present any program anomaly detection method in terms of its detection capability. We identify the detection capabilities of existing program anomaly detection methods (Section 4) and the theoretical accuracy limit (Section 3) in our framework.

5.1 Major Precision Levels of Program Anomaly Detection

We abstract any program anomaly detection method Λ through its equivalent abstract machine. Λ is unified according to the language L_Λ corresponding to the abstract machine. We summarize four major precision levels defined in our unified framework in Table 2. We describe them in detail below in the order of an increasing detection capability.

L-4: restricted regular language level. The most intuitive program anomaly detection model, which reasons events individually, e.g., a system call with or without arguments. No event correlation is recorded or analyzed.

An L-4 method corresponds to a restricted FSA, which accepts a simple type of regular languages L_4 that does not enforce specific adjacent elements for any element in a string (practical program trace $\ddot{\mathbf{T}}$).

[7] Probabilistic PDA has not been explored by the anomaly detection community.

Table 2. Precision levels in our framework (from the most to the least accurate).

Precision Levels	Limitation[a]	Chomsky Level
L-1 methods	Program execution equivalent	Type-1 grammars
L-2 methods	First-order reasoning	Type-2 grammars
L-3 methods	Cannot pair calls and returns	Type-3 grammars
L-4 methods	Individual event test	Type-3 grammars

[a] The key feature that distinguishes this level from a level of higher precision.
[b] The restricted regular language does not enforce specific adjacent events for any event in a program trace.

L-4 methods are the weakest detection model among the four. It is effective only when anomalous program executions can be indicated by individual events. For example, `sys_open()` with argument "/etc/passwd" indicates an anomaly.

A canonical example of L-4 methods is to analyze individual system events in system logs and summarize the result through machine learning mechanisms [16].

L-3: regular language level. The intermediate program anomaly detection model, which records and verifies *first-order event transitions* (i.e., the relation between a pair of adjacent events in a trace, which is an extra feature over L-4 methods) using type-3 languages (regular grammar).

An L-3 method corresponds to an FSA, which naturally describes first-order transitions between states. Each state can be defined as one or multiple events, e.g., a system call, n-grams of system calls. One state can be detailed using its arguments, call-sites, etc. The formal language L_3 used to describe normal traces in an L-3 method is a type-3 language.

L-3 methods consume black-box traces. The monitoring is efficient because internal activities are not exposed. However, L-3 methods cannot take advantage of exposed internal activities of an executing program. For example, procedure returns cannot be verified by L-3 methods because L_3 (regular grammar) cannot pair arbitrary events in traces; L-3 methods cannot model recursions well.

Canonical L-3 methods include DFA program anomaly detection [36], n-grams methods [23], statically built FSA [50], and FSA with call arguments [7].

L-2: context-free language level. The advanced program execution model, which verifies first-order event transitions with full knowledge (aware of any instructions) of program internal activities in the user space.

An L-2 method corresponds to a PDA, which expands the description of an FSA state with a stack (last in, first out). Procedure transitions (nested call-sites) can be stored in the stack so that L-2 methods can verify the return of each function/library/system call. The formal language L_2 used to describe normal traces in an L-2 method is a type-2 (context-free) language.

Gray-box or white-box traces are required to expose program internal activities (e.g., procedure transitions) so that the stack can be maintained in L-2 methods. Walking the stack when a system call occurs is an efficient stack expose technique [18]. However, the stack change between system calls is nondetermin-

istic. A more expensive approach exposes every procedure transition via code instrumentation [30], so that the stack is deterministic.

Canonical L-2 methods include VPStatic [19], VtPath [18], and Dyck [30]. Moreover, Bhatkar et al. applied argument analysis with data-flow analysis (referred to by us as DFAD) [4], and Giffin et al. correlated arguments and environmental variables with system calls (referred to by us as ESD) [28].

L-1: context-sensitive language level. The most accurate program anomaly detection model in theory, which verifies higher-order event transitions with full knowledge of program internal activities.

L-1 methods correspond to a higher-order PDA, which extends a PDA with non-adjacent event correlations, e.g., induction variables.

We develop Theorem 3 showing that higher-order PDA and $\tilde{M}$ (Section 3) are equivalent in their computation power. The proof of Theorem 2 points out $\tilde{M}$ and linear bounded automaton (LBA) are equivalent. Therefore, these three are abstract machines representing the most accurate program anomaly detection.

