Threat Detection in Program Execution and Data Movement: Theory and Practice

Xiaokui Shu

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science & Application

Danfeng Yao, Chair
Barbara G. Ryder
Naren Ramakrishnan
Patrick R. Schaumont
Trent Jaeger

April 6, 2016
Blacksburg, Virginia

Keywords: Cybersecurity, Program Anomaly Detection, Data Leak Detection

Threat Detection in Program Execution and Data Movement: Theory and Practice

Xiaokui Shu

(ABSTRACT)

Program attacks are one of the oldest and fundamental cyber threats. They compromise the confidentiality of data, the integrity of program logic, and the availability of services. This threat becomes even severer when followed by other malicious activities such as data exfiltration. The integration of primitive attacks constructs comprehensive attack vectors and forms advanced persistent threats.

Along with the rapid development of defense mechanisms, program attacks and data leak threats survive and evolve. Stealthy program attacks can hide in long execution paths to avoid being detected. Sensitive data transformations weaken existing leak detection mechanisms. New adversaries, e.g., semi-honest service provider, emerge and form threats.

This thesis presents theoretical analysis and practical detection mechanisms against stealthy program attacks and data leaks. The thesis presents a unified framework for understanding different branches of program anomaly detection and sheds light on possible future program anomaly detection directions. The thesis investigates modern stealthy program attacks hidden in long program executions and develops a program anomaly detection approach with data mining techniques to reveal the attacks. The thesis advances network-based data leak detection mechanisms by relaxing strong requirements in existing methods. The thesis presents practical solutions to outsource data leak detection procedures to semi-honest third parties and identify noisy or transformed data leaks in network traffic.

This work has been supported by grants ONR N00014-13-1-0016 and ARO YIP W911NF-14-1-0535 as well as Security and Software Engineering Research Center (S^2ERC).

Acknowledgments

I would like to express my deepest gratitude to my advisor, Dr. Danfeng Yao, for her guidance and support through my graduate study. Dr. Yao opens a door for me to create and improve freely in system and network security. Without her encouragement and inspiration, this dissertation would not have been possible.

I would like to thank my committee members, Dr. Barbara G. Ryder, Dr. Naren Ramakrishnan, Dr. Patrick R. Schaumont, and Dr. Trent Jaeger, who contribute time and effort to verify my models, give me writing suggestions, show me scientific discovery procedures, and support my attendance to system security summer schools and workshops.

I would like to thank my friends Hao Zhang, Kui Xu, Karim Elish, Fang Liu, Ke Tian, Tong Zhang, Long Cheng, Hussain Almohri, and Qingrui Liu, who help me dive into system security and discuss related topics to strengthen my understandings of the field.

I would like to thank Dr. Benjamin Jantzen, who discusses autonomous scientific discovery with me from the philosophical perspective and inspires me to develop the unified framework for program anomaly detection in this thesis.

I would like to thank Dr. Changhee Jung for the discussion and comments on practical tracing techniques in my program anomaly detection project.

I would like to acknowledge my collaborator Jing Zhang, who ports my code to CUDA and helps me evaluate the scalability of my alignment-based data leak detection solution.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Cyber Threats Against Programs and Data

Security problems in program executions, caused by program bugs, inappropriate program logics, or/and improper system designs, are first perceived by the Air Force, the Advanced Research Projects Agency (ARPA), and IBM in the early 1970s. *Malicious user threat* was then conceived and described by Anderson in 1972 [7], and it achieved major recognition as a real-world threat in the 1990s with the emergence of practical attacking techniques including buffer overflow [143], return-into-libc [141], denial of service [153], etc. Defenses have been proposed and adopted on hardware (e.g., NX bit), operating system (e.g., address space layout randomization), compiler (e.g., canaries) and software architecture (e.g., sandbox). On one hand, the defenses set barriers to exploiting a program. On the other hand, they motivate attackers to develop stealthy attacks, using subtle control flow manipulation tactics to exploit a program, or utilizing cunning usage patterns to conduct service abuse attacks. As a result, many modern attacks circumvent existing defenses and set up new challenges for attack detection and prevention solutions.

Many of the modern program attacks are deliberately planned and developed for profit. They compromise the confidentiality of sensitive data (e.g., data leak), the integrity of program logic (e.g., authentication bypass), the availability of services (e.g., denial of service), and even the financial interests directly (click fraud). Widely deployed defense mechanisms, such as anti-virus and Network Intrusion Detection Systems (NIDS), are mostly signature-based and not sufficiently effective against many modern attacks, especially general or newly developed attacks. Zero-day attacks, which take advantage of the gap between the discovery of vulnerabilities and the deployment of specific countermeasures, pose severe threats with respect to the aforementioned attack objectives. For instance, the `OpenSSL` Heartbleed bug was publicly disclosed on April 7, 2014 with the patched library released [82]. However, attackers have made use of this vulnerability before every affected server was patched around

the globe. The Community Health Systems in the U.S. was breached and 4.5 million social security numbers and addresses were leaked [64]. Canada Revenue Agency also reported the leak of 900 social insurance numbers [52].

Data exfiltration – one of the most profitable cyber attack objectives – is becoming a significant and severe threat from 2010. Reports show that the number of leaked sensitive data records has grown 10 times in the last 4 years, and it reached a record high of 1.1 billion in 2014 [157]. A significant portion of the data leak incidents are due to human errors, for example, a lost or stolen laptop containing unencrypted sensitive files, or transmitting sensitive data without using end-to-end encryption such as PGP [94]. A recent Kaspersky Lab survey shows that accidental leak by staff is the leading cause for internal data leaks in corporates [109], and I focus on the detection of the accidental leaks.

This thesis presents both theoretical and technical advances in program anomaly detection and data leak detection, the two important defenses against different elements in sophisticated modern attack vectors.

Program anomaly detection models normal program behaviors and detects anomalous ones. It is not restricted by attack signatures of known program attacks. Therefore, seeking anomalous program behaviors is an effective means of detecting new and unknown program attacks. Moreover, anomaly detection can reveal early phases of attacks, e.g., heap fengshui [176], and helps prevent subsequent attack steps from happening. This thesis presents a unified framework (Chapter 3) for understanding existing and future program anomaly detection methods [169] and an advanced data mining approach (Chapter 4) for program anomaly detection on long program execution traces [168]. The former serves as a guideline for developing new program anomaly detection approaches, and the latter validates my systematization for program anomaly model development towards the theoretical accuracy limit.

Data leak detection is a basic countermeasure against data exfiltration or data loss, and it serves as a primary step in building sophisticated leak prevention systems. Detecting and preventing data leaks requires a set of complementary solutions, which may include data-leak detection [15, 92], data confinement [16, 140, 215], stealthy malware detection [103, 106], and policy enforcement [108]. Among these techniques, network-based data leak detection is relatively practical and has been adopted by commercial products [76, 180]. Unfortunately, many common issues in network-based data leak detection have not been systemically studied in the literature. This thesis studies privacy-preserving data leak detection [166, 167] as well as the detection of transformed data leaks [165, 170–172]. The security model of privacy-preserving data leak detection is established and the requirements for transformed data leak detection is discussed. Two detection approaches are presented for privacy-preserving data leak detection (Chapter 5) and the detection of transformed data leaks (Chapter 6).

In the remaining chapter, I brief the problems and my contributions in *program anomaly detection* and *network-based data leak detection*, which include both theoretical and practical advances in the fields. These methodology and techniques enable the detection of multiple elements in complex modern attack vectors with various requirements.

1.2 Program Anomaly Detection: Theory and Practices

Program attacks are one of the oldest and fundamental cyber threats, which constitute one of the technical kernels of latest attack vectors and advanced persistent threats (APT). Standard defenses built upon retrospects of observed and inspected attacks bear time lags between emerging attacks and deployed countermeasures, while program anomaly detection analyzes normal program behaviors instead of the threats. The latter discovers aberrant executions caused by attacks, misconfigurations, program bugs, and unusual usage patterns. The merit of anomaly detection is its independence from attack signatures, which enables proactive defenses against new and unknown threats.

This thesis advances the area of program anomaly detection from both theoretical and pragmatic aspects. In Chapter 3, I present a unified framework to describe the detection capability of *any* program anomaly detection models [168]. My work on program anomaly detection abstraction systematizes significant knowledge discovered in the last decades and provides a unified understanding of detection capability across varieties of program anomaly detection systems, which establishes a field map for program anomaly detection and provides guidance for future model development. Guided by my framework, I present a program anomaly detection system in Chapter 4 that leverages machine learning techniques to discover event co-occurrences during program executions and to detect stealthy attacks that do not cause illegal control flows [169]. I identify and overcome multiple challenges in modeling program behaviors within extreme long execution trace segments including behavior space explosion, diverse normal behaviors, long tail cluster distribution, and flexible tracing granularity.

1.2.1 Program Anomaly Detection in a Unified Framework

Problem: Despite the decades of research and development of program anomaly detection models, the problem of program anomaly detection has not been formalized. Multiple program anomaly detection branches have been established. While each of them is advanced and supported by its theories and practices, it was difficult to compare the effectiveness of models between branches [192]. In this thesis, I focus on systematizing knowledge of this area and studying the following critical questions that have not been answered in the literature.

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

Contributions: I present a unified framework for program anomaly detection and answered all the aforementioned questions in Chapter 3. The framework bridges program anomaly detection and formal language theory [169]. Through my formalization, any program anomaly detection method can be abstracted into a formal language, the detection capability of which can be compared via Chomsky hierarchy. *Detection capability* and *scope of the norm* are identified as two properties of a detection method in my formalization, and the framework unifies deterministic and probabilistic detection approaches through different scopes of the norm. I prove the theoretical accuracy limit of program anomaly detection with an abstract machine $\tilde{M}$. $\tilde{M}$ can distinguish any two execution paths as the program itself, thus it can capture any anomalous program execution if properly trained. I summarize the evolution of program anomaly detection using my framework and envision several directions for future development from accuracy and practicality aspects.

1.2.2 Discovery of Event Correlation in Program Behaviors Against Execution Anomalies

Problem: Modern stealthy exploits can achieve attack goals without introducing illegal control flows, e.g., tampering with non-control data and waiting for the modified data to propagate and alter the control flow legally. Attacks can be constructed using normal program execution trace fragments and buried in extreme long execution paths. Existing program anomaly detection systems focusing on legal control flow attestation and short call sequence verification are inadequate to detect such stealthy attacks. In this thesis, I point out the need to analyze long program execution paths and discover event correlations in large-scale execution windows among millions of instructions for detecting stealthy program attacks such as non-control data attacks, and denial of service (DoS) attacks.

Contributions: I present a security model for efficient program behavior analysis through event correlations in large-scale execution windows in Chapter 4. Guided by my framework in Chapter 3, I invent a two-stage data mining approach for the recognition of diverse normal call-correlation patterns as well as the detection of program attacks at both inter- and intra-cluster levels [168]. This approach enables memorizing and reasoning call events in a large time window. It extends the modeling capability of pushdown automaton (PDA) methods by relaxing the memory of the detection system from the provisional stack to the entire program execution history. I implement a prototype of my approach using `Python`, `Pin`, and `SystemTap`, and demonstrate its effectiveness against three real-world attacks and four categories of synthetic anomalies with less than 0.01% false positive rates and 0.1 ms to 1.3 ms analysis overhead per behavior instance (1k to 50k function or system calls).

The proposed method initializes context-sensitive language level detection approaches, but it does not match the detection capability of the most precise context-sensitive language approach, e.g., $\tilde{M}$, due to the absence of element order reasoning in its abstract language, Bach. Future improvement can be made to include the reasoning of order information in

program traces for more precise detection.

1.3 Network-Based Data Leak Detection: Emerging Paradigms and System Design

Inspecting network traffic and comparing it with tagged sensitive data is a common practice adopted by commercial network-based data leak detection products to detect inadvertent/accidental leaks. The process relies on an effective document deduplication technique, i.e., shingling (producing n-grams) and fingerprinting (yielding hashes). A sensitivity score is calculated based on the similarity between the two sets of fingerprints – Jaccard index or other similarity measures can be used – to indicate whether the sensitive data is leaked.

In this thesis, I relax strong requirements and generalize the standard approach to meet the settings in various deployment scenarios [165–167, 170–172]. In Chapter 5, I remove the requirement that data leak detection should be processed privately and design a scheme to make the detection procedure transparent to the operator, which enables data leak detection outsourcing to semi-honest third parties to leverage their economical computation power and low system maintenance expenses. In Chapter 6, I generalize similarity measures between sensitive data and network traffic fingerprint sets to achieve more accurate detection and meet the high accuracy and performance standards of transformed data leak detection.

1.3.1 Privacy-Preserving Data Leak Detection

Problem: Straightforward detection realizations based on set similarity measures require precise knowledge of the sensitive data, either in plaintext or fingerprints. However, this requirement is undesirable, as it may threaten the confidentiality of the sensitive information. If a detection system is compromised, it may expose the plaintext sensitive data (in memory). In addition, the data owner may need to outsource the detection to semi-honest providers without revealing the precise knowledge about the sensitive data.

Contributions: I present a scheme named *fuzzy fingerprint* to relax the requirement of precise sensitive data used in the detection procedure in Chapter 5. Fuzzy fingerprints form a special set of sensitive data digests that are prepared by the data owner and released to the semi-honest data leak detection provider [166, 167]. The released digests are used to protect sensitive data from being reversed to genuine fingerprints as well as from being identified when actual leaks are detected. I describe how a cloud provider can offer their customers data-leak detection as an add-on service with strong privacy guarantees. The advantage of my method is that it enables the data owner to safely delegate the detection operation to a semi-honest provider without revealing the sensitive data to the provider. I implement a prototype of my privacy-preserving data leak detection protocol using fuzzy fingerprints and

evaluated both accuracy and performance of the design in various data leak scenarios.

The proposed approach balances the computation of leak detection between the data owner and the semi-honest service provider. The amount of computation remained at the data owner fulfills the privacy goal. In other words, my current design does not allow all computation to be outsourced to the service provider, and it remains an open question to perform privacy-preserving data leak detection without data owner in a practical manner.

1.3.2 Detection of Transformed Data Leaks

Problem: Transformation of sensitive data in network traffic poses a severe accuracy issue for network-based data leak detection solutions. Transformations vary from pervasive character substitution to data truncation (partial data leak). Standard countermeasures in the industry rely on human effort to list popular transformations and to write mapping rules for improving detection accuracy. However, this approach is not scalable and cannot be used for partial leak detection where set similarity test does not apply. I point out the need to develop a detection approach that can match generic leak transformations automatically. Regular expression is not feasible, because transformation rules are unknown in the security model.

Contributions: I bring the idea of sequence alignment to data leak detection and create an approach detecting generic transformed data leak in Chapter 6. My entire solution achieves detection objectives including accuracy, performance, and utility by pairing a specialized dynamic programming local alignment algorithm with a uniquely designed sequence sampling algorithm [165,170–172]. This alignment-based design achieves high detection accuracy against transformed data leaks: 100% detection rate and 0.8% false positive rate vs. 63.8% detection rate and 8.9% false positive rate yielded by traditional set-based approaches. I demonstrate the feasibility of performing data leak detection on high-performance coprocessors and the scalability of my design through a GPGPU prototype that achieves 400 Mbps detection throughput on one NVIDIA Tesla C2050 GPU.

The proposed approach utilizes alignment to tolerant transformations and sampling to achieve good performance. The combination is effective for detecting large pieces of sensitive data, e.g., contents of documents and emails. Alternative methods such as string matching should be used when dealing with short and regular data, e.g., credit card information, where transformation is usually not an issue. High-entropy sensitive data transformations such as encryption are beyond the scope of my design, and host-based data tracking systems could be combined to detect such threats.

Chapter 2

Literature Review

Defenses against program attacks and data leaks have been visited and studied in the literature. In this chapter, I review the development of two fields of defenses, namely program anomaly detection (Section 2.1) and data leak detection (Section 2.2). I point out unresolved issues in state-of-the-art solutions and discuss the differences between my approaches and existing detection methods from both academia and industry. Besides existing solutions to program anomaly detection and data leak detection, I discuss related fields, such as malware classification, string matching, and privacy-preserving computation. Security models in the related fields are explained and compared to the problems studied in this thesis.

2.1 Defenses Against Program Attacks

This section first reviews the development of existing program anomaly detection methods, or host-based intrusion detection systems (Section 2.1.1). Then I discuss related fields such as malware classification (Section 2.1.2), event correlation analysis (Section 2.1.3), and defenses developed against specific attacks (Section 2.1.4).

2.1.1 Program Anomaly Detection

Conventional program anomaly detection systems (aka host-based intrusion detection systems) follow Denning's intrusion detection vision [46]. They were designed to detect illegal control flows or anomalous system calls based on two basic paradigms: *n-gram-based dynamic normal program behavior modeling* and *automaton-based normal program behavior analysis*. The former was pioneered by Forrest [61], and the latter was formally introduced by Sekar et al. [160] and Wagner and Dean [192]. Other notable approaches include probabilistic modeling methods pioneered by Lee and Stolfo [122] and dynamically built state machine

first proposed by Kosoresow and Hofmeyr [112]. Later work explored more fine-grained models [13, 72, 74] and combined static and dynamic analysis [66].

Each paradigm leads to a fruitful line of models for detecting anomalous calls/returns in program traces. The n-gram methods are based on an observation that short trace segments are good indicators of normal program executions, which is true when detecting attacks with injected anomalous calls, e.g. `system()` or `exec()`. However, it is impractical to increase n for large-scale program trace analysis. The basic n-gram model was further studied in [60,93] and sophisticated forms were developed, e.g., machine learning model [95, 122]; first-order Markov model [217], [212]; hidden Markov model [68, 196]; and neural network [71]. Call arguments were used to precisely define states in [132]. n-gram/lookahead pair frequencies were studied in [91] and [90]. Beyond program anomaly detection, n-grams are also used in malware detection [27]. Multithreading handling is discussed in [47]. And pH in [175] mitigates attacks by delaying anomalous calls.

The other paradigm, i.e., automaton models, aims to model a program not limited to short system call sequences. It builds an automaton or a pushdown automaton to read the entire trace at a time. However, reading the entire trace is not equivalent to correlating events in the transition history. All automaton models in literature are first-order and they only verify each state transition on its own. Program counter and call stack information were used by [55, 56, 160] to help precisely define each state (a system call) in an automaton. Pushdown automaton or its equivalents were employed in many advanced models [55, 66, 74, 96, 131]. Hidden procedure transition information is revealed by inserting flags in particular procedures [74]. Call arguments are also added in an FSA model [72]. [65] improves FSA methods by providing an event frequency analysis extension, and it successfully detects DoS attacks.

Some FSA model develops into pure static analysis and enforcement solutions. One milestone is control flow integrity (CFI) [1], which statically instruments a program and embeds the checking for dynamic functions and jumps. It defeats advanced exploits such as return-oriented programming (ROP) [161] and ROP without returns [34].

In Chapter 3, I show that all anomaly detection paradigms and branches can be unified, as well as deterministic/probabilistic detection methods. I give a formal definition of program anomaly detection, and provide a unified framework to explain any program anomaly detection method. Guided by my framework, I envision a new program anomaly detection branch and propose a detection method in Chapter 4. The method validates my systematization in Chapter 3 and detects stealthy program attacks hidden in long program traces not captured by existing methods.

2.1.2 Malware Classification

Clustering and classification techniques are widely used in malware classification, e.g., [12,63, 149,150]. However, three unique perspectives distinguish the problem I study from malware detection: *i)* normal program behaviors are modeled in my anomaly detection problem instead of malicious ones; *ii)* inter-cluster detection seeking montage anomalies is critical and unique to my anomaly detection problem; and *iii)* program specific information, e.g., procedure symbols and addresses, are useful to precisely define normal program behaviors in my anomaly detection problem. But it is not general across multiple malware samples. Instead, system call dependencies [12] and traffic event features [150] were used in malware classification to define abstract malware behaviors. Advanced machine learning techniques, e.g., graph mining [63] and episodes mining [107], were used to extract significant malware behaviors as signatures. They are potentially useful for anomaly detection. However, it is unclear how they can be tailored to achieve my goal in Chapter 4: capturing infrequent-yet-important normal call patterns.

2.1.3 Event Correlation Analysis and Reasoning

Correlation analysis has been used in network intrusion detection system (NIDS) and botnet discovery. Comprehensive classification and clustering techniques were developed by Perdisci et al. to analyze related bytes in traffic payload [147, 148]. The problem is different from mine in that there is no control flow semantics in most payloads, so NIDS uses n-gram or lookahead pair underneath. The co-occurrence of synchronized and repetitive bot activities was studied by Gu et al. [79]. I analyze the co-occurrence of program events to correlate traversed control flow segments in program executions.

Causality reasoning on network events advances the relationship discovery procedure and provides detailed insights into event relations. Hao et al. leveraged machine learning methods to detect malware activities through network traffic analysis [219, 220] and designed visualization tools to aid security analysts identifying the threats [218]. This approach analyzes a program as a black box from the network perspective. Therefore, the problem is different from mine.

An interesting work by Gao [67] measures the distance between trace segments using alignment. An alignment compares two entire traces directly, thus event co-occurrence is taken into account. However, the security models are different between mine and Gao's. [67] is built for a replica system or N-variant systems [42]. The replica provides one and the only one reference of normal program behavior, which is different than detecting anomalies from all normal behavior instances.

2.1.4 Defenses Against Specific Categories of Attacks

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) [182]. These defense are built upon the understanding of specific categories of attacks and abstraction of the attacks. They do not prevent attacks in new categories or program attacks that have not been studied. And they they lead to an intertwined developments between intrusion and defense mechanisms.

For example, given the aforementioned defenses, program attacks were developed leveraging unattended/uninspected execution elements, such as return-oriented programming [161], jump-oriented programming [14, 34], and non-control data attacks [36]. These new attacks were dissected and studied for developing defenses against them, e.g., Control-flow integrity (CFI) [1], gadget defender [44, 77], data integrity [5, 216], and data-flow integrity [31]. However, new generation of non-control data attack, e.g., Control-Flow Bending [29], Data-Oriented Exploit [86], are developed to circumvent the CFI mechanisms.

Other solutions against specific attacks include: integer overflow [24, 137, 195], DoS attacks [156], and heap spraying was [49]. Swaddler [40] enforces logic sequences of high-level tasks in web applications, e.g., tax calculation follows total calculation, through training state-transition diagrams and statistical analysis representing properties of allowed workflow. This approach is inappropriate to be applied to program trace analysis, because of the high complexity caused by large amounts of program events.

2.2 Data Leak Detection

In this section, I first present commercial data leak detection tools (Section 2.2.1) and sensitive data flow enforcement in the literature (Section 2.2.2). String matching (Section 2.2.3), alignment (Section 2.2.4) and security application parallelization (Section 2.2.5) are then reviewed as three techniques tightly related to my transformed data leak detection solution. Next, privacy-preserving needs and techniques are discussed with respect to my privacy-preserving data leak detection solution (Section 2.2.6 and Section 2.2.7). Anomaly detection approaches for identifying data leaks are discussed (Section 2.2.8), and problems similar to leak detection such as remote execution verification are discussed at last (Section 2.2.9). Data leak issues due to insufficient anonymization [89], query sanitization [37], or side channels [19] require different techniques and are not further discussed.

2.2.1 Data Leak Detection with the Bag-of-words Model

Network-based data-leak detection (DLD) typically performs deep packet inspection (DPI) and searches for any occurrences of sensitive data patterns. DPI is a technique to analyze pay-

loads of IP/TCP packets for inspecting application layer data, e.g., HTTP header/content. Alerts are triggered when the amount of sensitive data found in traffic passes a threshold. The detection system can be deployed on a router or integrated into existing network intrusion detection systems (NIDS).

Existing commercial tools and services for data leak prevention include Symantec DLP [180], IdentityFinder [92], GlobalVelocity [75] and GoCloudDLP [76]. GlobalVelocity uses FPGA to accelerate the system. Their algorithms are likely based on set intersection, or the bag-of-words model. Set intersection operation is performed on two sets of n-grams, one from the content and one from sensitive data. The set intersection gives the amount of sensitive n-grams appearing in the content. The method has been used to detect similar documents on the web [22, 155], shared malicious traffic patterns [26], malware [100], as well as email spam [124]. The advantage of n-grams is the extraction of local features of a string, enabling the comparison to tolerate discrepancies. Some advanced versions of the set intersection method utilize Bloom filter [20], e.g., [180], which trades accuracy for space complexity and speed. Bloom filter configured with a small number of hash functions has collisions, which introduce additional unwanted false positives. IdentityFinder searches file systems for short patterns of numbers that may be sensitive (e.g., 16-digit numbers that might be credit card numbers). It does not provide any in-depth leak detection.

2.2.2 Enforcing Sensitive Data Flow to Prevent Leaks

Another category of approaches for data-leak detection is monitoring and enforcing the sensitive data flows. The approaches include data flow analysis [203], and taint analysis [215], legal flow marking [43], and file-descriptor sharing enforcement [140]. These approaches are different from mine because they do not aim to provide a remote service. However, a pure network-based solution cannot handle encrypted traffic [187], and these methods are complementary to my approach in detecting different forms (e.g., encrypted) of data leaks.

Several tools are developed for securing sensitive information on mobile platforms [85, 140, 209]. The work in [140] describes an approach to control the sharing of sensitive files among mobile applications such as Android apps. File descriptors (not the content) are stored, tracked and managed. The access control on files is realized through policies. In [209], the authors aim to detect the transmission of sensitive data that are not intended by smartphone users. The work analyzes mobile applications using symbolic execution. [85] describes a visualization method for informing mobile users of information exposure. The potential information exposure may be caused by improper setting and configuration of access policies. The visualization is through a novel avatar apparel approach. The security goals and requirements in all these studies are different from mine, leading to different techniques.

2.2.3 String Matching and Data Leak Detection

Network intrusion detection systems (NIDS) such as Snort [158] and Bro [146] use regular expressions to perform string matching [4, 17] in deep packet inspection [45, 127]. Nondeterministic finite automaton (NFA) with backtracking requires $O(2^n)$ time and $O(n)$ space, where n is the number of automaton states. Deterministic finite automaton (DFA) has a time complexity of $O(n)$ and a space complexity of $O(2^n)$ when used with quantification. Quantification is for expressing optional characters and multiple occurrences in a pattern. DFA's space complexity can be reduced by grouping similar patterns into one automaton [185]. Further studies reduce the number of edges in D^{2}FA [117] and CD^{2}FA [118]. These improvements provide a coefficient level of speedup.

However, neither DFA or NFA is designed to support arbitrary and unpredictable pattern variations. In comparison, my solution based on sequence alignment covers all possible pattern variations in long sensitive data without needing to explicitly specify them. Another drawback of automata is that it provides binary results. In comparison, alignment provides precise matching scores and allows customized weight functions. The proposed alignment gives more accurate detection than approximate string matching (e.g., [8, 9]).

String matching techniques lead to straightforward data leak detection solutions such as keyword search. For instance, iLeak is a system for preventing inadvertent information leaks on a personal computer [110]. iLeak monitors the file access activities of processes and searches for system call inputs that involve sensitive data. The sensitive data is matched using the keyword search functionality provided by the operating system. This method is easy to implement on a single machine, but it is not applicable to network-wide data leak detection. It does not capture transformed data leaks as well due to the limitation of automaton-based string matching methods.

2.2.4 Alignment Algorithms Developed for Security Applications

Alignment algorithms have been widely used in computational biology applications. In security literature, NetDialign [104] based on the well-known Dialign algorithms [138] is proposed for network privacy. It performs differential testing among multiple traffic flows. Kreibich and Crowcroft presented an alignment algorithm for traffic intrusion detection systems such as Bro [113]. It is a variant of Jacobson-Vo alignment that calculates the longest common subsequence with the minimum number of gaps. Data leak detection differs from the above network privacy and IDS problems, and it has new requirements. The alignment performs complex inferences needed for aligning sampled sequences, and my solution is also different from fast non-sample alignments in bioinformatics, e.g., BLAST [6].

2.2.5 Parallelization of Security Applications

Parallelization of conventional IDS has been explored on multi-core CPU, FPGA [186], and GPU [99]. The work in [201] describes a multi-core CPU Snort optimization. IDS utilizes a specialized pattern matching processor is reported in [38]. The work in [11] and [173] points out the technical advantages of GPU (such as flexibility) over FPGA in network intrusion detection. Several pattern matching algorithms have been implemented with GPU, e.g., Gnort [188] and a GPU version of Wu-Manber pattern matching algorithm [87]. The work in [189] presents a GPU-based architecture for intrusion detection that can achieve high-speed I/O performance. I also utilize GPU to parallelize my prototype. With parallel design inherent in my algorithms, the prototype shows good scalability.

2.2.6 Privacy-preserving Data Leak Detection

There have been several advances in understanding the privacy needs [111] or the privacy requirement of security applications [206]. In this thesis, I identify the privacy needs in an outsourced data-leak detection service and tackle the unique data-leak detection problem in an outsourced setting where the DLD provider is not fully trusted. Such privacy requirement does not exist in traditional security applications, e.g., the virus signatures are non-sensitive in the virus-scan paradigm [81].

Bloom filter is a space-saving data structure for set membership test [20], and it is a basic building block of many network security solutions, e.g., [69], [194]. The fuzzy Bloom filter introduced in [135] constructs a special Bloom filter that probabilistically sets the corresponding filter bits to `1`'s, which, on its own, does not prevent a DLD provider from learning leaked data in the traffic.

Most of commercial data leak detection tools do not provide the privacy-preserving feature, thus they cannot be outsourced. One exception is GoCloudDLP [76], which allows the customers to outsource the detection to a fully honest DLD provider. My fuzzy fingerprint method differs from these solutions and enables its adopter to provide data-leak detection as a service without the fully trust in the service provider.

2.2.7 General Privacy-preserving Frameworks

Privacy-preserving keyword search [178] or fuzzy keyword search [123] provide string matching approaches in semi-honest environments, but keywords usually do not cover enough sensitive data segments for data-leak detection.

Besides my fuzzy fingerprint solution for data-leak detection, there are other privacy-preserving techniques invented for specific processes, e.g., DNA matching [184], or for general purpose use, e.g., secure multi-party computation (SMC). Similar to string matching methods dis-

cussed above, [184] uses anonymous automata to perform the comparison. SMC [210] is a cryptographic mechanism, which supports a wide range of fundamental arithmetic, set, and string operations as well as complex functions such as knapsack computation [211], automated trouble-shooting [88], network event statistics [25, 199], private information retrieval [214], genomic computation [102], private database query [213], private join operations [28], and distributed data mining [97]. The provable privacy guarantees offered by SMC comes at a cost regarding computational complexity and realization difficulty. The advantage of my fingerprint filter approach is its concision and efficiency.

2.2.8 Discovering Traces of Data Leaks via Anomalous Traffic Detection

Anomaly detection in network traffic can be used to detect data leaks. Borders and Prakash proposed a solution to detect any substantial amount of new information in the traffic [15], and Fawcett analyzed entropy to identify anomalous leak patterns [53]. I present a signature-based model to detect data leaks and focus on the design that can be outsourced. Thus, the two approaches are different.

2.2.9 Remote Execution Verification

Another work similar to my outsourced data-leak detection is the verification of outsourced execution. Du et al. proposed a solution testing whether deliberately sent chaff (surrounded by normal input) is processed or not after harvest [48]. The goal is to verify whether the service provider performs the operation, and it is different from the privacy goal in Chapter 5.

Chapter 3

Program Anomaly Detection in a Unified Framework

3.1 Introduction

In this chapter, I formalize one area of intrusion detection, namely *program anomaly detection* or *host-based intrusion detection* [163]. 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 [59].

Evaluating the detection capability of program anomaly detection methods is always challenging [192]. 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 [192]. 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 [60]. Feng et al. formalized automaton based methods in [55]. Chandola et al. described program anomaly detection as a sequence analysis problem [32]. Chandola et al. provided a survey of machine learning approaches in [33]. The connection between an n-gram method and its automaton representation is first stated by Wagner [193]. Sharif et al. proved that any system call sequence based method can be simulated by a control-flow based method [163].

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?

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

The contributions of this chapter are summarized as follows.

1. I formalize the general security model for program anomaly detection. I prove that the detection capability of a method is determined by the expressiveness of its corresponding language (Section 3.2).
2. I point out two independent properties of program anomaly detection: *precision* and *the scope of the norm*. I explain the relation between precision and deterministic/probabilistic detection methods (Section 3.2).
3. I present the theoretical accuracy limit of program anomaly detection with an abstract machine $\tilde{M}$. I prove that $\tilde{M}$ can characterize traces as precise as the executing program (Section 3.3).
4. I develop a hierarchal framework unifying any program anomaly detection method according to its detection capability. I mark the positions of existing methods in my framework and point out the gap between the state-of-the-art methods and the theoretical accuracy limit (Section 3.5).
5. I explain the evolution of program anomaly detection solutions. I envision future program anomaly detection systems with features such as full path sensitivity and higher-order relation description (Section 3.6).
6. I compare program anomaly detection with control-flow enforcement. I point my their similarities in techniques/results and explain their different perspectives approaching program/process security (Section 3.7).

