Privacy-Preserving Distributed Event Corroboration

Janak J. Parekh

Submitted in partial fulfillment of the
Requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2007

ABSTRACT

Privacy-Preserving Distributed Event Correlation

Janak J. Parekh

Event correlation is a widely-used data processing methodology, and is useful for the distributed monitoring of software faults and vulnerabilities. Most existing solutions have focused on "intra-organizational" correlation; organizations typically employ *privacy policies* that prohibit the exchange of information outside of the organization. However, "inter-organizational" Internet-scale correlation holds promise given its potential role in both software fault maintenance *and* vulnerability detection.

In this thesis, I reconcile these opposing forces via the use of *privacy preservation* integrated into an event processing framework. I introduce the notion of *event corroboration*, a reduced yet flexible form of correlation, enabling *collaborative verification* without revealing sensitive information. The framework supports both *source anonymity* and *data privacy*, yet allows for temporal corroboration of a broad variety of data. It is designed as a lightweight collection of components to enable integration with existing COTS platforms and distributed systems. I also present an implementation: Worminator, a Collaborative Intrusion Detection System (CIDS); it is based on an earlier platform, XUES (**X**ML **U**niversal **E**vent **S**ervice), an event processor used as part of an autonomic software monitoring, reconfiguration and repair platform.

XUES collected and correlated information from sensors installed in legacy systems; while it was not privacy-preserving, it laid the groundwork for Worminator by supporting event typing, the use of publish-subscribe and extensibility support via pluggable modules. In turn, Worminator is a rewrite of XUES to support privacy-preserving event types and algorithms, enabling intrusion detection alerts

to be corroborated without revealing sensitive information about a contributor's identity, network or services. Worminator is implemented as a corroboration framework on top of existing IDS sensors, and can detect not only worms but also "stealthy" scans; traditional single-network sensors overlook or miss them entirely. Worminator corroborates packet metadata, packet content, and even aggregate models of network traffic.

The contributions of this thesis include the development of an event processing framework with native privacy-preserving types, the use of privacy-preserving corroboration, and the establishment of a practical deployed collaborative security system. The thesis also quantifies Worminator's effectiveness at attack detection and its privacy preservation techniques.

Contents

List of Figures

List of Tables

I dedicate this thesis to Sal Stolfo, whose mix of tireless encouragement and constant pressure was the main reason I ever finished this research. He thought nothing but of his students and gave selflessly of himself, to the point of illness shortly after my defense. Sal, may you have a speedy recovery and get back to what you do best: encouraging and supporting students to succeed in their research. Thank you so much for everything.

Acknowledgements

I would first like to thank Gail Kaiser, my advisor; she was instrumental in helping me start my research and guided me through the veritable Ph.D. minefield with aplomb. I would also like to thank my co-advisor, Sal Stolfo, and the rest of my committee (Al Aho, Wenke Lee, and Dan Rubenstein) for their involvement and useful feedback. I would also like to acknowledge my colleagues in this work, including Gabriela Cretu, Ke Wang, Vanessa Frias-Martinez, Weijen Li, Michael Locasto, Angelos Stavrou, and Angelos Keromytis for their help on Worminator; and Phil Gross, Suhit Gupta, Gaurav Kc, Peppo Valetto, and Enrico Buonnano for their work on KX. I also thank the other members of PSL, IDS, and NSL for putting up with me.

I would also like to thank Panagiotis Manolios and Peter Dillinger for their suggestions in Bloom filter design, and David Garlan, Bradley Schmerl, and George Heineman for their timely participation and assistance in developing and successfully demonstrating KX. Thanks also go to David Dagon and Philip Chan for their assistance in the deployment and testing of Worminator.

Finally, I would like to thank my family, including my mom and my dad for their infinite patience; my sister, for going through a program as long as mine, to demonstrate I'm not the only insane one in the family; and, last but not least, my dear friend (and now Professor) Katherine Compton for her tireless encouragement.

Chapter 1

Introduction

Event correlation is becoming increasingly useful in the context of distributed monitoring for software faults and vulnerabilities, as software continues to grow in size, configurations become more complex, and event/alert logs become increasingly difficult to understand. This thesis primarily concerns *distributed* event correlation, where application-specific data may be exchanged by any number of Internet peers.

The semantics (i.e., *types*) of this event data can differ greatly based on the application: it may contain information about method calls in an instrumented software program, database transactions being executed, or intrusion-detection alerts triggered by a pattern of suspicious behavior (which in turn can itself be viewed as correlation). Such events may contain significant amounts of data as it is transmitted and processed, *or* may contain metadata about system behavior. As a result, event streams are generally confined to an individual organization, much like other strategic organizational data—and correlation systems remain within the organization's network as well. However, application semantics are no longer local area network-based; instead, the pervasiveness of Internet communication dictates that applications be robust to a broader variety of errors and vulnerabilities. Additionally, Internet-scale correlation gives any individual node far richer correla-

tion data, as a wider range of nodes and their diversity enable a broader variety of correlation and debugging scenarios ([89], etc.).

To date, however, "inter-organizational" correlation has been limited at best to organizations that forge business and legal relationships with each other, in order to avoid releasing sensitive information, such as trade secrets or information restricted by compliance requirements (e.g., HIPAA [108]). This is at odds with most current event correlation techniques, which focus on the expressiveness of correlation capability, as the fundamental working assumption is that almost any organizational data can be encapsulated into an event. These approaches do not work with anonymized, limited data exchanges between nodes. Instead, a solution has to accommodate the notion of an organization's *privacy policy*, such as the disclaimer provided on websites disclosing appropriate practices with collected customer data.[1]

Therefore, any solution that forms Internet-scale collaborations must be able to comply with privacy and information disclosure policies, yet still meaningfully correlate event data. I directly address this tradeoff via the idea of *event corroboration*. Corroboration is a form of collaborative verification, e.g. "organization X corroborates organization Y's observation of event *e*", implying both X and Y observed an event of interest. As I describe in detail in this thesis, while corroboration is a subset of general correlation, it is still surprisingly flexible. It is not limited to simple event matching, and can cover a large number of useful scenarios, while offering privacy guarantees for sensitive information. Here, effective cross-site corroboration is accomplished via the design and implementation of a type-driven event processing framework adaptable to organizational privacy requirements.

I begin with definitions and a formal problem statement addressed by this thesis,

[1]My practical experience in dealing with a broad variety of organizations, including academic, governmental, and business entities, demonstrates that such entities are either ignorant of the possibilities of collaboration or are actively against it due to the information disclosure problem.

and briefly mention the thesis's contributions to this problem before discussing the model in chapter 3.

1.1 Definitions

In this section, I formalize some of the terms used in this thesis.

- **Events** are discrete, structured data objects generated at a specific point in time [111]. It is important to note that the structure of events are left to be defined as part of the problem domain. The data that is stored as part of an application method call, for example, may differ significantly from that of an intrusion detection alert. Most of the techniques in this thesis can be applied to almost any event semantics/event structure, although I focus on three:

    1. An "opaque" event is one whose contents appear unstructured to an event subsystem; for example, a network packet's payload does not have an *a priori* structure unless protocol-aware parsing and processing is first done. Such events can be treated as a fixed blob of data, or may undergo aggregate analysis, e.g., a frequency distribution over the contents of the packet.
    2. A "flat" event contains a collection of attribute-value pairs. This structure is common in event distribution systems [19, 135], and allows for fast processing and routing.
    3. A "hierarchical" event may contain collections of attribute-value pairs arranged into a tree hierarchy. A classic example of a standardized hierarchical wire format is XML, which has both *elements*, which can be nested, and *attributes*, which are flat and are associated with an element.

In addition to the overall structure of an event, I take note of two additional pieces of metadata:

1. The *type* of an event is a description specifying both the structural and semantic aspects of the data contained in that event. By typing events, we can build standardized mechanisms to process and transform such data. A type hierarchy relevant to this thesis is described in greater detail in chapter 3. Note that untyped events can be treated as opaque, which may reduce the expressiveness of the event to a subsystem, but allows for a general approach to all events instead of those for which information is known ahead of time.
2. The *timestamp* of an event is a global clock value associated with the generation time of an event. One can also store other timestamps with an event, such as the "detection" time(s) that triggered this event or the "publication" time when the event is pushed through a distribution infrastructure; for the purposes of this thesis, these are considered specialized timestamps but can substitute, if necessary, for the generation time. The key attribute a timestamp provides is an *ordering* on a sequence of events to accommodate applications that attach a semantic meaning to such ordering. A timestamp can also provide an *expiration* mechanism to ameliorate memory and processing overhead.