The formal language L_1 used to describe normal traces in an L-1 method is a type-1 (context-sensitive) language.

We formally describe an L-1 method, i.e., $\tilde{M}$, in Section 3. Any other LBA or $\tilde{M}$ equivalents are also L-1 methods.

Theorem 3. *L-1 methods are as precise as the target executing program.*

We provide a proof sketch for Theorem 3. First, $\tilde{M}$ is as precise as the executing program (Theorem 2 in Section 3). Next, we give the sketch of the proof that the abstract machine of L-1 methods, i.e., a higher-order PDA, is equivalent to $\tilde{M}$: a higher-order PDA characterizes cross-serial dependencies [6], i.e., correlations between non-adjacent events. Therefore, it accepts context-sensitive languages [53], which is type-1 languages accepted by $\tilde{M}$.

Although the general context-sensitive model (higher-order PDA or $\tilde{M}$) has not been realized in the literature, Shu et al. demonstrated the construction of a constrained context-sensitive language model (*co-oc* in Fig. 1) [54]. The model quantitatively characterizes the co-occurrence relation among non-adjacent function calls in a long trace. Its abstraction is the context-sensitive language Bach [49].

Probabilistic detection methods and our hierarchy are orthogonal. The reason is that probabilistic models affect the scope of the norm definition, but not the precision of the detection (explained in Section 2.3). For instance, a probabilistic FSA method (e.g., HMM [61,64], classification based on n-grams [16,46]) is an L-3 method. It cannot model recursion well because there is no stack in the model. The precision of a probabilistic FSA method is the same as the precision of a deterministic FSA method, except that the scope of the norm is defined probabilistically. A similar analysis holds for N-variant methods. All existing N-variant methods [25,26] are L-3 methods.

Instruction arguments are part of events in T. However, argument analysis does not increase the precision level of a detection method, e.g., an n-gram approach with argument reasoning is still an L-3 approach.

Table 3. Terminology of Sensitivity in Program Anomaly Detection.

	Calling context	**Flow**	**Path**	**Environment**
Sensitive Objects	Call sites	Instruction order	Branch dependency	Arguments configurations
Precision Level[a]	L-4	L-3	L-2	L-2
Description[b]	RL	RL	CFL	CFL

[a] The least precise level required to specify the sensitivity.
[b] The least powerful formal language required for describing the sensitivity. RL: regular language. CFL: context-free language.

5.2 Sensitivity in a Nutshell

We describe optional properties (sensitivities) within L-1 to L-3 in our hierarchical framework with respect to sensitivity terms introduced from program analysis. We summarize the terminology of sensitivity in Table 3 and explain them and their relation to our framework.

Calling context sensitivity concerns the call-site of a call. In other words, it distinguishes a system/function call through different callers or call-sites. Calling-context-sensitive methods[8] are more precise than non-calling-context-sensitive ones because mimicked calls from incorrect call-sites are detected.

Flow sensitivity concerns the order of events according to control-flow graphs (CFG). Only legal control flows according to program binaries can be normal, e.g., [50]. Flow sensitive methods bring static program analysis to anomaly detection and rule out illegal control flows from the scope of the norm.

Path sensitivity concerns the branch dependencies among the binary (in a single CFG or cross multiple CFGs). Infeasible paths (impossibly co-occurring basic blocks or branches) can be detected by a path-sensitive method. Full path sensitivity requires exponential time to discover. Existing solutions take some path-sensitive measures, e.g., Giffin et al. correlated less than 20 branches for a program in ESD [28].

Environment sensitivity correlates execution paths with executing environments, e.g., arguments, configurations, environmental variables. Several types of infeasible paths such as an executed path not specified by the corresponding command line argument can be detected by an environment-sensitive method [28]. Environment sensitivity is a combination of techniques including data-flow analysis, path-sensitive analysis, etc.

[8] Calling context sensitivity (or context sensitivity in short) in program analysis should be distinguished from the term *context-sensitive* in formal languages. The latter characterizes cross-serial dependencies in a trace, while the former identifies each event (e.g., a system call) in a trace more precisely.