3.2 Formal Definitions for Program Anomaly Detection

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

3.2.1 Security Model

Considering both transactional (terminating after a transaction/computation) and continuous (constantly running) program executions, I 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 3.2.1. *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.

I formalize the problem of program anomaly detection in Definition 3.2.2.

Definition 3.2.2. *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 3.2.2, 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 3.2.3. *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.*

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

I 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 [66], 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.

I describe the general security model of a real-world program anomaly detection system in Definition 3.2.4. The security model derives from Definition 3.2.2 but measures program executions using $\ddot{\mathbf{T}}$ instead of $\mathbf{T}$.

Definition 3.2.4. *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 $\{\ddot{\mathbf{T}} \mid \ddot{\mathbf{T}} \text{ is normal}\}$. 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.

3.2.2 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. I define *precision* of Λ as the ability of Λ to distinguish different precise program traces in Definition 3.2.5. This concept is independent of whether the scope of the norm is deterministically or probabilistically established (discussed in Section 3.2.3).

Definition 3.2.5. *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 $\ddot{\mathbf{T}}$.*

My 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) [192] and *average reachability measure* (using context-free grammar to approximate the description of precise program traces) [72]. The generation is achieved through the using of $\mathbf{T}$, the most precise description of a program execution. `average` in Definition 3.2.5 can be replaced by other aggregation function for customized evaluation.

I formalize the relation between the expressive power of L_Λ (defined by detection method Λ) and the detection capability of Λ in Theorem 3.2.1.

Theorem 3.2.1. *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 3.2.1 enables the comparison between detection capabilities of different detection systems through their corresponding languages. It lays the foundation of my 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.

3.2.3 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 3.2.6. *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 [55] (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 [56] (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.

3.2.4 Overview of My Unified Framework

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

I illustrate my unified framework in Fig. 3.1 showing its hierarchical structure[3]. In Fig. 3.1, 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]. My 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)

[2]Context-sensitive languages correspond to pushdown automata.

[3]The hierarchy is reasoned via Chomsky hierarchy [39], 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.

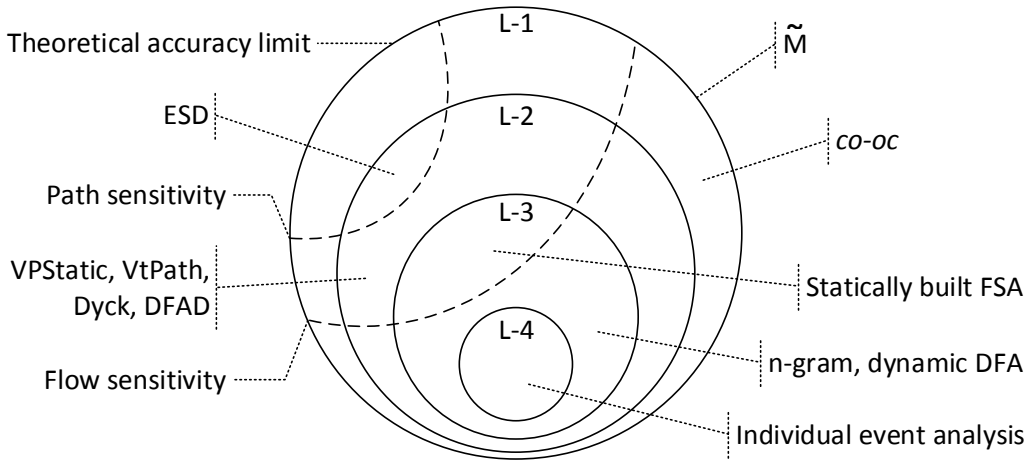


Figure 3.1: The hierarchy of my program anomaly detection framework. L-1 to L-4 are four major precision levels with decreasing detection capabilities.

L-2: context-free language level

L-3: regular language level

L-4: restricted regular language level (least powerful level)

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 3.5.2). I prove the theoretical accuracy limit (the outmost circle in Fig. 3.1) in Section 3.3 with an abstract detection machine $\tilde{M}$. I abstract existing methods in Section 3.4 and identify their positions in my unified framework in Section 3.5. I present details of my framework and point out the connection between levels in my framework and grammars in Chomsky hierarchy in Section 3.5. I describe the evolution from L-4 methods to L-1 methods in Section 3.6.2.

3.3 Accuracy Limit of Program Anomaly Detection

I 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 $\mathbf{T}$. I 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.

Table 3.1: 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.

3.3.1 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 3.1. $\tilde{M}$ 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. I keep Π in $\tilde{M}$ to mimic the execution of a real-world program. It helps extend $\tilde{M}$ for multi-threading (Section 3.3.3) and unify $\tilde{M}$ in my framework (Section 3.5.1).

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 $\{\omega_i\}$.

3.3.2 The Equivalent Abstract Machine of An Executing Program

I state the precision of the abstract detection machine $\tilde{M}$ in Theorem 3.3.1 and interpreter both *sufficiency* and *necessity* aspects of the theorem.

Theorem 3.3.1. *$\tilde{M}$ 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 [84]. I prove Theorem 3.3.1 by Lemma 3.3.1 and Lemma 3.3.2.

Lemma 3.3.1. *A program that is executing on a real-world machine is equivalent to a linear bounded automaton (LBA).*

Lemma 3.3.2. *$\tilde{M}$ is equivalent to a linear bounded automaton.*

Proof. I 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. $size(\Upsilon) = size(\Pi_0) + size(\Pi_1)$.

I 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 [144]. 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 3.3.2 holds.

□

3.3.3 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 [60], first-order automaton transition verification [55]).

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.

3.4 Abstractions of Existing Detection Methods

In this section, I analyze existing program anomaly detection models and abstract them in five categories. I identify their precision (or detection capability) in my framework in Section 3.5.

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 [112] 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 [160]. 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 (3.1). 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.

$$p_i \xrightarrow{s*} p_{i+1} \tag{3.1}$$

***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* [61]. A basic n-gram method tests whether every n-gram is in the known set of normal n-grams [60].

Abstraction: a set of n-gram (of normal program behaviors) is equivalent to an FSA where each state is an n-gram [193]. A transition of such an FSA can be described in (3.2). 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.

$$(s_0, s_1, \ldots, s_{n-1}) \xrightarrow{s_n} (s_1, \ldots, s_{n-1}, s_n) \tag{3.2}$$

Since n-gram methods are built on a membership test, various deterministic [134, 197] and probabilistic [51, 196] 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 [27, 177, 183].

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 [56].

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 [78, 192]. A PDA, in contrast, keeps CFGs isolated and utilizes a stack to record and verify function calls or returns [55, 56, 73]. Thus, it can describe recursions. However, only exposing the stack when

[5] n can be either a fixed value or a variable [134, 198].

[6] Lookahead pair methods are subsequent variants of n-gram methods [93].

system calls occur is not enough to construct a deterministic DPA [55]. 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 [74] by embedding loggers for function calls and returns via binary rewriting.

Abstraction: a typical PDA method consumes white-box level traces [55] or gray-box level traces [131]. 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. I describe the general PDA transition in (3.3) where Γ_i/Γ_{i+1} is the stack before/after the transition, respectively.

$$p_i, \Gamma_i \xrightarrow{f \text{ or } s} p_{i+1}, \Gamma_{i+1} \tag{3.3}$$

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 [13]. Giffin et al. extended system call arguments to environment values, e.g., configurations, and built an environment-sensitive method [72].

Probabilistic methods differ from deterministic program anomaly detection approaches that they use stochastic languages to define the scope of the norm (Section 3.2.3). 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) [196, 205], classification methods [50, 114, 125, 139], artificial neural network [70], data mining approaches [122], etc. Gu et al. presented a supervised statistical learning model, which uses control-flow graphs to help the training of its probabilistic model [80].

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 3.2.3. 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.

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

Table 3.2: Precision levels in my 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.

In N-variant methods, a program is executed with n replicas [42]. 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 [67] and probabilistic hidden Markov model [68] 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.

3.5 Unification Framework

I develop a hierarchical framework to uniformly present any program anomaly detection method in terms of its detection capability. I identify the detection capabilities of existing program anomaly detection methods (Section 3.4) and the theoretical accuracy limit (Section 3.3) in my framework.

3.5.1 Major Precision Levels of Program Anomaly Detection

I 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. I summarize four major precision levels defined in my unified framework in Table 3.2. I 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 $\ddot{\mathbf{T}}$).

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 [50].

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 [112], n-grams methods [61], statically built FSA [160], and FSA with call arguments [27].

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 [56]. However, the stack

change between system calls is nondeterministic. A more expensive approach exposes every procedure transition via code instrumentation [74], so that the stack is deterministic.

Canonical L-2 methods include VPStatic [55], VtPath [56], and Dyck [74]. Moreover, Bhatkar et al. applied argument analysis with data-flow analysis (referred to by us as DFAD) [13], and Giffin et al. correlated arguments and environmental variables with system calls (referred to by us as ESD) [72].

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.

I develop Theorem 3.5.1 showing that higher-order PDA and $\tilde{M}$ (Section 3.3) are equivalent in their computation power. The proof of Theorem 3.3.1 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.

I formally describe an L-1 method, i.e., $\tilde{M}$, in Section 3.3. Any other LBA or $\tilde{M}$ equivalents are also L-1 methods.

Theorem 3.5.1. *L-1 methods are as precise as the target executing program.*

I provide a proof sketch for Theorem 3.5.1. First, $\tilde{M}$ is as precise as the executing program (Theorem 3.3.1 in Section 3.3). Next, I 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 [18], i.e., correlations between non-adjacent events. Therefore, it accepts context-sensitive languages [164], 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. 3.1) [168]. 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 [152].

Probabilistic detection methods and my 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 3.2.3). For instance, a probabilistic FSA method (e.g., HMM [196,205], classification based on n-grams [50,139]) is an L-3 method. It cannot model recursion well because there is no stack in the model. The precision of a probabilistic FSA

Table 3.3: 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.

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 [67, 68] 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.

3.5.2 Sensitivity in a Nutshell

I describe optional properties (sensitivities) within L-1 to L-3 in my hierarchical framework with respect to sensitivity terms introduced from program analysis. I summarize the terminology of sensitivity in Table 3.3 and explain them and their relation to my 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., [160]. 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

[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.

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 [72].

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 [72]. Environment sensitivity is a combination of techniques including data-flow analysis, path-sensitive analysis, etc.

3.6 Attack/Detection Evolution and Open Problems

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

3.6.1 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 [193]. 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 [55, 56, 74], invalidate this type of mimicry attacks with context-free grammar description of program traces. However, mimicry attacks using context-free grammars, e.g., [58, 115], 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 3.3 requires exponential time to build.

3.6.2 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. I describe the upgrade of detection systems from a lower

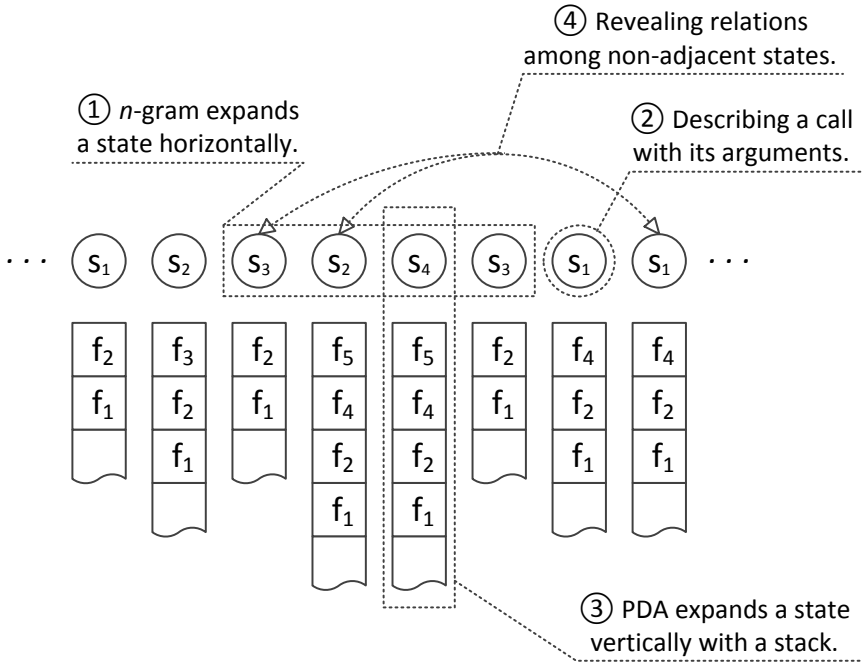


Figure 3.2: Four approaches for improving a basic L-3 method (FSA).

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. I summarize four features to upgrade an L-3 method (abstracted as a general FSA) to L-2 and L-1 methods in Fig. 3.2.

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 upgrades an L-3 method to an L-2 method. ④ discovers cross-serial dependencies and establishes a context-sensitive language [164], 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 [61]. L-3 with ② was studied in [132]. 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 [177] and Gaurav et al. [183]. L-3 with ② and ③ was studied by Bhatkar

et al. [13] and Giffin et al. [72]. $\tilde{M}$ (described in Section 3.3) provides ③ and ④ as basic features. ② can be added to $\tilde{M}$ to describe each state in more details.

3.6.3 Open Problems

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

Precision As illustrated in my framework (Fig. 3.1), 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 [72]) and constrained L-1 level (e.g., co-oc [168]).

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 [168]), 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 [83]. 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 [10]. However, Szekeres et al. found that the industry usually tolerates at most 5% overhead for a security solution [182].

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.

3.7 Control-Flow Enforcement Techniques

Control-flow enforcements, e.g., Control-Flow Integrity (CFI) [1] and Code-Pointer Integrity (CPI) [119], enforce control-flow transfers and prevent illegal function calls/pointers from executing. They evolve from the perspective of attack countermeasures [182]. They are equivalent to one category of program anomaly detection that defines the scope of the norm as legal control flows [163].

3.7.1 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 [126]. I list milestones in the development of control-flow enforcement techniques below (with an increasing protection capability).

1. Return address protection: Stack Guard [41], Stack Shield [190].
2. Indirect control-flow transfer protection: CFI [1], Modular CFI [142].
3. All code pointer protection: CPI [119].

3.7.2 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 3.2.3); 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. [160] and PDA method proposed by Feng et al. [56].

3.7.3 Comparison of the Two Methods

I 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

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

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.

Chapter 4

Program Event Correlation Discovery and Anomaly Detection

4.1 Introduction

In this chapter, I present a program anomaly detection approach for detecting modern exploits that are developed with subtle control flow manipulation tactics to escape existing detection mechanisms. One example is the `sshd` flag variable overwritten attack (an example of non-control data attacks [36]). An attacker overwrites a flag variable, which indicates the authentication result, before the authentication procedure. As a result, the attacker bypasses critical security control and logs in after a failed authentication.

Besides the aforementioned `sshd` attack, stealthy attacks can also be constructed based on existing exploits. Wagner and Soto first diluted a compact exploit (several system calls) into a long sequence (hundreds of system calls) [193]. Kruegel et al. further advanced this approach by building an attack into an extremely long execution path [115]. In their proposed exploit, the attacker accomplishes one element of an attack vector, relinquishes the control of the target program, and waits for another opportunity (exploited vulnerability) to construct the next attack element. Therefore, the elements of the attack vector are buried in an extreme long execution path (millions of instructions). I refer stealthy attacks whose construction and/or consequence are buried into long execution paths and cannot be revealed by any small fragment of the entire path as *aberrant path attacks*.

Existing anomaly detection solutions, e.g., [55, 56, 60, 74] are effective as long as an attack can be discovered in a small detection window on attack traces, e.g., an invalid n-gram or an illegal control flow transition (the latter can be accompanied by data-flow analysis). The aforementioned diluting attack [193] may be detected if it involves illegal control flows. However, there does not exist effective solutions for detecting general aberrant path attacks, because these attacks cannot be revealed in a small detection window on traces.

Mining correlations among arbitrary events in a large-scale execution window is the key to the detection of aberrant path attacks that are buried in long execution paths. The scale of the window may vary from thousands to millions of instructions. However, straightforward generalization of existing approaches is inadequate for large-scale execution window analysis because of two challenges described below.

Training scalability challenge: existing automaton-based methods are first-order and only verify state transition individually. One needs a linear bounded automaton or a Turing machine to enforce the relation among arbitrary events. The generalization results in exponential time complexity for training. n-gram based methods (e.g., lookahead pair, practical hidden Markov model) have a similar exponential convergence complexities in terms of n; large n (e.g., 40) usually leads to false positives due to insufficient training.

Behavior diversity challenge: real-world programs usually realize various functionalities, which result in diverse program behaviors within large-scale execution windows. The distance between a normal program behavior and an anomalous one can be less than the distance between two normal ones. The diversity of normal behaviors makes traditional single-threshold probabilistic methods (e.g., hidden Markov model, one-class SVM) difficult to fine-tune for achieving both a low false positive rate and a high detection rate.

To defend against aberrant path attacks, I propose a detection approach that analyzes program behaviors in large-scale execution windows. My approach maps program behavior instances extracted from large-scale execution windows into data points in a high-dimensional detection space. It then leverages specifically designed machine learning techniques to *i)* recognize diverse program behaviors, *ii)* discover event correlations, and *iii)* detect anomalous program behaviors in various subspaces of the detection space.

In addition to the *binary representation of event relations* in an execution window, my approach further models *quantitative frequency relations among occurred events*. Some aberrant path attacks deliberately or unintentionally result in anomalous event frequency relations, e.g., Denial of Service (DoS), directory harvest attack. The advantage of modeling frequency relations over individual event frequencies (used in existing anomaly detection [65]) is the low false positive rates in case of program/service workload variation.

The contributions of my work are summarized as follows.

- I study the characteristics of aberrant path attacks and identify the need to analyze program behaviors in large-scale execution windows. I present a security model for efficient program behavior analysis through event correlations in large-scale execution windows. The security model covers the detection of two types of anomalous program behaviors abstracted from four known categories of aberrant path attacks. The first type contains events (and their corresponding control-flow segments) that are incompatible in a single large-scale execution window, e.g., non-control data attacks. The second type contains aberrant relations among event occurrence frequencies, e.g., service abuse attacks.

- I design a two-stage detection approach to discover anomalous event correlations in large-scale execution windows and detect aberrant path attacks. My approach contains a *constrained* agglomerative clustering algorithm for addressing the behavior diversity challenge and dividing the detection problem into subproblems. My approach addresses the scalability challenge by utilizing fixed-size profiling matrices and by estimating normal behavior patterns from an incomplete training set through probabilistic methods in each cluster. The unique two-stage design of my approach enables effective detections of *i)* legal-but-incompatible control-flow segments and *ii)* aberrant event occurrence frequency relations at inter- and intra-cluster levels.
- I implement a prototype of my approach on Linux and evaluate its detection capability, accuracy, and performance with `sshd`, `libpcre` and `sendmail`. The evaluation contains over 22,000 normal profiles and over 800 attack traces. My approach successfully detects all attack attempts with less than 0.01% false positive rates. I demonstrate the high detection accuracy of my clustering design through the detection of four types of synthetic anomalies. My prototype takes 0.3 ms to 1.3 ms to analyze a single program behavior instance, which contains 1k to 50k function/system call events.

4.2 Security Model

I describe the attack model, explain my security goals, and discuss three basic solutions toward the goals in this section.

4.2.1 Aberrant Path Attack

I aim to detect aberrant path attacks, which contain infeasible/inconsistent/aberrant execution paths but obey legitimate control-flow graphs. Aberrant path attacks can evade existing detection mechanisms because of the following properties of the attacks:

- not conflicting with any control-flow graph
- not incurring anomalous call arguments
- not introducing unknown short call sequences

Aberrant path attacks are realistic threats and gain popularity since early-age attacks have been efficiently detected and blocked. Concrete aberrant path attack examples are:

1. *Non-control data attacks* hijack programs without manipulating their control data (data loaded into program counter in an execution, e.g., return addresses). One such attack, first described by Chen et al. [36], takes advantage of an integer overflow vulnerability found in several implementations of the `SSH1` protocol [120]. Illustrated in

```
 1: void do_authentication(char *user, ...) {
 2:   int authenticated = 0;
      ...
 3:   while (!authenticated) {
 4:     type = packet_read();
 5:     switch (type) {
          ...
 6:       case SSH_CMSG_AUTH_PASSWORD:
            ...
 7:         if (auth_password(user, password)) {
 8:           memset(password, 0, strlen(password));
 9:           xfree(password);
10:           log_msg("...", user);
11:           authenticated = 1;
12:           break;
            }
13:         memset(password, 0, strlen(password));
14:         debug("...", user);
15:         xfree(password);
16:         break;
          ...
        }
17:     if (authenticated) break;
        ...
```

Figure 4.1: `sshd` flag variable overwritten attack [36].

Fig. 4.1, an attacker can overwrite the flag integer `authenticated` when the vulnerable procedure `packet_read()` is called. If `authenticated` is overwritten to a nonzero value, line 17 is always `True` and `auth_password()` on line 7 is no longer effective.

2. *Workflow violation attacks* can be used to bypass access control [40], leak critical information, disable a service (e.g., trigger a deadlock), etc. One example is *presentation layer access control bypass* in web applications. If the authentication is only enforced by the presentation layer, an attacker can directly access the business logic layer (below presentation layer) and read/write data.

3. *Exploit preparation* is a common step preceding the launch of an exploit payload. It usually utilizes *legal control flows* to load essential libraries, arranges memory space (e.g., heap feng shui [176]), seeks addresses of useful code and data fragments (e.g., ASLR probing [162]), and/or triggers particular race conditions.

4. *Service abuse attacks* do not take control of a program. Instead, the attacks utilize *legal control flows* to compromise the availability (e.g., Denial of Service attack), con-

fidentiality (e.g., Heartbleed data leak [82]), and financial interest (e.g., click fraud) of target services.

4.2.2 Anomalous Program Behaviors within Large-scale Execution Windows

Aberrant path attacks cannot be detected by analyzing events in small windows on program traces. I define semantically meaningful execution windows and unearth aberrant path attacks in large-scale execution windows.

Definition 4.2.1. *An execution window W is the entire or an autonomous portion of a transactional or continuous program execution.*

Execution windows can be partitioned based on boundaries of program functionalities, e.g., login, session handling, etc. Since aberrant path attacks can lead to delayed attack consequences, e.g., non-control data attacks, the analysis should be performed on large-scale execution windows. One such window could contain tens of thousands of system calls and hundreds of times more function calls.

I give some examples of practical large-scale execution window partitioning for security analysis purposes:

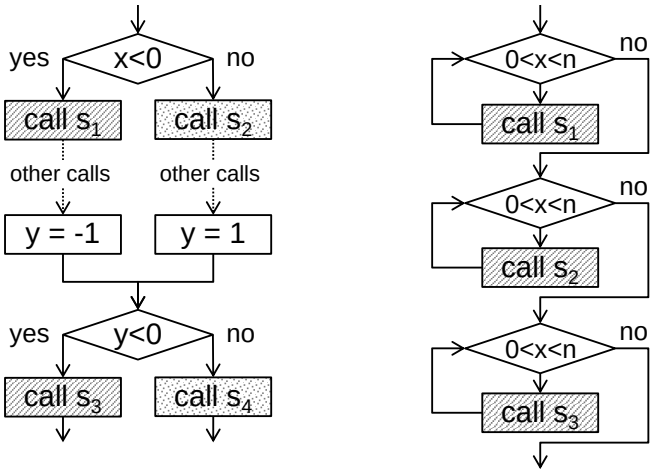
1. partitioning by routines/procedures/functions,
2. partitioning by threads or forked processes,
3. partitioning by activity intervals, e.g., `sleep()`,
4. an entire execution of a small program.

In large-scale execution windows, I abstract two common anomalous behavior patterns of aberrant path attacks.

1. *Montage anomaly* is an anomalous program behavior composed of multiple legitimate control flow fragments that are incompatible in a single execution.

   One example of a montage anomaly is the `sshd` flag variable overwritten attack presented in Fig. 4.1. The attack incurs an execution path that contains two incompatible execution segments: *i)* fail-auth handling (line 13-16) and *ii)* pass-auth execution (line 18-).

2. *Frequency anomaly* is an anomalous program behavior with aberrant ratios/relations between/among event occurrence frequencies. Normal relations among frequencies are established by: *i)* mathematical relations among induction variables that are specified in the binary (e.g., Fig. 4.2b), and *ii)* normal usage patterns of the program.



(a) The executions of s_1 and s_3 occur in the same run, similarly for s_2 and s_4.

(b) s_1, s_2 and s_3 occur at the same frequency in a run.

Figure 4.2: Examples of event co-occurrence and occurrence frequency relations.

One example of a frequency anomaly is a *directory harvest attack* against a mail server. The attack probes legitimate usernames on the server with a batch of emails targeting possible users. The attack results in an aberrant ratio between event frequencies in the server's handling procedures of existent/nonexistent receivers.

Sometimes an event occurrence frequency alone can indicate an attack, e.g., DoS. However, the workload of a real-world service may vary rapidly, and the individual frequencies are imprecise to model program behaviors.

4.2.3 Security Goals

The key to the detection of montage anomalies and frequency anomalies is to model and analyze the *relations among control-flow segments that occur in a large-scale execution window*. I further deduce two practical security goals for detecting aberrant path attacks. The deduction is based on the fact that *events* (e.g., call, jmp, or generic instructions) in dynamic program traces mark/indicate the control-flow segment to which they belong.

1. *Event co-occurrence analysis* examines the patterns of co-occurred events in a large-scale execution window[1]. I illustrate an event co-occurrence analysis in Fig. 4.2a. Rules should be learned that events $\langle s_1, s_3 \rangle$ or $\langle s_2, s_4 \rangle$ always occur together, but not $\langle s_1, s_4 \rangle$ or $\langle s_2, s_3 \rangle$.

[1]I define the co-occurrence of events in the scope of an execution window, not essentially at the same time.

2. *Event occurrence frequency analysis* examines the event occurrence frequencies and the relations among them. For instance, s_1, s_2 and s_3 always occur at the same frequency in Fig. 4.2b. Another type of event occurrence frequency relation is generated utterly due to specific usage patterns (mail server example in Sect. 4.2.2), which can be only learned from dynamic traces.

4.2.4 Basic Solutions and Their Inadequacy

There are several straightforward solutions providing event co-occurrence and occurrence frequency analysis. I point out their limitations, which help motivate my work.

Basic Solution I: One can utilize a large n in an n-gram approach (either deterministic approaches, e.g., [61], or probabilistic approaches, e.g., hidden Markov model [68,196]). This approach detects aberrant path attacks because long n-grams are large execution windows. However, it results in exponential training convergence complexity and storage complexity. Unless the detection system is trained with huge number of normal traces, which is exponential to n, a large portion of normal traces will be detected as anomalous. The exponential convergence complexity explains why no n-gram approach employs $n > 40$ in practice [60].

Basic Solution II: One can patch existing solutions with frequency analysis components to detect some aberrant path attacks, e.g., DoS. The possibility has been explored by Hubballi et al. on n-grams [91] and Frossi et al. on automata state transitions [65]. Their solutions successfully detect DoS attacks through unusually high frequencies of particular n-grams and individual automata state transitions. However, the underlying detection paradigms restrict the solutions from correlating arbitrary events in a long trace. Thus, their solutions do not detect general aberrant path attacks.

Basic Solution III: One can perform episodes mining within large-scale execution windows. It extends existing *frequent episode mining* [107,122] by extracting episodes (featured subsequences) at all frequencies so that infrequent-but-normal behaviors can be characterized. In order to analyze all episodes (the power set of events in a large-scale execution window), this approach faces a similar exponential complexity of training convergence as Basic Solution I.

4.3 Overview of my Approach

I present an overview of my approach analyzing event co-occurrence and event occurrence frequencies in large-scale execution windows. I develop a constrained agglomerative clustering algorithm to overcome the behavior diversity challenge. I develop a compact and fixed-length matrix representation to overcome the scalability problem for storing variable-length trace segments. I utilize probabilistic methods to estimate normal behaviors in an incomplete training dataset for overcoming the training scalability issue.

4.3.1 Profiling Program Behaviors

I design my approach to expose user-space program activities (executed control flow segments) via `call` instructions. `call` and `ret`[2] are responsible for call stack changes and provide a natural boundary for determining execution windows as discussed in Section 4.2.2.

I denote the overall activity of a program P within an execution window W as a behavior instance b. Instance b recorded in a program trace is profiled in two matrices:

Definition 4.3.1. *An event co-occurrence matrix O is an $m \times n$ Boolean matrix recording co-occurred call events in a behavior instance b. $o_{i,j}$ = `True` indicates the occurrence of the call from the i-th row symbol (a routine) to the j-th column symbol (a routine). Otherwise, $o_{i,j}$ = `False`.*

Definition 4.3.2. *A transition frequency matrix F is an $m \times n$ nonnegative matrix containing occurrence frequencies of all calls in a behavior instance b. $f_{i,j}$ records the occurrence frequency of the call from the i-th row symbol (a routine) to the j-th column symbol (a routine). $f_{i,j} = 0$ if the corresponding call does not occur in W.*

For one specific b, O is a Boolean interpretation of F that

$$o_{i,j} = \begin{cases} \texttt{True} & \text{if } f_{i,j} > 0 \\ \texttt{False} & \text{if } f_{i,j} = 0 \end{cases} \tag{4.1}$$

O and F are succinct representations of the dynamic call graph of a running program. m and n are total numbers of possible callers and callees in the program, respectively. Row/column symbols in O and F are determined through static analysis. m may not be equal to n, in particular when calls inside libraries are not counted.

Bitwise operations, such as `AND`, `OR`, and `XOR` apply to co-occurrence matrices. For example, O' `AND` O'' computes a new O that $o_{i,j} = o'_{i,j}$ `AND` $o''_{i,j}$.

Profiles at different granularities Although designed to be capable of modeling user-space program activities via function calls, my approach can also digest coarse level program traces for learning program behaviors. For example, system calls can be traced and profiled into O and F to avoid excessive tracing overheads in performance-sensitive deployments. The semantics of the matrices changes in this case; each cell in O and F represents a statistical relation between two system calls. The detection is not as accurate as my standard design because system calls are coarse descriptions of program executions.

[2] `ret` is paired with `call`, which can be verified via existing CFI technique. I do not involve the duplicated correlation analysis of `ret` in my model, but I trace `ret` to mark function boundaries for execution window partitioning.

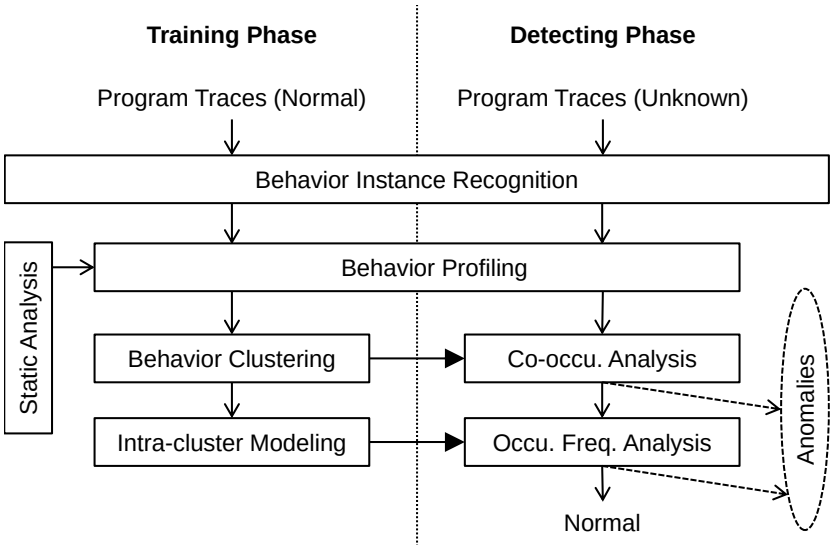


Figure 4.3: Overview of two-stage program anomaly detection. Information flows among operations in two stages and two phases of my program anomaly detection approach illustrated.

4.3.2 Architecture of My Approach

My approach consists of two complementary stages of modeling and detection where montage/frequency anomalies are detected in the first/second stage, respectively.

The first stage models the binary representation of event co-occurrences in a large-scale execution window via event co-occurrence matrix O. It performs event co-occurrence analysis against montage anomalies. It consists of a training operation Behavior Clustering and a detection operation Co-occurrence Analysis.

The second stage models the quantitative frequency relation among events in a large-scale execution window via transition frequency matrix F. It performs event occurrence frequency analysis against frequency anomalies. It consists of a training operation Intra-cluster Modeling and a detection operation Occurrence Frequency Analysis.

I illustrate the architecture of my approach in Fig. 4.3 and brief the functionalities of each operation below.

1. Behavior Profiling recognizes target execution windows $\{W_1, W_2, \dots\}$ in traces and profiles b from each W into O and F. Symbols in F and O are retrieved via static program analysis or system call table lookup.