   Such timestamps can be explicit or implicit; if a timestamp is not associated with an event on its generation, one can be created during publication of the event *or* upon receipt of the event. This may have ramifications on the precision of event ordering, but the techniques in this these are orthogonal to timestamping precision.

- **Distributed** refers to Internet-scale communication and information sharing.

The work in this thesis may also apply to large private networks, but the focus is on the most general Internet-based distribution mechanisms. *Peers* or *sites* may form collaboration groups via a variety of different communication protocols and topologies. It is **important to note**, however, that this thesis does *not* focus on the communication substrate itself—rather, it is a framework that supports, and plugs into, a broad variety of *publish-subscribe* mechanisms. I discuss publish-subscribe and desiderata for such substrates in chapter 3; the implementations discussed in chapters 2 and 6 use well-known publish-subscribe event systems.

- **Event Corroboration** is precisely defined in section 3.2. Briefly stated, a corroboration by a participant is a set intersection with its local set of events against set(s) of events received from another party or parties. Much of corroboration's flexibility, and equivalence to the aforementioned event correlation, comes from careful definition of a "set" of events and appropriate comparison operations during set intersections.

- **Privacy** also takes many different forms [43]. Some of the more relevant ones to event correlation include:

    1. **Data privacy** refers to the semantics of the data in the event and whether they contain information that may be deemed sensitive by the producer of the event. A *privacy transform* can be employed to translate a non-privacy-preserving event to one that enforces data privacy.

    2. **Source anonymity** guarantees the privacy of an event producer's identity. A source that is anonymous cannot be traced by recipients of the event: there is no explicit identifier linking the event to a known producer, the data in the event cannot reliably be linked to the producer, and the source cannot be traced from the receipt of the event (e.g., source of a network

connection).

3. **Physical privacy** refers to the access of sensitive information or resources via direct access to the repositories or interference with servers of data. This includes intruders, malicious insiders, and resource starvation (e.g., denial-of-service) mechanisms.

4. **Time privacy** corresponds to the fact that the distribution of event arrival times could yield some aggregate information; more interestingly, the correlation of curious or insidious activities with event arrival times could potentially violate the source anonymity stated above.

This thesis focuses on the first two forms of privacy. It is possible to maintain data privacy without maintaining source anonymity (e.g., an event came from source X but it is free of what X deems sensitive), as well as vice-versa (e.g., it is unknown exactly who the event came from, but it contains classified information privy to only a small number of organizations). Of course, both can exist in tandem. With both, I argue that recipients cannot trace the source or information for relevant applications (i.e., those that are fulfilled by the requirements in section 1.3). In this thesis, I develop novel concepts in data privacy and combine them with several well-known anonymity techniques to accomplish a complete synthesis of the two.

As for the latter two forms, physical privacy poses a unique set of challenges on its own—most systems secure from remote access have physical backdoors—and is considered outside the scope of this thesis. Meanwhile, the definition of events and event correlation assume an ordering amongst events. Some of the data privacy approaches in the thesis do indirectly provide time privacy, but full time privacy poses its own unique correlation challenges; a complete discussion is outside the scope of this work.

- A **privacy-preserving transformation** is a data transformation, $d' = p(d)$ for some arbitrary datum d, where the inverse function $p^{-1}(d')$ cannot be algorithmically derived *and* for which brute-force guessing d given a d' is not tractable. A classic example of a privacy-preserving transform are one-way functions, such as the hash function SHA-1 [105]. This thesis uses several different privacy-preserving transforms, including hashing (§5.2.1) and frequency transforms (§5.2.3)—some of them well-established but are used here in novel contexts, while others are novel concepts.

- Finally, a **privacy policy** is both a promise by an organization to originators of data contained within the organization, as well as a compliance statement to consumers of data produced by the organization. It may contain one or both of the first two privacy requirements, as well as other additional requirements. Such a policy may be expressed as a legal/paper contract *or* as a machine-encoded representation, such as P3P [167]. Chapter 5 briefly discusses the relevance of privacy policies to this thesis, and chapter 7 discusses future work possibilities with respect to encoding such privacy policies into our framework to direct the corroboration done in this thesis.

1.2 Problem Statement

As implied, the problem explored in this thesis is to

Design an event processing methodology, appropriate event transformation techniques, and a distribution and corroboration architecture to process transformed events that:

- Supports Internet-scale collaboration;
- Approximates generalized event correlation for software reliability and network security;
- Enables information sharing between organizations whose privacy requirements would ordinarily forbid such event-driven information exchange.

1.3 Requirements

Based on our definitions and initial problem statement, I now establish a set of requirements necessary for an effective solution to the problem. Ideally, these requirements should be established as a minimum, but not *imposed* on the target application, as certain collaboration groups *may* choose to share more information where appropriate.

1. **Support data privacy.** First and foremost, data in a published event must correspond to the producer's privacy requirements. At the simplest level, this may refer to data sanitization, where rules are established to determine what part of an event is kept, and what part of an event is removed prior to publication [110]. However, the applications cited in this thesis are interested in correlating the sensitive data itself; scrubbing the data from the event eliminates this possibility, and so a more general approach to transforming event data is required that enables correlation without revealing content.

2. **Support event source anonymity.** As described in section 1.1, an event source must, if it prefers, be able to contribute anonymously. While data privacy helps in this regard, consideration at the event distribution level must be taken

into account. Note that this *does not* imply that a given event source cannot be differentiated from other sources, but rather that any one event source cannot be traced back to its origin. As discussed in section 5.3, differentiation is an extremely useful technique for enriching correlation. In certain situations, one may also want to support *classification* in addition to differentiation, e.g., classify source identifiers into aggregate groups without revealing individual identities.

3. **Support event corroboration.** Clearly, it is impossible to support every correlation scenario when the original data is not present. Instead of requiring completely generalized correlation, which is the research focus of many existing approaches, I relax this definition to require *corroboration* as a minimum, i.e., collaborative verification of observed application behavior. While the correlator should, if possible, support richer event correlation when shared data allows it, corroboration is the baseline operation. The techniques and applications described in this thesis validate that corroboration is effective for a broad variety of applications.

4. **Support privacy-preserving authentication.** While the correlator may be source-agnostic, the producers of events should ideally be authenticated as a means of providing simple trust management (e.g., to differentiate benign and malicious entities). However, any such authentication should not require a producer to reveal their identity to collaborating peers unless they volunteer that information. Approaches that provide an authentication-free trust model do exist, but pose their own problems, as described in section 5.3.3. It is important to note that this requirement does *not* require that peers actually transmit event streams through the authenticator.

5. **Support heterogeneous privacy requirements.** As already indicated, an

organization may volunteer to contribute more information than necessary as per the above requirements. Others may employ a finer-grained differentiation; for example, internal identities may be prohibited from being exchanged, while externally-originating identities may be communicable. The framework must be able to support these heterogeneous requests and support correlation between them.

6. **Heterogeneous application support.** Not only should the framework support different privacy policies, it should support different datatypes for different applications. The notion of privacy-preservation is broadly applicable to more than just one specific problem, and this thesis puts forward the notion of separating the privacy-preservation requirement from the application at hand.

7. **Near real-time performance.** While the framework does not need to support hard real-time requirements, it should support the ability to keep up with event streams during runtime. This is especially significant for applications that generate large numbers of events.

Other requirements have been previously implied, so we simply list them here: The framework must be *Internet-scale*, supporting cross-site and cross-organization collaboration, and it must support *temporal correlation*, enabling the creation of corroboration rules with temporal constraints.

I now state the two hypotheses of this research: privacy-preserving mechanisms and event typing.