6 Attack/Detection Evolution and Open Problems

In this section, we describe the evolution of program anomaly detection systems using the precision levels in our framework. New solutions aim to achieve better precision and eliminate mimicry attacks. We point out future research directions from both precision and practicality perspectives.

6.1 Inevitable Mimicry Attacks

Mimicry attacks are stealthy program attacks designed to subvert program anomaly detection systems by mimicking normal behaviors. A mimicry attack exploits false negatives of a specific detection system Λ. The attacker constructs a precise trace $\mathbf{T}'$ (achieving the attack goal) that shares the same practical trace $\ddot{\mathbf{T}}_\Lambda$ with a normal $\mathbf{T}$ to escape the detection.

The first mimicry attack was described by Wagner and Soto [60]. They utilized an FSA (regular grammar) to exploit the limited detection capability of n-gram methods (L-3 methods). In contrast, L-2 methods, such as [18,19,30], invalidate this type of mimicry attacks with context-free grammar description of program traces. However, mimicry attacks using context-free grammars, e.g., [20,38], are developed to subvert these L-2 methods.

As program anomaly detection methods evolve from L-4 to L-1, the space for mimicry attacks becomes limited. The functionality of mimicry attacks decreases since the difference between an attack trace and a normal trace attenuates. However, an attacker can always construct a mimicry attack against any real-world program anomaly detection system. The reason is that the theoretical limit of program anomaly detection (L-1 methods) cannot be efficiently reached, i.e., $\tilde{M}$ described in Section 3 requires exponential time to build.

6.2 Evolution From L-4 to L-1

A detection system Λ_1 rules out mimicry traces from a less precise Λ_2 to achieve a better detection capability. We describe the upgrade of detection systems from a lower precision level to a higher precision level. Intuitively, L-3 methods improve on L-4 methods as L-3 methods analyze the order of events. We summarize four features to upgrade an L-3 method (abstracted as a general FSA) to L-2 and L-1 methods in Fig. 2.

① expanding a state horizontally (with neighbor states)
② describing details of states (call-sites, arguments, etc.)
③ expanding a state vertically (using a stack)
④ revealing relations among non-adjacent states

The four features are not equally powerful for improving the precision of an anomaly detection method. ① and ② are complementary features, which do not change the major precision level of a method. ③ introduces a stack and

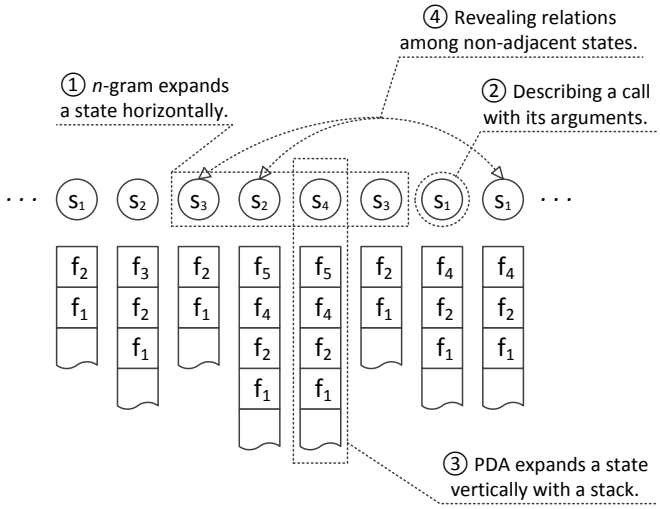

Fig. 2. Four approaches for improving a basic L-3 method (FSA).

upgrades an L-3 method to an L-2 method. ④ discovers cross-serial dependencies and establishes a context-sensitive language [53], which results in an L-1 method.

Most of the existing program anomaly detection methods can be explained as a basic L-3 method plus one or more of these features. L-3 with ① yields an n-gram method [23]. L-3 with ② was studied in [44]. L-3 with ③ is a basic L-2 method. More than one feature can be added in one program anomaly detection system. L-3 with ① and ② was studied by Sufatrio and Yap [55] and Gaurav et al. [57]. L-3 with ② and ③ was studied by Bhatkar et al. [4] and Giffin et al. [28]. $\tilde{M}$ (described in Section 3) provides ③ and ④ as basic features. ② can be added to $\tilde{M}$ to describe each state in more details.