2. Behavior Clustering is a training operation. It takes in all normal behavior instances $\{b_1, b_2, \dots\}$ and outputs a set of behavior clusters $\mathbb{C} = \{C_1, C_2, \dots\}$ where $C_i = \{b_{i_1}, b_{i_2}, \dots\}$.

3. Intra-cluster Modeling is a training operation. It is performed in each cluster. It takes in all normal behavior instances $\{b_{i_1}, b_{i_2}, \dots\}$ for C_i and constructs one deterministic model and one probabilistic model for computing the refined normal boundary in C_i.

4. Co-occurrence Analysis is an inter-cluster detection operation that analyzes O (of b) against clusters in $\mathbb{C}$ to seek montage anomalies. If behavior instance b is normal, it reduces the detection problem to subproblems within a set of behavior clusters $\mathbb{C}_b = \{C_{b_1}, C_{b_2}, \dots\}$, in which b closely fits.

5. Occurrence Frequency Analysis is an intra-cluster detection operation that analyzes F (of b) in each C_b to seek frequency anomalies. Behavior instance b is normal if F abides by the rules extracted from C_b and F is within the normal boundary established in C_b.

4.4 Inter-/intra-cluster Detection

I detail the training/modeling and detection operations in my two-stage approach. The key to the first stage is a customized clustering algorithm, which differentiates diverse program behaviors and divides the detection problem into subproblems. Based on the clustering, inter-/intra-cluster detection is performed in the first/second stage, respectively.

4.4.1 Behavior Clustering (Training)

I develop a constrained agglomerative clustering algorithm that addresses two special needs to handle program behavior instances for anomaly detection: *i)* long tail elimination, and *ii)* borderline behavior treatment. Standard agglomerative clustering algorithms result in a large number of tiny clusters in a long-tail distribution (shown in Section 4.6.1). Tiny clusters do not provide sufficient numbers of samples for statistical learning of the refined normal boundary inside each cluster. Standard algorithms also do not handle borderline behaviors, which could be trained in one cluster and tested in another, resulting in false alarms.

My algorithm (presented below) clusters program behavior instances based on the co-occurred events shared among instances. To deal with the borderline behavior issue, I alter the standard process into a two-step process:

1. generate scopes of clusters in an agglomerative way (line 13-28), and

2. add behavior instances to generated clusters (line 30-44).

Require: a set of normal program behavior instances B and a termination threshold T_d. $dist()$ is the distance function between behaviors/clusters. $pen()$ is the penalty function for long tail elimination.

Ensure: a set of behavior clusters $\mathbb{C}$.

$h \leftarrow \varnothing_{heap}$
$v \leftarrow \varnothing_{hashtable}$
$V \leftarrow \varnothing_{set}$
for all $b \in B$ **do**
    $O \leftarrow O_b$
    $v[O] \leftarrow v[O] + 1$
    for all $O' \in V$ **do**
        $d_p \leftarrow dist(O, O') \times pen(v[O], v[O'])$
        push $\langle d_p, O, v[O], O', v[O'] \rangle$ onto h
    end for
    add O to V
end for
while $h \neq \varnothing_{heap}$ **do**
    pop $\langle d_p, O_1, v_{O_1}, O_2, v_{O_2} \rangle$ from h
    break if $d_p > T_d$
    if $O_1 \in V$ and $O_2 \in V$ **then**
        continue if $v_{O_1} < v[O_1]$ or $v_{O_2} < v[O_2]$
        $O \leftarrow O_1$ `OR` O_2
        $v[O] \leftarrow v[O_1] + v[O_2]$
        remove O_1 from V
        remove O_2 from V
        for all $O' \in V$ **do**
            $d_p \leftarrow dist(O, O') \times pen(v[O], v[O'])$
            push $\langle d_p, O, v[O], O', v[O'] \rangle$ onto h
        end for
        add O to V
    end if
end while
$w[O] \leftarrow \varnothing_{set}$ **for all** $O \in V$
for all $b \in B$ **do**
    $O \leftarrow O_b$
    $m \leftarrow$ MAXINT
    for all $O' \in V$ **do**
        if O `OR` $O' = O'$ **then**
            if $dist(O, O') < m$ **then**
                $m \leftarrow dist(O, O')$
                $V' \leftarrow \{O'\}$
            else if $dist(O, O') = m$ **then**
                add O' to V'
            end if

41: **end if**
42: **end for**
43: add b to $w[O]$ **for all** $O \in V'$
44: **end for**
45: $\mathbb{C} \leftarrow \{w[O]$ **for all** $O \in V\}$

The basic idea of my agglomerative clustering algorithm is to put each behavior in a cluster (line 4 to line 12) and then merge the nearest clusters (line 13 to line 28) until the stopping criteria is reached (line 15). I use a lazily updated heap h in my clustering algorithm to minimize the calculation and sorting of distances between intermediate clusters. Each entry in h contains the distance between two clusters and h is sorted based on the distance. The design of the lazily updated heap ensures that a previously merged cluster is not removed proactively in h until the entry containing it is popped and abandoned.

The scope of a cluster $C = \{b_i \mid 0 \leq i \leq k\}$ is represented by its event co-occurrence matrix O_C. O_C records occurred events in any behavior instances in C. It is calculated using (4.2) where O_{b_i} is the event co-occurrence matrix of b_i.

$$O_C = O_{b_1} \texttt{ OR } O_{b_2} \texttt{ OR } \ldots \texttt{ OR } O_{b_k},\ 0 \leq i \leq k \tag{4.2}$$

The distances between *i)* two behavior instances, *ii)* two clusters, and *iii)* a behavior instance and a cluster are all measured by their co-occurrence matrices O_1 and O_2 in (4.3) where $|O|$ counts the number of `True` in O.

$$dist(O_1, O_2) = \frac{\text{Hamming}(O_1, O_2)}{\min(|O_1|, |O_2|)} \tag{4.3}$$

Hamming distance alone is insufficient to guide the cluster agglomeration: it loses the semantic meaning of O, and it weighs `True` and `False` the same. However, in co-occurrence matrices, only `True` contributes to the co-occurrence of events.

I explain the unique features of my constrained agglomerative clustering algorithm over the standard design:

- *Long tail elimination* A standard agglomerative clustering algorithm produces clusters with a long tail distribution of cluster sizes – there are a large number of tiny clusters, and the unbalanced distribution remains at various clustering thresholds. Tiny clusters provide insufficient number of behavior instances to train probabilistic models in Intra-cluster Modeling.

  In order to eliminate tiny/small clusters in the long tail, my algorithm penalizes $dist(O_1, O_2)$ by (4.4) before pushing it onto h. $|C_i|$ denotes the size of cluster C_i.

$$pen(|C_1|, |C_2|) = \max(\log(|C_1|), \log(|C_2|)) \tag{4.4}$$

- *Penalty maintenance* The distance penalty between C_1 and C_2 changes when any size of C_1 and C_2 changes. In this case, all entries in h containing a cluster whose size changes should be updated or nullified.

  I use a version control to mark the latest and deprecated versions of clusters in h. The version of a cluster C is recorded as its current size (an integer). It is stored in $v[O]$ where O is the event co-occurrence matrix of C. v is a hashtable that assigns `0` to an entry when the entry is accessed for the first time. A heap entry contains two clusters, their versions and their distance when pushed to h (line 9 and line 24). An entry is abandoned if any of its two clusters are found deprecated at the moment the entry is popped from h (line 17).

- *Borderline behavior treatment* It may generate a false positive when *i)* $dist(b, C_1) = dist(b, C_2)$, *ii)* b is trained only in C_1 during Intra-cluster Modeling, and *iii)* a similar behavior instance b' is tested against C_2 in operation Occurrence Frequency Analysis (intra-cluster detection).

  To treat this type of borderline behaviors correctly, my clustering algorithm duplicates b in every cluster, which b may belong to (line 30-44). This operation also increases cluster sizes and results in sufficient training in Intra-cluster Modeling.

4.4.2 Co-occurrence Analysis (Detection)

This operation performs inter-cluster detection to seek montage anomalies. A behavior instance b is tested against all normal clusters $\mathbb{C}$ to check whether the co-occurred events in b are consistent with co-occurred events found *in a single cluster*. An alarm is raised if no such cluster is found. Otherwise, b and its most closely fitted clusters $\mathbb{C}_b = \{C_1, \ldots, C_k\}$ are passed to Occurrence Frequency Analysis for intra-cluster detection.

An incoming behavior instance b fits in a cluster C if O_b `OR` $O_C = O_C$ where O_C and O_b are the event co-occurrence matrices of C and b. The detection process searches for all clusters in which b fits. If this set of clusters is not empty, distances between b and each cluster in this set are calculated using (4.3). The clusters with the nearest distance (there could be more than one cluster) are selected as $\mathbb{C}_b$.

4.4.3 Intra-cluster Modeling (Training)

Within a cluster C, my approach analyzes behavior instances through their transition frequency matrices $\{F_b \mid b \in C\}$. The matrices are vectorized into data points in a high-dimensional detection space where each dimension records the occurrence frequency of a specific event across profiles. Two analysis methods reveal relations among frequencies.

The probabilistic method. I employ a one-class SVM, i.e., ν-SVM [159], to seek a frontier $\mathcal{F}$ that envelops all behavior instances $\{b \mid b \in C\}$.

1. Each frequency value is preprocessed with a logarithmic function $f(x) = \log_2(x+1)$ to reduce the variance between extreme values (empirically proved necessary).
2. A subset of dimensions are selected through *frequency variance analysis* (FVA)[3] or *principle component analysis* (PCA)[4] before data points are consumed by ν-SVM. This step manages the *curse of dimensionality*, a common concern in high-dimensional statistical learning.
3. I pair the ν-SVM with a kernel function, i.e., radial basis function (RBF)[5], to search for a non-linearly $\mathcal{F}$ that envelops $\{b \mid b \in C\}$ tightly. The kernel function transforms a non-linear separating problem into a linearly separable problem in a high-dimensional space.

The deterministic method. I employ *variable range analysis* to measure frequencies of events with zero or near zero variances across all program behaviors $\{b \mid b \in C\}$.

Frequencies are discrete integers. If all frequencies of an event in different behavior instances are the same, PCA simply drops the corresponding dimension. In some clusters, all behavior instances (across all dimensions) in C are the same or almost the same. Duplicated data points are treated as a single point, and they cannot provide sufficient information to train probabilistic models, e.g., one-class SVM.

Therefore, I extract deterministic rules for events with zero or near zero variances. This model identifies the frequency range $[f_{min}, f_{max}]$ for each of such events. f_{min} can equal to f_{max}.

4.4.4 Occurrence Frequency Analysis (Detection)

This operation performs intra-cluster detection to seek frequency anomalies: *i)* deviant relations among multiple event occurrence frequencies, and/or *ii)* aberrant occurrence frequencies. Given a program behavior instance b and its closely fitted clusters $\mathbb{C}_b = \{C_1, \ldots, C_k\}$ discovered in Co-occurrence Analysis, this operation tests b in every C_i $(0 \leq i \leq k)$ and aggregates the results using (4.5).

$$\exists C \in \mathbb{C}\ N_{clt}(b, C) \Rightarrow b \text{ is normal} \tag{4.5}$$

[3]FVA selects dimensions/events with larger-than-threshold frequency variances across all behavior instances in C.

[4]PCA selects linear combinations of dimensions/events with larger-than-threshold frequency variances, which is a generalization of FVA.

[5]Multiple functions have been tested for selection.

The detection inside C is performed with 3 rules, and the result is aggregated into $N_{clt}(b, C)$.

$$N_{clt}(b, C) = \begin{cases} \texttt{True} & \text{normal by all 3 rules} \\ \texttt{False} & \text{anomalous by any rule} \end{cases} \tag{4.6}$$

- *Rule 1: normal if the behavior instance b passes the probabilistic model detection.* The frequency transition matrix F of b is vectorized into a high-dimensional data point and tested against the one-class SVM model built in Intra-cluster Modeling. This operation computes the distance d between b and the frontier $\mathcal{F}$ established in the ν-SVM. If b is within the frontier or b is on the same side as normal behavior instances, then $d > 0$. Otherwise, $d < 0$. d is compared with a detection threshold T_f that $T_f \in (-\infty, +\infty)$. b is abnormal if $d < T_f$.

- *Rule 2: normal if the behavior instance b passes the range model detection.* Events in b with zero or near zero variances are tested against the range model (the deterministic method) built in Intra-cluster Modeling. b is abnormal if any event frequency of b exceeds its normal range.

- *Rule 3: presumption of innocence in tiny clusters.* If no frequency model is trained in C because the size of C is too small, the behavior instance b is marked as normal. This rule generates false negatives. It sacrifices the detection rate for reducing false alarms in insufficiently trained clusters.

4.4.5 Discussion

My program anomaly detection approach is a context-sensitive language parser from the formal language perspective, i.e., Bach language parser [152]. In comparison, existing automata methods are at most context-free language parsers (pushdown automata methods) [55]. n-gram methods are regular language parsers (finite state machine equivalents [193]). Existing probabilistic methods are stochastic languages parsers (probabilistic regular language parsers).

A context-sensitive language parser is more precise than a context-free language parser or a regular language parser in theory. It is accepted by a linear bounded automaton (LBA), which is a restricted Turing machine with a finite tape. The advantage of a context-sensitive parser is its ability to characterize cross-serial dependencies, or to correlate far away events in a long program trace.

My approach explores the possibility to construct an efficient program anomaly detection approach on the context-sensitive language level. Potential mimicry attacks could be constructed to exploit the gap between Bach and the most precise program execution description. However, it is more difficult to do so than constructing mimicry attacks against regular or

context-free language level detection tools. For example, padding is a simple means to construct regular language level mimicry attacks, and my approach can detect padding attacks. My analysis characterizes whether two function calls should occur in one execution window, so padding rarely occurred calls can be detected. My approach recognizes the ratios between call pairs in one execution window. Thus, excessive padding elements can be discovered.

Potential mimicry attacks may exploit the monitoring granularity of a detection approach. My current approach utilizes `call` instructions to mark control-flow segments, which can be generalized to any instruction for detecting mimicry attacks that do not involve `call` in any part of their long attack paths. Another potential threat is machine learning poisoning [207], which could exploit the boundary of the probabilistic scope of the norm (Section 3.2.3).

4.5 Implementation

I implement a prototype of my detection approach on Linux (Fedora 21, kernel 3.19.3). The static analysis is realized through `C` (`ParseAPI` [145]). The profiling, training, and detection phases are realized in `Python`. The dynamic tracing and behavior recognition are realized through Intel `Pin`, a leading dynamic binary instrumentation framework, and `SystemTap`, a low-overhead dynamic instrumentation framework for Linux kernel. Tracing mechanisms are independent of my detection design; more efficient tracing techniques can be plugged in replacing `Pin` and `SystemTap` to improve the overall performance in the future.

Static analysis before profiling: symbols and address ranges of routines/functions are discovered for programs and libraries. The information helps to identify routine symbols if not found explicitly in dynamic tracing. Moreover, I leverage static analysis to list legal caller-callee pairs.

Profiling: My prototype *i)* verifies the legality of events (function calls) in a behavior instance b and *ii)* profiles b into two matrices (Sect. 4.3.1). The event verification filters out simple attacks that violate control flows before my approach detects stealthy aberrant path attacks. I implement profile matrices in Dictionary of Keys (DOK) format to minimize storage space for sparse matrices.

Dynamic tracing and behavior recognition: I develop a Pintool in JIT mode to trace function calls in the user space and to recognize execution windows within entire program executions. My Pintool is capable of tracing *i)* native function calls, *ii)* library calls *iii)* function calls inside dynamic libraries, *iv)* kernel thread creation and termination. Traces of different threads are isolated and stored separately. My Pintool recognizes whether a call is made within a given routine and on which nested layer the given routine executes (if nested execution of the given routine occurs). This functionality enables the recognition of large-scale execution windows through routine boundary partitioning.

I demonstrate that my approach is versatile recognizing program behaviors at different gran-

Table 4.1: Normal profile statistics.

Program	**Version**	**Events in Profile**	**Execution Window**	**#(N.P.)**
`sshd`	1.2.30	function calls	routine boundary	4800
`libpcre`	8.32	function calls	library call	11027
`sendmail`	8.14.7	system calls[†]	continuous operation	6579

N.P. is short for *normal profile*.

[†]Function calls are not traced due to its complex process spawning logic. Customization of my Pintool is needed to trace them.

Table 4.2: Statistics of average single normal profile.

Program	**#(Symbols)**	**#(Event)**	**#(Unique Event)**
`sshd`	415	34511	180
`libpcre`	79	44893	45
`sendmail`	350	1134	213

ularities. I develop a `SystemTap` script to trace system calls with timestamps. It enables execution window partitioning via activity intervals when the program is monitored as a black box.

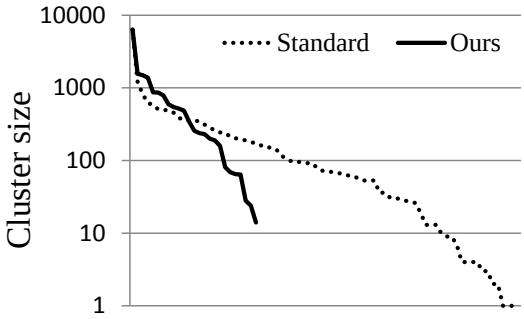
4.6 Evaluations

To verify the detection capability of my approach, I test my prototype against different types of aberrant path attacks (Sect. 4.6.2). I investigate its detection accuracy using real and synthetic program traces (Sect. 4.6.3). I evaluate the performance of my prototype with different tracing and detection options (Sect. 4.6.4).

4.6.1 Experiment Setup

I study three programs/libraries (Table 4.1) in distinct categories. I demonstrate that my approach is a versatile detection solution to be applied to programs and dynamic libraries with various large-scale execution window definitions and event definitions. I detail the programs/libraries and their training dataset (normal profiles) in Table 4.2 and below.

- `[sshd]` Execution window definition: program activities of `sshd` within routine `do_authentication()`. The routine `do_authentication()` is called in a forked thread



(a) `libpcre` cluster size distribution (sorted in descending order).

	`sshd`	`libpcre`	`sendmail`
T_d	1.6	2.0	1.6
$\|\mathbb{C}\|$	6	29	20

(b) Overview of program behavior clustering.

Figure 4.4: Clustering of program behavior instances.

after a client initializes its connection to `sshd`. All session activities are within the execution window if the authentication is passed. Normal runs cover three authentication methods (password, public key, rhost), each of which contains 800 successful and 800 failed connections. 128 random commands are executed in each successful connection.

- `[libpcre]` Execution window definition: program activities of `libpcre` when a library call is made. Library calls are triggered through `grep -P`. Over 10,000 normal tests are used from the `libpcre` package.

- `[sendmail]` Execution window definition: a continuous system call sequence wrapped by long no-op (no system call) intervals. `sendmail` is an event-driven program that only emits system calls when sending/receiving emails or performing a periodical check. I set up this configuration to demonstrate that my detection approach can consume various events, e.g., system calls. I collect over 6,000 normal profiles on a public `sendmail` server during 8 hours.

I list clustering threshold T_d used for the three studied programs/libraries in Fig. 4.4b[6]. $|\mathbb{C}|$ denotes the number of clusters computed with the specific T_d. In Fig. 4.4a, I demonstrate the effectiveness of my constrained agglomerative clustering algorithm to eliminate tiny clusters. The standard agglomerative clustering approach results in a long-tail distribution of cluster sizes shown in Fig. 4.4a.

In operation Occurrence Frequency Analysis, the detection threshold T_f is determined by a given false positive rate (FPR) upper bound, i.e., FPR^u, through cross-validation. In the training phase of cross-validation, I perform multiple random 10-fold partitioning. Among distances from all training partitions, T_f is initialized as the kth smallest distance within distances[7] between a behavior instance and the ν-SVM frontier $\mathcal{F}$. k is calculated

[6]The value is empirically chosen to keep a balance between an effective recognition of diverse behaviors and an adequate elimination of tiny clusters.

[7]The distance can be positive or negative. More details are specified in Rule 1 (Sect. 4.4.4).

Table 4.3: Overview of reproduced attacks.

Attack Name	**Target**	**Attack Settings**
flag variable overwritten	`sshd`	an inline virtual exploit that matches a username
ReDoS	`libpcre`	3 deleterious patterns paired with 8-23 input strings
directory harvest attack	`sendmail`	probing batch sizes: 8, 16, 32, 64, 100, 200, and 400

Table 4.4: Overview of detection results.

Attack Name	**#(Attack Attempt)**	**Detection Rate**	**FPR^u**
flag variable overwritten	800	100%	0.0001
ReDoS	46	100%	0.0001
directory harvest attack	14	100%	0.0001

FPR^u is the false positive rate upper bound (details in Sect. 4.6.1).

using FPR^u and the overall number of training cases. The FPR is calculated in the detection phase of cross-validation. If FPR $>$ FPR^u, a smaller k is selected until FPR $\leq$ FPR^u.

4.6.2 Discovering Real-World Attacks

I reproduce three known aberrant path attacks to test the detection capability of my approach. My detection approach detects all attack attempts with less than 0.0001 false positive rate. The overview of the attacks and detection results are presented in Table 4.3 and Table 4.4, respectively.

Flag Variable Overwritten Attack

Flag variable overwritten attack is a non-control data attack. An attacker tampers with decision-making variables. The exploit takes effect when the manipulated data affects the control flow at some later point of execution.

I reproduce the flag variable overwritten attack against `sshd` introduced by Chen et al. [36]. I describe the attack in Sect. 4.2.1, bullet (a) and in Fig. 4.1. I simplify the attack procedure by placing an *inline virtual exploit* in `sshd` right after the vulnerable routine `packet_read()`:

```
if (user[0] == 'e' && user[1] == 'v' && user[2] == 'e') authenticated = 1;
```

This inline virtual exploit produces the immediate consequence of a real exploit – overwriting `authenticated`. It does not interfere with my tracing/detection because no `call` instruction is employed. For each attack attempt, 128 random commands are executed after a successful login.

Normal[a]	**Normal**[b]	**Attack**
...	...	...
`auth_p > xfree`	`auth_p > xfree`	`auth_p > xfree`
`do_auth > xfree`	`do_auth > debug`	`do_auth > debug`
`do_auth > log_msg`	`do_auth > xfree`	`do_auth > xfree`
`do_auth > p_start`	`do_auth > p_start`	`do_auth > p_start`
`p_start > buf_clr`	`p_start > buf_clr`	`p_start > buf_clr`
...	...	...
`phdtw > buf_len`	`phdtw > buf_len`	`phdtw > buf_len`
`do_auth > do_autd`	`do_auth > p_read`	`do_auth > do_autd`
...	...	...

[a] A successfully authenticated session.
[b] A failed (wrong password) authentication.
"`caller > callee`" denotes a function call.
Routine names are abbreviated to save space.

Figure 4.5: Samples of normal and anomalous `sshd` traces.

My approach (configured at FPR^u 0.0001) successfully detects all attack attempts in inter-cluster detection (Co-occurrence Analysis)[8]. I present normal and attack traces inside the execution window (selected routine `do_authentication()`) in Fig. 4.5 to illustrate the detection.

In Fig. 4.5, the *Attack* and *Normal*^{b} bear the same trace prior to the last line, and the *Attack* and *Normal*^{a} bear the same trace after (including) the last line. My approach detects the attack as a montage anomaly: the control-flow segment containing `do_auth > debug` should not co-occur with the control-flow segment containing `do_auth > do_authed` (and following calls) in a single execution window.

In the traces, there are identical 218 `call` events including library routines (36 `call`s excluding library ones) between the third line and the last line in Fig. 4.5. I test an n-gram detection tool, and it requires at least $n = 37$ to detect the specific attack without libraries routine traced. The 37-gram model results in an FPR of 6.47% (the FPR of my approach is less than 0.01%). This indicates that n-gram models with a large n is difficult to converge at training. I do not test automaton-based detection because they cannot detect the attack in theory. The attack does not contain any illegal function calls.

Regular Expression Denial of Service

Regular expression Denial of Service (ReDoS) is a service abuse attack. It exploits the exponential time complexity of a regex engine when performing backtracking. The attacks

[8] One-class SVM in Occurrence Frequency Analysis only detects 3.8% attack attempts if used alone.

Table 4.5: Deleterious patterns used in ReDoS attacks.

	Deleterious Pattern	**#(attack)**
Pattern 1	`^(a+)+$`	15
Pattern 2	`((a+)*)+$`	8
Pattern 3	`^(([a-z])+.)+[A-Z]([a-z])+$`	23

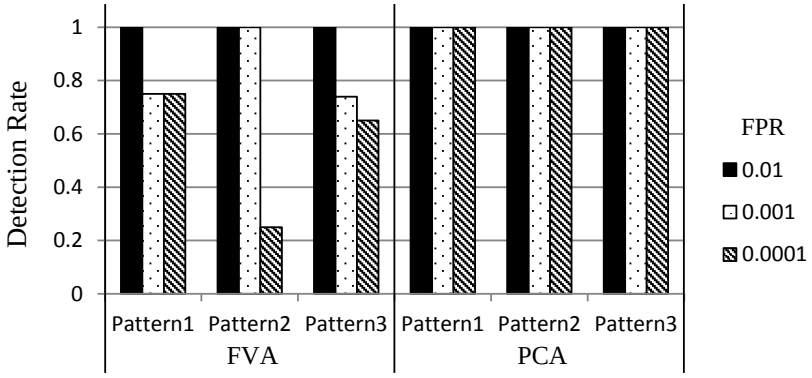


Figure 4.6: Detection rates of ReDoS attacks.

construct extreme matching cases where backtracking is involved. All executed control flows are legal, but the regex engine hangs due to the extreme complexity.

I produce 46 ReDoS attack attempts targeting `libpcre`[9]. Three deleterious patterns are used (Table 4.5). For each deleterious pattern, attacks are constructed with an increasing length of `a` in the input string starting at 6, e.g., `aaaaaaaab`. I stop attacking `libpcre` at different input string lengths so that the longest hanging time periods for different deleterious patterns are about the same (a few seconds). A longer input string incurs a longer hanging time; it results in a more severe ReDoS attack than a shorter one.

ReDoS attacks are detected in intra-cluster detection operation (Occurrence Frequency Analysis) by the probabilistic method, i.e., ν-SVM. I test my approach with both PCA and FVA feature selection (Sect. 4.4.3, the probabilistic method, bullet b). The detection results (Fig. 4.6) show that my approach configured with PCA is more sensitive than it configured with FVA. My approach (with PCA) detects all attack attempts at different FPRs[10]. The undetected attack attempts (with FVA) are all constructed with the small amount of `a` in the input strings, which do not result in very severe ReDoS attacks.

[9]Internal deep recursion prevention of `libcpre` is disabled.

[10]No attack is detected if only Co-occurrence Analysis is performed.

Directory Harvest Attack

Directory harvest attack (DHA) is a service abuse attack. It probes valid email users through brute force. I produce 14 DHA attack attempts targeting `sendmail`. Each attack attempt consists of a batch of closely sent probing emails with a dictionary of possible receivers. I conduct DHA attacks with 7 probing batch sizes from 8 to 400 (Table 4.3). Two attack attempts are conducted for each batch size.

My approach (configured at FPR^u 0.0001) successfully detects all attack attempts with either PCA or FVA feature selection[10]. DHA attacks are detected in intra-cluster detection (Occurrence Frequency Analysis) by the probabilistic method, i.e., ν-SVM. The attacks bypass the inter-cluster detection (Co-occurrence Analysis) because invalid usernames occur in normal training dataset.

This experiment demonstrates that my approach can consume coarse program behavior descriptions (e.g., system calls) to detect attacks. Most of the probing emails do not have valid receivers. They result in a different processing procedure than that for normal emails; the batch of DHA emails processed in an execution window gives anomalous ratios between frequencies of *valid email processing control flows* and frequencies of *invalid email processing control flows*. In `sendmail`, these different control flows contain different sets of system calls, so they are revealed by system call profiles. More precise detection requires the exposure of internal program activities, such as function calls.

4.6.3 Systematic Accuracy Evaluation

I systematically demonstrate how sensitive and accurate my approach is through receiver operating characteristic (ROC). Besides normal program behaviors ground truth (Sect. 4.6.1), I generate four types of synthetic aberrant path anomalies. I first construct F' for each synthetic anomalous behavior instance b', and then I use (4.1) to derive O' (of b') from F'[11].

1. *Montage anomaly*: two behavior instance b_1 and b_2 are randomly selected from two different behavior clusters. For a cell $f'_{i,j}$ in F', if one of $f_{1_{i,j}}$ (of F_1) and $f_{2_{i,j}}$ (of F_2) is 0, the value of the other is copied into $f'_{i,j}$. Otherwise, one of them is randomly selected and copied.

2. *Incomplete path anomaly*: random one-eighth of non-zero cells of a normal F are dropped to 0 (indicating events that have not occurred) to construct F'.

3. *High-frequency anomaly*: three cells in a normal F are randomly selected, and their values are magnified 100 times to construct F'.

[11]The synthetic anomalies are tested not in the normal program behavior set. The generated profiles are defined to be anomalous (outside the scope of the normal program behaviors) for testing purposes.

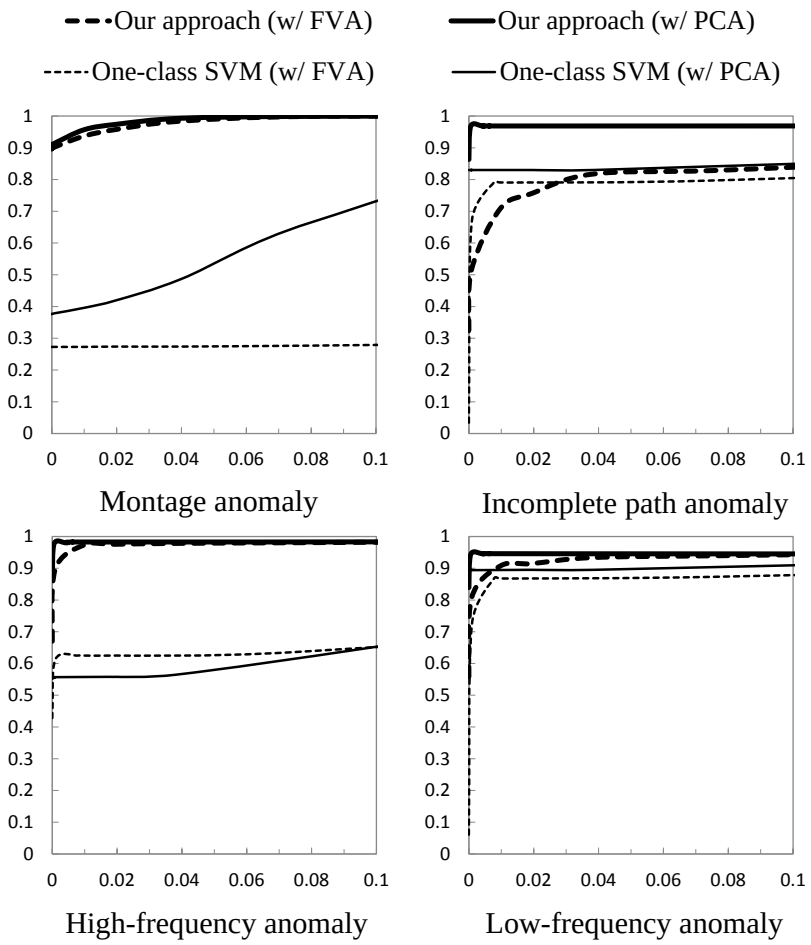


Figure 4.7: `libpcre` ROC of my approach and basic one-class SVM. X-axis is false positive rate, and y-axis is detection rate.

4. *Low-frequency anomaly*: similar to high-frequency anomalies, but the values of the three cells are reduced to 1.

To demonstrate the effectiveness of my design in handling diverse program behaviors, I compare my approach with a basic one-class SVM (the same ν-SVM and same configurations, e.g., kernel function, feature selection, and parameters, as used in my INTRA-CLUSTER MODELING operation).

I present the detection accuracy results on `libpcre` in Fig. 4.7, which has the most complicated behavior patterns among the three studied programs/libraries[12]. In any subfigure of Fig. 4.7, each dot is associated with a false positive rate (multi-round 10-fold cross-validation with 10,000 test cases) and a detection rate (1,000 synthetic anomalies). I denote an *anomaly*

[12]Results of the other two programs share similar characteristics as `libpcre` and are not presented.

result as a *positive*.

Taking the montage anomaly `libpcre` ROC figure (the first subfigure) as an example, I describe the process of ROC curve computation. First, I pick up an FPR and train the four detection systems to obtain proper parameters, e.g., T_f, so that different ν-SVMs in different clusters yield the same FPR. 10-fold cross-validation is used in the training. Second, every trained detection system is tested against synthetic anomalies (montage anomalies in this case). Each point in the ROC figure is computed with 10,000 test cases in repeated 10-fold cross-validations and 1,000 synthetic anomalies in system testing. The two steps are applied for different FPRs until the whole figure is drawn.

$$TPR = TP/(TP + FN) \tag{4.7}$$

$$TPR = FP/(FP + TN) \tag{4.8}$$

Fig. 4.7 shows the effectiveness of my clustering design. The detection rate of my prototype (with PCA[13]) is usually higher than 0.9 with FPR less than 0.01. Because of diverse patterns, basic one-class SVM fails to learn tight boundaries that wrap diverse normal patterns as expected. A loose boundary results in false negatives and low detection rates.