1.4 Hypotheses

1. *The addition of privacy-preserving mechanisms (source anonymity, data privacy) will enable effective correlation despite organizational privacy-preserving requirements.*

More precisely, the implementation in section 5 will demonstrate that temporal event corroboration remains practical even after the addition of both source anonymity and data privacy-preserving transforms.

2. *A typed event-driven framework provides a solution for matching heterogeneous organizational information-sharing policies with different privacy mechanisms.* The model presented in chapter 3, using privacy-preserving event structures and a type-enabled event processing infrastructure, maps to Worminator's use of Bloom filters, frequency models, Z-Strings, etc., as discussed in chapter 6.

1.5 Thesis Outline

The rest of this thesis is organized as follows.

- First, chapter 2 introduces the decentralized correlation model and motivates the development of privacy-preserving mechanisms. It details the architecture and implementation of XUES (**X**ML **U**niversal **E**vent **S**ervice), an event processor used as part of an autonomic software monitoring, control and effector platform called KX (**K**inesthetics e**X**treme), which originally was *not* privacy-preserving. At the end of this chapter, several example applications are discussed that need privacy-preserving mechanisms to be feasible.

- Next, given the definitions and problems introduced by the introduction and the middleware monitoring chapters, chapter 3 introduces the core corroboration model—later used in, but not limited to, the applications described in chapter 6. Chapter 4 discusses work related to this model in several fields, including software engineering and intrusion detection.

- Chapter 5 then abstracts the problem and discusses, at an abstract theoretical level, techniques that can be used to make such frameworks privacy-preserving

and requirements in general to attain a desirable level of privacy. It addresses both the creation of new frameworks and the concept of *retrofitting* privacy onto existing architectures, such as XUES.

- Chapter 6 finally details Worminator—an architecture and implementation that, from the ground up, contains the necessary privacy-preserving and event typing features necessary for the area of *collaborative intrusion detection*. A significant evaluation is conducted for three main applications—IP-based intrusion alert dissemination, privacy-preserving payload-based anomaly exchange/signature generation, and traffic model-based intrusion detection/profiling.

- Finally, the thesis concludes with chapter 7 and discusses possible venues of future work.

Chapter 2

Middleware Event Monitoring

This chapter describes *XUES*, or *XML Universal Event Service*, the initial event-processing middleware developed in this thesis, and in particular the *Event Packager* and *Event Distiller* components, used as part of a larger framework called KX, or Kinesthetics eXtreme.

In this chapter, I[1] briefly describe the background and motivation for developing this middleware. After the overview, KX's architecture and implementation are detailed; several example applications of the framework are also described. Finally, I then conclude with a discussion of how privacy becomes an issue once such a system is deployed in a distributed environment. The later chapters in this thesis tie the middleware motivated here to privacy mechanisms, including an abstracted model (ch. 3), via *retrofitting* (§5.4), and via a new ground-up design (ch. 6).

2.1 Background

KX was originally designed and implemented as part of an attempt to provide self-management and self-healing facilities to increasingly complex networked computer systems and applications that lack such solutions. A surprising number

[1] KX is joint work with other members of the Programming Systems Lab.

of applications today have minimal or nonexistent self-management capabilities; the result has led to a tremendous interest in what some have termed *autonomic computing* [125, 48]. However, most approaches described in the literature for developing autonomic software systems (e.g., see [66]) ignore *legacy* software and the increasingly common assembly of large scale systems from components supplied by multiple sources, instead assuming the customer or user will be willing and able to migrate to this new generation of systems. In fact, even when not mandated by archaic hardware, legacy software may persist indefinitely, even though it was implemented in "unsafe" languages like C, or in languages no longer in common use where many expert maintainers are now past retirement age [36], making the need for autonomic diagnosis, and possibly repair, even greater [123].

Note that by specifying a legacy system here, we mean any system, no matter how recent, that does not include its own built-in self-management capabilities. Further, some subsystems of the "systems of systems" of interest may indeed include their own autonomic or analogous previous-generation fault-tolerance, dependability, reliability, survivability, etc. facilities, but these alone will not necessarily provide an "end-to-end" self-management capability for the composite system as a whole; this issue is argued in greater detail in [159].

A few general-purpose facilities have been developed to automate problem detection and/or problem correction for pre-existing software. For example, some new operating systems include engines to automate the collection of crash data [11]; other tools help detect anomalous behavior by monitoring system and application logs [133]; and a few tools provide administrative control over application behavior [84]. However, these tools generally leave analysis of what the system is doing (or not doing), how and why, to a human administrator, who must then determine, plan and carry out the reconfiguration or repair.

In an attempt to do better, we have developed a generic framework for not just

collecting but also interpreting application-specific behavioral and performance data at runtime. We tailor this interpretation to the application and/or domain by the introduction of system models that can describe expected correct behaviors and possibly anticipate error situations (that can automatically be recognized as having occurred, or not occurred). The models may be relatively simple as well as incremental in the sense that new rules for system behavior (or misbehavior) can easily be added as they are gleaned; deep analysis and formal representation of the target system is not required, but of course would increase value if available. Further, the framework can support a (software) feedback control loop [62] to automatically decide when corrections are required, select and instantiate repair plans, and coordinate the execution (and handle contingencies) of the possibly many interdependent elements required for target system reconfiguration. Ideally, this is done with no downtime, while the system continues operation (possibly at a temporarily reduced level of service).

Our autonomic computing framework consists of four main kinds of components: sensors, gauges, controllers, and effectors—as one rendition of a reference architecture we developed together with a consortium of researchers, as explained in [68].[2] The gauges and controllers are informed by models of the target system, and thus may be themselves rather generic, with the same components usable over a range of target systems, whereas the sensors and effectors are typically more tightly coupled to the target system and/or its operating environment.[3]

Sensors[4] watch the target system to collect primitive data, while separate gauges aggregate, filter and interpret the sensor data according to system models. This

[2]The consortium included BBN, CMU, OBJS, Teknowledge and WPI as well as Columbia.

[3]It is important to note that controllers, which actively *adapt* software, is outside the scope of this thesis, but is described briefly in this chapter to provide a "holistic" view of the architecture and possible applications in which it can be deployed.

[4]In earlier research, we used the term "probe" instead of "sensor", but this terminology became confusing when considering network security, where a "probe" refers to an attacker scan of open ports.

monitoring framework can be used with or without a feedback loop that automatically performs dynamic adaptations. Without the feedback loop, gauges would typically generate alerts and/or be visualized on a human systems management console—but providing deeper understanding and more of a "big picture" of the target system's activities than earlier human-oriented systems management.

An adaptation framework, like one implemented by our research group in [160], can supplement the monitoring framework with decision, coordination and actuation capabilities. Based on the coalesced and interpreted sensor data relayed by gauges and on modeled information about the target system, a controller makes decisions on what adaptations (if any) need to be done. This triggers a controller facility to orchestrate the work of one or more effectors—which interact with the target system to carry out the low-level tweaks and tuning, and/or coarser subsystem restarts and reconfigurations, as directed by the adaptation plan.

Our implementation of the monitoring framework is called the *XML Universal Event Service*, or XUES; it, along with the Workflakes adaptation framework described in [160], served as an integral component in our larger *Kinesthetics eXtreme* framework (KX, pronounced "kicks", for short). KX, and XUES in particular, run as a lightweight, decentralized, easily integrable collection of active middleware components, loosely coupled via a publish-subscribe event notification facility. We show how XUES can be used to monitor and analyze, and direct Workflakes to repair, a variety of target applications employing models of application-level semantics, protocols and performance requirements, thus effectively achieving the self-configuring, self-healing, self-managing goals of autonomic computing—but for legacy systems and/or systems of systems, rather than applying only to new systems with autonomic properties explicitly designed-in.

Of course, our approach does not necessarily work for all legacy systems; it is limited by the degree to which the target system enables placement of sensors

and effectors and by the availability of application-specific models that can support analysis of sensor data and definition of repair plans using the effectors. Further, the configuration of XUES itself as opposed to the (distributed) target system can become rather complicated under some circumstances, and may require manual configuration. We briefly describe integration work with the ACME architectural model [50, 16] to facilitate automatic configuration and deployment; a commercial implementation could extend our model to make the monitoring system itself (partially) autonomic.