6.3 Open Problems

We point out several open problems in program anomaly detection research.

Precision As illustrated in our framework (Fig. 1), there is a gap between the theoretical accuracy limit (the best L-1 method) and the state-of-the-art approaches in L-2 (e.g., ESD [28]) and constrained L-1 level (e.g., co-oc [54]).

L-2 models: existing detection methods have not reached the limit of L-2 because none of them analyze the complete path sensitivity. Future solutions can explore a more complete set of path sensitivity to push the detection capability of a method towards the best L-2 method.

L-1 models: higher-order relations among states can then be discovered to upgrade an L-2 method to L-1. However, heuristics algorithms need to be developed to avoid exponential modeling complexity. Another choice is to develop constrained L-1 approaches (e.g., co-oc [54]), which characterize some aspects of higher-order relations (e.g., co-occurrence but not order).

Probabilistic models: existing probabilistic approaches, i.e., probabilistic FSA equivalents, are at precision level L-3. Probabilistic PDA and probabilistic LBA can be explored to establish L-2 and even L-1 level probabilistic models.

Practicality In contrast to the extensive research in academia, the security industry has not widely adopted program anomaly detection technologies. No products are beyond L-3 level with black-box traces [33]. The main challenges are *eliminating tracing overhead* and *purifying training dataset*.

Tracing overhead issue: L-2 and L-1 methods require the exposure of user-space program activities, which could result in over 100% tracing overhead on general computing platforms [3]. However, Szekeres et al. found that the industry usually tolerates at most 5% overhead for a security solution [56].

Polluted training dataset issue: most existing program anomaly detection approaches assume the training set contains only normal traces. Unless the scope of the norm is defined as *legal control flows*, which can be extracted from the binary, the assumption is not very practical for a real-world product. A polluted training dataset prevents a precise learning of the scope of the norm for a detection model, which results in false negatives in detection.

7 Control-Flow Enforcement Techniques

Control-flow enforcements, e.g., Control-Flow Integrity (CFI) [1] and Code-Pointer Integrity (CPI) [39], enforce control-flow transfers and prevent illegal function calls/pointers from executing. They evolve from the perspective of attack countermeasures [56]. They are equivalent to one category of program anomaly detection that defines the scope of the norm as legal control flows [52].

7.1 Control-Flow Enforcement

Control-flow enforcement techniques range from the protection of return addresses, the protection of indirect control-flow transfers (CFI), to the protection of all code pointers (CPI). They aim to protect against control-flow hijacks, e.g., stack attacks [42]. We list milestones in the development of control-flow enforcement techniques below (with an increasing protection capability).

Return address protection: Stack Guard [13], Stack Shield [58].
Indirect control-flow transfer protection: CFI [1], Modular CFI [47].
All code pointer protection: CPI [39].

7.2 Legal Control Flows as the Scope of the Norm

In program anomaly detection, one widely adopted definition of the scope of the norm S_Λ is *legal control flows* (Section 2.3); only basic block transitions that obey the control flow graphs are recognized as normal. The advantage of such definition is that the boundary of S_Λ is clear and can be retrieved from the binary. No labeling is needed to train the detection system. This definition

of S_Λ leads to a fruitful study of constructing automata models through static program analysis[9], e.g., FSA method proposed by Sekar et al. [50] and PDA method proposed by Feng et al. [18].

7.3 Comparison of the Two Methods

We discuss the connection and the fundamental difference between control-flow enforcement and program anomaly detection.

Connection Modern control-flow enforcement prevents a program from executing any illegal control flow. It has the same effect as the category of program anomaly detection that defines the scope of the norm as legal control flows. From the functionality perspective, control-flow enforcement even goes one step further; it halts illegal control flows. Program anomaly detection should be paired with prevention techniques to achieve the same functionality.

Difference A system can either learn from attacks or normal behaviors of a program to secure the program. Control-flow enforcement evolves from the former perspective while program anomaly detection evolves from the latter. The specific type of attacks that control-flow enforcement techniques tackle is *control-flow hijacking*. In other words, control-flow enforcement techniques do not prevent attacks those obey legal control flows, e.g., brute force attacks. Program anomaly detection, in contrast, detects attacks, program bugs, anomalous usage patterns, user group shifts, etc. Various definitions of the scope of the norm result in a rich family of program anomaly detection models. One family has the same detection capability as control-flow enforcement.