4.6.4 Performance Analysis

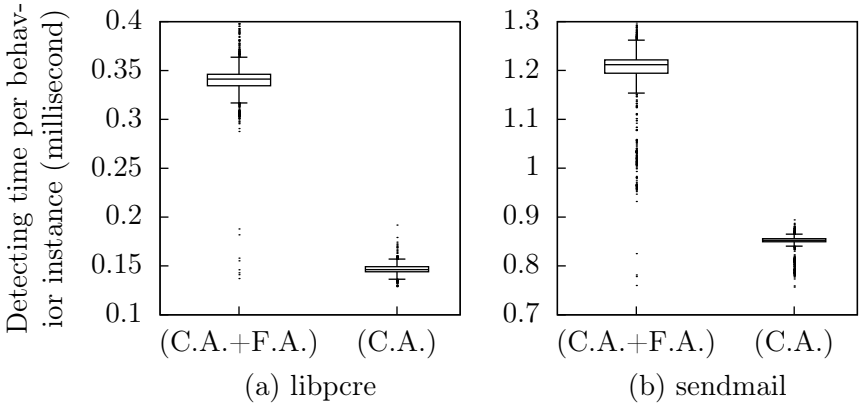
Although performance is not a critical issue for the training phase, a fast and efficient detection is important for enabling real-time protection and minimizing negative user experience [136]. The overall overhead of a program anomaly detection system comes from *tracing* and *analysis* in general.

I evaluate the performance of my analysis procedures (inter- and intra-cluster detections) with either function call profiles (`libpcre`) or system call profiles (`sendmail`). I test the analysis on all normal profiles (`libpcre`: 11027, `sendmail`: 6579) to collect overhead for *inter-cluster detection alone* and *the combination of inter- and intra-cluster detection*[14]. The analysis of each behavior instance is repeated 1,000 times to obtain a fair timing. The performance results in Fig. 4.8 illustrate that

- It takes 0.1 ms to 1.3 ms to analyze a single behavior instance, which contains 44893 function calls (`libpcre`) or 1134 system calls (`sendmail`) on average (Table 4.1).
- The analysis overhead is positively correlated with the number of unique events in a profile (Table 4.1), which is due to my DOK implementation of profile matrices.

[13] PCA proves itself more accurate than FVA in Fig. 4.7.

[14] PCA is used for feature selection. FVA (results omitted) yields a lower overhead due to its simplicity.



C.A.+F.A.: Inter- and intra-cluster detection combined.
C.A.: Inter-cluster detection only.

Figure 4.8: Detection (analysis) overhead of my approach.

- Montage anomalies takes less time to detect than frequency anomalies, because they are detected at the first stage (Co-occurrence Analysis).

Compared with the analysis procedure, dynamic function call tracing incurs a noticeable overhead. `sshd` experiences a 167% overhead on average when my Pintool is loaded. A similar 141% overhead is reported by Jalan and Kejariwal in their dynamic call graph Pintool `Trin-Trin` [98]. Advanced tracing techniques, e.g., probe mode pintool, branch target store [202], etc., can potentially reduce the tracing overhead to less than 10% toward a real-time detection system.

Another choice to deploy my detection solution is to profile program behaviors through system calls as I demonstrate using `sendmail`. System calls can be traced through `SystemTap` with near-zero overhead [181], but it sacrifices the capability to reveal user-space program activities and downgrades the modeling/detection accuracy.

My approach can support offline detection or forensics of program attacks, in which case accuracy is the main concern instead of performance [179]. My Pintool enables analysts to locate anomalies within execution windows, and my matrices provide caller information for individual function calls. This information helps analysts quickly reduce false alarms and locate vulnerable code segments.

Summary I evaluate the detection capability, accuracy, and performance of my detection prototype on Linux.

- My approach successfully detects all reproduced aberrant path attack attempts against `sshd`, `libpcre` and `sendmail` with less than 0.0001 false positive rates.
- My approach is accurate in detecting different types of synthetic aberrant path anoma-

lies with a high detection rate (> 0.9) and a low false positive rate (< 0.01).

- My approach analyzes program behaviors fast; it only incurs 0.1 ms to 1.3 ms analysis overhead (excluding tracing) per behavior instance (1k to 50k function/system calls in my experiments).

Chapter 5

Privacy-Preserving Detection of Sensitive Data Exposure

5.1 Introduction

In this chapter, I present a data-leak detection solution which can be outsourced and deployed in a semi-honest detection environment where the detection does not require the sensitive data in plaintext. I design, implement, and evaluate my *fuzzy fingerprint* technique that enhances data privacy during data-leak detection operations. My approach is based on a fast and practical one-way computation on the sensitive data (SSN records, classified documents, sensitive emails, etc.). It enables the data owner to securely delegate the content-inspection task to DLD providers without exposing the sensitive data. Using my detection method, the DLD provider, who is modeled as an honest-but-curious (aka semi-honest) adversary, can only gain limited knowledge about the sensitive data from either the released digests, or the content being inspected. Using my techniques, an Internet service provider (ISP) can perform detection on its customers' traffic securely and provide data-leak detection as an add-on service for its customers. In another scenario, individuals can mark their own sensitive data and ask the administrator of their local network to detect data leaks for them.

In my detection procedure, the data owner computes a special set of digests or fingerprints from the sensitive data and then discloses only a small amount of them to the DLD provider. The DLD provider computes fingerprints from network traffic and identifies potential leaks in them. To prevent the DLD provider from gathering exact knowledge about the sensitive data, the collection of potential leaks is composed of real leaks and noises. It is the data owner, who post-processes the potential leaks sent back by the DLD provider and determines whether there is any real data leak.

This chapter details my solution and provides extensive experimental evidences and theoretical analyses to demonstrate the feasibility and effectiveness of my approach. My contribu-

tions are summarized as follows.

1. I describe a privacy-preserving data-leak detection model for preventing inadvertent data leak in network traffic. My model supports detection operation delegation and ISPs can provide data-leak detection as an add-on service to their customers using my model.

   I design, implement, and evaluate an efficient technique, fuzzy fingerprint, for privacy-preserving data-leak detection. Fuzzy fingerprints are special sensitive data digests prepared by the data owner for release to the DLD provider.

2. I implement my detection system and perform extensive experimental evaluation on 2.6 GB Enron dataset, Internet surfing traffic of 20 users, and also 5 simulated real-world data-leak scenarios to measure its privacy guarantee, detection rate and efficiency. The results indicate high accuracy achieved by my underlying scheme with very low false positive rate. The results also show that the detection accuracy does not degrade much when only partial (sampled) sensitive-data digests are used. In addition, I give an empirical analysis of my fuzzification as well as of the fairness of fingerprint partial disclosure.

5.2 Model and Overview

I abstract the privacy-preserving data-leak detection problem with a threat model, a security goal and a privacy goal. First I describe the two most important players in my abstract model: the organization (i.e., data owner) and the data-leak detection (DLD) provider.

- *Organization* owns the sensitive data and authorizes the DLD provider to inspect the network traffic from the organizational networks for anomalies, namely inadvertent data leak. However, the organization does not want to directly reveal the sensitive data to the provider.
- *DLD provider* inspects the network traffic for potential data leaks. The inspection can be performed offline without causing any real-time delay in routing the packets. However, the DLD provider may attempt to gain knowledge about the sensitive data.

I describe the security and privacy goals in Section 5.2.1 and Section 5.2.2.

5.2.1 Security Goal and Threat Model

I categorize three causes for sensitive data to appear on the outbound traffic of an organization, including the legitimate data use by the employees.

- Case I *Inadvertent data leak*: The sensitive data is accidentally leaked in the outbound traffic by a legitimate user. This thesis focuses on detecting this type of accidental data leaks over supervised network channels. Inadvertent data leak may be due to human errors such as forgetting to use encryption, carelessly forwarding an internal email and attachments to outsiders, or due to application flaws (such as described in [105]). A supervised network channel could be an unencrypted channel or an encrypted channel where the content in it can be extracted and checked by an authority. Such a channel is widely used for advanced NIDS where MITM (man-in-the-middle) SSL sessions are established instead of normal SSL sessions [101].

- Case II *Malicious data leak*: A rogue insider or a piece of stealthy software may steal sensitive personal or organizational data from a host. Because the malicious adversary can use strong private encryption, steganography or covert channels to disable content-based traffic inspection, this type of leaks is out of the scope of my network-based solution. Host-based defenses (such as detecting the infection onset [204]) need to be deployed instead.

- Case III *Legitimate and intended data transfer*: The sensitive data is sent by a legitimate user intended for legitimate purposes. In this thesis, I assume that the data owner is aware of legitimate data transfers and permits such transfers. So the data owner can tell whether a piece of sensitive data in the network traffic is a leak using legitimate data transfer policies.

The security goal in this thesis is to detect Case I leaks, that is inadvertent data leaks over supervised network channels. In other words, I aim to discover sensitive data appearance in network traffic over supervised network channels. I assume that: *i)* plaintext data in supervised network channels can be extracted for inspection; *ii)* the data owner is aware of legitimate data transfers (Case III); and *iii)* whenever sensitive data is found in network traffic, the data owner can decide whether or not it is a data leak. Network-based security approaches are ineffective against data leaks caused by malware or rogue insiders as in Case II, because the intruder may use strong encryption when transmitting the data, and both the encryption algorithm and the key could be unknown to the DLD provider.

5.2.2 Privacy Goal and Threat Model

To prevent the DLD provider from gaining knowledge of sensitive data during the detection process, I need to set up a privacy goal that is complementary to the security goal above. I model the DLD provider as a semi-honest adversary, who follows my protocol to carry out the operations, but may attempt to gain knowledge about the sensitive data of the data owner. My privacy goal is defined as follows. The DLD provider is given digests of sensitive data from the data owner and the content of network traffic to be examined. The DLD provider should not find out the exact value of a piece of sensitive data with a probability

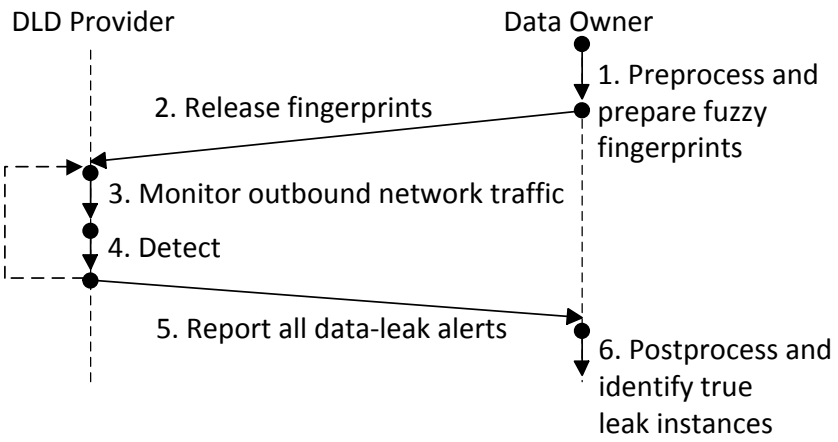


Figure 5.1: Overview of privacy-preserving data-Leak detection model.

greater than $\frac{1}{K}$, where K is an integer representing the number of all possible sensitive-data candidates that can be inferred by the DLD provider.

I present a privacy-preserving DLD model with a new fuzzy fingerprint mechanism to improve the data protection against semi-honest DLD provider. I generate digests of sensitive data through a one-way function, and then hide the sensitive values among other non-sensitive values via fuzzification. The privacy guarantee of such an approach is much higher than $\frac{1}{K}$ when there is no leak in traffic, because the adversary's inference can only be gained through brute-force guesses.

The traffic content is accessible by the DLD provider in plaintext. Therefore, in the event of a data leak, the DLD provider may learn sensitive information from the traffic, which is inevitable for all deep packet inspection approaches. My solution confines the amount of maximal information learned during the detection and provides quantitative guarantee for data privacy.

5.2.3 Overview of Privacy-Enhancing DLD

My privacy-preserving data-leak detection method supports practical data-leak detection as a service and minimizes the knowledge that a DLD provider may gain during the process. Fig. 5.1 lists the six operations executed by the data owner and the DLD provider in my protocol. They include PREPROCESS run by the data owner to prepare the digests of sensitive data, RELEASE for the data owner to send the digests to the DLD provider, MONITOR and DETECT for the DLD provider to collect outgoing traffic of the organization, compute digests of traffic content, and identify potential leaks, REPORT for the DLD provider to return data-leak alerts to the data owner where there may be false positives (i.e., false alarms), and POSTPROCESS for the data owner to pinpoint true data-leak instances. Details are presented

in the next section.

The protocol is based on strategically computing data similarity, specifically the quantitative similarity between the sensitive information and the observed network traffic. High similarity indicates potential data leak. For data-leak detection, the ability to tolerate a certain degree of data transformation in traffic is important. I refer to this property as *noise tolerance*. My key idea for fast and noise-tolerant comparison is the design and use of a set of *local features* that are representatives of local data patterns, e.g., when byte b_2 appears in the sensitive data, it is usually surrounded by bytes b_1 and b_3 forming a local pattern b_1, b_2, b_3. Local features preserve data patterns even when modifications (insertion, deletion, and substitution) are made to parts of the data. For example, if a byte b_4 is inserted after b_3, the local pattern b_1, b_2, b_3 is retained though the global pattern (e.g., a hash of the entire document) is destroyed. To achieve the privacy goal, the data owner generates a special type of digests, which I call fuzzy fingerprints. Intuitively, the purpose of fuzzy fingerprints is to hide the true sensitive data in a crowd. It prevents the DLD provider from learning its exact value. I describe the technical details next.

5.3 Fuzzy Fingerprint Method and Protocol

I describe technical details of my fuzzy fingerprint mechanism in this section.

5.3.1 Shingles and Fingerprints

The DLD provider obtains digests of sensitive data from the data owner. The data owner uses a sliding window and Rabin fingerprint algorithm [154] to generate short and hard-to-reverse (i.e., one-way) digests through the fast polynomial modulus operation. The sliding window generates small fragments of the processed data (sensitive data or network traffic), which preserves the local features of the data and provides the noise tolerance property. Rabin fingerprints are computed as polynomial modulus operations, and can be implemented with fast XOR, shift, and table look-up operations. The Rabin fingerprint algorithm has a unique min-wise independence property [23], which supports fast random fingerprints selection (in uniform distribution) for partial fingerprints disclosure.

The shingle-and-fingerprint process is defined as follows. A sliding window is used to generate q-grams on an input binary string first. The fingerprints of q-grams are then computed.

A shingle (q-gram) is a fixed-size sequence of contiguous bytes. For example, the 3-gram shingle set of string `abcdefgh` consists of six elements {`abc`, `bcd`, `cde`, `def`, `efg`, `fgh`}. Local feature preservation is accomplished through the use of shingles. Therefore, my approach can tolerate sensitive data modification to some extent, e.g., inserted tags, small amount of character substitution, and lightly reformatted data. The use of shingles for finding duplicate

web documents first appeared in [21, 22].

The use of shingles alone does not satisfy the one-wayness requirement. Rabin fingerprint is utilized to satisfy such requirement after shingling. In fingerprinting, each shingle is treated as a polynomial $q(x)$. Each coefficient of $q(x)$, i.e., c_i $(0 < i < k)$, is one bit in the shingle. $q(x)$ is mod by a selected irreducible polynomial $p(x)$. The process shown in (5.1) maps a k-bit shingle into a p_f-bit fingerprint f where the degree of $p(x)$ is $p_f + 1$.

$$f = c_1 x^{k-1} + c_2 x^{k-2} + \ldots + c_{k-1} x + c_k \bmod p(x) \tag{5.1}$$

From the detection perspective, a straightforward method is for the DLD provider to raise an alert if any sensitive fingerprint matches the fingerprints from the traffic[1]. However, this approach has a privacy issue. If there is a data leak, there is a match between two fingerprints from sensitive data and network traffic. Then, the DLD provider learns the corresponding shingle, as it knows the content of the packet. Therefore, the central challenge is *to prevent the DLD provider from learning the sensitive values even in data-leak scenarios*, while allowing the provider to carry out the traffic inspection.

I propose an efficient technique to address this problem. The main idea is to relax the comparison criteria by strategically introducing matching instances on the DLD provider's side *without increasing false alarms for the data owner*. Specifically, *i)* the data owner perturbs the sensitive-data fingerprints before disclosing them to the DLD provider, and *ii)* the DLD provider detects leaking by a range-based comparison instead of the exact match. The range used in the comparison is pre-defined by the data owner and correlates to the perturbation procedure. I define the notions of *fuzzy length* and *fuzzy set* next and then describe how they are used in my detailed protocol in Section 5.3.2.

Definition 5.3.1. *Given a p_f-bit-long fingerprint f, the fuzzy length p_d $(p_d < p_f)$ is the number of bits in f that may be perturbed by the data owner.*

Definition 5.3.2. *Given a fuzzy length p_d, and a collection of fingerprints, the fuzzy set S_{f,p_d} of a fingerprint f is the set of fingerprints in the collection whose values differ from f by at most $2^{p_d} - 1$.*

In Definition 5.3.2, the size of the fuzzy set $|S_{f,p_d}|$ is upper bounded by 2^{p_d}, but the actual size may be smaller due to the sparsity of the fingerprint space.

5.3.2 Operations in My Protocol

1. PREPROCESS: This operation is run by the data owner on each piece of sensitive data.

[1] In reality, data-leak detection solutions usually utilize more complex statistical models to raise alerts instead of alerting individual fingerprints. Statistical approaches, e.g., packet sensitivity in Section V, eliminate accidental matches.

(a) The data owner chooses four public parameters $(q, p(x), p_d, M)$. q is the length of a shingle. $p(x)$, is an irreducible polynomial (degree of $p_f + 1$) used in Rabin fingerprint. Each fingerprint is p_f-bit long and the fuzzy length is p_d. M is a bitmask, which is p_f-bit long and contains p_d `0`'s at random positions. The positions of `1`'s and `0`'s in M indicate the bits to preserve and to randomize in the fuzzification, respectively.

(b) The data owner computes $\mathbb{S}$, which is the set of all Rabin fingerprints of the piece of sensitive data.

(c) The data owner transforms each fingerprint $f \in \mathbb{S}$ into a fuzzy fingerprint f^* with randomized bits (specified by the mask M). The procedure is described as follows: for each $f \in \mathbb{S}$, the data owner generates a random p_f-bit binary string $\dot{f}$, mask out the bits not randomized by $\dot{f}' = (\texttt{NOT}\ M)\ \texttt{AND}\ \dot{f}$ (`1`'s in M indicate positions of bits not to randomize), and fuzzify f with $f^* = f\ \texttt{XOR}\ \dot{f}'$. The overall computation is described in (5.2).

$$f^* = ((\texttt{NOT}\ M)\ \texttt{AND}\ \dot{f})\ \texttt{XOR}\ f \tag{5.2}$$

All fuzzy fingerprints are collected and form the output of this operation, the fuzzy fingerprint set, $\mathbb{S}^*$.

2. RELEASE: This operation is run by the data owner. The fuzzy fingerprint set $\mathbb{S}^*$ obtained by PREPROCESS is released to the DLD provider for use in the detection, along with the public parameters $(q, p(x), p_d, M)$. The data owner keeps $\mathbb{S}$ for use in the subsequent POSTPROCESS operation.

3. MONITOR: This operation is run by the DLD provider. The DLD provider monitors the network traffic T from the data owner's organization. Each packet in T is collected and the payload of it is sent to the next operation as the network traffic (binary) string $\tilde{T}$.

   The payload of each packet is not the only choice to define $\tilde{T}$. A more sophisticated approach could identify TCP flows and extract contents in a TCP session as $\tilde{T}$. Contents of other protocols can also be retrieved if required by the detection metrics.

4. DETECT: This operation is run by the DLD provider on each $\tilde{T}$ as follows.

   (a) The DLD provider first computes the set of Rabin fingerprints of traffic content $\tilde{T}$ based on the public parameters. The set is denoted as $\mathbb{T}$.

   (b) The DLD provider tests whether each fingerprint $f' \in \mathbb{T}$ is also in $\mathbb{S}^*$ using the fuzzy equivalence test (5.3).

$$E(f', f^*) = \texttt{NOT}\ (M\ \texttt{AND}\ (f'\ \texttt{XOR}\ f^*)) \tag{5.3}$$

   $E(f', f^*)$ is either `True` or `False`. $f'\ \texttt{XOR}\ f^*$ gives the difference between f' and f^*. $M\ \texttt{AND}\ (f'\ \texttt{XOR}\ f^*)$ filters the result leaving only the interesting bits (preserved

bits with `1`'s in M). Because `XOR` yields `0` for equivalent bits, `NOT` is used to turn `0`-bits into `1`'s (and `1`'s into `0`'s). The overall result from (5.3) is read as a boolean indicating whether or not f' is equivalent to a fuzzy fingerprint $f^* \in \mathbb{S}^*$.

(5.2) and (5.3) are designed in a pair, and M works the same in both equations by masking out fuzzified bits at same positions in each f, f^* and f'. All f' with `True` values are record in a set $\hat{\mathbb{T}}$.

(c) The DLD provider aggregates the outputs from the preceding step and raises alerts based on a threshold. My concrete aggregation formula is given in section 5.5.

5. Report: If Detection on $\tilde{T}$ yields an alert, the DLD provider reports the set of detected candidate leak instances $\hat{\mathbb{T}}$ to the data owner.

6. Postprocess: After receiving $\hat{\mathbb{T}}$, the data owner test every $f' \in \hat{\mathbb{T}}$ to see whether it is in $\mathbb{S}$. A precise likelihood of data leaking is computed at the data owner's, which I discuss more in section 5.5.

In the protocol, because S_{f^*,p_d}, the fuzzy set of f^*, includes the original fingerprint f, the true data leak can be detected (i.e., true positive). Yet, due to the increased detection range, multiple values in S_{f^*,p_d} may trigger alerts. Because S_{f^*,p_d} is large for the given network flow, the DLD provider has a low probability of pinpointing the sensitive data, which can be bounded as shown in Section 5.4.

The Detect operation can be performed between $\mathbb{T}$ and $\mathbb{S}^*$ via set intersection test with a special equivalence test function (e.g. Formula 5.5 in Section 5.5 as one realization). The advantage of my method is that the additional matching instances introduced by fuzzy fingerprints protect the sensitive data from the DLD provider; yet they do not cause additional false alarms for the data owner, as the data owner can quickly distinguish true and false leak instances. Given the digest f of a piece of sensitive data, a large collection T of traffic fingerprints, and a positive integer $K \ll |T|$, the data owner can choose a fuzzy length p_d such that there are at least $K-1$ other distinct digests in the fuzzy set of f, assuming that the shingles corresponding to these K digests are equally likely to be candidates for sensitive data and to appear in network traffic. A tight fuzzy length (i.e., the smallest p_d value satisfying the privacy requirement) is important for efficient Postprocess operation. Due to the dynamic nature of network traffic, p_d needs to be estimated accordingly. There exists an obvious tradeoff between privacy and detection efficiency – large fuzzy set allows a fingerprint to hide among others and confuses the DLD provider, yet this indistinguishability results in more work in Postprocess. I provide quantitative analysis on fuzzy fingerprint including empirical results on different sizes of fuzzy sets.

5.3.3 Extensions

Fingerprint Filter I develop this extension to use Bloom filter in the Detect operation for

efficient set intersection test. Bloom filter [20] is a well-known space-saving data structure for performing set-membership test. It applies multiple hash functions to each of the set elements and stores the resulting values in a bit vector; to test whether a value v belongs to the set, the filter checks each corresponding bit mapped with each hash function. Bloom filter in combination with Rabin fingerprint is referred to by us as the *fingerprint filter*. I use Rabin fingerprints with variety of modulus's in fingerprint filter as the hash functions, and I perform extensive experimental evaluation on both fingerprint filter and bloom filter with MD5/SHA in Section 5.5.

Partial disclosure Using the min-wise independent property of Rabin fingerprint, the data owner can quickly disclose partial fuzzy fingerprints to the DLD provider. The purpose of partial disclosure is two-fold: *i)* to increase the scalability of the comparison in the DETECT operation, and *ii)* to reduce the exposure of data to the DLD provider for privacy. The method of partial release of sensitive data fingerprints is similar to the suppression technique in database anonymization [3, 35].

This extension requires a good uniform distribution random selection to avoid disclosure bias. The min-wise independence feature of Rabin fingerprint guarantees that the minimal fingerprint is coming from a (uniformly distributed) random shingle. The property is also valid for a minimum set of fingerprints and so the data owner can just select r smallest elements in $\mathbb{S}^*$ to perform partial disclosure. The r elements are then sent to the DLD provider in RELEASE operation instead of $\mathbb{S}^*$. I implement the partial disclosure policy, evaluate its influence on detection rate, and verify the min-wise independence property of Rabin fingerprint in Section 5.5.

5.4 Analysis and Discussion

I analyze the security and privacy guarantees provided by my data-leak detection system, as well as discuss the sources of possible false negatives – data leak cases being overlooked and false positives – legitimate traffic misclassified as data leak in the detection. I point out the limitations associated with the proposed network-based DLD approaches.

Privacy Analysis My privacy goal is to prevent the DLD provider from inferring the exact knowledge of all sensitive data, both the outsourced sensitive data and the matched digests in network traffic. I quantify the probability for the DLD provider to infer the sensitive shingles as follows.

A polynomial-time adversary has no greater than $\frac{2^{p_f - p_d}}{n}$ *probability of correctly inferring a sensitive shingle, where* p_f *is the length of a fingerprint in bits,* p_d *is the fuzzy length, and* $n \in [2^{p_f - p_d}, 2^{p_f}]$ *is the size of the set of traffic fingerprints, assuming that the fingerprints of shingles are uniformly distributed and are equally likely to be sensitive and appear in the traffic.*

I explain my quantification in two scenarios:

1. There is a match between a sensitive fuzzy fingerprint f^* (derived from the sensitive fingerprint f) and fingerprints from the network traffic. Because the size of fuzzy set S_{f,p_d} is upper bounded by 2^{p_d} (Definition 5.3.2), there could be at most 2^{p_d} (sensitive or non-sensitive) fingerprints fuzzified into the identical f^*. Given a set (size n) of traffic fingerprints, the DLD provider expects to find K fingerprints matched to f^* where $K = \frac{n}{2^{p_f}} \times 2^{p_d}$.

   (a) If f corresponds to a sensitive shingle leaked in the traffic, i.e., f is within the K traffic fingerprints, the likelihood of correctly pinpointing f from the K fingerprints is $\frac{1}{K}$, or $\frac{2^{p_f-p_d}}{n}$. The likelihood is fare because both sensitive data and network traffic contain binary data. It is difficult to predict the subspace of the sensitive data in the entire binary space.

   (b) If the shingle form of f is not leaked in the traffic, the DLD provider cannot use the K traffic fingerprints, which match f^*, to infer f. Alternatively, the DLD provider needs to brute force f^* to get f, which is discussed as in the case of no match.

2. There is no match between sensitive and traffic fingerprints. The adversarial DLD provider needs to brute force reverse the Rabin fingerprinting computation to obtain the sensitive shingle. There are two difficulties in reversing a fingerprint: *i)* Rabin fingerprint is a one-way hash. *ii)* Multiple shingles can map to the same fingerprint. It requires to searching the complete set of possible shingles for a fingerprint and to identify the sensitive one from the set. This brute-force attack is difficult for a polynomial-time adversary, thus the success probability is not included.

In summary, the DLD provider cannot decide whether the alerts (traffic fingerprints matched f^*) contain any leak or not (case i.a or i.b). Even if it is known that there is a real leak in the network traffic, the polynomial-time DLD provider has no greater than $\frac{2^{p_f-p_d}}{n}$ probability of correctly pinpointing a sensitive shingle (case i.a).

Alert Rate I quantify the rate of alerts expected in the traffic for a sensitive data entry (the fuzzified fingerprints set of a piece of sensitive data) given the following values: the total number of fuzzified sensitive fingerprints τ, the expected traffic fingerprints set size n, fingerprint length p_f, fuzzy length p_d, partial disclosure rate $p_s \in (0, 1]$, and the expected rate α, which is the percentage of fingerprints in the sensitive data entry that appear in the network traffic. The expected alert rate R is presented in (5.4).

$$R = \frac{\alpha p_s K \tau}{n} = \frac{\alpha p_s \tau}{2^{p_f - p_d}} \tag{5.4}$$

R is used to derive threshold $\mathcal{S}_{thres}$ in the detection; $\mathcal{S}_{thres}$ should be lower than R. The overhead of my privacy-preserving approach over traditional fingerprinting data-leak detection solutions is tightly related to R and $\mathcal{S}_{thres}$, because there is an extra POSTPROCESS step in my approach after the DLD provider detects potential leaks. The less potential leaks the DLD provider reports back to the data owner, the less overhead is introduced by my privacy-preserving approach, while the less privacy is achieved since K is small.

Collisions Collisions may be due to where the legitimate traffic happens to contain the partial sensitive-data fingerprints by coincidence. The collision may increase with shorter shingles, or smaller numbers of partial fingerprints, and may decrease if additional features such as the order of fingerprints are used for detection. A previous large-scale information-retrieval study empirically demonstrated the low rate of this type of collisions in Rabin fingerprint [22], which is a desirable property suggesting low unwanted false alarms in my DLD setting. Collisions due to two distinct shingles generating the same fingerprint are proved to be low [21] and are negligible.

Space of sensitive data The space of all text-based sensitive data may be smaller than the space of all possible shingles. Yet, when including non-ASCII sensitive data (text in UTF-8 or binaries), the space of sensitive data can be significantly expanded. A restricted space limits K and can expose the fuzzified fingerprint. For instance, one may assume that a password has higher entropy than normal English shingles, thus a fuzzy fingerprint of a password rules out much space of S_{f,p_d} where normal English lives. The full space with a variety of text encodings and binaries ensures that the major space is still there shadowing the fuzzy fingerprint.

Short polynomial modulus A naive alternative to the fuzzy fingerprint mechanism is to use a shorter polynomial modulus to compute Rabin fingerprints (e.g., 16-bit instead of 32-bit). This approach increases collisions for fuzzification purpose. However, one issue of this approach is that true positive and false positives yield the same fingerprint value due to collision, which prevents the data owner from telling true positives apart from false positives. In addition, my fuzzy fingerprint approach is more flexible from the deployment perspective, as the data owner can adjust and fine-tune the privacy and accuracy in the detection without recomputing the fingerprints. In contrast, the precision is fixed in the naive shorter polynomial approach unless fingerprints are recomputed.

Limitations: I point out three major limitations of my detection approach within my threat model.

Modified data leak The underlying shingle scheme of my approach has limited power to capture heavily modified data leaks. False negatives (i.e., failure to detect data leak) may occur due to the data modification (e.g., reformatting). The new shingles/fingerprints may not resemble the original ones, and cannot be detected. As a result, a packet may evade the detection. In my experiments, I evaluate the impact of several types of data transformation in real world scenarios. The modified data-leak detection problem is a general problem for all comparison-based data-leak detection solutions. More advanced content comparison

techniques than shingles/fingerprints are needed to fully address the issue.

Dynamic sensitive data For protecting dynamically changing data such as source code or documents under constant development or keystroke data, the digests need to be continuously updated for detection, which may not be efficient or practical. I raise the issue of how to efficiently detect dynamic data with a network-based approach as an open problem to investigate by the community.

Selective fragments leak The partial disclosure scheme may result in false negatives, i.e., the leaked data may evade the detection because it is not covered by the released fingerprints. This issue illustrates the tradeoff among detection accuracy, privacy guarantee and detection efficiency. Fortunately, it is expensive for an attacker to escape the detection with partial disclosure. On one hand, Rabin fingerprint guarantees that every fingerprint has the same probability to be selected and released through its min-wise independence property. Deliberately choosing unreleased segments from sensitive data is not easy. On the other hand, even figuring out which fingerprints are not released, one needs leaking inconsecutive bytes to bypass the detection. It usually makes no sense to leak inconsecutive bytes from sensitive data. Some format, e.g., binary, may be destroyed through the leaking.

5.5 Experimental Evaluation

I implement my fuzzy fingerprint framework in Python, including packet collection, shingling, Rabin fingerprinting, as well as partial disclosure and fingerprint filter extensions. My implementation of Rabin fingerprint is based on cyclic redundancy code (CRC). I use the padding scheme mentioned in [154] to handle small inputs. In all experiments, the shingles are in 8-byte, and the fingerprints are in 32-bit (33-bit irreducible polynomials in Rabin fingerprint). I set up a networking environment in VirtualBox, and make a scenario where the sensitive data is leaked from a local network to the Internet. Multiple users' hosts (Windows 7) are put in the local network, which connect to the Internet via a gateway (Fedora). Multiple servers (HTTP, FTP, etc.) and an attacker-controlled host are put on the Internet side. The gateway dumps the network traffic and sends it to a DLD server/provider (Linux). Using the sensitive-data fingerprints defined by the users in the local network, the DLD server performs off-line data-leak detection. The speed aspect of privacy-preserving data-leak detection is another topic and I study it in [128].