2.2 Architecture

Our Kinesthetics eXtreme (KX) architecture, shown in figure 2.1, is based on a common "reference platform", developed together with other participants in the DARPA Dynamic Assembly for Systems Adaptability, Dependability, and Assurance (DASADA) program [32]. As an externalized platform, KX is not intertwined and tries to avoid interfering with the target system's conventional functional and extra-functional communications and computations. Since the gauges and controllers typically run on separate hosts, the only direct points of possible impact are sensors and effectors; this can be ameliorated with careful placement and use of these components. The only *a priori* knowledge of the specific target system comes from *behavioral models*, e.g., architectural models describing the system, as in [49]; system-specific models must necessarily be supplied to a given KX instance in order for it to monitor and/or dynamically adapt in accordance with those models. Most runtime knowledge is collected by sensors, although it is sometimes possible for effectors to perform the equivalent of (limited) sensing duties when their repair tasks involve localized checks.

In KX, we chose to communicate among sensors, gauges and controllers solely via

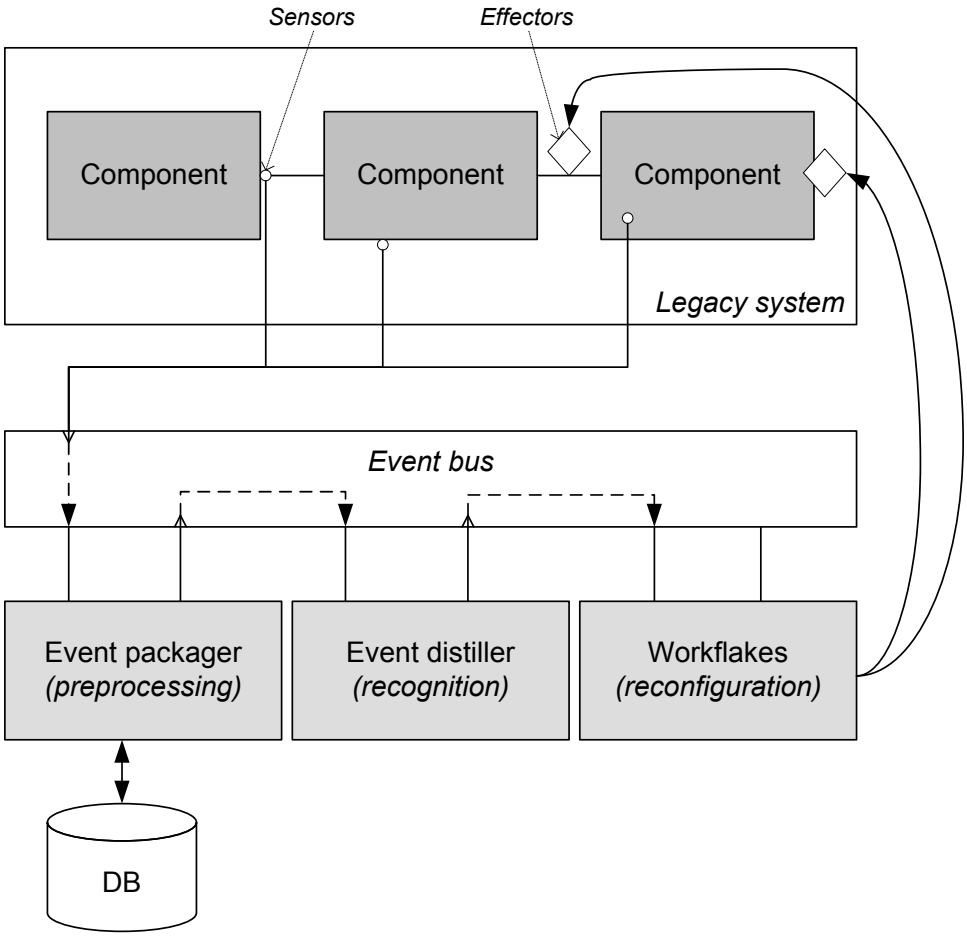

Figure 2.1: KX Architectural Overview.

publish/subscribe event notification, using content-based asynchronous messaging middleware. By leveraging event notification middleware, KX components can be easily rearranged on-the-fly, with multiple instances of KX gauges and controllers introduced as needed to address scalability requirements. Any of several available event notification systems could be used (see related work, §4.2); we chose Siena (Scalable Internet Event Notification Architecture) from U. Colorado at Boulder [20], as among the most advanced distributed event propagation systems where both source code and support were readily available. Although we have experimented with publish/subscribe communication between controllers and effectors, point-to-point communication is normally employed because the controller usually invokes

specific effectors, who must report back unambiguously to that controller to achieve coordination (although this communication may sometimes be asynchronous, as explained in [159]).

No particular sensor or effector technology is formally part of the KX infrastructure, as the best selection among potential technologies must consider the implementation details of the target system and can vary widely. We have to date used mainly our own Worklets mobile agent technology for effectors [161]. Our development of the Worklets platform was originally intended for other purposes [70] and preceded the rest of KX, but it applied nicely to the reconfiguration and repair requirements of our case studies. Other techniques we have experimented with for effectors include JMX management beans [149] and SOAP-based interfaces for synchronous remote calls.

We have employed a number of different sensor solutions developed by others. Some of these inject callbacks into source code (when source code is available and recompilation is feasible), such as WPI's AIDE [61, 51]. Others modify bytecode or binaries, such as OBJS' ProbeMeister [119]; replace DLLs or other dynamic libraries, such as Teknowledge's mediator connectors [8]; operate in the surrounding environment, e.g., to inspect network traffic, such as [128] (and chapter 6); or monitor operating system resource usage. We have also experimented with using Worklet mobile agents to deploy and modify sensors, dubbed "Probelets". Many other instrumentation technologies are described in the literature, with some commercially available.

Since the various sensor technologies do not necessarily output the event format presumed by our gauges, we introduced the Event Packager component as a preprocessor event translation service to transform into a common format. The Event Packager also removes duplicates, timestamps sensor events according to a globally synchronized clock (using NTP [98]) and acts as a "flight recorder" to

persistently log the event streams, for later replay or data mining.

The Event Distiller is our main gauge component. It performs sophisticated, possibly cross-stream temporal event pattern analysis and correlation across continuous data streams from multiple sensors, to monitor desirable and undesirable behaviors. When undesirable behaviors occur (or desired behaviors do not occur within the requisite time-bound), the Event Distiller generates meta-level events indicating this interpretation; these higher-level notification events also carry information about the lower-level sensor data that contributed to the analysis. The Event Distiller is dynamically configured with the rules defining complex event patterns of interest—that is, the behavioral models regarding what to monitor—so new models can be added and previous models replaced or removed on the fly.

KX fulfills the controller role by employing our workflow technology called Workflakes [160, 159]. Workflakes is a decentralized process enactment engine, specialized towards automated coordination of software entities as previously suggested in [175], as opposed to the more conventional use of workflow to organize human activities (e.g., [156]). Workflakes is triggered by gauge outputs to select and tailor a dynamic adaptation plan to the problem at hand, then instantiates and superintends a collection of effectors to enact the tasks specified in the workflow.

It is important to note that all of the KX components are separately usable. Depending on the problem domain, one or more of these components may be used. For example, if only a few very well-defined repair scenarios are to be performed, or if KX is being used only to do high-level monitoring and report to a human systems management console, one may choose to omit the Workflakes controller component, etc. The Event Packager and Event Distiller components, which together comprise XUES, are discussed in greater detail in the next few sections, which together with the experiments presented in section 2.5 constitute the main contributions of this chapter.

2.3 Event Packager: Adapting Events from Sensors to Gauges

The Event Packager (EP) component is architected to support a number of event input services, such as duplicate removal and persistent spooling. It utilizes a plug-in architecture to support a broad variety of incoming event formats (inputs); a variety of transformations, including timestamping; and a variety of output event formats and other options (outputs). New plugins can easily be synthesized[5]; for example, Instant Messaging (IM) messages can be represented as a form of event input.