8 Conclusion

Program anomaly detection is a powerful paradigm discovering program attacks without the knowledge of attack signatures. In this paper, we provided a general model for systematically analyzing *i)* the detection capability of any model, *ii)* the evolution of existing solutions, *iii)* the theoretical accuracy limit, and *iv)* the possible future paths toward the limit.

Our work filled a gap in the literature to unify deterministic and probabilistic models with our formal definition of program anomaly detection. We presented and proved the theoretical accuracy limit for program anomaly detection. We developed a unified framework presenting any existing or future program anomaly detection models and orders them through their detection capabilities. According to our unified framework, most existing detection approaches belong to the regular and the context-free language levels. More accurate context-sensitive language models can be explored with pragmatic constraints in the future. Our framework has the potential to serve as a roadmap and help researchers approach the ultimate program defense without attack signature specification.

[9] Dynamically assigned transitions cannot be precisely pinpointed from static analysis.

Acknowledgments. This work has been supported by ONR grant N00014-13-1-0016. The authors would like to thank Trent Jaeger, Gang Tan, R. Sekar, David Evans and Dongyan Xu for their feedback on this work. The authors would like to thank anonymous reviewers for their comments on stochastic languages.

References

1. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity. In: Proceedings of ACM CCS. pp. 340–353 (2005)
2. Anderson, J.P.: Computer security technology planning study. Tech. rep., DTIC (October 1972)
3. Bach, M., Charney, M., Cohn, R., Demikhovsky, E., Devor, T., Hazelwood, K., Jaleel, A., Luk, C.K., Lyons, G., Patil, H., Tal, A.: Analyzing parallel programs with Pin. Computer 43(3), 34–41 (March 2010)
4. Bhatkar, S., Chaturvedi, A., Sekar, R.: Dataflow anomaly detection. In: Proceedings of IEEE S & P (May 2006)
5. Bletsch, T., Jiang, X., Freeh, V.W., Liang, Z.: Jump-oriented programming: A new class of code-reuse attack. In: Proceedings of ASIACCS. pp. 30–40 (2011)
6. Bresnan, J., Kaplan, R.M., Peters, S., Zaenen, A.: Cross-serial dependencies in Dutch. In: The formal complexity of natural language, pp. 286–319. Springer (1987)
7. Canali, D., Lanzi, A., Balzarotti, D., Kruegel, C., Christodorescu, M., Kirda, E.: A quantitative study of accuracy in system call-based malware detection. In: Proceedings of ISSTA. pp. 122–132 (2012)
8. Chandola, V., Banerjee, A., Kumar, V.: Anomaly detection for discrete sequences: A survey. IEEE TKDE 24(5), 823–839 (May 2012)
9. Chandola, V., Banerjee, A., Kumar, V.: Anomaly detection: A survey. ACM Computing Surveys 41(3), 1–58 (July 2009)
10. Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.R., Shacham, H., Winandy, M.: Return-oriented programming without returns. In: Proceedings of ACM CCS. pp. 559–572 (2010)
11. Chen, S., Xu, J., Sezer, E.C., Gauriar, P., Iyer, R.K.: Non-control-data attacks are realistic threats. In: Proceedings of USENIX Security. vol. 14, pp. 12–12 (2005)
12. Chomsky, N.: Three models for the description of language. IRE Transactions on Information Theory 2(3), 113–124 (1956)
13. Cowan, C., Pu, C., Maier, D., Hintony, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q.: StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: Proceedings of USENIX Security. vol. 7, pp. 5–5 (1998)
14. Cox, B., Evans, D., Filipi, A., Rowanhill, J., Hu, W., Davidson, J., Knight, J., Nguyen-Tuong, A., Hiser, J.: N-variant systems: A secretless framework for security through diversity. In: Proceedings of USENIX Security. vol. 15 (2006)
15. Denning, D.E.: An intrusion-detection model. IEEE TSE 13(2), 222–232 (February 1987)
16. Endler, D.: Intrusion detection: Applying machine learning to Solaris audit data. In: Proceedings of ACSAC. pp. 268–279 (December 1998)
17. Eskin, E., Lee, W., Stolfo, S.: Modeling system calls for intrusion detection with dynamic window sizes. In: Proceedings of DARPA Information Survivability Conference and Exposition II. vol. 1, pp. 165–175 (2001)