In my prototype system, the DLD server detects the sensitive data within each packet on basis of a stateless filtering system. I define the sensitivity of a packet in (5.5), which is used by the DLD provider in DETECTION. It indicates the likelihood of a packet containing

sensitive data.

$$\mathcal{S}_{packet} = \frac{|\underset{p_d}{\gg}\ddot{\mathbb{S}}^* \cap \underset{p_d}{\gg}\mathbb{T}|}{min(|\mathbb{S}^*|, |\mathbb{T}|)} \times \frac{|\mathbb{S}^*|}{|\ddot{\mathbb{S}}^*|} \tag{5.5}$$

$\mathbb{T}$ is the set of all fingerprints extracted in a packet. $\mathbb{S}^*$ is the set of all sensitive fuzzy fingerprints. For each piece of sensitive data, the data owner computes $\mathbb{S}^*$ and reveals a partial set $\ddot{\mathbb{S}}^*$ ($\ddot{\mathbb{S}}^* \subseteq \mathbb{S}^*$) to the DLD provider. The operator $\underset{t}{\gg}$ indicates right shifting every fingerprint in a set by t bits, which is the implementation of a simple mask M in my protocol (Section 5.3.2). $|\mathbb{S}^*|/|\ddot{\mathbb{S}}^*|$ estimates the leaking level of $\mathbb{S}^*$ according to the revealed and tested partial set $\ddot{\mathbb{S}}^*$. When too few fuzzy fingerprints are revealed, e.g., 10%, the samples may not sufficiently describe the leaking characteristic of the traffic, and the estimation can be imprecise. For each packet, the DLD server computes $\mathcal{S}_{packet}$ ($\mathcal{S}_{packet} \in [0, 1]$). If it is higher than a threshold $\mathcal{S}_{thres} \in (0, 1)$, $\mathbb{T}$ is reported back to the data owner, and the data owner uses (5.6) to determine whether it is a real leak in POSTPROCESS.

$$\mathcal{S}_{packet} = \frac{|\mathbb{S} \cap \mathbb{T}|}{min(|\mathbb{S}|, |\mathbb{T}|)} \tag{5.6}$$

The difference between (5.5) operated by the DLD provider and (5.6) by the data owner is that the original fingerprints $\mathbb{S}$ are used in (5.6) instead of the sampled and fuzzified set $\ddot{\mathbb{S}}^*$ in (5.5), so the data owner can pinpoint the exact leaks.

The use of $\mathcal{S}_{packet}$ and $\mathcal{S}_{thres}$ for detection is important because individual shingles or fingerprints are not accurate features to represent an entire piece of sensitive data. Sensitive data can share strings with non-sensitive data, e.g., formatting strings, which results in occasionally reported sensitive fingerprints. $\mathcal{S}_{packet}$ is an accumulated score and $\mathcal{S}_{thres}$ filters out packets with occasionally discovered sensitive fingerprints.

The evaluation goal is to answer the following questions:

1. Can my solution accurately detect sensitive data leak in the traffic with low false positives (false alarms) and high true positives (real leaks)?
2. Does using partial sensitive-data fingerprints reduce the detection accuracy in my system?
3. What is the performance advantage of my *fingerprint filter* over traditional Bloom filter with SHA-1?
4. How to choose a proper fuzzy length and make a balance between the privacy need and the number of alerts?

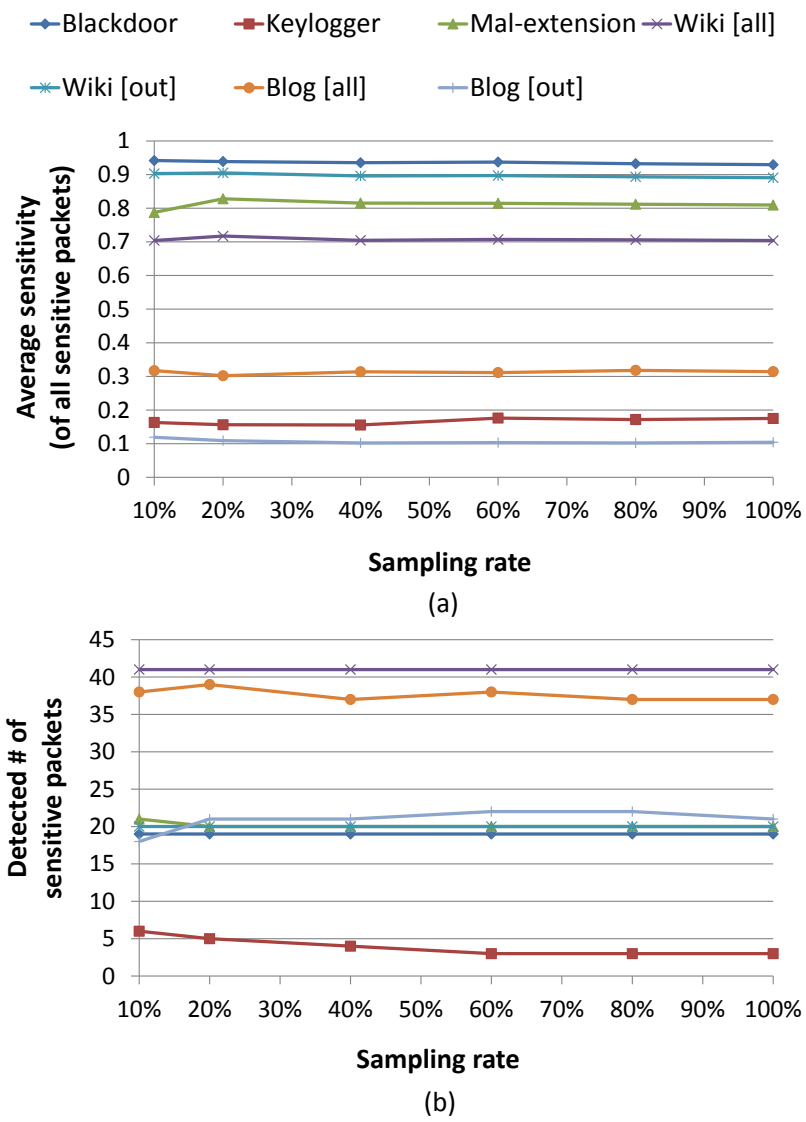


Figure 5.2: Detection accuracy comparison, in terms of (a) the averaged sensitivity and (b) the number of detected sensitive packets. X-axis is the partial disclosure rate, or the percentage of sensitive-data fingerprints revealed to the DLD server and used in the detection. [out] indicates outbound traffic only, while [all] means both outbound and inbound traffic captured and analyzed.

In the following subsection, I experimentally addressed and answered all the questions. For the first three questions, I present results based on the $\mathcal{S}_{packet}$ value calculated in (5.6). The first and second questions are answered in Section 5.5.1. The third question is discussed in Section 5.5.2. The last question is designed to understand the properties of fuzzification and partial disclosure, and it is addressed in Section 5.5.3.

Table 5.1: Mean and standard deviations of the sensitivity per packet. For Exp.1, the higher sensitivity, the better; for the other two (negative control), the lower sensitivity, the better

Dataset	Exp.1	Exp.2	Exp.3
$\mathcal{S}_{packet}$ Mean	0.952564	0.000005	0.001849
$\mathcal{S}_{packet}$ STD	0.004011	0.000133	0.002178

5.5.1 Accuracy Evaluation

I evaluate the detection accuracy in simple and complex leaking scenarios. First I test the detection rate and false positive rate in three simple experiments where the sensitive data is leaked in its original form or not leaked. Then I present accuracy evaluation on more complex leaking experiments to reproduce various real-world leaking detection scenarios.

Simple leaking scenarios. I test my prototype without partial disclosure in simple leaking scenarios, i.e., $\ddot{\mathbb{S}}^* = \mathbb{S}^*$. I generate 20,000 personal financial records as the sensitive data and store them in a text file. The data contains *person name*, *social security number*, *credit card number*, *credit card expiration date*, and *credit card CVV*.

To evaluate the accuracy of my strategy, I perform three separate experiments using the same sensitive dataset:

Exp.1 *True leak* A user leaks the entire set of sensitive data via FTP by uploading it to a remote FTP server.

Exp.2 *No leak* The non-related outbound HTTP traffic of 20 users is captured (30 minutes per user), and given to the DLD server to analyze. No sensitive data (i.e., zero true positive) should be confirmed.

Exp.3 *No leak* The Enron dataset (2.6 GB data, 150 users' 517,424 emails) as a virtual network traffic is given to the DLD server to analyze. Each virtual network packet created is based on an email in the dataset. No sensitive data (i.e., zero true positive) should be confirmed by the data owner.

The detection results are shown in Table 5.1. Among the three experiments, the first one is designed to infer true positive rate. I manually check each packet and the DLD server detects *all* 651 real sensitive packets (all of them have sensitivity values greater than 0.9). The sensitivity value is less than one, because the high-layer headers (e.g., HTTP) in a packet are not sensitive. The next two experiments are designed to estimate the false positive rate. I found that none of the packets has a sensitivity value greater than 0.05. The results indicate that my design performs as expected on plaintext.

Complex leaking scenarios. The data owner may reveal a subset of sensitive data's fingerprints to the DLD server for detection, as opposed to the entire set. I are particularly

interested in measuring the percentage of revealed fingerprints that can be detected in the traffic, assuming that fingerprints are equally likely to be leaked[2]. I reproduce four real-world scenarios where data leaks are caused by human users or software applications.

Exp.4 *Web leak:* a user posts sensitive data on wiki (`MediaWiki`) and blog (`WordPress`) pages.

Exp.5 *Backdoor leak:* a program (`Glacier`) on the user's machine (Windows 7) leaks sensitive data.

Exp.6 *Browser leak:* a malicious Firefox extension `FFsniFF` records the information in sensitive web forms, and emails the data to the attacker.

Exp.7 *Keylogging leak:* a keylogger `EZRecKb` exports intercepted keystroke values on a user's host[3]. It connects to a SMTP server on the Internet side and sends its log of keystrokes periodically.

In these four experiments, the source file of TCP/IP page on Wikipedia (24KB in text) is used as the sensitive data. The detection is performed at various partial disclosure rate. The subset of the sensitive fingerprints is selected randomly. The sensitivity threshold is $\mathcal{S}_{thres} = 0.05$.

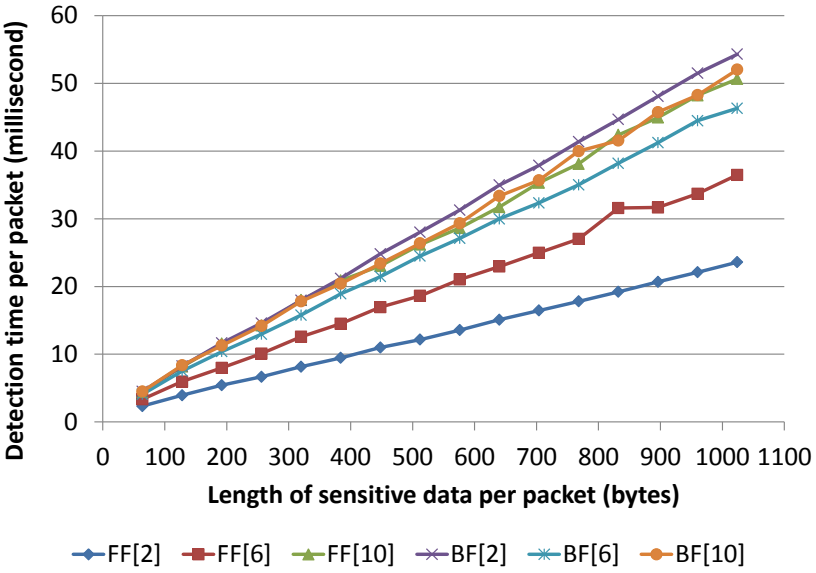
Fig. 5.2 shows the comparison of performance across various size of fingerprints used in the detection, in terms of the averaged sensitivity per packet in (a) and the number of detected sensitive packets in (b). These accuracy values reflect results of the POSTPROCESS operation.

The results show that the use of partial sensitive-data fingerprints does not much degrade the detection rate compared to the use of full sets of sensitive-data fingerprints. However, extreme small sampling rates, e.g., 10%, may not provide sufficient numbers of fingerprints to describe the leaking characteristic of the traffic. The packet sensitivity estimation ($|\mathbb{S}|/|\ddot{\mathbb{S}}|$ in (5.6)) magnifies the signal (the real sensitivity of a packet) as well as the noise produced by fingerprint sampling. The precision could be affected and drops, e.g., 6 packets with 10% vs. 3 packets with 100% for `Keylogger` in Fig. 5.2 (b).

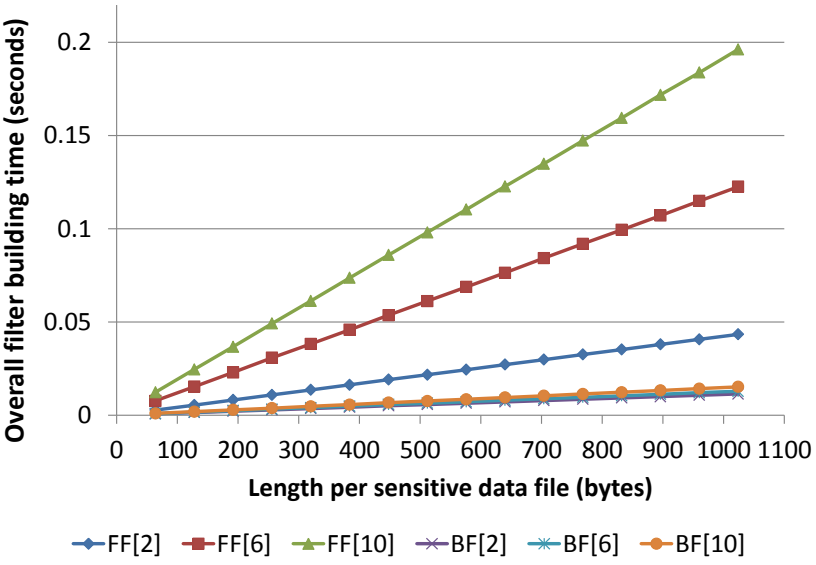
In Fig. 5.2 (a), the sensitivities of experiments vary due to different levels of modification by the leaking programs, which makes it difficult to perform detection. `WordPress` substitutes spaces with `+`'s when sending the HTTP POST request. `EZRecKb` inserts function-key as labels into the original text. Typing typos and corrections also bring in modification to the original sensitive data. In Fig. 5.2 (b), [all] results contain both outbound and inbound traffic and double the real number of sensitive packets in Blog and Wiki scenarios due to HTML fetching and displaying of the submitted data.

[2]Given the *subset independence* property, sensitive-data's fingerprints are equally likely to be selected for detection.

[3]`EZRecKb` records every key stroke and replaces the function keys with labels, such as `[left shift]`.



(a) Detection Time



(b) Filter Building Time

Figure 5.3: Overhead of filters for detecting data leaks. The detection time is averaged from 100 packets against all 10 pieces of sensitive data.

5.5.2 Runtime Comparison

My fingerprint filter implementation is based on the Bloom filter library in Python (`Pybloom`). I compare the runtime of Bloom filter provided by standard `Pybloom` (with dynamically

selected hash function from MD5, SHA-1, SHA-256, SHA-384 and SHA-512) and that of fingerprint filter with Rabin fingerprint. For Bloom filters and fingerprint filters, I test their performance with 2, 6, and 10 hash functions. I inspect 100 packets with random content against 10 pieces sensitive data at various lengths – there are a total of 1,625,600 fingerprints generated from the traffic and 76,160 pieces of fingerprints from the sensitive data.

I present the time for detection per packet in Fig. 5.3 (a). It shows that fingerprint filters run faster than Bloom filters, which is expected as Rabin fingerprint is easier to compute than MD5/SHA. The gap is not significant due to the fact that Python uses a virtualization architecture. I have the core hash computations implemented in Python C/C++ extension, but the remaining control flow and function call statements are in pure Python. The performance difference between Rabin fingerprint and MD5/SHA is largely masked by the runtime overhead spent on non-hash related operations.

In Fig. 5.3 (a), the number of hash functions used in Bloom filters does not significantly impact their runtime, because only one hash function is operated in most cases for Bloom filters. `Pybloom` automatically chooses SHA-256 for Bloom filter with 6 hash functions and SHA-384 for Bloom filter with 10 hash functions. One hash is sufficient to distinguish 32-bits fingerprints. MD5 is automatically chosen for the Bloom filter with 2 hash functions, which gives more collisions and the second hash could be involved. I speculate this is the reason why Bloom filter with 2 hashes is slower than Bloom filters with 6 or 10 hashes. All of my fingerprint filters use 32-bit Rabin fingerprint functions. The small output space requires more than one hash for a membership test, so there is more significant overhead when a fingerprint filter is equipped with more hashes (6 vs. 2 and 10 vs. 6).

The filter construction time is shown in Fig. 5.3 (b). It shares similar characteristics with the detection time. Filters with more hash functions require more time to initialize, because every hash function need to be computed. The construction of fingerprint filters, especially assigning the irreducible polynomials $p(x)$ for each Rabin fingerprint, is written in pure Python, which is significantly slower than SHA-256 and SHA-384 encapsulated using Python C/C++ extension.

5.5.3 Sizes of Fuzzy Sets vs. Fuzzy Length

The size of fuzzy set corresponds to the K value in my definition of privacy goal. The higher K is, the more difficult it is for a DLD provider to infer the original sensitive data using my fuzzy fingerprinting mechanism – the fingerprint of the sensitive data hides among its neighboring fingerprints.

I empirically evaluate the average size of the fuzzy set associated with a given fuzzy length with both Brown Corpus (text) and real-world network traffic (text & binary).

- *Brown Corpus:* The Brown University Standard Corpus of Present-Day American

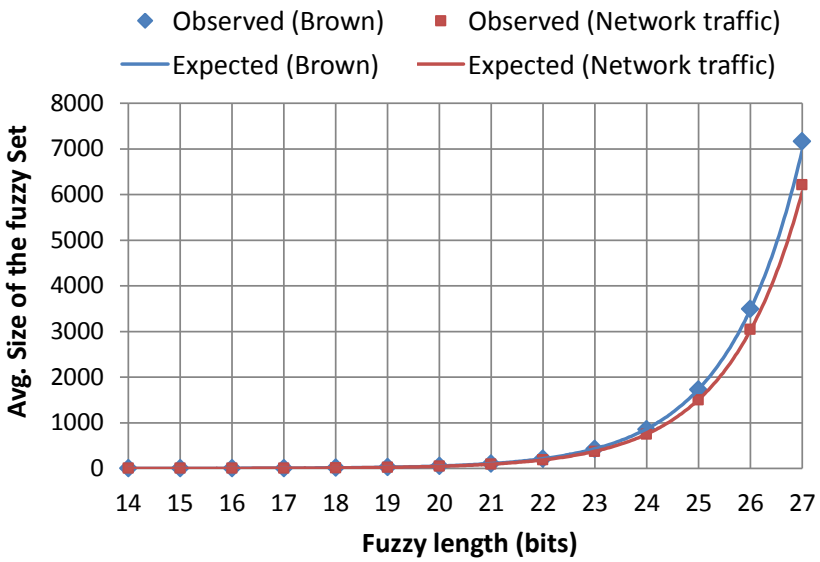


Figure 5.4: The observed and expected sizes of fuzzy sets per fingerprint. Experiments in Brown Corpus dataset (in blue) and network traffic (in red) with different fuzzy lengths.

English [62]. It contains 500 samples of English text across 15 genres, and there are 1,014,312 words in total.

- *Network traffic:* 500MB Internet traffic dump collected by us on a single host. It includes a variety of network traffic: multimedia Internet surfing (images, video, etc.), binary downloading, software and system updates, user profile synchronization, etc.

I aim to show the trend of how the fuzzy-set sizes changes with the fuzzy length, which can be used to select the optimal fuzzy length used in the algorithm. I compute 32-bit fingerprints from the datasets, and then count the number of neighbors for each fingerprint. Fig. 5.4 shows the estimated and observed sizes of fuzzy sets for fuzzy lengths in the range of $[14, 27]$ for 218,652 and 189,878 fingerprints generated from the Brown Corpus dataset and the network traffic dataset. The figure shows that the empirical results observed are very close with the expected values of the fuzzy set sizes computed based on my analysis in Section 5.4. This close fit also indicates the uniform distribution of the fingerprints.

The fuzzy set is small when the fuzzy length is small, which is due to the sparsity nature of Rabin fingerprints. Given an estimated composition of traffic content, the data owner can use the result of this experiment to determine the optimal fuzzy length. In the datasets evaluated in the experiments, for fuzzy length of 26 and 27 bits, the K values are above 1,500 and 3,000, respectively. Because the data owner can defuzzify in PostProcess very quickly, the false positives can be sifted out by the data owner. I also find that for a fixed fuzzy length the distribution of fuzzy-set sizes follows a Gaussian distribution. Different datasets may have different K size characteristics. I demonstrate the feasibility of estimating the

fuzzy set sizes, which illustrates how fuzzy fingerprintings can be used to realize a privacy goal.

Summary My detection rates in terms of the number of sensitive packets found do not decrease much with the decreasing size of disclosed fingerprint sets in Fig. 5.2, even when only 10% of the sensitive-data fingerprints are used for detection. My experiments evaluate several noisy conditions such as *noise insertion* – `MediaWiki`-based leak scenario, and *data substitution* – for the keylogger- and `WordPress`-based leak scenarios. The results indicate that my fingerprint filter can tolerate these three types of noises in the traffic to some degree. My approach works well especially in the case where consecutive data blocks are leaked (i.e., *local* data features are preserved). When the noises spread across the data and destroy the local features (e.g., replacing every space with another character), the detection rate decreases as expected. The use of shorter shingles mitigates the problem, but it may increase false positives. How to improve the noise tolerance property in those conditions remains an open problem. My fuzzy fingerprint mechanism supports the detection of data-leak at various sizes and granularities. I study the fuzzy set size and also verify the min-wise independence property of Rabin fingerprint, which are the building blocks of my fuzzy fingerprint method.

Chapter 6

Fast Detection of Transformed Data Leaks

6.1 Introduction

In this chapter, I present a high-performance detection method for addressing the following two challenges in network-based data leak detection. My method follows the basic network-based data leak detection paradigm and searches for the occurrences of *plaintext sensitive data* in the *content of network traffic* (retrieved by deep packet inspection techniques [45, 127]). It alerts users and administrators of the identified data exposure vulnerabilities.

- *Data transformation.* The exposed data in the content may be unpredictably transformed or modified by users or applications, and it may no longer be identical to the original sensitive data, e.g., insertions of metadata or formatting tags, substitutions of characters, and data truncation (partial data leak). Thus, the detection algorithm needs to recognize different kinds of sensitive data variations.
- *Scalability.* The heavy workload of data leak screening is due to two reasons.
    1. *Long sensitive data patterns.* The sensitive data (e.g., customer information, documents, source code) can be of arbitrary length (e.g., megabytes).
    2. *Large amount of content.* The detection needs to rapidly screen content (e.g., gigabytes to terabytes). Traffic scanning is more time sensitive than storage scanning, because the leak needs to be discovered before the message is transmitted.

My solution to the detection of transformed data leaks is a sequence alignment algorithm, executed on the sampled sensitive data sequence and the sampled content being inspected.

The alignment produces scores indicating the amount of sensitive data contained in the content. My alignment-based solution measures the order of n-grams. It also handles arbitrary variations of patterns without an explicit specification of all possible variation patterns. Experiments show that my alignment method substantially outperforms the set intersection method in terms of detection accuracy in a multitude of transformed data leak scenarios.

I solve the scalability issue by sampling both the sensitive data and content sequences before aligning them. I enable this procedure by providing the pair of a *comparable* sampling algorithm and a *sampling-oblivious* alignment algorithm. The comparable sampling algorithm yields constant samples of a sequence wherever the sampling starts and ends. The sampling-oblivious alignment algorithm infers the similarity between the original unsampled sequences with sophisticated traceback techniques through dynamic programming. The algorithm infers the lost information (i.e., sampled-out elements) based on the matching results of their neighboring elements. Evaluation results show that my design boosts the performance, yet only incurs a very small amount of mismatches.

Existing network traffic sampling techniques, e.g., [57], only sample the content. The problem I am dealing with differs from existing sampling problems that both sensitive data and content sequences are sampled. The alignment is performed on the sampled sequences. Therefore, the samples of similar sequences should be similar so that they can be aligned. I define a *comparable sampling* property, where the similarity of two sequences is preserved. For example, if x is a substring of y, then x' should be a substring of y', where x' and y' are sampled sequences of x and y, respectively. None of the existing sampling solutions satisfies this comparable sampling requirement. Deterministic sampling, e.g., [191], does not imply comparable sampling, either. The key to my comparable sampling is to consider the local context of a sequence while selecting items. Sample items are selected deterministically within a sliding window. The same sampled items are selected in spite of different starting/ending points of sampling procedures.

Both of my algorithms are designed to be efficiently parallelized. I parallelize my prototype on a multicore CPU and a GPU. I demonstrate the strong scalability of my design and the high performance of my prototypes. My GPU-accelerated implementation achieves nearly 50 times of speedup over the CPU version. My prototype reaches 400Mbps analysis throughput. This performance potentially supports the rapid security scanning of storage and communication required by a sizable organization.

In this chapter, I formalize the description and analysis of my comparable sampling algorithm and sampling-oblivious alignment algorithm in Section 6.3 and Section 6.4. I detail the design of both algorithms and conduct extensive experiments to systematically understand how sensitive my system is in response to data transformation in various degrees (Section 6.5) and how scalable the system is detecting different sizes of the sensitive data (Section 6.6).

6.2 Models and Overview

In my data leak detection model, I analyze two types of sequences: sensitive data sequence and content sequence.

- *Content sequence* is the sequence to be examined for leaks. The content may be data extracted from file systems on personal computers, workstations, and servers; or payloads extracted from supervised network channels (details are discussed below).
- *Sensitive data sequence* contains the information (e.g., customers' records, proprietary documents) that needs to be protected and cannot be exposed to unauthorized parties. The sensitive data sequences are known to the analysis system.

In this chapter, I focus on detecting inadvertent data leaks (Case I in Section 5.2.1), and I assume the content in file system or network traffic (over supervised network channels) is available to the inspection system. A supervised network channel could be an unencrypted channel or an encrypted channel where the content in it can be extracted and checked by an authority. Such a channel is widely used for advanced NIDS where MITM (man-in-the-middle) SSL sessions are established instead of normal SSL sessions [101]. I do not aim at detecting stealthy data leaks that an attacker encrypts the sensitive data secretly before leaking it. Preventing intentional or malicious data leak, especially encrypted leaks, requires different approaches and remains an active research problem [15].

6.2.1 Technical Challenges

High detection specificity. In my data-leak detection model, high specificity refers to the ability to distinguish true leaks from coincidental matches. *Coincidental matches* are false positives, which may lead to false alarms. Existing set-based detection is orderless, where the order of matched shingles (n-grams) is ignored. Orderless detection may generate coincidental matches, and thus having a lower accuracy of the detection. In comparison, my alignment-based method has high specificity. For example, a detection system can use 3-grams to represent the sensitive data.

Sensitive data	`abcdefg`
3-grams	`abc`, `bcd`, `cde`, `def`, `efg`

Then, consider the content streams 1 and 2 below. Stream 1 contains a true leak, whereas stream 2 does not.

Content stream 1	`....abcdefg...`
Content stream 2	`....efg...cde...abc...`

However, set intersection between 3-grams of the sensitive data and the 3-grams of content stream 2 results in a significant number of matching 3-grams (`efg`, `cde`, and `abc`), even though they are out of order compared to the sensitive data pattern. This problem is eliminated in alignment, i.e., the content stream 2 receives a low sensitivity score when aligned against the sensitive data.

Pervasive and localized modification. Sensitive data could be modified before it is leaked out. The modification can occur throughout a sequence (pervasive modification). The modification can also only affect a local region (local modification). I describe some modification examples:

- Character replacement, e.g., `WordPress` replaces every space character with a `+` in HTTP POST requests.
- String insertion, e.g., HTML tags inserted throughout a document for formatting or embedding objects.
- Data truncation or partial data leak, e.g., one page of a two-page sensitive document is transmitted.

6.2.2 Discussions on Existing Solutions

None of existing techniques satisfies the two requirements. I discuss *automata-based string matching* and *set intersection on n-grams* below, the two most important techinques that could be used to detect long and inexact data leak.

Directly applying automata-based string matching, e.g., [4, 116, 208], to data leak detection is inappropriate and inefficient, because automata are not designed to support unpredictable and arbitrary pattern variations (Section 2.2.3). In data leak detection scenarios, the transformation of leaked data (in the description of regular expression) is unknown to the detection method. Creating comprehensive automata models covering all possible variations of a pattern is infeasible, which leads to $O(2^n)$ space complexity (for deterministic finite automata) or $O(2^n)$ time complexity (for nondeterministic finite automata) where n is the number of automaton states. Therefore, automata approaches cannot be used for detecting long and transformed data leaks.

Set intersection on n-grams is widely used in commercial leak detection tools. However, set intersection is *orderless*, i.e., the ordering of shared n-grams is not analyzed. Thus, set-based detection generates undesirable false alerts, especially when n is set to a small value to tolerant data transformation. In addition, set intersection cannot effectively characterize the scenario when partial data is leaked, which results in false negatives. Therefore, none of the existing techniques is adequate for detecting transformed data leaks.

6.2.3 Overview of My Approach

My work presents an efficient sequence comparison technique needed for analyzing a large amount of content for sensitive data exposure. My detection approach consists of a comparable sampling algorithm and a sampling oblivious alignment algorithm. The pair of algorithms computes a quantitative similarity score between the sensitive data and the content. Local alignment – as opposed to global alignment [151] – is used to identify similar sequence segments. The design enables the detection of partial data leaks.

My detection runs on continuous sequences of n bytes (n-grams). n-grams are obtained from the content and sensitive data, respectively. Local alignment is performed between the two (sampled) sequences to compute their similarity. The purpose of my comparable sampling operation is to enhance the analysis throughput. I discuss the tradeoff between security and performance related to sampling in my evaluation sections. Finally, I report the content that bears higher-than-threshold similarity with respect to sensitive patterns. Given a threshold T, content with a greater-than-T sensitivity is reported as a leak.

6.3 Comparable Sampling

In this section, I define the sampling requirement needed in data leak detection. Then I present my solution and its analysis.

6.3.1 Definitions

One great challenge in aligning sampled sequences is that the sensitive data segment can be exposed at an arbitrary position in a network traffic stream or a file system. The sampled sequence should be deterministic despite the starting and ending points of the sequence to be sampled. Moreover, the leaked sensitive data could be inexact but similar to the original string due to unpredictable transformations. I first define substring and subsequence relations in Definition 6.3.1 and Definition 6.3.2. Then I define the capability of giving comparable results from similar strings in Definition 6.3.3.

Definition 6.3.1. *(Substring) a substring is a consecutive segment of the original string.*

If x is a substring of y, one can find a prefix string (denoted by y_p) and a suffix string (denoted by y_s) of y, so that y equals to the concatenation of y_p, x, and y_s. y_p and y_s could be empty.

Definition 6.3.2. *(Subsequence) subsequence is a generalization of substring that a subsequence does not require its items to be consecutive in the original string.*

One can generate a subsequence of a string by removing items from the original string and keeping the order of the remaining items. The removed items can be denoted as gaps in the subsequence, e.g., `lo-e` is a subsequence of `flower` (`-` indicates a gap).

Definition 6.3.3. *(Comparable sampling) Given a string x and another string y that x is similar to a substring of y according to a similarity measure M, a comparable sampling on x and y yields two subsequences x' (the sample of x) and y' (the sample of y), so that x' is similar to a substring of y' according to M.*

If I restrict the similarity measure M in Definition 6.3.3 to *identical relation*, I get a specific instance of comparable sampling in Definition 6.3.4.

Definition 6.3.4. *(Subsequence-preserving sampling) Given x as a substring of y, a subsequence-preserving sampling on x and y yields two subsequences x' (the sample of x) and y' (the sample of y), so that x' is a substring of y'.*

Because a subsequence-preserving sampling procedure is a restricted comparable sampling, so the subsequence-preserving sampling is deterministic, i.e., the same input always yields the same output. The vice versa may not be true.