The various plugins are coordinated via a user-definable metabase (metadata database) that dictates what should be done to the data (transforms) and where the data should be sent. Transforms can include single-event processing tasks, such as event clock/timestamp synchronization, static event reformatting/rewriting, augmentation, and selective or complete event persistence. Typically, the goal is to have a number of different input formats streamlined, spooled, and aggregated onto one event stream for the other KX components.

The Event Packager implementation (figure 2.2) was designed from the ground up to be easy to extend. Developing a new input, output, transform or store only requires that one Java class be extended, and some simple methods filled in. This enables the quick and easy creation of wrappers around existing sensors and middleware. The Event Packager uses its own opaque event format container to allow future support for new event formats without breaking compatibility with existing plugins (although for optimal performance, certain plugins might support introspection into event formats for specialized processing). By using opaque event containers, minimal per-event decision-making is needed, which enables the

[5]This includes, but is not limited to, privacy transforms, e.g., Worminator.

creation of fast pseudo-pipelined datapaths. If more complex processing is needed, a transform can be applied, although this may affect event processing speed.

We have developed input plugins that support and standardize Siena and Elvin messaging, TCP socket streams (transporting both serialized Java objects and XML messages), console input, email (via sendmail), and AOL Instant Messaging (AIM) messaging as event sources. A broad array of output formats closely mirrors these inputs. Transforms include event conversion (from Siena and flat ASCII formats) and event timestamp synchronization (to compensate for distributed clock environments).

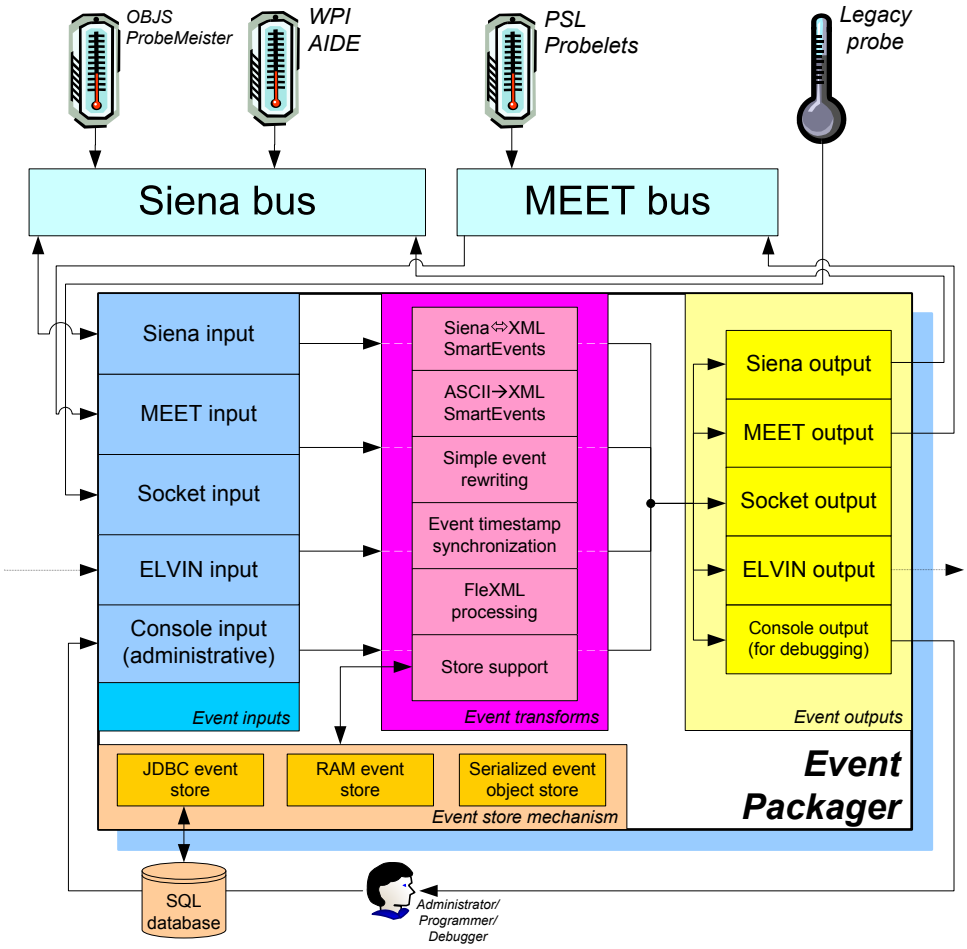

Figure 2.2: Event Packager Internal Architecture.

EP also supports in-memory, JDBC-backed SQL, and flat-file (serialized object)

stores. Persistent logging enables the Event Packager to support the reanalysis of previously-received event streams, particularly as new Event Distiller gauges are deployed. Multiple persistence techniques may be simultaneously employed via the plug-in model, so that rules can specify persistence to one or more data repositories, such as an SQL database, enabling the use of efficient offline analysis and data mining.

The above components are arranged on-the-fly. Upon startup, the Event Packager reads its configuration file and instantiates the necessary plugins and begins routing events. However, plugins can be added (via Java late-binding reflection mechanisms) and removed during runtime.

EP consists of about 9,000 lines of Java code; the core engine that coordinates inputs, transforms, outputs and stores is about 2,000 lines, while the bundled plugins to deal with input, output, transform and store comprise the rest. Some C glue code handles legacy integration. A typical rulebase is usually a few hundred lines of XML; a small example is given in the Appendix.

Flexible XML (FleXML) is an XML-based technology used by the Event Packager as an optional plugin, to intelligently convert from XML-formatted sensor output to the XML or non-XML event vocabulary expected by the Event Distiller, which uses Siena-style events consisting of unordered sequences of typed attribute/value pairs. Siena supports a naïve translation from XML data into its flat event namespace, but cannot handle hierarchical formats [44]. FleXML supports intelligent tag semantic discovery via a *Metaparser*, *Tag Processors* that can intelligently interpret and convert a XML tag, and an *Oracle* that acts as a semantic backend for the metaparser. FleXML itself is outside the scope of this thesis; see [114] for more detail. FleXML's primary use in KX was to support intelligent conversion from XML-formatted sensor output (such as generated by the Java AIDE probe generation tool [51]) to the vocabulary expected by the Event Distiller.

2.4 Event Distiller: Recognizing Event Sequences

The Event Distiller (ED) is responsible for detecting problematic or anomalous system activities by matching (gauging) sequences of events emitted by one or more sensors. An event sequence is defined here as being a nondeterministic ordering of events ultimately indicating correct vs. incorrect behavior. Such event sequences' transitions (i.e., between subsequent events) will almost always have timebounds so as to emphasize the real-time nature of the application domain and to act as a check on the state overhead needed to support nondeterministic matching.

ED rules define the event patterns of interest as derived from behavioral models, in an XML vocabulary; the full notation is given in the Appendix. Note that Event Distiller rules are not related to the Event Packager rulebase specifying plugin configuration. Each ED rule is partitioned into "states" and "actions", where matches amongst the former are mapped to (meta-)events that are emitted corresponding to the latter. This state/transition representation closely corresponds to a nondeterministic finite automaton; the idea is that one event may lead to many different possible subsidiary events and one wants to match whichever ones are appropriate. Transitions are inherently timebound, enabling temporal validation as well as a control on the size of the nondeterministic matching problem—an expiration implies that a transition is no longer possible, and ED can then garbage collect from the pool of potential matches for incoming events, to reduce the amount of system state required during execution. An alternative approach would be to provide backtracking, but this is impractical given the runtime requirements of such a system and the potentially huge number of events it may witness at any given time.

The Event Distiller internal architecture (figure 2.3) supports several additional first-level constructs as defined in the rule language:

- Rule chaining is accomplished by allowing published actions from one rule to match other rules' states. This late-binding approach enables dynamic rules to be created and to immediately support chaining.

- Looping provides Kleene star-like functionality, but can also match a specific number of times.

- Success and failure actions can be made at any matched state. A success action is published immediately upon reaching the state. A failure action is one where all the transitions from that state to another state are eliminated and no further transitions can be done, and is sent upon successful garbage collection of the current rule instance. Multiple success and failure states can be specified at each state if desired. Such actions may trigger a rule chaining within the Event Distiller architecture, may be used by other interested components (such as controllers that begin applying a repair or reconfiguration workflow), or may even trigger human notification via some immediate communication channel, such as a pager.