18. Feng, H.H., Kolesnikov, O.M., Fogla, P., Lee, W., Gong, W.: Anomaly detection using call stack information. In: Proceedings of IEEE S & P (2003)
19. Feng, H., Giffin, J., Huang, Y., Jha, S., Lee, W., Miller, B.: Formalizing sensitivity in static analysis for intrusion detection. In: Proceedings of IEEE S & P. pp. 194–208 (May 2004)
20. Fogla, P., Sharif, M., Perdisci, R., Kolesnikov, O., Lee, W.: Polymorphic blending attacks. In: Proceedings of USENIX Security. pp. 241–256 (2006)
21. Forrest, S., Hofmeyr, S., Somayaji, A.: The evolution of system-call monitoring. In: Proceedings of ACSAC. pp. 418–430 (December 2008)
22. Forrest, S., Perelson, A., Allen, L., Cherukuri, R.: Self-nonself discrimination in a computer. In: Proceedings of IEEE S & P. pp. 202–212 (May 1994)
23. Forrest, S., Hofmeyr, S.A., Somayaji, A., Longstaff, T.A.: A sense of self for Unix processes. In: Proceedings of IEEE S & P. pp. 120–128 (1996)
24. Gao, D., Reiter, M.K., Song, D.: On gray-box program tracking for anomaly detection. In: Proceedings of USENIX Security. vol. 13, pp. 8–8 (2004)
25. Gao, D., Reiter, M.K., Song, D.: Behavioral distance for intrusion detection. In: Proceedings of RAID. pp. 63–81 (2006)
26. Gao, D., Reiter, M.K., Song, D.: Behavioral distance measurement using hidden Markov models. In: Proceedings of RAID. pp. 19–40 (2006)
27. Ghosh, A.K., Schwartzbard, A.: A study in using neural networks for anomaly and misuse detection. In: Proceedings of USENIX Security. vol. 8, pp. 12–12 (1999)
28. Giffin, J.T., Dagon, D., Jha, S., Lee, W., Miller, B.P.: Environment-sensitive intrusion detection. In: Proceedings of RAID. pp. 185–206 (2006)
29. Giffin, J.T., Jha, S., Miller, B.P.: Detecting manipulated remote call streams. In: Proceedings of USENIX Security. pp. 61–79 (2002)
30. Giffin, J.T., Jha, S., Miller, B.P.: Efficient context-sensitive intrusion detection. In: Proceedings of NDSS (2004)
31. Gopalakrishna, R., Spafford, E.H., Vitek, J.: Efficient intrusion detection using automaton inlining. In: Proceedings of IEEE S & P. pp. 18–31 (May 2005)
32. Gu, Z., Pei, K., Wang, Q., Si, L., Zhang, X., Xu, D.: Leaps: Detecting camouflaged attacks with statistical learning guided by program analysis. In: Processing of DSN (June 2015)
33. Hofmeyr, S.: Primary response technical white paper, `http://www.ttivanguard.com/austinreconn/primaryresponse.pdf`, accessed August 2015
34. Hopcroft, J.E.: Introduction to automata theory, languages, and computation. Pearson Education India (1979)
35. Inoue, H., Somayaji, A.: Lookahead pairs and full sequences: a tale of two anomaly detection methods. In: Proceedings of ASIA. pp. 9–19 (2007)
36. Kosoresow, A., Hofmeyer, S.: Intrusion detection via system call traces. IEEE Software 14(5), 35–42 (September 1997)
37. Kruegel, C., Mutz, D., Robertson, W., Valeur, F.: Bayesian event classification for intrusion detection. In: Proceedings of ACSAC. pp. 14–23 (December 2003)
38. Kruegel, C., Kirda, E., Mutz, D., Robertson, W., Vigna, G.: Automating mimicry attacks using static binary analysis. In: Proceedings of USENIX Security. vol. 14, pp. 11–11 (2005)
39. Kuznetsov, V., Szekeres, L., Payer, M., Candea, G., Sekar, R., Song, D.: Code-pointer integrity. In: Proceedings of USENIX OSDI. pp. 147–163 (2014)
40. Lee, W., Stolfo, S.J.: Data mining approaches for intrusion detection. In: Proceedings of USENIX Security. vol. 7, pp. 6–6 (1998)
41. Liao, Y., Vemuri, V.: Use of k-nearest neighbor classifier for intrusion detection. Computers & Security 21(5), 439–448 (2002)