In Example 1 with two sequences of integers, I illustrate the differences between a comparable sampling algorithm and a random sampling method, where a biased coin flipping at each position decides whether to sample or not. The input is a pair of two similar sequences. There is one modification (9 to 8), two deletions (7) and (3), and suffix padding (1, 4) in the second sequence. Local patterns are preserved in a comparable sampling method, whereas the random sampling does not. The local patterns can then be digested by my sampling-oblivious alignment algorithm to infer the similarity between the two original input sequences.

Example 1. `Comparable sampling.`

```
Inputs:
1 1 9 4 5 7 3 5 9 7 6 6 3 3 7 1 6
1 1 9 4 5 7 3 5 8 6 6 3 7 1 6 1 4

Comparable sampling may give:
1 1 - 4 - - 3 5 - - - - 3 3 - 1 -
1 1 - 4 - - 3 - - 6 - 3 - 1 - 1 4

Random sampling may give:
1 - - 4 - - 3 5 - 7 - 6 - - 7 1 -
- 1 9 - 5 - - 5 - 6 - 3 7 - 6 1 -
```

Table 6.1: Illustration of my sampling procedure.

Step	w	m_c	m_p	e_n	e_o	Sampled list
0	[1, 5, 1, 9, 8, 5]	1, 1, 5	N/A	N/A	N/A	<-, -, -, -, -, -, -, -, -, ->
1	[5, 1, 9, 8, 5, 3]	1, 3, 5	1, 1, 5	3	1	<1, -, -, -, -, -, -, -, -, ->
2	[1, 9, 8, 5, 3, 2]	1, 2, 3	1, 3, 5	2	5	<1, -, -, -, -, -, -, 2, -, ->
3	[9, 8, 5, 3, 2, 4]	2, 3, 4	1, 2, 3	4	1	<1, -, 1, -, -, -, -, 2, -, ->
4	[8, 5, 3, 2, 4, 8]	2, 3, 4	2, 3, 4	N/A	N/A	<1, -, 1, -, -, -, -, 2, -, ->

6.3.2 My Sampling Algorithm

I present my comparable sampling algorithm. The advantage of my algorithm is its **context-aware selection**, i.e., the selection decision of an item depends on how it compares with its surrounding items according to a selection function. As a result, the sampling algorithm is *deterministic* and *subsequence-preserving*.

My comparable sampling algorithm takes in $\mathcal{S}$, an input list of items (preprocessed n-grams of sensitive data or content[1]), and outputs $\mathcal{T}$, a sampled list of the same length; the sampled list contains null values, which correspond to items that are not selected. The null regions in $\mathcal{T}$ can be aggregated, and $\mathcal{T}$ can be turned into a *compact representation* $\mathcal{L}$. Each item in $\mathcal{L}$ contains the value of the sampled item and the length of the null region between the current sampled item and the preceding one.

$\mathcal{T}$ is initialized as an empty list, i.e., a list of null items. The algorithm runs a small sliding window w on $\mathcal{S}$. w is initialized with the first $|w|$ items in $\mathcal{S}$ (line 2 in Algorithm 1). The algorithm then utilizes a selection function to decide what items in w should be selected for $\mathcal{T}$. The selection decision is made based on not only the value of that item, but also the values of its neighboring items in w. Therefore, unlike a random sampling method where a selection decision is stochastic, my method satisfies the subsequence-preserving and comparable sampling requirements.

In Algorithm 1, without loss of generality, I describe my sampling method with a specific selection function $f = \min(w, N)$. f takes in an array w and returns the N smallest items (integers) in w. f is *deterministic*, and it *unbiasedly* selects items when items (n-grams) are preprocessed with the min-wise independent Rabin's fingerprint [23]. f can be replaced by other functions that are also min-wise independent. The selection results at each sliding window position determine what items are chosen for the sampled list. The parameters N and $|w|$ determine the sampling rate. `collectionDiff(A,B)` in lines 10 and 11 outputs the collection of all items of collection `A` that are not in collection `B`. The operation is similar to the set difference, except that it works on collections and does not eliminate duplicates.

$\mathcal{T}$ output by Algorithm 1 takes the same space as $\mathcal{S}$ does. Null items can be combined, and

[1] I preprocess n-grams with Rabin's fingerprint to meet the min-wise independent requirement of selection function f described next. Each item in $\mathcal{S}$ is a fingerprint/hash value (integer) of an n-gram.

Algorithm 1 A subsequence-preserving sampling algorithm.

Require: an array $\mathcal{S}$ of items, a size $|w|$ for a sliding window w, a selection function $f(w, N)$ that selects N smallest items from a window w, i.e., $f = \min(w, N)$
Ensure: a sampled array $\mathcal{T}$

initialize $\mathcal{T}$ as an empty array of size $|\mathcal{S}|$
$w \leftarrow \text{read}(\mathcal{S}, |w|)$
let $w.head$ and $w.tail$ be indices in $\mathcal{S}$ corresponding to the higher-indexed end and lower-indexed end of w, respectively
collection $m_c \leftarrow \min(w, N)$
while w is within the boundary of $\mathcal{S}$ **do**
    $m_p \leftarrow m_c$
    move w toward high index by 1
    $m_c \leftarrow \min(w, N)$
    if $m_c \neq m_p$ **then**
        item $e_n \leftarrow \text{collectionDiff}(m_c, m_p)$
        item $e_o \leftarrow \text{collectionDiff}(m_p, m_c)$
        if $e_n < e_o$ **then**
            write value e_n to $\mathcal{T}$ at $w.head$'s position
        else
            write value e_o to $\mathcal{T}$ at $w.tail$'s position
        end if
    end if
end while

$\mathcal{T}$ is turned into a *compact representation* $\mathcal{L}$, which is consumed by my sampling-oblivious alignment algorithm in the next phase.

I show how my sampling algorithm works in Table 6.1. I set my sampling procedure with a sliding window of size 6 (i.e., $|w| = 6$) and $N = 3$. The input sequence is `1,5,1,9,8,5,3,2,4,8`. The initial sliding window $w =$ `[1,5,1,9,8,5]` and collection $m_c =$ `{1,1,5}`.

Sampling Algorithm Analysis

My sampling algorithm is deterministic, i.e., given a fixed selection function f: same inputs yield the same sampled string. However, deterministic sampling (e.g., [191]) does not necessarily imply subsequence preserving. One can prove using a counterexample. Consider a sampling method that selects the first of every 10 items from a sequence, e.g., 1-st, 11-th, 21-st, ... It is deterministic, but it does not satisfy the subsequence-preserving requirement. Some sampling methods such as coresets [2, 54] do not imply determinism.

My sampling algorithm is not only deterministic, but also subsequence-preserving as presented in Theorem 6.3.1.

Theorem 6.3.1. *Algorithm 1 (denoted by Ψ) is subsequence-preserving. Given two strings x and y, where x is a substring of y, then $\Psi(x)$ is a substring of $\Psi(y)$.*

of Theorem 6.3.1. Let $L[m:n]$ denote the substring of L starting from the m-th item and ending at the n-th item. Consider strings L_1 and L_2 and their sampled sequences S_1 and S_2, respectively. I prove that the theorem holds in four cases below.

Case 1: L_2 equals to L_1. Because my comparable sampling algorithm is deterministic, the same string yields the same sampled sequence. Thus, the theorem holds.

Case 2: L_2 is a prefix of L_1. The sampling of L_1 can be split into two phases.

Phase 1 The head of the sliding moves within $L_1[size(win) : size(L_2)]$, i.e., from the start of L_1 to the exact position in L_1 where L_2 ends. Since L_2 is a prefix of L_1, and the window only moves within the scope of the prefix, the sample of L_1 generated in this subprocess is the same as S_2, the final sample of L_2.

Phase 2 The head of the sliding window moves within $L_1[size(L_2)+1 : size(L_2)+size(win)]$. The tail of the sample window sweeps $L_1[size(L_2)-size(win)+1 : size(L_2)]$ and yields zero or more sampled items on $S_1[size(L_2)-size(win)+1 : size(L_2)]$.

$S_1[1 : size(L_2)-size(win)]$ is solely generated in **Phase 1**. Thus, it is the same as $S_2[1 : size(L_2)-size(win)]$. In **Phase 2**, I know that $S_1[size(L_2)-size(win)+1 : size(L_2)]$ contains zero or more sample items than $S_2[size(L_2)-size(win)+1 : size(L_2)]$. Thus, $S_2[size(L_2)-size(win)+1 : size(L_2)]$ is a substring of $S_1[size(L_2)-size(win)+1 : size(L_2)]$. The theorem holds.

Case 3: L_2 is a suffix of L_1. The proof is similar to **Case 2**. The sampling of L_1 can be split into two phases.

Phase 1 The tail of the sliding window moves within $L_1[size(L_1)-size(L_2)+1 : size(L_1)-size(win)]$. The generated sampled sequence is the same as S_2, which is the final sample of L_2.

Phase 2 The tail of the sliding window moves within $L_1[size(L_1)-size(L_2)-size(win)+1 : size(L_1)-size(L_2)]$. The head of the window sweeps $L_1[size(L_1)-size(L_2)+1 : size(L_1)-size(L_2)+size(win)]$ and yields zero or more sampled items on $L_1[size(L_1)-size(L_2)+1 : size(L_1)-size(L_2)+size(win)]$.

$S_1[size(L_1)-size(L_2)+size(win)+1 : size(L_1)-size(L_2)]$ is the same as $S_2[size(L_1)-size(L_2)+size(win)+1 : size(L_1)-size(L_2)]$. In addition, $S_2[size(L_1)-size(L_2)+1 : size(L_1)-size(L_2)+size(win)]$ is a substring of $S_1[size(L_1)-size(L_2)+1 : size(L_1)-size(L_2)+size(win)]$. Thus, the theorem holds.

Case 4: All others. This case is when L_2 is a substring of L_1, but not a prefix or suffix, i.e., $L_2[1 : size(L_2)] = L_1[m : n]$. I align L_1 and L_2 and cut the two strings at a position where they are aligned. Denote the position in L_2 by k. I obtain $L_2[1 : k]$ as a suffix of $L_1[m : m + k]$ and $L_2[k + 1 : size(L_2)]$ as a prefix of $L_1[m + k + 1 : n]$. Based on the proofs in **Case 2** and **Case 3**, I conclude that $S_2[1 : k]$ is a substring of $S_1[m : m + k]$, and $S_2[k + 1 : size(L_2)]$ is a substring of $S_1[m + k + 1 : n]$. Thus, S_2 is a substring of S_1.

In summary, Theorem 6.3.1 holds in all cases. □

My algorithm is unbiased, meaning that it gives an equal probability for every unit in the string to be selected. To achieve bias-free property, I hash inputs using a min-wise independent function, namely Rabin's fingerprint [154]. It guarantees that the smallest N items come equally from any items in the original string. This hashing is performed in PREPROCESSING operation in my prototypes.

The complexity of sampling using the $\min(w, N)$ selection function is $O(n \log |w|)$, or $O(n)$ where n is the size of the input, $|w|$ is the size of the window, The factor $O(\log |w|)$ comes from maintaining the smallest N items within the window w.

Sampling rate $\alpha \in [\frac{N}{|w|}, 1]$ approximates $\frac{N}{|w|}$ for random inputs, where $|w|$ is the size of the sliding window, and N is the number of items selected within the sliding window. For arbitrary inputs, the actual sampling rate depends on the characteristics of the input space and the selection function used. The sampling rate in my evaluations on common Internet traffic is around $1.2\frac{N}{|w|}$.

Sufficient number of items need to be sampled from sequences in order to warrant an accurate detection. My empirical result in Section 6.5.2 shows that with 0.25 sampling rate my alignment method can detect as short as 32-byte-long sensitive data segments.

6.4 Alignment Algorithm

In this section, I describe the requirements for a sample-based alignment algorithm and present my solution.

6.4.1 Requirements and Overview

I design a specialized alignment algorithm that runs on compact sampled sequences $\mathcal{L}^a$ and $\mathcal{L}^b$ to infer the similarity between the original sensitive data sequence $\mathcal{S}^a$ and the original content sequence $\mathcal{S}^b$. It needs to satisfy the requirement of **sampling oblivion**, i.e., the result of a sampling-oblivious alignment on sampled sequences $\mathcal{L}^a$ and $\mathcal{L}^b$ should be consistent with the alignment result on the original $\mathcal{S}^a$ and $\mathcal{S}^b$.

Conventional alignment may underestimate the similarity between two substrings of the sampled lists, causing misalignment. Regular local alignment without the sampling oblivion property may give inaccurate alignment on sampled sequences as illustrated in Example 2.

Example 2. *Sampling-oblivious alignment vs. regular local alignment*

```
Original lists:
      562798385743254639782436 6
      5627983966432546395

Sampled sequences need to be aligned as:
      --2---3-5---2---3-7-2-3--
      --2---3-6---2---3--

However, regular local alignment may give:
      23523723
         23623
```

Because values of unselected items are unknown to the alignment, the decision of match or mismatch cannot be made solely on them during the alignment. I observe that leaked data region is usually consecutive, e.g., spans at least dozens of bytes. Thus, my algorithm achieves sampling oblivion by inferring the similarity between null regions (consecutive sampled-out elements) and counts that similarity in the overall comparison outcomes between the two sampled sequences. The inference is based on the comparison outcomes between items surrounding null regions and sizes of null regions. For example, given two sampled sequences `a-b` and `A-B`, if `a == A` and `b == B`, then the two values in the positions of the null regions are likely to match as well. In case of mismatch surrounding the null region, penalty is applied. My experimental results confirm that this inference mechanism is effective.

I develop my alignment algorithm using dynamic programming. A string alignment problem is divided into three prefix alignment subproblems: the current two items (from two sequences) are aligned with each other, or one of them is aligned with a gap. In my algorithm, not only the sampled items are compared, but also comparison outcomes between null regions are inferred based on their non-null neighboring values and their sizes/lengths. The comparison results include *match*, *mismatch* and *gap*, and they are rewarded (match) or penalized (mismatch or gap) differently for sampled items or null regions according to a weight function $f_w()$.

My alignment runs on sampled out elements. I introduce *i)* extra fields of scoring matrix cells in dynamic programming, *ii)* extra steps in recurrence relation for bookkeeping the null region information, and *iii)* a complex weight function estimating similarities between null regions.

Security Advantages of Alignment. There are three major advantages of my alignment-based method for detecting data leaks: order-aware comparison, high tolerance to pattern

variations, and the capability of partial leak detection. All features contribute to high detection accuracy.

- *Order-aware comparison.* Existing data leak filtering methods based on set intersection are orderless. An orderless comparison brings undesirable false alarms due to coincidental matches, as explained in Section 6.2. In comparison, alignment is order-aware, which significantly reduces the number of false positives.
- *High tolerance to pattern variations.* The optimal alignment between the sensitive data sequence and content sequence ensures high accuracy for data leak detection. The alignment-based detection tolerates pattern variations in the comparison, thus can handle transformed data leaks. The types of data transformation in my model include localized and pervasive modifications such as insertion, deletion, and substitution, but exclude strong encryption.
- *Capability of detecting partial leaks.* Partial data leak is an extreme case of truncation in transformation. In set-intersection methods, the size of sensitive data and that of the inspected content are usually used to diminish the score of coincidental matches, which incurs false negatives when only partial sensitive data is leaked. Local alignment searches for similar substrings in two sequences, thus it can detect a partial data leak.

6.4.2 Recurrence Relation

I present the recurrence relation of my dynamic program alignment algorithm in Algorithm 2. For the i-th item $\mathcal{L}_i$ in a sampled sequence $\mathcal{L}$ (the compact form), the field $\mathcal{L}_i.value$ denotes the value of the item and a new field $\mathcal{L}_i.span$ denotes the size of null region between that item and the preceding non-null item. My local alignment algorithm takes in two sampled sequences $\mathcal{L}^a$ and $\mathcal{L}^b$, computes a non-negative score matrix H of size $|\mathcal{L}^a|$-by-$|\mathcal{L}^b|$, and returns the maximum alignment score with respect to a weight function. Each cell $H(i,j)$ has a score field $H(i,j).score$ and two extra fields recording sizes of neighboring null regions, namely $null_{row}$ and $null_{col}$.

The intermediate solutions are stored in matrix H. For each subproblem, three previous subproblems are investigated: *i)* aligning the current elements without a gap, which leads to a *match* or *mismatch*, *ii)* aligning the current element in $\mathcal{L}^a$ with a gap, and *iii)* aligning the current element in $\mathcal{L}^b$ with a gap. A cell candidate h is generated for each situation; its score $h.score$ is computed via the weight function f_w (lines 1 to 3 in Algorithm 2). The other two fields, $null_{row}$ and $null_{col}$, are updated for each candidate cell (lines 4 to 9). This update may utilize the null region value stored in the *span* field of an item. All three cell candidates h^{up}, h^{left}, and h^{dia} are prepared. The cell candidate having the highest score is chosen as $H(i,j)$, and the score is stored in $H(i,j).score$.

Algorithm 2 Recurrence relation in dynamic programming.

Require: A weight function f_w, visited cells in H matrix that are adjacent to $H(i,j)$: $H(i-1,j-1), H(i,j-1)$, and $H(i-1,j)$, and the i-th and j-th items $\mathcal{L}_i^a, \mathcal{L}_j^b$ in two sampled sequences $\mathcal{L}^a$ and $\mathcal{L}^b$, respectively.

Ensure: $H(i,j)$

1: $h^{up}.score \leftarrow f_w(\mathcal{L}_i^a, \text{-}, H(i-1,j))$

2: $h^{left}.score \leftarrow f_w(\text{-}, \mathcal{L}_j^b, H(i,j-1))$

3: $h^{dia}.score \leftarrow f_w(\mathcal{L}_i^a, \mathcal{L}_j^b, H(i-1,j-1))$

4: $h^{up}.null_{row} \leftarrow 0$

5: $h^{up}.null_{col} \leftarrow 0$

6: $h^{left}.null_{row} \leftarrow 0$

7: $h^{left}.null_{col} \leftarrow 0$

8: $h^{dia}.null_{row} \leftarrow \begin{cases} 0, & \text{if } \mathcal{L}_i^a = \mathcal{L}_j^b \\ H(i-1,j).null_{row} & \\ \quad + \mathcal{L}_i^a.span + 1, & \text{else} \end{cases}$

9: $h^{dia}.null_{col} \leftarrow \begin{cases} 0, & \text{if } \mathcal{L}_i^a = \mathcal{L}_j^b \\ H(i,j-1).null_{col} & \\ \quad + \mathcal{L}_j^b.span + 1, & \text{else} \end{cases}$

10: $H(i,j) \leftarrow \arg\max_{h.score} \begin{cases} h^{up} \\ h^{left} \\ h^{dia} \end{cases}$

11: $H(i,j).score \leftarrow \max \begin{cases} 0 \\ H(i,j).score \end{cases}$

6.4.3 Weight Function

A weight function computes the score for a specific alignment configuration. My weight function $f_w()$ takes three inputs: the two items being aligned (e.g., $\mathcal{L}_i^a$ from sensitive data sequence and $\mathcal{L}_j^b$ from content sequence) and a reference cell c (one of the three visited adjacent cells $H(i-1,j-1), H(i,j-1)$, or $H(i-1,j)$). It then outputs a score of an alignment configuration. One of $\mathcal{L}_i^a$ and $\mathcal{L}_j^b$ may be a gap ($-$) in the alignment. The computation is based on the penalty given to mismatch and gap conditions and reward given to match conditions. My weight function differs from the one in Smith-Waterman algorithm [174] in its ability to infer comparison outcomes for null regions. This inference is done efficiently accordingly to the values of their adjacent non-null neighboring items. The inference may trace back to multiple preceding non-null items up to a constant factor.

In my $f_w()$, r is the reward for a single unit match, m is the penalty for a mismatch, and g is the penalty for a single unit aligned with a gap. As presented in Section 6.4.2, the field *value* is the value of a sampled item (e.g., $x.value$ or $y.value$ in $f_w()$ below), and the field *span* stores the length of the null region preceding the item. For the input cell c, the fields n_r (short for $null_{row}$) and n_c (short for $null_{col}$) record the size of the accumulated null regions

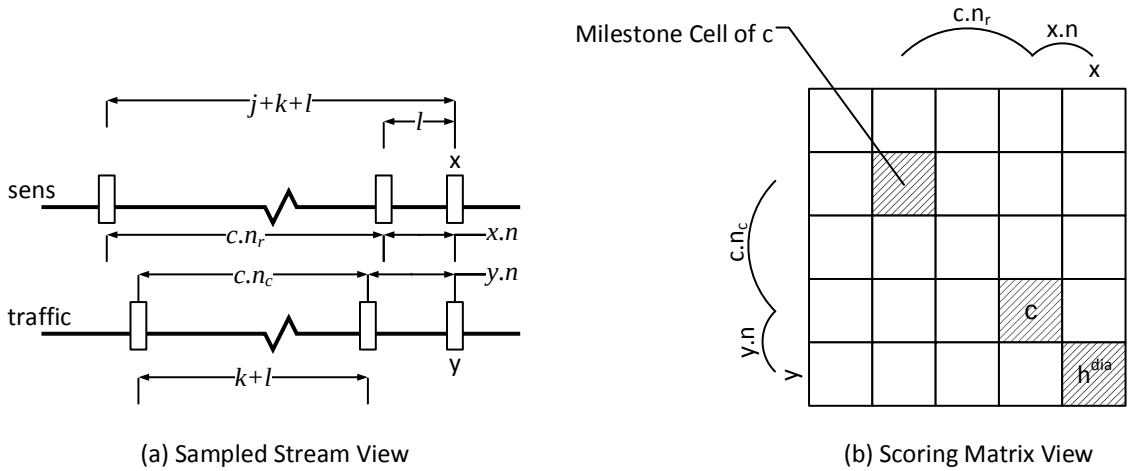


Figure 6.1: Illustration of the notation used in the weight function $f_w()$, the match case (i. e., $x.value = y.value$) in the alignment view (a) and matrix view (b). The milestone cell in (b) is for inference due to sampling.

in terms of row and column from the nearest milestone cell (explained next in my traceback strategy) to the current cell. $\text{diff}(m, n) = |m - n|$. Values p, q, l, k, and j serve as weight coefficients in my penalty and reward mechanisms. I detail my weight function $f_w()$ below and illustrate the lengths l, k and j for the match case in Figure 6.1.

1. (Gap) h^{up}

$$f_w(x, -, c) = c.score + m \times p + g \times q$$
where
$$p = \min(c.n_r + x.span + 1, c.n_c)$$
$$q = \text{diff}(c.n_r + x.span + 1, c.n_c)$$

2. (Gap) h^{left}

$$f_w(-, y, c) = c.score + m \times p + g \times q$$
where
$$p = \min(c.n_r, c.n_c + y.span + 1)$$
$$q = \text{diff}(c.n_r, c.n_c + y.span + 1)$$

3. (Mismatch) $h^{dia} | x.value \neq y.value$

$$f_w(x, y, c) = cell.score$$

4. (Match) $h^{dia}|x.value = y.value$

$$\begin{aligned} f_w(x, y, c) = & \, cell.score \\ & + r \times l \\ & + m \times k \\ & + g \times j \end{aligned}$$

where

$$\begin{aligned} l &= \min(x.span, y.span) + 1, \\ k &= \min(c.n_r, c.n_c) - l, \\ j &= \text{diff}(c.n_r, c.n_c) + \text{diff}(x.span, y.span) + l \end{aligned}$$

Traceback in my weight function is for inferring matching outcomes based on preceding null regions, including the adjacent one. My traceback operation is efficient. It extends to a constant number of preceding null regions. To achieve this property, I define a special type of cells (referred to as milestone cells) in matrix H with zero $null_{row}$ and $null_{col}$ fields. These milestone cells mark the boundary for the traceback inference; the subproblems (upper left cells) of a milestone cell are not visited. A milestone cell is introduced in either match or gap cases in f_w.

6.4.4 Algorithm Analysis

The complexity of my alignment algorithm is $O(|\mathcal{L}^a||\mathcal{L}^b|)$, where $|\mathcal{L}^a|$ and $|\mathcal{L}^b|$ are lengths of compact representations of the two sampled sequences. The alignment complexity for a single piece of sensitive data of size l is the same as that of a set of shorter pieces with a total size l, as the total amounts of matrix cells to compute are the same.

In a real-world deployment, the overall sensitive data sequence $\mathcal{S}^a$ is usually close to a fixed length, and more attention is commonly paid to the length of the content sequence $\mathcal{S}^b$. In this case, the complexity of my alignment is $O(|\mathcal{L}^b|)$ where $\mathcal{L}^b$ is the sampled list of $\mathcal{S}^b$. I experimentally evaluate the throughput of my prototype in Section 6.6, which confirms the $O(|\mathcal{L}^b|)$ complexity in the analysis.

The correctness of my alignment is ensured by dynamic programming and the recurrence relation among the subproblems of string alignment. The preciseness of similarity inference between sampled-out elements is achieved by my specifically designed weight function. Empirical results show that the alignment of sampled sequences $\mathcal{L}^a$ and $\mathcal{L}^b$ is very close to the alignment of original sequences $\mathcal{S}^a$ and $\mathcal{S}^b$, confirming the sampling oblivion property.

My alignment of two sampled sequences achieves a speedup in the order of $O(\alpha^2)$, where $\alpha \in (0, 1)$ is the sampling rate. There is a constant damping factor due to the overhead introduced by sampling. The expected value is 0.33 because of the extra two fields, besides

Table 6.2: Datasets in accuracy & scalability experiments

Dataset	Size	Details
A. Enron [30]	2.6 GB	517,424 email (with full headers and bodies) of 150 users
B. Source-code	3.8 MB	288 source files in projects `tar`, `net-tools`, `gzip`, `procps`, and `rsync`
C. HTTP requests	12 MB	HTTP requests of 20 users (30-minute Internet activities recorded for each user)
D. MiscNet	500MB	Miscellaneous web traffic containing text and multimedia content

the score field, to maintain for each cell in H. I experimentally verify the damping factor in my evaluation.

Permutation-based data transformation (e.g., position swaps) affects the alignment precision and reduces the overall detection accuracy.

6.5 Evaluation on Detection Accuracy

I extensively evaluate the accuracy of my solution with several types of datasets under a multitude of real-world data leak scenarios. My experiments in this section aim to answer the following questions.

1. Can my method detect leaks with pervasive modifications, e.g., character substitution throughout a sensitive document?
2. Can my method detect localized modifications, especially partial data leaks?
3. How specific is my detection, that is, the evaluation of false positives?
4. How does my method compare to the state-of-the-art collection intersection method in terms of detection accuracy?

6.5.1 Implementation and Experiment Setup

I implement a single-threaded prototype (referred to as *AlignDLD* system) and a collection intersection method (referred to as *Coll-Inter* system), which is a baseline. Both systems are written in `C++`, compiled using `g++` 4.7.1 with flag `-O3`. I also provide two parallel versions of my prototype in Section 6.6 for performance demonstration.

Table 6.3: Semantics of true/false positives/negatives.

	True Leak	No Leak
Leak detected	*TP*	*FP*
No leak detected	*FN*	*TN*

- *AlignDLD:* my sample-and-align data leak detection method with sampling parameters $N = 10$ and $|w| = 100$. 3-grams and 32-bit Rabin's fingerprints[2] are used.
- *Coll-Inter:* a data leak detection system based on collection intersection[3], which is widely adopted by commercial tools such as GlobalVelocity [75] and GoCloudDLP [76]. 8-grams and 64-bit Rabin's fingerprints are used, which is standard with collection intersection.

I use four datasets (Table 6.2) in my experiments. *A. Enron* and *B. Source-code* are used either as the sensitive data or the content to be inspected. *C. Outbound HTTP requests* and *D. MiscNet* are used as the content. Detailed usages of these datasets are specified in each experiment.

I report the detection rate in Equation (6.1) with respect to a certain threshold for both *AlignDLD* and *Coll-Inter* systems. The detection rate gives the percentage of leak incidents that are successfully detected. I also compute standard false positive rate defined in Equation (6.2). I detail the semantic meaning for primary cases, true positive (TP), false positive (FP), true negative (TN), and false negative (FN), in Table 6.3.

$$\text{Detection rate (Recall)} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{6.1}$$

$$\text{False positive rate} = \frac{\text{FP}}{\text{FP} + \text{TP}} \tag{6.2}$$

I define the *sensitivity* $\mathbb{S} \in [0, 1]$ of a content sequence in Equation (6.3). It indicates the similarity of sensitive data D and content $C_{D'}$ with respect to their sequences $\mathcal{S}^a$ and $\mathcal{S}^b$ after PREPROCESS. ξ is the maximum score in the alignment, i.e., the maximum score calculated in the scoring matrix of my dynamic programming alignment. r is the reward for one-unit match in the alignment (details in Section 6.4.3).

$$\mathbb{S} = \frac{\xi}{r \times \min\left(|\mathcal{S}^a|, |\mathcal{S}^b|\right)} \tag{6.3}$$

[2]Rabin's fingerprint is used for unbiased sampling discussed in Section 6.3.2.
[3]Set and collection intersections are used interchangeably.

I reproduce four leaking scenarios in a virtual network environment using VirtualBox. I build a virtual network and deploy the detection systems at the gateway of the virtual network. The detection systems intercept the outbound network traffic, perform deep packet inspection, and extract the content at the highest known network layer[4]. Then the detection systems compare the content with predefined sensitive data to search for any leak.

1. *Web leak:* a user publishes sensitive data on the Internet via typical publishing services, e.g., `WordPress`,
2. *FTP:* a user transfers unencrypted sensitive files to an FTP server on the Internet,
3. *Backdoor:* a malicious program, i.e., `Glacier`, on the user's machine exfiltrates sensitive data,
4. *Spyware:* a `Firefox` extension `FFsniFF` [200] exfiltrates sensitive information via web forms.

It is not a challenge to detect intact data leaks. My *AlignDLD* system successfully detects intact leaks in all these leaking scenarios with a small sampling rate between 5% and 20%. In the following subsections, I analyze the detection accuracy to answer the questions at the beginning of this section.

6.5.2 Detecting Modified Leaks

I evaluate three types of modifications: *i)* real-world pervasive substitution by `WordPress`, *ii)* random pervasive substitution, and *iii)* truncated data (localized modifications).

Pervasive Substitution

I test *AlignDLD* and *Coll-Inter* on content extracted from three kinds of network traffic.

1. Content without any leak, i.e., the content does not contain any sensitive data.
2. Content with unmodified leak, i.e., sensitive data appearing in the content is not modified.
3. Content with modified leaks caused by `WordPress`, which substitutes every space with a "+" in the content.

[4]The content is obtained at the TCP layer when unknown protocols are used at higher network layers.

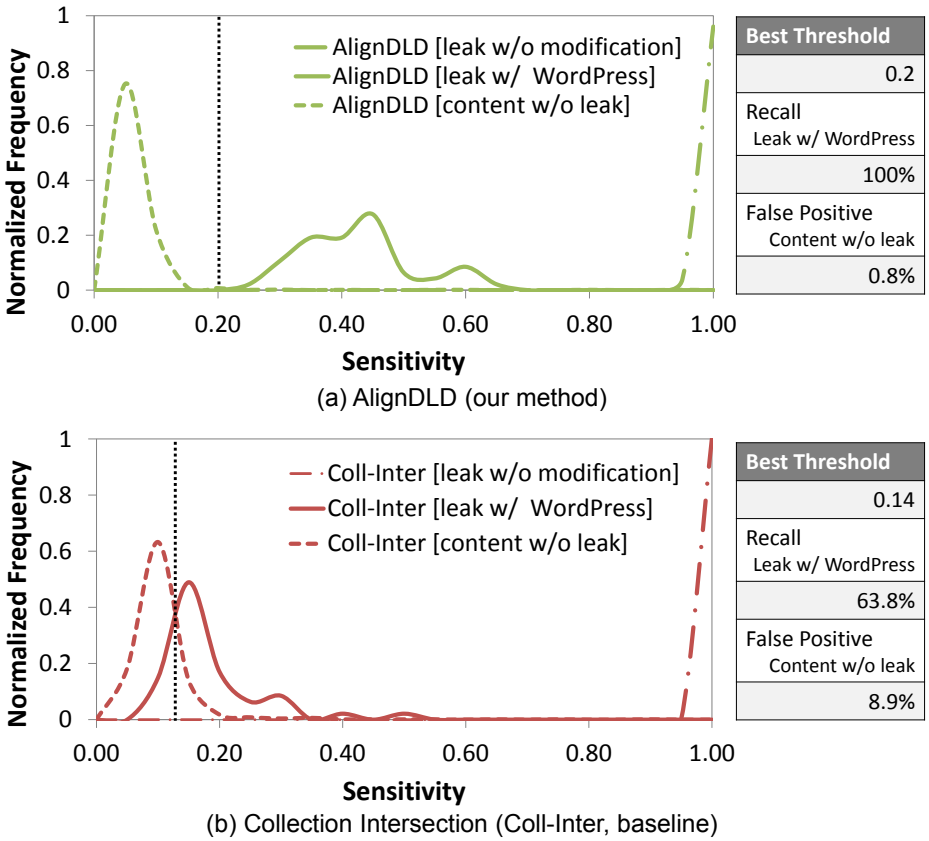


Figure 6.2: Detection comparison of AlignDLD and collection intersection, leak through WordPress in AlignDLD (a) and collection intersection (b). In each subfigure, each of the 3 curves shows the distribution of sensitivity values under one of 3 scenarios: leak without transformation, leak with WordPress transformation, or content without leak. With a threshold of 0.2, AlignDLD detects all the leaks. In comparison, collection intersection performs worse as shown in the table on the right.