- Absorption enables a given state match to be exclusive, e.g., if a particular state of a particular rule enables absorption, all subsequent rules will not match that state, even if they specify the exact same criteria as the first matched rule. Note that this implies a partial ordering upon the rulebase—rules at the top have absorption capability over all other rules in the rulebase, whereas rules at the bottom can declare absorption but such a declaration has no effect.

- Variable binding enables conditional matching—a value can be bound by the first match, and further states in a rule may require that value to appear in subsequent events. This is useful for any sequence of events that refer to a common shared value, such as the name or unique identifier of a service being monitored.

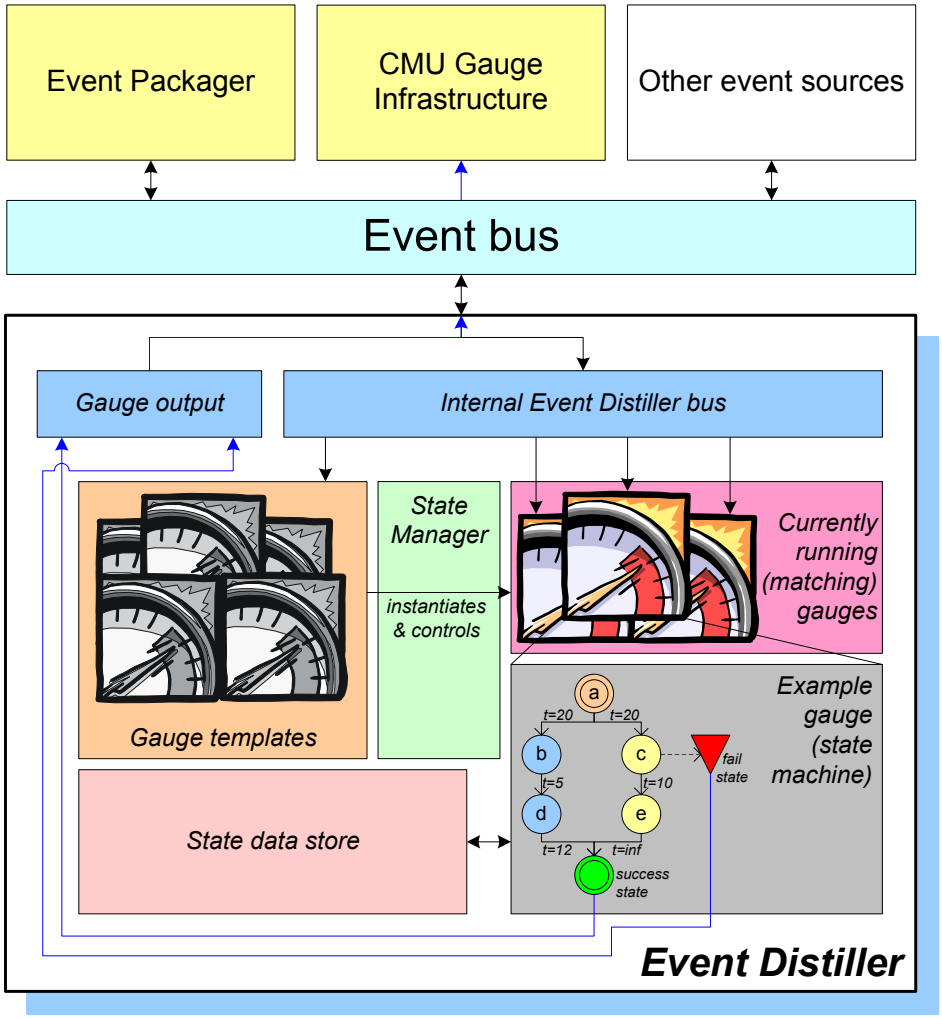

Figure 2.3: Event Distiller Internal Architecture.

Internally, the Event Distiller uses a collection of nondeterministic state engines for temporal complex event pattern matching. The rulebase is loaded into memory, and forms a series of "state machine templates"; once an event matches the first state of one of these templates, an instance of the template is automatically created to keep progress of the matching through the state machine. While this is memory-intensive, it allows a richer representation of event sequences: logic constructs are supported, as are loops, rule chaining, and variable binding as required by the architecture. Memory usage is mitigated by supporting timeouts and automatic garbage collection. Timestamped event reordering is also supported, so if events

arrive out-of-order within a certain window (1 second by default), the Event Distiller will rearrange them appropriately so that sequences, and causality, can still be recognized correctly. Note that such reordering, if done with many sources, requires some authoritative time declaration as close as possible to the sources themselves, as network latencies may be unpredictable. If the generator of the events being matched doesn't support timestamping, Event Packagers may be placed at the generation point or at its immediate network peer to create timestamps to enable reordering.

ED's repertoire of event patterns may be populated in one of several ways: First, an XML-format rulebase is supported, where event sequence patterns are specified, along with timebound parameters among sequence elements as well as success and failure notifications. There is also a GUI to assist a KX integrator; it also works as a systems management console for human engineers, although a major goal of the effort was to automate many repairs within a KX feedback loop (via notifications to Workflakes). Second, the Event Distiller supports dynamic rule generation—messages can be sent to the Event Distiller with XML snippets specifying a rule or a segment of a rule (e.g., to construct new rules on the fly or modify existing rules). Such rule modifications are received through the publish-subscribe channel, potentially containing an XML snippet that contains a full rule (e.g., states and actions) to be matched for all subsequent incoming events. Such rule changes affect templates for future matches and not currently-matching rules. Third, as with the Event Packager, other sources can be easily integrated: For instance, support for CMU's Acme architectural description language constraints [49] has been integrated: the Event Distiller can act as a "reporting gauge" onto the Acme Gauge Bus [18], thereby providing feedback to the corresponding architecturally-oriented repair tools.

The Event Distiller implementation is about 7,000 lines of Java code. The event

pattern rulebase may vary in length depending on the complexity of the behavioral model, but in experiments were typically a few hundred lines of XML.

2.5 Example Applications

In order to validate our approach to introducing autonomic properties into legacy systems, we developed several scenarios and corresponding experiments with real, deployed complex distributed software. We describe three such scenarios in this section, dealing with service failures, load balancing, and quality of service across two different legacy systems; additional applications are discussed in [160] and [120]. Because the authors were not involved in the prior development, deployment and systems management of these real-world systems, it is not possible to offer full "before" and "after" comparisons with detailed quantitative data. However, these case studies demonstrate that it is possible and advantageous to apply our approach to real-world legacy systems. To show the diversity of plausible application domains, we also discuss a simple "toy" example that detects and blocks email spam. Finally, several of these applications and their privacy concerns are reexamined in section 2.7.

2.5.1 Service Failures

We integrated KX with a complex GIS (Geographical Information System) intelligence analysis tool developed at the USC Information Sciences Institute (ISI) and used experimentally at the US PACOM (Pacific Command), known as GeoWorlds [27]. GeoWorlds uses a distributed set of services glued together by Jini [150]. While the system generally works well, services sometimes stop running, with no recourse except to wait for the request to time out and to manually restart the appropriate backend subsystem. For example, GeoWorlds' reliance on harvesting external

websites (e.g., www.bbc.co.uk)—for news items that it then maps to locations in the GIS system—is subject to frequent glitches (DDoS, server failure, etc.), requiring restart of the GeoWorlds service that is trying to access the external website, possibly substituting an alternative news site.

Using WPI's AIDE (Active Interface Development Environment) sensor technology [61], we were able to automatically instrument the GeoWorlds Java source code, and in particular the mechanism that dealt with request-to-service dispatch, with sensors that would monitor the start and end of method calls that were relevant to contacting external services. The Event Distiller incorporated rules to monitor a variety of method calls, making sure that a "termination" call matched up with each "initiation" call within an appropriate timebound (ranging from seconds to a minute). AIDE reports method calls in an XML format; these calls were then translated to a simple attribute/value set via the FleXML Metaparser and fed into the Event Distiller.

```
<state name="Start" timebound="-1" children="End" actions=""
       fail_actions="">
 <attribute name="Service" value="*service"/>
 <attribute name="Status" value="Started"/>
 <attribute name="ipAddr" value="*ipaddr"/>
 <attribute name="ipPort" value="*ipport"/>
 <attribute name="time" value="*time"/>
</state>

<state name="End" timebound="15000" children="" actions="Debug"
       fail_actions="Crash">
 <attribute name="Service" value="*service"/>
 <attribute name="State" value="FINISHED_STATE"/>
 <attribute name="ipAddr" value="*ipaddr"/>
 <attribute name="ipPort" value="*ipport"/>
 <attribute name="time" value="*time2"/>
</state>
```

Figure 2.4: Failure Detection Pattern.