42. Liebchen, C., Negro, M., Larsen, P., Davi, L., Sadeghi, A.R., Crane, S., Qunaibit, M., Franz, M., Conti, M.: Losing control: On the effectiveness of control-flow integrity under stack attacks. In: Proceedings of ACM CCS (2015)
43. Liu, Z., Bridges, S.M., Vaughn, R.B.: Combining static analysis and dynamic learning to build accurate intrusion detection models. In: Proceedings of IWIA. pp. 164–177 (March 2005)
44. Maggi, F., Matteucci, M., Zanero, S.: Detecting intrusions through system call sequence and argument analysis. IEEE TDSC 7(4), 381–395 (October 2010)
45. Marceau, C.: Characterizing the behavior of a program using multiple-length n-grams. In: Proceedings of NSPW. pp. 101–110 (2000)
46. Mutz, D., Valeur, F., Vigna, G., Kruegel, C.: Anomalous system call detection. ACM TISSEC 9(1), 61–93 (February 2006)
47. Niu, B., Tan, G.: Modular control-flow integrity. SIGPLAN Notices 49(6), 577–587 (June 2014)
48. Papadimitriou, C.H.: Computational complexity. John Wiley and Sons Ltd. (2003)
49. Pullum, G.K.: Context-freeness and the computer processing of human languages. In: Proceedings of ACL. pp. 1–6. Stroudsburg, PA, USA (1983)
50. Sekar, R., Bendre, M., Dhurjati, D., Bollineni, P.: A fast automaton-based method for detecting anomalous program behaviors. In: Proceedings of IEEE S & P. pp. 144–155 (2001)
51. Shacham, H.: The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: Proceedings of ACM CCS. pp. 552–561 (2007)
52. Sharif, M., Singh, K., Giffin, J., Lee, W.: Understanding precision in host based intrusion detection. In: Proceedings of RAID. pp. 21–41 (2007)
53. Shieber, S.M.: Evidence against the context-freeness of natural language. Springer (1987)
54. Shu, X., Yao, D., Ramakrishnan, N.: Unearthing stealthy program attacks buried in extremely long execution paths. In: Proceedings of ACM CCS (2015)
55. Sufatrio, Yap, R.: Improving host-based IDS with argument abstraction to prevent mimicry attacks. In: Proceedings of RAID. pp. 146–164 (2006)
56. Szekeres, L., Payer, M., Wei, T., Song, D.: SoK: Eternal war in memory. In: Proceedings of IEEE S & P. pp. 48–62 (2013)
57. Tandon, G., Chan, P.K.: On the learning of system call attributes for host-based anomaly detection. IJAIT 15(6), 875–892 (2006)
58. Vendicator: StackShield, `http://www.angelfire.com/sk/stackshield/`, accessed August 2015
59. Wagner, D., Dean, D.: Intrusion detection via static analysis. In: Proceedings of IEEE S & P. pp. 156–168 (2001)
60. Wagner, D., Soto, P.: Mimicry attacks on host-based intrusion detection systems. In: Proceedings of ACM CCS. pp. 255–264 (2002)
61. Warrender, C., Forrest, S., Pearlmutter, B.: Detecting intrusions using system calls: alternative data models. In: Proceedings of IEEE S & P. pp. 133–145 (1999)
62. Wee, K., Moon, B.: Automatic generation of finite state automata for detecting intrusions using system call sequences. In: Proceedings of MMM-ACNS (2003)
63. Wespi, A., Dacier, M., Debar, H.: Intrusion detection using variable-length audit trail patterns. In: Proceedings of RAID. pp. 110–129 (2000)
64. Xu, K., Yao, D., Ryder, B.G., Tian, K.: Probabilistic program modeling for high-precision anomaly classification. In: Proceedings of IEEE CSF (2015)