The sensitive dataset in this experiment is English text, 50 randomly chosen email messages from the Enron dataset[5]. The content without leak consists of other 950 randomly chosen Enron email messages. I compute the sensitivities of the content according to Equation (6.3).

I evaluate and compare my *AlignDLD* method with the *Coll-Inter* method. The distributions of sensitivity values in all 6 experiments are shown in Figure 6.2. The table to the right of each figure summarizes the detection accuracy under a chosen threshold. The dotted lines in both Figure 6.2 (a) and (b) (on the left) represent the content without leak. Low sensitivities are observed in them by both systems as expected. The dashed lines (on the right) represent the content with unmodified leak. High sensitivities are reported by both systems as expected.

The solid lines in Figure 6.2 represent the detection results of leaks with `WordPress` modifi-

[5]Headers are included.

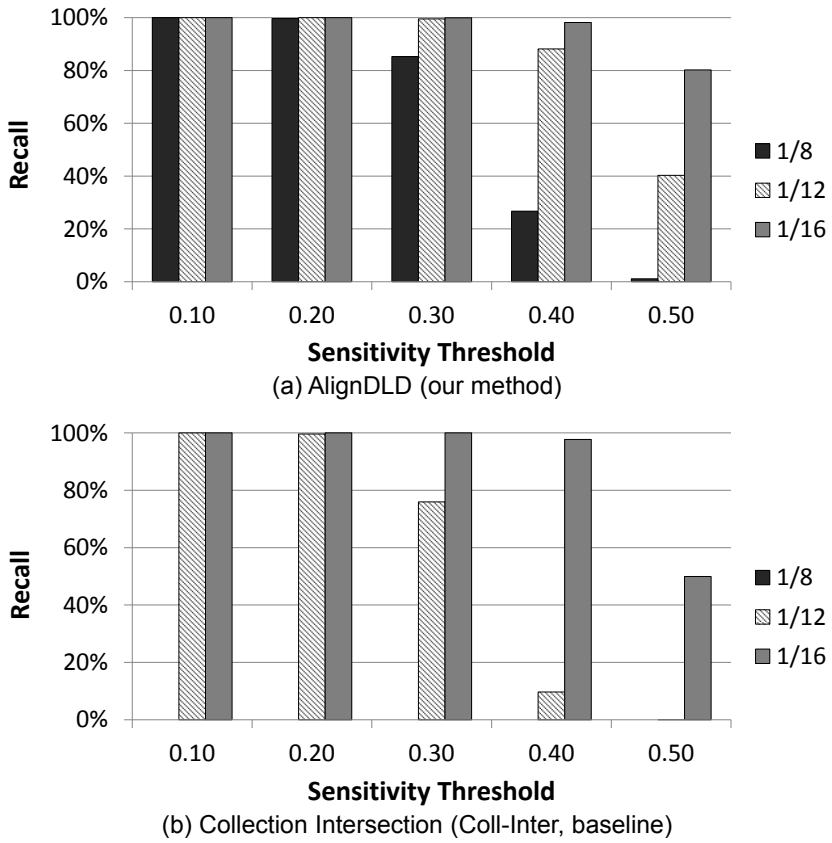


Figure 6.3: Sensitivity values of the content under various transformation ratios. *A. Enron* dataset. Transformation ratio (X-axis) denotes the fraction of leaked sensitive data that is randomized.

cations. My *AlignDLD* method (in Figure 6.2 (a)) gives much higher sensitivity scores to the transformed data leak than the *Coll-Inter* method. ***AlignDLD* detects all transformed email leaks with a threshold of 0.2,** i.e., it achieves 100% recall. The false positive rate is low. In contrast, *Coll-Inter* in Figure 6.2 (b) results in a significant overlap of sensitivity values between messages with no leak and messages with transformed leaks. Its accuracy is much lower than that of *AlignDLD*, e.g., 63.8% recall and a 10 times higher false positive rate. Further analysis of false positives caused by coincidental matches (dotted lines on the left) is given in Section 6.5.3.

Random and Pervasive Substitution

The sensitive data in this experiment is the same as above, i.e., randomly chosen 50 Enron emails (including headers). For the content sequences, I randomize one byte out of every m bytes, where $m \in \{8, 12, 16\}$. The smaller m is, the harder the detection is, as the similarity

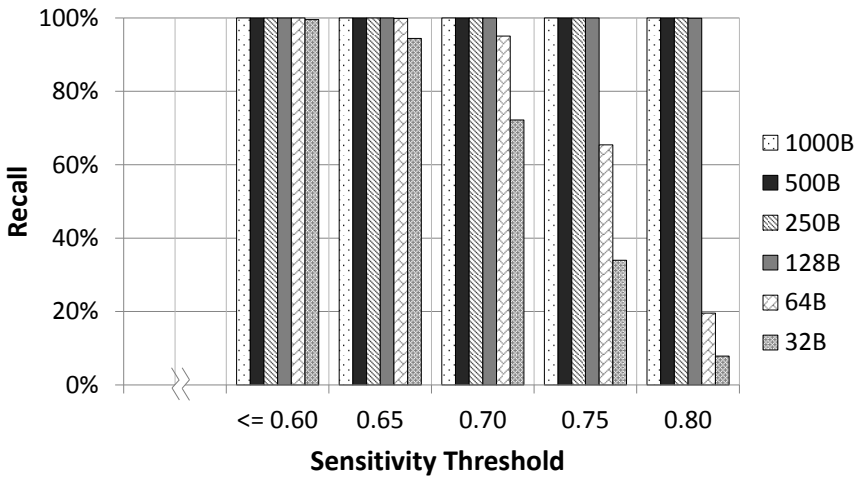


Figure 6.4: The detection success rate of AlignDLD in partial data leaks. Detection under various detection thresholds. Each content sequence contains a consecutive portion of a 1KB sensitive text, ranging from 32 bytes to 1KB. AlignDLD achieves 100% detection rates when the threshold is equal or smaller than 0.6.

between the content and sensitive data becomes lower. The detection results with respect to various thresholds are shown in Figure 6.3.

The recall values decrease as the substitution frequency increases for both the alignment and collection intersection methods as expected. **My alignment method degrades more gracefully under the pervasive substitution scenario.** For example, under threshold 0.3, the detection rate is over 80% even when one out of every 8 bytes is substituted. The collection intersection cannot detect the leak (0% detection rate) in the same scenario.

Data Truncation

In data truncation or partial data leak scenarios, consecutive portions of the sensitive data are leaked. In this experiment, a content sequence contains a portion of sensitive text. The total length of the sensitive text is 1KB. The size of the leaked portion appearing in the content sequence ranges from 32 bytes to 1KB. Each content sequence is 1KB long with random padding if needed.

I measure the *unit sensitivity* $\tilde{\mathbb{S}} \in [0, 1]$ on segments of content sequences. Unit sensitivity $\tilde{\mathbb{S}}$ is the normalized *per-element* sensitivity value for the aligned portion of two sequences. It is defined in Equation (6.4), where $\tilde{\xi}$ is the maximum local alignment score obtained between aligned segments $\tilde{\mathcal{S}}^a$ and $\tilde{\mathcal{S}}^b$, which are sequence segments of sensitive data D and content $C_{D'}$. The higher $\tilde{\mathbb{S}}$ is, the better the detection is. Threshold l is a predefined length describing

Table 6.4: Sampling rates of AlianDLD. A.Enron and B.source-code data sets. $|w| = 100$

N	2	3	5	10	20	40
Enron	2.83%	4.14%	6.67%	12.32%	22.78%	43.1%
S.Code	2.81%	4.01%	6.30%	11.81%	22.33%	43.04%

the shortest segment to invoke the measure. $l = 16$ in my experiments.

$$\tilde{\mathbb{S}} = \frac{\tilde{\xi}}{r \times \min\left(|\tilde{\mathcal{S}}^a|, |\tilde{\mathcal{S}}^b|\right)} \quad \text{where} \quad \min\left(|\tilde{\mathcal{S}}^a|, |\tilde{\mathcal{S}}^b|\right) \geq l \tag{6.4}$$

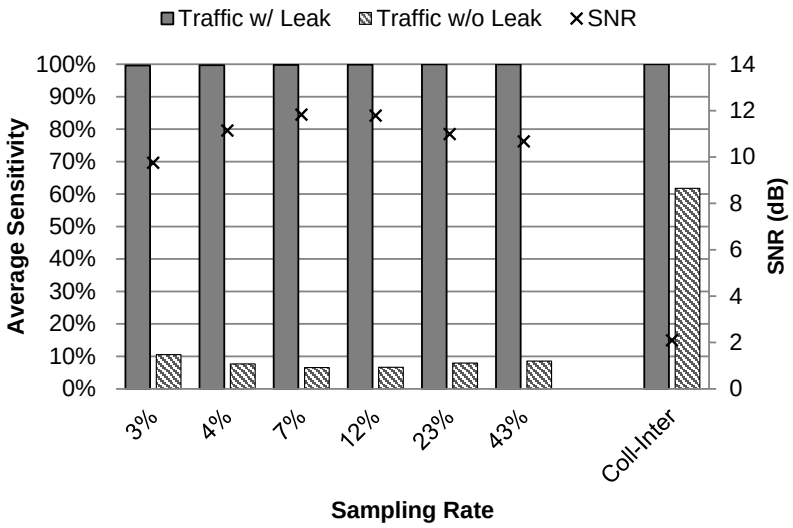
The detection results are shown in Figure 6.4, where X-axis shows the threshold of sensitivity, and Y-axis shows the recall rates of AlignDLD. Content with longer sensitive text is easier to detection as expected. Nevertheless, **my method detects content with short truncated leaks as small as 32 bytes with high accuracy.** The detection rate decreases with higher thresholds. I observe that high thresholds (e.g., higher than 0.6) are not necessary for detection when 8-byte shingles are used; false positives caused by coincidental matches are low in this setup. These experiments show that my detection is resilient to data truncation.

6.5.3 Low False Positive Rate

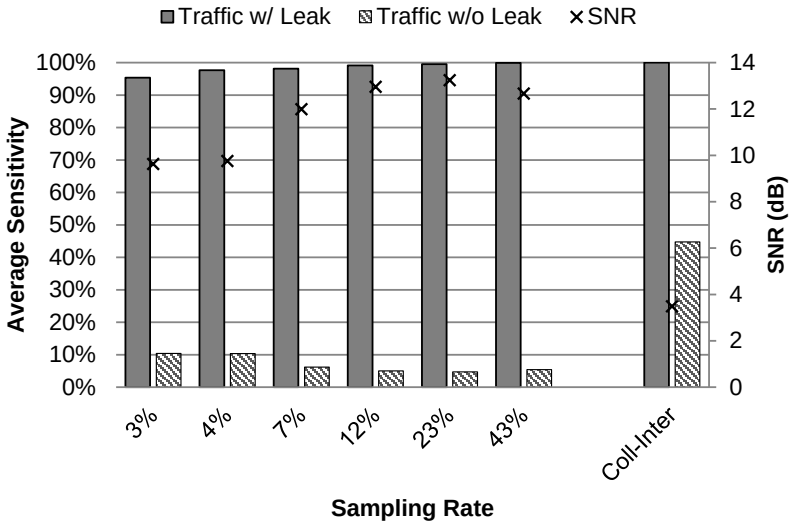
The purpose of this experiment is to evaluate how specific my alignment-based data leak detection is, i.e., reporting leaks and only leaks. I compute and compare the amount of coincidental matches (defined in Section 6.2) found by my method and the collection intersection method. I conduct two sets of experiments using *A. Enron* and *B. Source-code* datasets. In *A. Enron*, I use 50 random email messages (including headers) as the sensitive data and other 950 messages as the content. In *B. Source-code*, I use 5 random files as the sensitive data and other 283 files as the content. None of the contents contain any intentional leaks. Sensitivity scores are computed for each email message and source code file. Small amounts of coincidental matches are expected in these two datasets, because of shared message structures and C/C++ code structures.

I test the impact of sampling in this experiment. I chose screen size $N = 2, 3, 5, 10, 20, 40$ and window size $|w| = 100$. The sampling rates (Table 6.4) on the two datasets are similar when rounded to percentages. This is because Rabin's fingerprint maps any n-gram uniformly to the item space before sampling.

I measure the signal-to-noise ratios (SNR_{dB}) between sensitive scores of real leaks and sensitive scores of non-leak traffic. I calculate SNR_{dB} as in Equation 6.5, where the signal value is the averaged sensitivity score of traffic containing leaks, and the noise value is the averaged



(a) Enron



(b) Source-code

Figure 6.5: Capability of differentiating real leak from coincidental matches, experiments for AlignDLD with different sampling rates and Coll-Inter.

sensitivity score of regular traffic with no leaks.

$$\mathrm{SNR}_{\mathrm{dB}} = 10 \log_{10} \frac{\mathrm{Signal}}{\mathrm{Noise}} \tag{6.5}$$

The results in Figure 6.5 show that the sensitivities are equal or less than 0.1 for almost all detection using my AlignDLD system. With a reasonable threshold (e.g., 0.2), none of

these coincidental matches triggers a false alarm. The detection capability of my approach is generally stable with respect to different sampling rates. I observe that sampling rates have a noticeable but insignificant impact on the results. SNR_{dB} slightly increases when the sampling rate is small, e.g., 3%.

My previous experiments in Section 6.5.2 show that thresholds ≥ 0.2 give a strong separation between true leaks and coincidental matches. Thus, the evidence shows that my method achieves high recall with zero or low false positive rate. In comparison, the collection intersection method reports higher sensitivity scores for the content without any leak, e.g., 62% for Enron emails. High sensitivity scores in coincidental matches lead to a high false positive rate for the collection intersection method as illustrated in Figure 6.2.

Summary. The experimental results provide strong evidences supporting that my method is resilience against various types of modifications evaluated. My alignment algorithm provides a high specificity (i.e., low number of coincidental matches), compared to the collection intersection method. My approach is capable of detecting leaks of various sizes, ranging from tens of bytes to megabytes.

6.6 Parallelization and Evaluation

In order to achieve high analysis throughput, I parallelize my algorithms on CPU as well as on general-purpose GPU platforms. In this section, I aim to answer the following questions:

1. How well does my detection scale? (Sections 6.6.2 and 6.6.3)
2. What is the speedup of sampling? (Section 6.6.4)

6.6.1 Parallel Detection Realization

I implement two parallel versions of my prototype on a hybrid CPU-GPU machine equipped with an Intel Core i5 2400 (Sandy-Bridge micro-architecture) and an NVIDIA Tesla C2050 GPU (Fermi architecture with 448 GPU cores):

1. a multithreading AlignDLD program on CPU [6],
2. a parallel AlignDLD program on GPU [7].

[6] The multithreaded CPU version is written in C, compiled using gcc 4.4.5 with flag -O2.

[7] The GPU version is written in CUDA compiled using CUDA 4.2 with flag -O2 -arch sm 20 and NVIDIA driver v295.41.

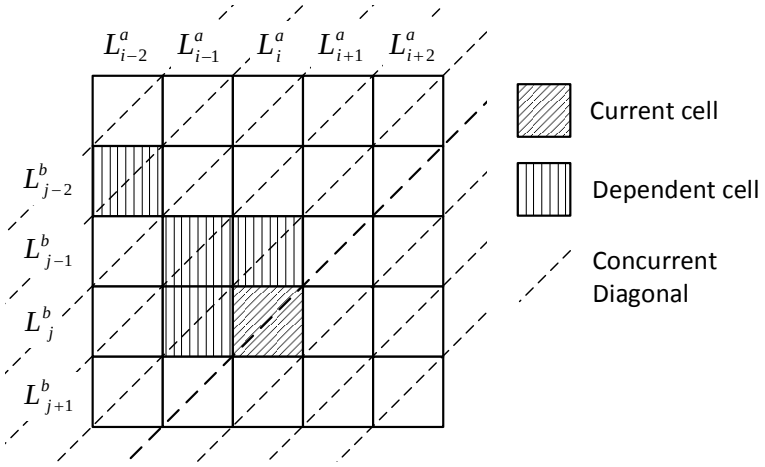


Figure 6.6: Parallel realization of my alignment algorithm. $\mathcal{L}^a_i$ and $\mathcal{L}^b_j$ are the current items to be aligned. All cells on the concurrent diagonal of $(\mathcal{L}^a_i, \mathcal{L}^b_j)$ can be computed simultaneously.

Smith-Waterman alignment was parallelized in OpenGL [129] and CUDA [133]. My parallel alignment algorithms differ from the existing ones, as I address several implementation issues in parallel computing due to my complex weight function.

In the multithreading CPU version, I parallelize both the SAMPLING and ALIGNMENT procedures with the `pthread` library. I parallelize the SAMPLING operation by loading different strings onto different threads. Long streams are split into multiple substrings. Substrings are sampled in parallel by different threads and then assembled for output. ALIGNMENT is the most time-consuming procedure and is made parallel on both CPU and GPU.

I use a parallelized score matrix filling method to compute a diagonal of cells at the same time. My parallel matrix filling strategy is illustrated in Figure 6.6. The scoring matrix is filled from the top left corner to the bottom right corner. At any stage of the process, cells on the concurrent diagonal (dashed lines in Figure 6.6) can be computed simultaneously. My strategy is a variant of the standard Smith-Waterman parallelism [130]. Dependent cells in my algorithm include traditional three adjacent cells as well as all previous cells on the diagonal that is orthogonal to the concurrent diagonal.

The alignment between a sampled sensitive data sequence and a sampled content sequence is assigned to a block of threads on the GPU, and every thread in the block is responsible for an element on the moving diagonal. This method consumes linear space and is efficient. It allows us to fully utilize the memory bandwidth, putting reusable data into fast but small (32KB in my case) shared memory on GPU.

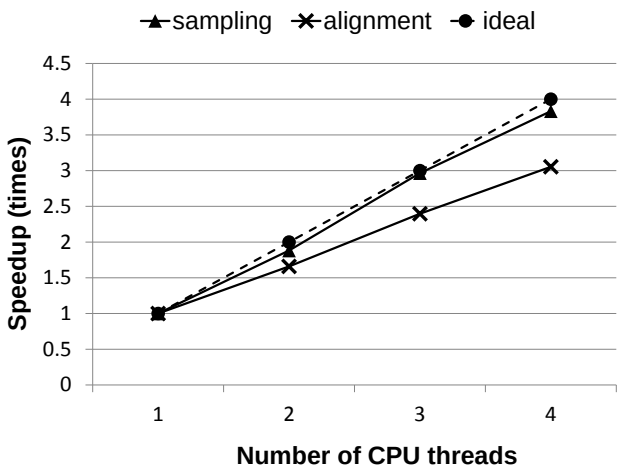


Figure 6.7: High scalability of parallel sampling and alignment algorithms.

6.6.2 Scalability

In this experiment, I parallelize SAMPLING and ALIGNMENT in AlignDLD through various numbers of threads. The times of speedup in analyzing *A. Enron* dataset are reported in Figure 6.7. The results show the close-to-ideal scalability for SAMPLING when parallelized onto an increasing number of threads. My unoptimized multithreaded CPU ALIGNMENT scales up less well in comparison, which I attribute to poor memory cache utilization. The score matrices are too large to fit into the cache for some alignments. The interaction between threads may evict reusable data from the cache. These operations in turn may cause cache misses. An optimized program should possess better data locality to minimize cache misses, and the optimization can be achieved in real-world detection products.

6.6.3 GPU Acceleration

I evaluate the performance of the most time-consuming ALIGNMENT procedure on a GPU with 448 cores grouped in 14 stream multiprocessors and a quad-core CPU. Times of speedup in detecting sensitive data of types `txt`, `png`, or `pdf`[8] against *A. Enron* or *D. MiscNet* traffic, respectively, are shown in Figure 6.8. The result shows that the GPU-accelerated ALIGNMENT achieves over 40 times of speedup over the CPU version on large content datasets (for both *A. Enron* and *D. MiscNet*). GPU speedup with *A. Enron* data is nearly 50 times of the CPU version.

Due to the limited bandwidth between CPU and GPU, data transfer is the bottleneck of my GPU implementation and dominates the execution time. A common strategy to solve

[8] 43KB txt data, 21KB png data, and 633KB pdf data.

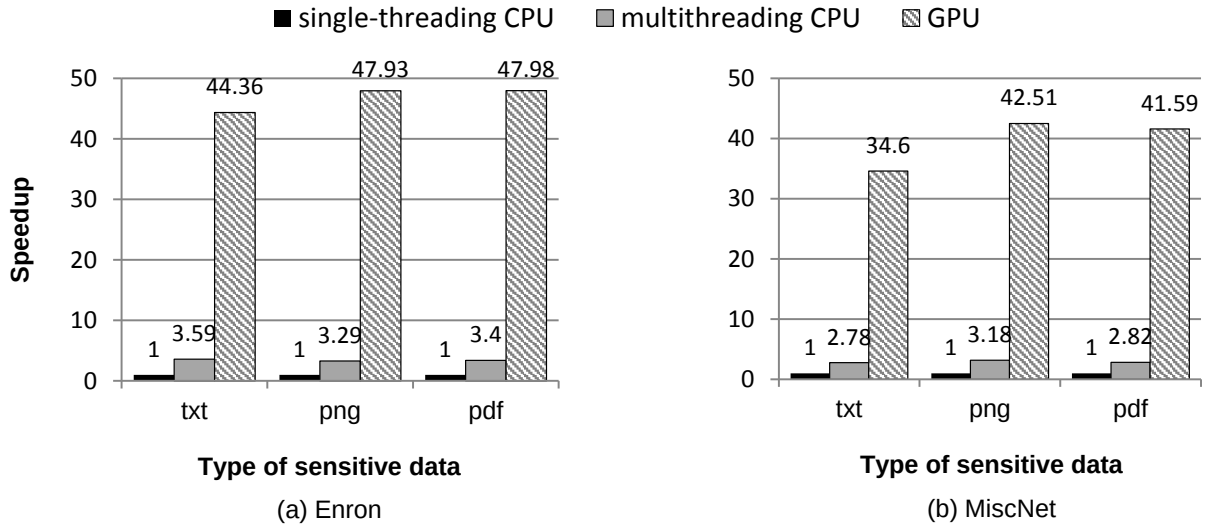


Figure 6.8: Speedup of multithreading alignment and GPU-accelerated alignment, compared with the single thread version on different combinations of sensitive data and traffic data.

Table 6.5: Throughput (in Mbps) of the Alignment operation on GPU.

Sensitive data size (KB)	250	500	1000	2500
Sampling rate				
0.03	426	218	110	44
0.12	23	11	5	2

the issue is to overlap data transfer and kernel execution or to batch the GPU input [99]. Another possible approach from the hardware perspective is to use a CPU-GPU integrated platform, such as AMD APU or Intel MIC, which benefits from the shared memory between CPU and GPU [121].

I report the throughput of ALIGNMENT in my GPU implementation under various parameters. Other procedures – that are faster than alignment – can be carried out in parallel with ALIGNMENT in real-world deployment. I randomly generate sensitive data pieces, 500 bytes for each, and run the detection against 500MB misc network traffic (*D. MiscNet*). The results in Table 6.5 show that I can achieve over 400Mbps throughput on a single GPU. This throughput is comparable to that of a moderate commercial firewall. More optimizations on data locality and memory usage can be performed in real-world detection products.

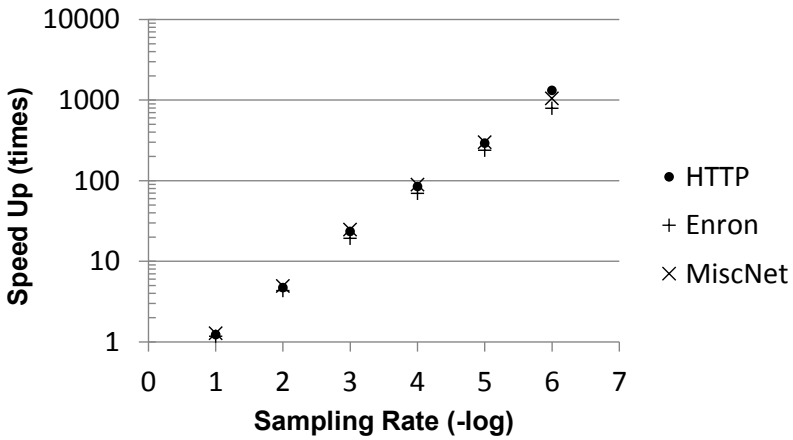


Figure 6.9: Alignment speedup through sampling.

6.6.4 Sampling Speedup

I measure the performance gain brought by sampling and compare the empirical results with the theoretical expectation. Measurements are performed on *A. Enron*, *C. HTTP*, and *D. MiscNet* datasets. Figure 6.9 shows the speedup of ALIGNMENT through different sampling rates α $(0.5, 0.25, 0.125, \dots)$. $-\log_2 \alpha$ is shown on the X-axis. The well fitted lines (R^2 at 0.9988, 0.9977 and 0.9987) from the results have slope coefficients between 1.90 and 2.00, which confirms the α^2 speedup by my sampling design. I calculate the damping factor 0.33 from intercept coefficients of fitted lines.

Chapter 7

Conclusions and Future Work

In this dissertation, I presented theories and defense mechanisms to understand and advance program anomaly detection and network-based data leak detection – two important defense paradigms against sophisticated attack vectors and modern cyber threats. Program anomaly detection discovers program attacks without the knowledge of attack signatures. Network-based data leak detection seeks sensitive data in insecure or inappropriate transmitting channels for potential leaks.

My theoretical work presented in Chapter 3 provided a general framework 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. It filled a gap in the literature to unify deterministic and probabilistic models with my formal definition of program anomaly detection. According to my 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. My framework has the potential to serve as a roadmap and help researchers approach the ultimate program defense without attack signature specification.

My event correlation analysis approach presented in Chapter 4 is a two-stage anomaly detection method that unearths attacks from extreme long program traces. The significance of this work is its capability to discover subtle program inconsistencies efficiently. This work validated my systematization of program anomaly detection and advanced the state-of-the-art program anomaly detection by demonstrating the effectiveness of large-scale program behavioral modeling and enforcement against runtime anomalies that are buried in extremely long execution paths.

I proposed fuzzy fingerprint, a privacy-preserving data-leak detection model, in Chapter 5. Using special digests, the exposure of the sensitive data is kept to a minimum during the detection. I realized a prototype based on my approach and conducted extensive experiments to validate the accuracy, privacy, and efficiency of our solutions. The application of the

privacy-preserving approach is described in the cloud computing environments, where the cloud provider naturally serves as the DLD provider.

I presented a content inspection technique for detecting leaks of sensitive information in chapter 6. My detection approach is based on aligning two sampled sequences for similarity comparison. The technical enabler is a new comparable sampling technique. It preserves local features of inputs and thus supports the similarity comparison between two sampled sequences. This unique property allows the local alignment of shorter sampled sequences to efficiently compute their similarity and detect leaks. The experimental results suggested that the alignment method is useful for detecting multiple common data leak scenarios where the leaked data is transformed. The parallel CPU and GPU versions of my prototype provided substantial speedup and indicate high scalability of my design.

For future work, I plan to further advance practical solutions for both program anomaly detection and data leak detection. With the help of the latest hardware, e.g., Intel Processor Tracing, it is possible to achieve practical user-space monitoring with less than 5% overhead. The combination of fast tracing and my event correlation analysis method will enable the development of accurate real-time incidence response systems that can autonomously recognize and interrupt ongoing attacks. Host-based data tracking and network-based data leak detection methods are complimentary, and I plan to leverage the characterization of sensitive data in host-based methods to assist my network-based solutions detect rapidly changing sensitive data such as source code.

Bibliography

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 340–353, 2005.

[2] Pankaj K. Agarwal and R. Sharathkumar. Streaming algorithms for extent problems in high dimensions. In *SODA*, pages 1481–1489, 2010.

[3] Gagan Aggarwal, Tomás Feder, Krishnaram Kenthapadi, Rajeev Motwani, Rina Panigrahy, Dilys Thomas, and An Zhu. Anonymizing tables. In *Database Theory-ICDT 2005*, pages 246–258. Springer, 2005.

[4] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.

[5] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 263–277, May 2008.

[6] Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990.

[7] James P Anderson. Computer security technology planning study. Technical report, DTIC Document, 1972.

[8] Mikhail J. Atallah, Frédéric Chyzak, and Philippe Dumas. A randomized algorithm for approximate string matching. *Algorithmica*, 29(3):468–486, 2001.

[9] Mikhail J. Atallah, Elena Grigorescu, and Yi Wu. A lower-variance randomized algorithm for approximate string matching. *Inf. Process. Lett.*, 113(18):690–692, 2013.

[10] M. Bach, M. Charney, R. Cohn, E. Demikhovsky, T. Devor, K. Hazelwood, A. Jaleel, Chi-Keung Luk, G. Lyons, H. Patil, and A. Tal. Analyzing parallel programs with Pin. *Computer*, 43(3):34–41, March 2010.

[11] Zachary K. Baker and Viktor K. Prasanna. Time and area efficient pattern matching on FPGAs. In *FPGA*, pages 223–232, 2004.

[12] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Krugel, and Engin Kirda. Scalable, behavior-based malware clustering. In *NDSS*, 2009.

[13] S. Bhatkar, A. Chaturvedi, and R. Sekar. Dataflow anomaly detection. In *Proceedings of IEEE Symposium on Security and Privacy*, May 2006.

[14] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented programming: A new class of code-reuse attack. In *Proceedings of ASIACCS*, pages 30–40, 2011.

[15] Kevin Borders and Atul Prakash. Quantifying information leaks in outbound web traffic. In *Security & Privacy, 2009 30th IEEE Symposium on*, pages 129–140. IEEE, 2009.

[16] Kevin Borders, Eric Vander Weele, Billy Lau, and Atul Prakash. Protecting confidential data on personal computers with storage capsules. *Ann Arbor*, 1001:48109, 2009.

[17] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, October 1977.

[18] Joan Bresnan, Ronald M Kaplan, Stanley Peters, and Annie Zaenen. Cross-serial dependencies in Dutch. In *The formal complexity of natural language*, pages 286–319. Springer, 1987.

[19] Eric Brier, Quentin Fortier, Roman Korkikian, K. W. Magld, David Naccache, Guilherme Ozari de Almeida, Adrien Pommellet, A. H. Ragab, and Jean Vuillemin. Defensive leakage camouflage. In *CARDIS*, pages 277–295, 2012.

[20] Andrei Broder and Michael Mitzenmacher. Network applications of Bloom filters: A survey. *Internet mathematics*, 1(4):485–509, 2004.

[21] Andrei Z Broder. Some applications of Rabins fingerprinting method. In *Sequences II*, pages 143–152. Springer, 1993.

[22] Andrei Z. Broder. Identifying and filtering near-duplicate documents. In *COM '00: Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*, pages 1–10, London, UK, 2000. Springer-Verlag.

[23] Andrei Z. Broder, Moses Charikar, Alan M. Frieze, and Michael Mitzenmacher. Min-wise independent permutations. *J. Comput. Syst. Sci.*, 60(3):630–659, 2000.

[24] David Brumley, Dawn Xiaodong Song, Tzi cker Chiueh, Rob Johnson, and Huijia Lin. RICH: Automatically protecting against integer-based vulnerabilities. In *NDSS*, 2007.

[25] Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas Dimitropoulos. SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. *Network*, 1:101101, 2010.

[26] Min Cai, Kai Hwang, Yu-Kwong Kwok, Shanshan Song, and Yu Chen. Collaborative Internet worm containment. *IEEE Security and Privacy*, 3(3):25–33, 2005.

[27] Davide Canali, Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. A quantitative study of accuracy in system call-based malware detection. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 122–132. ACM, 2012.

[28] Bogdan Carbunar and Radu Sion. Joining privately on outsourced data. In *Secure Data Management*, pages 70–86. Springer, 2010.

[29] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 161–176, Washington, D.C., August 2015. USENIX Association.

[30] Vitor R. Carvalho and William W. Cohen. Preventing information leaks in email. In *SDM*, 2007.

[31] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 147–160, Berkeley, CA, USA, 2006. USENIX Association.

[32] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection for discrete sequences: A survey. *IEEE TKDE*, 24(5):823–839, May 2012.

[33] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM Computing Surveys (CSUR)*, 41(3):15, 2009.

[34] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 559–572. ACM, 2010.

[35] Rui Chen, Benjamin Fung, Noman Mohammed, Bipin C Desai, and Ke Wang. Privacy-preserving trajectory data publishing by local suppression. *Information Sciences*, 231:83–97, 2013.