Figure 2.4 shows an example of a simple event pattern used to perform such

failure detection. The incoming sensors reporting Status and State values track method completion. If for some reason a "FINISHED_STATE" is not received within 15 seconds after a method had initiated, the system sends out the "Crash" event; otherwise, the "Debug" notification is emitted, signifying a "success" and acting as a record of the operation for future debugging purposes if desired. Note that the strings prefixed with an asterisk ("*") designate a variable binding, e.g., the Event Distiller substitutes all instances of "*service" by the first source that it sees for this instance of the rule. Thus, this one rule can match a large number of different sources and subjects.

When the repair system (Workflakes) received a "Crash" event, the repair involved a simple restart of the service as specified in the message generated by the Event Distiller. A more sophisticated repair (which was not implemented) would have coordinated multiple services to prevent having to restart a long transaction from scratch, instead using partial results leading up to the individual service failure. Even with the simple repair, however, we were able to automate a process that previously had been done manually.

2.5.2 Load Balancing

Several GeoWorlds execution scripts rely on computationally-intensive backend services hosted at ISI, such as a noun phraser that would analyze incoming news articles and extract nouns for mapping to GIS attributes; crash avoidance and performance maximization through request relocation was clearly desirable. To accomplish this, the relocatability of Jini services was exploited to build a load-balancing solution for GeoWorlds. A system monitor sensor was built in C# to measure the overall load on the backend system(s) running the noun phraser, and results were piped into a custom plugin for the Event Packager.

CMU's Acme architectural description language [16] was used to specify Ge-

oWorlds' architectural composition and system load constraints on the various services. The Acme Gauge Extractor then generated Event Distiller rules based on these constraints. During the execution of various services, if this load exceeded a predetermined threshold for an extended period of time, the Event Distiller detected and reported it as a violation of the architectural constraints. The triggered repair caused the service to move to a different Jini-enabled host. We were able to visualize the load and service state using AcmeStudio's architectural diagram visualization tools [17], so one could watch the feedback loop in action, in concert with the architectural model.

Additional logic was programmed into the Event Distiller rulebase to detect oscillation, utilizing the feature whereby ED can listen to itself (more generally, a hierarchy of EDs could be constructed to analyze meta-level events). In particular, if many (outgoing) Event Distiller messages requesting a load-balance were detected within a short timespan, one of two strategies could be selected: either eliminate load balancing between the two oscillating hosts for future repair plans (by notifying the Workflakes controller), or increase the load threshold in the architectural constraints (either by patching ED's own rulebase or by manipulating the Acme constraints used to generate the rulebase). We implemented only the former, but the latter approach can also be implemented given appropriate API exposure on the part of AcmeStudio.

2.5.3 Quality of Service

We had the opportunity to experiment with a commercial J2EE-based multi-channel Instant Messaging (IM) service used daily by thousands of real-world end-users. First, on-demand scalability was added: by monitoring user sign-on events and server request queues, KX was able to determine the load of each member of the IM server farm and take appropriate actions whenever needed. Repairs, selected

on the basis of inferences carried out using Event Distiller rules, encompassed modifications to the threading model of active servers, or even on-the-fly deployment and activation of additional server instances and corresponding reconfiguration of the commercial load-balancer of the IM server farm to redirect client traffic to these new servers. Failure detection was also supported from a load-balancing standpoint: information on server failures and interconnections between servers and backend DBMS entities was similarly captured to facilitate load balancer reconfiguration to direct client traffic to still-functional servers. The same set of sensors and effectors, coupled with slightly different Event Distiller gauge rules and Workflakes repairs, were also used to support controlled and graceful staging of the service infrastructure; this enabled automated software release deployment without necessitating a complete shutdown (and service interruption) during transitions.

A set of quantitative results were derived from running and observing the adapted IM system in lab conditions, with both manual and automated traffic simulation that reproduced in-the-field demands on the IM service. These results focus on the improvement via automation in the support, maintenance and management activities typically carried out on the IM service under field conditions. Also, some measurements about the development and integration effort necessary to implement the case study were taken. The most significant results are:

- Substantial reduction in effort for deployment and configuration of an IM service in the field, originally approximately person-days, with locally present experts. With KX, that was reduced to 1-2 minutes from a remote location.

- Reduced monitoring and maintenance effort necessary to ensure the health of the running service. KX completely automated the 24/7/365 monitoring of a set of major service parameters, as well as the counter-measures to be taken for a set of well-known critical conditions.

- Reduced reaction times and improved reliability: for example, KX recognized the overload of an IM server in 1-2 seconds and deployed an additional server replica in approximately 40 seconds. Overload detection was originally manual, starting with accumulated application logs—a clearly error-prone approach, potentially endangering service availability.

- Manageable coding complexity: KX sensors, gauges and effectors were derived from generic code instrumentation templates that were then customized with situational logic. This resulted in rather compact code: 15 lines of Java code on average for sensors, and usually less than 100 lines for effectors. The total code written for this specific case experiment on top of the generic monitoring and dynamic adaptation facilities provided by the KX infrastructure was approximately 2,000 lines of Java and XML.

This study demonstrated the utility of a KX end-to-end feedback loop for service management and application-level QoS in an industrial context. Traditional application management practices report warnings, alarms and other information to some knowledgeable human operator who can recognize situations as they occur and take actions as needed—with very limited automation in the management platform. Instead, our approach offers a high level of guidance, coordination and automation to enforce what is a complex but often repeatable and codifiable process.

2.5.4 Spam Detecting and Blocking

In order to demonstrate KX's flexibility beyond the more conventional system management cases above, we instrumented Sendmail [137], a popular email Message Transfer Agent (MTA), to capture messages being received in a target network. More precisely, a Sendmail milter [136] was created and installed to capture incoming traffic. Specific attributes about each message (such as source address, subject,

and Message-ID) were captured by sensors, encapsulated into events by the Event Packager, and sent to the Event Distiller. The Event Distiller rules (figure 2.5) would trigger if multiple (3+) messages containing the same source and Message-ID were received, by one or more recipients in the administrative domain, within a very short timespan (less than 10 seconds). Once detection occurred, a mobile agent effector was dispatched to reconfigure the Sendmail MTA in the target network to block all further messages from that source address by rewriting the configuration file and sending a hangup signal (SIGHUP) to Sendmail to reload its configuration.

```
<state name="a" timebound="-1" children="b">
 <attribute name="from" value="*1"/>
 <attribute name="messageID" value="*2"/>
</state>
<state name="b" timebound="100" count="1" children="" actions="A,B"
       fail_actions="F" absorb="true">
 <attribute name="from" value="*1"/>
 <attribute name="messageID" value="*2"/>
</state>
```

Figure 2.5: Sample Pattern to Detect Repeated Emails.

This solution worked for simple spam—i.e., one message sent by a spammer to sufficient people in the same organization would verifiably get caught and future communication from that spammer would be blocked. Of course, the organizational newsletter might also be blocked. While this technique is superseded by better spam-specific technologies, such as SpamAssassin [7], which uses dynamic rules and Bayesian learning to distinguish more "stealthy" spam, this example demonstrates the broad utility of our Event Distiller's timebound-based pattern matching, in this case with email-specific semantics. In essence, we were able to add (limited) autonomic behavior to Sendmail.