[36] Shuo Chen, Jun Xu, Emre C Sezer, Prachi Gauriar, and Ravishankar K Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th conference on USENIX Security Symposium*, volume 14, pages 12–12, 2005.

[37] Yikan Chen and David Evans. Auditing information leakage for distance metrics. In *SocialCom/PASSAT*, pages 1131–1140, 2011.

[38] Young H. Cho and William H. Mangione-Smith. A pattern matching coprocessor for network security. In *DAC*, pages 234–239, 2005.

[39] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, 1956.

[40] Marco Cova, Davide Balzarotti, Viktoria Felmetsger, and Giovanni Vigna. Swaddler: An approach for the anomaly-based detection of state violations in web applications. In *Recent Advances in Intrusion Detection*, pages 63–86. Springer, 2007.

[41] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of USENIX Security*, volume 7, pages 5–5, 1998.

[42] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-variant systems: A secretless framework for security through diversity. In *Proceedings of USENIX Security*, volume 15, 2006.

[43] Jason Croft and Matthew Caesar. Towards practical avoidance of information leakage in enterprise networks. *USENIX HotSec (August 2011)*, 2011.

[44] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Ropdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 40–51, New York, NY, USA, 2011. ACM.

[45] Lorenzo De Carli, Robin Sommer, and Somesh Jha. Beyond pattern matching: A concurrency model for stateful deep packet inspection. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1378–1390. ACM, 2014.

[46] Dorothy E Denning. An intrusion-detection model. *Software Engineering, IEEE Transactions on*, pages 222–232, 1987.

[47] Anne Dinning and Edith Schonberg. *An empirical comparison of monitoring algorithms for access anomaly detection.* ACM, 1990.

[48] Wenliang Du and Michael T Goodrich. Searching for high-value rare events with uncheatable grid computing. In *Applied Cryptography and Network Security*, pages 122–137. Springer, 2005.

[49] Manuel Egele, Peter Wurzinger, Christopher Kruegel, and Engin Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *DIMVA*, pages 88–106, 2009.

[50] D. Endler. Intrusion detection. applying machine learning to Solaris audit data. In *Proceedings of the 14th Annual Computer Security Applications Conference*, pages 268–279, December 1998.

[51] E. Eskin, Wenke Lee, and S.J. Stolfo. Modeling system calls for intrusion detection with dynamic window sizes. In *Proceedings of DARPA Information Survivability Conference and Exposition II*, volume 1, pages 165–175, 2001.

[52] Pete Evans. Heartbleed bug: Rcmp asked revenue canada to delay news of sin thefts. *CBC News*, April 2014.

[53] Tyrell Fawcett. *ExFILD: A tool for the detection of data exfiltration using entropy and encryption characteristics of network traffic*. PhD thesis, University of Delaware, 2010.

[54] Dan Feldman, Morteza Monemizadeh, Christian Sohler, and David P. Woodruff. Coresets and sketches for high dimensional subspace approximation problems. In *SODA*, pages 630–649, 2010.

[55] Henry Hanping Feng, Jonathon T Giffin, Yong Huang, Somesh Jha, Wenke Lee, and Barton P Miller. Formalizing sensitivity in static analysis for intrusion detection. In *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on*, pages 194–208. IEEE, 2004.

[56] Henry Hanping Feng, Oleg M Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong. Anomaly detection using call stack information. In *Security and Privacy. Proceedings. the Symposium on*, pages 62–75. IEEE, 2003.

[57] Domenico Ficara, Gianni Antichi, Andrea Di Pietro, Stefano Giordano, Gregorio Procissi, and Fabio Vitucci. Sampling techniques to accelerate pattern matching in network intrusion detection systems. In *Communications (ICC), 2010 IEEE International Conference on*, pages 1–5. IEEE, 2010.

[58] Prahlad Fogla, Monirul Sharif, Roberto Perdisci, Oleg Kolesnikov, and Wenke Lee. Polymorphic blending attacks. In *Proceedings of the 15th USENIX Security Symposium*, pages 241–256, 2006.

[59] S. Forrest, A.S. Perelson, L. Allen, and R. Cherukuri. Self-nonself discrimination in a computer. In *Proceedings of the 1994 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 202–212, May 1994.

[60] Stephanie Forrest, Steven Hofmeyr, and Anil Somayaji. The evolution of system-call monitoring. In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*, pages 418–430. IEEE, 2008.

[61] Stephanie Forrest, Steven A Hofmeyr, Anil Somayaji, and Thomas A Longstaff. A sense of self for unix processes. In *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on*, pages 120–128. IEEE, 1996.

[62] W Nelson Francis and Henry Kucera. Brown corpus manual. *Brown University Department of Linguistics*, 1979.

[63] Matt Fredrikson, Somesh Jha, Mihai Christodorescu, Reiner Sailer, and Xifeng Yan. Synthesizing near-optimal malware specifications from suspicious behaviors. In *Security and Privacy, 2010 IEEE Symposium on*, pages 45–60. IEEE, 2010.

[64] Sam Frizell. Report: Devastating Heartbleed flaw was used in hospital hack. *Time Magazine*, August 2014.

[65] Alessandro Frossi, Federico Maggi, Gian Luigi Rizzo, and Stefano Zanero. Selecting and improving system call models for anomaly detection. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 206–223. Springer, 2009.

[66] Debin Gao, Michael K. Reiter, and Dawn Song. On gray-box program tracking for anomaly detection. In *Proceedings of USENIX Security*, volume 13, pages 8–8, 2004.

[67] Debin Gao, Michael K Reiter, and Dawn Song. Behavioral distance for intrusion detection. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses*, pages 63–81. Springer, 2006.

[68] Debin Gao, Michael K Reiter, and Dawn Song. Behavioral distance measurement using hidden Markov models. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses*, pages 19–40, 2006.

[69] Shahabeddin Geravand and Mahmood Ahmadi. Bloom filter applications in network security: A state-of-the-art survey. *Computer Networks*, 57(18):4047–4064, 2013.

[70] Anup K. Ghosh and Aaron Schwartzbard. A study in using neural networks for anomaly and misuse detection. In *Proceedings of USENIX Security*, volume 8, pages 12–12, 1999.

[71] Anup K Ghosh, James Wanken, and Frank Charron. Detecting anomalous and unknown intrusions against programs. In *14th Annual Computer Security Applications Conference, 1998. Proceedings.*, pages 259–267. IEEE, 1998.

[72] Jonathon T Giffin, David Dagon, Somesh Jha, Wenke Lee, and Barton P Miller. Environment-sensitive intrusion detection. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses*, pages 185–206, 2006.

[73] Jonathon T. Giffin, Somesh Jha, and Barton P. Miller. Detecting manipulated remote call streams. In *Proceedings of USENIX Security*, pages 61–79, 2002.

[74] Jonathon T Giffin, Somesh Jha, and Barton P Miller. Efficient context-sensitive intrusion detection. In *NDSS*, 2004.

[75] Global Velocity Inc. http://www.globalvelocity.com/.

[76] GoCloudDLP. GTB Technologies Inc. http://www.gocloudlp.com/.

[77] Enes Göktaş, Elias Athanasopoulos, Michalis Polychronakis, Herbert Bos, and Georgios Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 417–432, San Diego, CA, August 2014. USENIX Association.

[78] Rajeev Gopalakrishna, Eugene H Spafford, and Jan Vitek. Efficient intrusion detection using automaton inlining. In *Security and Privacy, 2005 IEEE Symposium on*, pages 18–31. IEEE, 2005.

[79] Guofei Gu, Phillip A Porras, Vinod Yegneswaran, Martin W Fong, and Wenke Lee. BotHunter: Detecting malware infection through IDS-driven dialog correlation. In *USENIX Security*, volume 7, pages 1–16, 2007.

[80] Zhongshu Gu, Kexin Pei, Qifan Wang, Luo Si, Xiangyu Zhang, and Dongyan Xu. LEAPS: Detecting camouflaged attacks with statistical learning guided by program analysis. In *Proceedings of the Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 491–502, 2014.

[81] Fang Hao, Murali Kodialam, TV Lakshman, and Hui Zhang. Fast payload-based flow estimation for traffic monitoring and network security. In *Architecture for networking and communications systems, 2005. ANCS 2005. Symposium on*, pages 211–220. IEEE, 2005.

[82] The Heartbleed bug, http://heartbleed.com/.

[83] Steven Hofmeyr. Primary response technical white paper. http://www.ttivanguard.com/austinreconn/primaryresponse.pdf.

[84] John E Hopcroft. *Introduction to automata theory, languages, and computation*. Pearson Education India, 1979.

[85] Roberto Hoyle, Sameer Patil, Dejanae White, Jerald Dawson, Paul Whalen, and Apu Kapadia. Attire: conveying information exposure through avatar apparel. In *CSCW Companion*, pages 19–22, 2013.

[86] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. Automatic generation of data-oriented exploits. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 177–192, Washington, D.C., August 2015. USENIX Association.

[87] Nen-Fu Huang, Hsien-Wei Hung, Sheng-Hung Lai, Yen-Ming Chu, and Wen-Yen Tsai. A GPU-based multiple-pattern matching algorithm for network intrusion detection systems. In *AINA Workshops*, pages 62–67, 2008.

[88] Qiang Huang, David Jao, and Helen J Wang. Applications of secure electronic voting to automated privacy-preserving troubleshooting. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 68–80. ACM, 2005.

[89] Zhengli Huang, Wenliang Du, and Biao Chen. Deriving private information from randomized data. In *SIGMOD Conference*, pages 37–48, 2005.

[90] N. Hubballi. Pairgram: Modeling frequency information of lookahead pairs for system call based anomaly detection. In *Communication Systems and Networks (COMSNETS), 2012 Fourth International Conference on*, pages 1–10, Jan 2012.

[91] N. Hubballi, S. Biswas, and S. Nandi. Sequencegram: n-gram modeling of system calls for program based anomaly detection. In *Communication Systems and Networks (COMSNETS), 2011 Third International Conference on*, pages 1–10, Jan 2011.

[92] Identity Finder. http://www.identityfinder.com/.

[93] Hajime Inoue and Anil Somayaji. Lookahead pairs and full sequences: a tale of two anomaly detection methods. In *Proceedings of the 2nd Annual Symposium on Information Assurance*, pages 9–19, 2007.

[94] Ponemon Institute. 2013 cost of data breach study: Global analysis.

[95] Md Rafiqul Islam, Md Saiful Islam, and Morshed U Chowdhury. Detecting unknown anomalous program behavior using API system calls. In *Informatics Engineering and Information Science*, pages 383–394. Springer, 2011.

[96] Jafar Haadi Jafarian, Ali Abbasi, and Siavash Safaei Sheikhabadi. A gray-box dpda-based intrusion detection technique using system-call monitoring. In *Proceedings of the 8th Annual Collaboration, Electronic messaging, Anti-Abuse and Spam Conference*, pages 1–12. ACM, 2011.

[97] Geetha Jagannathan and Rebecca N Wright. Privacy-preserving distributed k-means clustering over arbitrarily partitioned data. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 593–599. ACM, 2005.

[98] Rohit Jalan and Arun Kejariwal. Trin-Trin: Who's calling? a Pin-based dynamic call graph extraction framework. *International Journal of Parallel Programming*, 40(4):410–442, 2012.

[99] Muhammad Asim Jamshed, Jihyung Lee, Sangwoo Moon, Insu Yun, Deokjin Kim, Sungryoul Lee, Yung Yi, and KyoungSoo Park. Kargus: a highly-scalable software-based intrusion detection system. In *ACM Conference on Computer and Communications Security*, pages 317–328, 2012.

[100] Jiyong Jang, David Brumley, and Shobha Venkataraman. BitShred: feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 309–320, New York, NY, USA, 2011. ACM.

[101] Yeongjin Jang, Simon Chung, Bryan Payne, and Wenke Lee. Gyrus: A framework for user-intent monitoring of text-based networked applications. In *Proceedings of the 23rd USENIX Security Symposium*, pages 79–93, 2014.

[102] S. Jha, L. Kruger, and V. Shmatikov. Towards practical privacy for genomic computation. In *IEEE Symposium on Security and Privacy*, 2008.

[103] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy malware detection and monitoring through VMM-based "out-of-the-box" semantic view reconstruction. *ACM Transactions on Information and System Security (TISSEC)*, 13(2):12, 2010.

[104] Jaeyeon Jung, Anmol Sheth, Ben Greenstein, David Wetherall, Gabriel Maganis, and Tadayoshi Kohno. Privacy Oracle: a system for finding application leaks with black box differential testing. In *ACM Conference on Computer and Communications Security*, pages 279–288, 2008.

[105] Jaeyeon Jung, Anmol Sheth, Ben Greenstein, David Wetherall, Gabriel Maganis, and Tadayoshi Kohno. Privacy oracle: a system for finding application leaks with black box differential testing. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 279–288. ACM, 2008.

[106] Alexandros Kapravelos, Yan Shoshitaishvili, Marco Cova, Christopher Kruegel, and Giovanni Vigna. Revolver: An automated approach to the detection of evasive web-based malware. In *USENIX Security Symposium*, 2013.

[107] Sandeep Karanth, Srivatsan Laxman, Prasad Naldurg, Ramarathnam Venkatesan, J. Lambert, and Jinwook Shin. Pattern mining for future attacks. Technical Report MSR-TR-2010-100, Microsoft Research, 2010.

[108] Gunter Karjoth and Matthias Schunter. A privacy policy model for enterprises. In *Computer Security Foundations Workshop, 2002. Proceedings. 15th IEEE*, pages 271–281. IEEE, 2002.

[109] Kaspersky Lab. Global corporate IT security risks, 2014. http://media.kaspersky.com/en/business-security/Kaspersky_Global_IT_Security_Risks_Survey_report_Eng_final.pdf.

[110] Vasileios P. Kemerlis, Vasilis Pappas, Georgios Portokalidis, and Angelos D. Keromytis. iLeak: A lightweight system for detecting inadvertent information leaks. In *Proceedings of the 6th European Conference on Computer Network Defense (EC2ND)*, October 2010.

[111] Jon Kleinberg, Christos H Papadimitriou, and Prabhakar Raghavan. On the value of private information. In *Proceedings of the 8th conference on Theoretical aspects of rationality and knowledge*, pages 249–257. Morgan Kaufmann Publishers Inc., 2001.

[112] Andrew P Kosoresow and Steven A Hofmeyr. Intrusion detection via system call traces. *IEEE software*, 14(5):35–42, 1997.

[113] Christian Kreibich and Jon Crowcroft. Efficient sequence alignment of network traffic. In *Internet Measurement Conference*, pages 307–312, 2006.

[114] C. Kruegel, D. Mutz, W. Robertson, and F. Valeur. Bayesian event classification for intrusion detection. In *Proceedings of the 19th Annual Computer Security Applications Conference*, pages 14–23, December 2003.

[115] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Automating mimicry attacks using static binary analysis. In *Proceedings of the 14th conference on USENIX Security Symposium-Volume 14*, pages 11–11. USENIX Association, 2005.

[116] Sailesh Kumar, Balakrishnan Chandrasekaran, Jonathan S. Turner, and George Varghese. Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. In *ANCS*, pages 155–164, 2007.

[117] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan S. Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *SIGCOMM*, pages 339–350, 2006.

[118] Sailesh Kumar, Jonathan S. Turner, and John Williams. Advanced algorithms for fast and scalable deep packet inspection. In *ANCS*, pages 81–92, 2006.

[119] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. In *Proceedings of USENIX OSDI*, pages 147–163, 2014.

[120] Jeffrey P. Lanza. SSH CRC32 attack detection code contains remote integer overflow. *Vulnerability Notes Database*, 2001.

[121] K. Lee, H. Lin, and W. Feng. Performance characterization of data-intensive kernels on AMD fusion architectures. *Computer Science-Research and Development*, pages 1–10, 2012.

[122] Wenke Lee and Salvatore J Stolfo. Data mining approaches for intrusion detection. In *Proceedings of the 7th conference on USENIX Security Symposium*, pages 6–6. USENIX Association, 1998.

[123] Jin Li, Qian Wang, Cong Wang, Ning Cao, Kui Ren, and Wenjing Lou. Fuzzy keyword search over encrypted data in cloud computing. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–5. IEEE, 2010.

[124] Kang Li, Zhenyu Zhong, and Lakshmish Ramaswamy. Privacy-aware collaborative spam filtering. *IEEE Transactions on Parallel and Distributed systems*, 20(5), May 2009.

[125] Yihua Liao and V.Rao Vemuri. Use of k-nearest neighbor classifier for intrusion detection. *Computers & Security*, 21(5):439–448, 2002.

[126] Christopher Liebchen, Marco Negro, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Stephen Crane, Mohaned Qunaibit, Michael Franz, and Mauro Conti. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *Proceedings of ACM CCS*, 2015.

[127] Po-Ching Lin, Ying-Dar Lin, Yuan-Cheng Lai, and Tsern-Huei Lee. Using string matching for deep packet inspection. *IEEE Computer*, 41(4):23–28, 2008.

[128] Fang Liu, Xiaokui Shu, Danfeng Yao, and Ali Raza Butt. Privacy-preserving scanning of big content for sensitive data exposure with MapReduce. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy (CODASPY)*, pages 195–206, San Antonio, TX, USA, March 2015.

[129] W. Liu, B. Schmidt, G. Voss, A. Schroder, and W. Muller-Wittig. Bio-sequence database scanning on a GPU. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 8–pp. IEEE, 2006.

[130] Yongchao Liu, Douglas Maskell, and Bertil Schmidt. CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units. *BMC Research Notes*, 2(1):73, 2009.

[131] Zhen Liu, Susan M Bridges, and Rayford B Vaughn. Combining static analysis and dynamic learning to build accurate intrusion detection models. In *Information Assurance, 2005. Proceedings. Third IEEE International Workshop on*, pages 164–177. IEEE, 2005.

[132] Federico Maggi, Matteo Matteucci, and Stefano Zanero. Detecting intrusions through system call sequence and argument analysis. *IEEE Transactions on Dependable and Secure Computing*, 7(4):381–395, 2010.

[133] S.A. Manavski and G. Valle. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC bioinformatics*, 9(Suppl 2):S10, 2008.

[134] Carla Marceau. Characterizing the behavior of a program using multiple-length n-grams. In *Proceedings of NSPW*, pages 101–110, 2000.

[135] Christoph P Mayer. Bloom filters and overlays for routing in pocket switched networks. In *Proceedings of the 5th international student workshop on Emerging networking experiments and technologies*, pages 43–44. ACM, 2009.

[136] J. Sukarno Mertoguno. Human decision making model for autonomic cyber systems. *International Journal on Artificial Intelligence Tools*, 23(06):1460023, 2014.

[137] David Moore, Colleen Shannon, Douglas J Brown, Geoffrey M Voelker, and Stefan Savage. Inferring internet Denial-of-Service activity. *ACM Transactions on Computer Systems (TOCS)*, 24(2):115–139, 2006.

[138] B. Morgenstern, A. Dress, and T. Werner. Multiple DNA and protein sequence alignment based on segment-to-segment comparison. *Proc. Natl. Acad. Sci. U.S.A.*, 93(22):12098–12103, Oct 1996.

[139] Darren Mutz, Fredrik Valeur, Giovanni Vigna, and Christopher Kruegel. Anomalous system call detection. *ACM Trans. Inf. Syst. Secur.*, 9(1):61–93, February 2006.

[140] Adwait Nadkarni and William Enck. Preventing accidental data disclosure in modern operating systems. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, 2013.

[141] Nergal. The advanced return-into-lib(c) exploits. *Phrack magazine*, 11(58), 2001.

[142] Ben Niu and Gang Tan. Modular control-flow integrity. *SIGPLAN Notices*, 49(6):577–587, June 2014.

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

[144] Christos H Papadimitriou. *Computational complexity*. John Wiley and Sons Ltd., 2003.

[145] The Paradyn Project, http://www.paradyn.org/.

[146] Vern Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23-24):2435–2463, 1999.

[147] R. Perdisci, Guofei Gu, and Wenke Lee. Using an ensemble of one-class SVM classifiers to harden payload-based anomaly detection systems. In *Data Mining, 2006. ICDM '06. Sixth International Conference on*, pages 488–498, Dec 2006.

[148] Roberto Perdisci, Davide Ariu, Prahlad Fogla, Giorgio Giacinto, and Wenke Lee. McPAD: A multiple classifier system for accurate payload-based anomaly detection. *Computer Networks*, 53(6):864 – 881, 2009.

[149] Roberto Perdisci et al. VAMO: towards a fully automated malware clustering validity analysis. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 329–338. ACM, 2012.

[150] Roberto Perdisci, Wenke Lee, and Nick Feamster. Behavioral clustering of http-based malware and signature generation using malicious network traces. In *NSDI'10*, pages 26–26. USENIX Association, 2010.

[151] V. O. Polyanovsky, M. A. Roytberg, and V. G. Tumanyan. Comparative analysis of the quality of a global algorithm and a local algorithm for alignment of two sequences. *Algorithms Mol Biol*, 6(1):25, 2011.

[152] Geoffrey K. Pullum. Context-freeness and the computer processing of human languages. In *Proceedings of the 21st Annual Meeting on Association for Computational Linguistics*, ACL '83, pages 1–6, Stroudsburg, PA, USA, 1983. Association for Computational Linguistics.

[153] Josh Quittner. Panix attack. *Time Magazine*, September 2006.

[154] M. O. Rabin. Fingerprinting by random polynomials. Technical report, Center for Research in Computing Technology, Harvard University, 1981. TR-15-81.

[155] Lakshmish Ramaswamy, Arun Iyengar, Ling Liu, and Fred Douglis. Automatic detection of fragments in dynamically generated web pages. In *Proceedings of the 13th international conference on World Wide Web*, pages 443–454. ACM, 2004.

[156] Paruj Ratanaworabhan, V. Benjamin Livshits, and Benjamin G. Zorn. NOZZLE: A defense against heap-spraying code injection attacks. In *USENIX Security Symposium*, pages 169–186, 2009.

[157] RiskBasedSecurity. Data breach quickview: 2014 data breach trends, February 2015.

[158] Martin Roesch. Snort: Lightweight intrusion detection for networks. In *LISA*, pages 229–238, 1999.

[159] Bernhard Schölkopf, Robert C Williamson, Alex J Smola, John Shawe-Taylor, and John C Platt. Support vector method for novelty detection. In *NIPS*, volume 12, pages 582–588, 1999.

[160] R Sekar, Mugdha Bendre, Dinakar Dhurjati, and Pradeep Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*, pages 144–155. IEEE, 2001.

[161] Hovav 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.

[162] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM.

[163] Monirul Sharif, Kapil Singh, Jonathon Giffin, and Wenke Lee. Understanding precision in host based intrusion detection. In *Recent Advances in Intrusion Detection*, pages 21–41. Springer, 2007.

[164] Stuart M Shieber. *Evidence against the context-freeness of natural language*. Springer, 1987.

[165] Xiaokui Shu, Fang Liu, and Danfeng Yao. Rapid screening of big data against inadvertent leaks. In Shui Yu and Song Guo, editors, *Big Data Concepts, Theories and Applications*. Springer International Publishing, March 2016.

[166] Xiaokui Shu and Danfeng Yao. Data leak detection as a service. In *Proceedings of the 8th International Conference on Security and Privacy in Communication Networks (SecureComm)*, pages 222–240, Padua, Italy, September 2012.

[167] Xiaokui Shu, Danfeng Yao, and Elisa Bertino. Privacy-preserving detection of sensitive data exposure. *IEEE Transactions on Information Forensics and Security (TIFS)*, 10(5):1092–1103, May 2015.

[168] Xiaokui Shu, Danfeng Yao, and Naren Ramakrishnan. Unearthing stealthy program attacks buried in extremely long execution paths. In *Proceedings of the 2015 ACM Conference on Computer and Communications Security (CCS)*, pages 401–413, Denver, CO, USA, October 2015.

[169] Xiaokui Shu, Danfeng Yao, and Barbara G. Ryder. A formal framework for program anomaly detection. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 270–292, Kyoto, Japan, November 2015.

[170] Xiaokui Shu, Jing Zhang, Danfeng Yao, and Wu-chun Feng. Rapid and parallel content screening for detecting transformed data exposure. In *Proceedings of the Third International Workshop on Security and Privacy in Big Data (BigSecurity)*, pages 191–196, Hongkong, China, April 2015.

[171] Xiaokui Shu, Jing Zhang, Danfeng Yao, and Wu-chun Feng. Rapid screening of transformed data leaks with efficient algorithms and parallel computing. In *Proceedings of*

the 5th ACM Conference on Data and Application Security and Privacy (CODASPY), pages 147–149, San Antonio, TX, USA, March 2015. Extended abstract.

[172] Xiaokui Shu, Jing Zhang, Danfeng Yao, and Wu-chun Feng. Fast detection of transformed data leaks. *IEEE Transactions on Information Forensics and Security (TIFS)*, 11(3):528–542, March 2016.

[173] Randy Smith, Neelam Goyal, Justin Ormont, Karthikeyan Sankaralingam, and Cristian Estan. Evaluating GPUs for network packet signature matching. In *ISPASS*, pages 175–184, 2009.

[174] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *J. Mol. Biol.*, 147(1):195–197, Mar 1981.

[175] Anil Somayaji and Stephanie Forrest. Automated response using system-call delays. In *Proceedings of the 9th USENIX Security Symposium*, volume 70, 2000.

[176] Alexander Sotirov. Heap feng shui in javascript. *Black Hat Europe*, 2007.

[177] Sufatrio and RolandH.C. Yap. Improving host-based IDS with argument abstraction to prevent mimicry attacks. In *Recent Advances in Intrusion Detection*, volume 3858 of *Lecture Notes in Computer Science*, pages 146–164. Springer Berlin Heidelberg, 2006.

[178] Wenhai Sun, Wenjing Lou, Y Thomas Hou, and Hui Li. Privacy-preserving keyword search over encrypted data in cloud computing. In *Secure Cloud Computing*, pages 189–212. Springer, 2014.

[179] S.C. Sundaramurthy, J. McHugh, X.S. Ou, S.R. Rajagopalan, and M. Wesch. An anthropological approach to studying CSIRTs. *IEEE Security & Privacy*, 12(5):52–60, September 2014.

[180] Symantec. Symantec data loss prevention, 2015. http://www.symantec.com/data-loss-prevention.

[181] Systemtap overhead test, https://sourceware.org/ml/systemtap/2006-q3/msg00146.html.

[182] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal war in memory. In *Proceedings of IEEE S & P*, pages 48–62, 2013.

[183] Gaurav Tandon and Philip K. Chan. On the learning of system call attributes for host-based anomaly detection. *International Journal on Artificial Intelligence Tools*, 15(6):875–892, 2006.

[184] Juan Ramón Troncoso-Pastoriza, Stefan Katzenbeisser, and Mehmet Celik. Privacy preserving error resilient DNA searching through oblivious automata. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 519–528. ACM, 2007.

[185] Nathan Tuck, Timothy Sherwood, Brad Calder, and George Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. In *INFOCOM*, 2004.

[186] Jan van Lunteren. High-performance pattern-matching for intrusion detection. In *INFOCOM*, 2006.

[187] Vijay Varadharajan. Internet filtering-issues and challenges. *IEEE Security & Privacy*, 8(4):0062–65, 2010.

[188] Giorgos Vasiliadis, Spyros Antonatos, Michalis Polychronakis, Evangelos P. Markatos, and Sotiris Ioannidis. Gnort: High performance network intrusion detection using graphics processors. In *RAID*, pages 116–134, 2008.

[189] Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. MIDeA: a multi-parallel intrusion detection architecture. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 297–308, New York, NY, USA, 2011. ACM.

[190] Vendicator. StackShield. http://www.angelfire.com/sk/stackshield/.

[191] U. Vishkin. Deterministic sampling – a new technique for fast pattern matching. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, STOC '90, pages 170–180, New York, NY, USA, 1990. ACM.

[192] David Wagner and R Dean. Intrusion detection via static analysis. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*, pages 156–168. IEEE, 2001.

[193] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, CCS '02, pages 255–264, New York, NY, USA, 2002. ACM.

[194] Bing Wang, Shucheng Yu, Wenjing Lou, and Y Thomas Hou. Privacy-preserving multi-keyword fuzzy search over encrypted data in the cloud. In *IEEE INFOCOM*, 2014.

[195] Tielei Wang, Tao Wei, Zhiqiang Lin, and Wei Zou. IntScope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *NDSS*, 2009.

[196] Christina Warrender, Stephanie Forrest, and Barak Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *Security and Privacy. Proceedings. the IEEE Symposium on*, pages 133–145. IEEE, 1999.

[197] Kyubum Wee and Byungeun Moon. Automatic generation of finite state automata for detecting intrusions using system call sequences. In *Proceedings of MMM-ACNS*, 2003.

[198] Andreas Wespi, Marc Dacier, and Hervé Debar. Intrusion detection using variable-length audit trail patterns. In *Proceedings of RAID*, pages 110–129, 2000.

[199] Peter Williams and Radu Sion. Usable PIR. In *NDSS*, 2008.

[200] Candid Wuest and Elia Florio. Firefox and malware: When browsers attack. Technical report, Symantec Corporation, October 2009.

[201] Benjamin Wun, Patrick Crowley, and Arun Raghunath. Parallelization of Snort on a multi-core platform. In Peter Z. Onufryk, K. K. Ramakrishnan, Patrick Crowley, and John Wroclawski, editors, *Proceedings of the 2009 ACM/IEEE Symposium on Architecture for Networking and Communications Systems, ANCS 2009, Princeton, New Jersey, USA, October 19-20, 2009*, pages 173–174. ACM, 2009.

[202] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. CFIMon: Detecting violation of control flow integrity using performance counters. In *Proceedings of the Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 1–12, June 2012.

[203] Gaoyao Xiao, Jun Wang, Peng Liu, Jiang Ming, and Dinghao Wu. Program-object level data flow analysis with applications to data leakage and contamination forensics. In *Proceedings of the Sixth ACM on Conference on Data and Application Security and Privacy*, pages 277–284. ACM, 2016.

[204] Kui Xu, Danfeng Yao, Qiang Ma, and Alexander Crowell. Detecting infection onset with behavior-based policies. In *Network and System Security (NSS), 2011 5th International Conference on*, pages 57–64. IEEE, 2011.

[205] Kui Xu, Danfeng Yao, Barbara G. Ryder, and Ke Tian. Probabilistic program modeling for high-precision anomaly classification. In *Proceedings of IEEE CSF*, 2015.

[206] Shouhuai Xu. Collaborative attack vs. collaborative defense. In *Collaborative Computing: Networking, Applications and Worksharing*, pages 217–228. Springer, 2009.

[207] Weilin Xu, Yanjun Qi, and David Evans. Automatically evading classifiers: A case study on pdf malware classifiers. In *NDSS*, 2016.

[208] L. Yang, R. Karim, V. Ganapathy, and R. Smith. Improving NFA-based signature matching using ordered binary decision diagrams. In *13th International Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2010.

[209] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X Sean Wang. AppIntent: Analyzing sensitive data transmission in Android for privacy leakage detection. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, 2013.

[210] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *Foundations of Computer Science, 1986., 27th Annual Symposium on*, pages 162–167. IEEE, 1986.

[211] Danfeng Yao, Keith B Frikken, Mikhail J Atallah, and Roberto Tamassia. Private information: To reveal or not to reveal. *ACM Transactions on Information and System Security (TISSEC)*, 12(1):6, 2008.

[212] Nong Ye and X Li. A markov chain model of temporal behavior for anomaly detection. In *Proceedings of the 2000 IEEE Systems, Man, and Cybernetics Information Assurance and Security Workshop*, volume 166, page 169. Oakland: IEEE, 2000.

[213] Xun Yi, Md. Golam Kaosar, Russell Paulet, and Elisa Bertino. Single-database private information retrieval from fully homomorphic encryption. *IEEE Trans. Knowl. Data Eng.*, 25(5):1125–1134, 2013.

[214] Xun Yi, Russell Paulet, and Elisa Bertino. *Private Information Retrieval*. Synthesis Lectures on Information Security, Privacy, and Trust. Morgan & Claypool Publishers, 2013.

[215] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 116–127. ACM, 2007.

[216] Suan Hsi Yong and Susan Horwitz. Protecting C programs from attacks via invalid pointer dereferences. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-11, pages 307–316, New York, NY, USA, 2003. ACM.

[217] Stefano Zanero. Behavioral intrusion detection. In *Computer and Information Sciences*, pages 657–666. Springer, 2004.

[218] Hao Zhang, Maoyuan Sun, Danfeng Yao, and Chris North. Visualizing traffic causality for analyzing network anomalies. *Proceedings of International Workshop on Security and Privacy Analytics*, pages 37–42, 2015.

[219] Hao Zhang, Danfeng Yao, and Naren Ramakrishnan. Detection of stealthy malware activities with traffic causality and scalable triggering relation discovery. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 39–50. ACM, 2014.

[220] Hao Zhang, Danfeng Yao, Naren Ramakrishnan, and Zhibin Zhang. Causality reasoning about network events for detecting stealthy malware activities. *Computers & Security*, 58:180–198, 2016.