2.6 Summary

We have presented a general approach to software monitoring via event processing, enabling the retrofitting of autonomic capabilities onto pre-existing systems designed and developed without monitoring and dynamic adaptation in mind. Our KX architecture provides sample components for gauges (Event Distiller) and controllers (Workflakes), as well as additional components (Event Packager and FleXML) that act as adaptors for diverse sensor technologies developed by others. KX, and XUES in particular, has been used to add self-management and self-healing functionality to several legacy systems and systems of systems, spanning service failures, load balancing, quality of service and an experiment in spam detection and blocking.

Further, most of the behavioral models employed to date have been relatively ad-hoc, based on the authors' understanding of the external behaviors of each system as opposed to formal models—which are difficult to obtain for many real-world legacy systems. However, preliminary work with *a posteori* architectural models of GeoWorlds, defined in an architectural description language, shows long-term promise; this concept is discussed further in section 7.3.

2.7 Privacy Preservation

The implementation and applications described in sections 2.2 and 2.5, respectively, did *not* use privacy-preserving mechanisms; they were limited-scale or closed-network deployments, and maximum event expressiveness was desired. We can envision a number of extensions, however, that demand privacy:

- A broader service failure detection system than that described in section 2.5.1 may collect information on the same process from many different organizations, either for the benefit of the software developer or for decentralized debugging

and response strategies. For example, server crash reports could be correlated between different organizations' networks to see if there is a pattern triggering them. Some research has already been conducted in this regard; *application communities* [90] can be viewed as a large-scale application trace corroboration problem. More precisely, an application community is a decentralized network of nodes who all run the same application; this homogeneity is leveraged to distribute instrumentation, minimizing overhead for any one node, while providing the potential for aggressive response mechanisms for *all* participants if one node experiences anomalous behavior.

However, such information may be subject to privacy policies, as they may contain instance/organization-specific information. Transmitting a privacy-transformed version may be more palatable. A potential future extension of this work to support application communities is discussed in section 7.3.

- If a quality-of-service problem was extended to the P2P domain, different administrative domains would be required to exchange information on login patterns and server availability. Transmitting such information without any security could enable any curious participant to glean significant amounts of information about service deployment, and more seriously, weaknesses in the system, which could then be exploited via a denial-of-service or other form of attack. Instead, sharing privacy-transformed representations of the system may enable response and rebalancing strategies without having to reveal specific entities and/or their limitations and vulnerabilities.

- Distributed spam detection is an increasingly common application as a means to combat the fact that spam networks distribute near-identical spam to many different domains. Tools such as Vipul's Razor [163] already implement a simple form of privacy-preserving distributed spam detection; they exchange

message hashes, and suspicious hashes may be reported, which essentially populates a blacklist to combat spam. Spam has evolved, however, to introduce randomness in the message to defeat these techniques. More advanced privacy-preservation transforms may be implementable to encourage collaboration and gain a better global picture of the spam problem—both at the message and the spammer's network level.

Adding such privacy mechanisms also enables the development of *new* applications for XUES.

- Intrusion detection can be viewed as a software monitoring problem, specifically for *malicious* behavior. This application is a core privacy-enabled correlation scenario in this thesis, and is discussed in great detail in chapter 6. In fact, the Worminator architecture is derived from the Event Packager, and leverages its pluggable mechanisms to add privacy-enabled event types, with conversion facilities from "raw" intrusion detection alerts. In addition, the use of publish/subscribe as a fundamental communications model enabled interest-based collaboration and eliminated dependence on source identity in event distribution.

- *Document flow analysis* is interested in enabling the distributed detection of "documents of interest"; this can include both documents which contain malcode (and whose spreading may suggest a worm or a malware attack) *or* documents which contain confidential information (and thereby suggesting a content leak). This application, and an extension of Worminator to support it, is also discussed briefly in section 7.3.

The key idea here is that all three applications may transmit events amongst a broad array of interested subscribers but whose content may not be disclosable. For instance, a "confidential document" may contain ideas which may not be

disclosed, even in the documents' filters. While we would like to use the correlation capabilities outlined in this chapter, they cannot be directly applied—instead, a *privacy-preserving* version must be used. I discuss specific privacy transformations to support such applications in chapter 5, and as previously mentioned, the realized architecture in chapter 6. Of course, not all existing event correlation tools may be trivially migratable to such a privacy-enabled version, and so I also discuss the notion of *retrofitting* privacy preservation onto existing event correlation systems in section 5.4.

Chapter 3

Model

In this chapter, I discuss the basic corroboration model and the abstract architecture necessary to realize it. The model features three main components: a typed event model, type-driven event processing and correlation middleware, and a publish-subscribe event infrastructure. This thesis focuses on building solutions for the first two components, and supports integration with existing publish-subscribe architectures. Figure 3.1 shows a high-level overview.

As the diagram illustrates, the system runs on top of and connects to an event producer; see figure 3.1. These events are read, converted to a standardized type (§3.1), and a privacy-preserving transform of the data is applied (§5.1) before it is anonymously published (§5.3) via event middleware[1]. Meanwhile, anonymized events are received from other sources and are corroborated, with temporal constraints (§5.1.4), against existing local event data by applying the same privacy-preserving transforms. Finally, corroborated events may optionally be transmitted back to peers in raw form to enhance their source data set for further corroboration (§5.1.6).

[1]The model supports anonymity even if the event middleware is not designed for it, as described in section 5.3

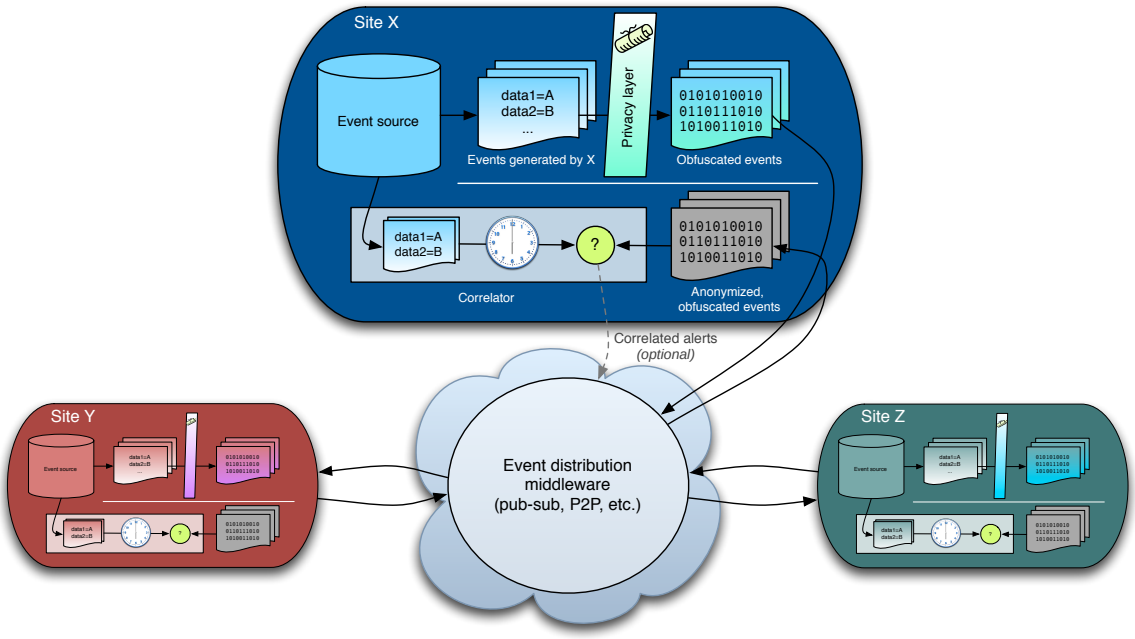

Figure 3.1: High-level view of model.

3.1 Event Model

Type-driven event models are not new; [92] suggests that anything from XML to Java objects can be used for typing. Sun's Java Message Specification (JMS; see [151]) is a event message broker that supports the transport of arbitrary Java types. In fact, the model specified here builds on top of standard OO type structures.

However, a distinction is made as the type hierarchy in this approach, as shown in figure 3.2, directly contains privacy-preserving mechanisms; additionally, metadata associated with such events supports heterogeneous correlation. I describe each of the types in greater detail below.

- **Abstract event.** The abstract event type at the top of the hierarchy represents a common underlying event type, and is characterized by metadata that apply to all event types in the model. The more important among these include the following:

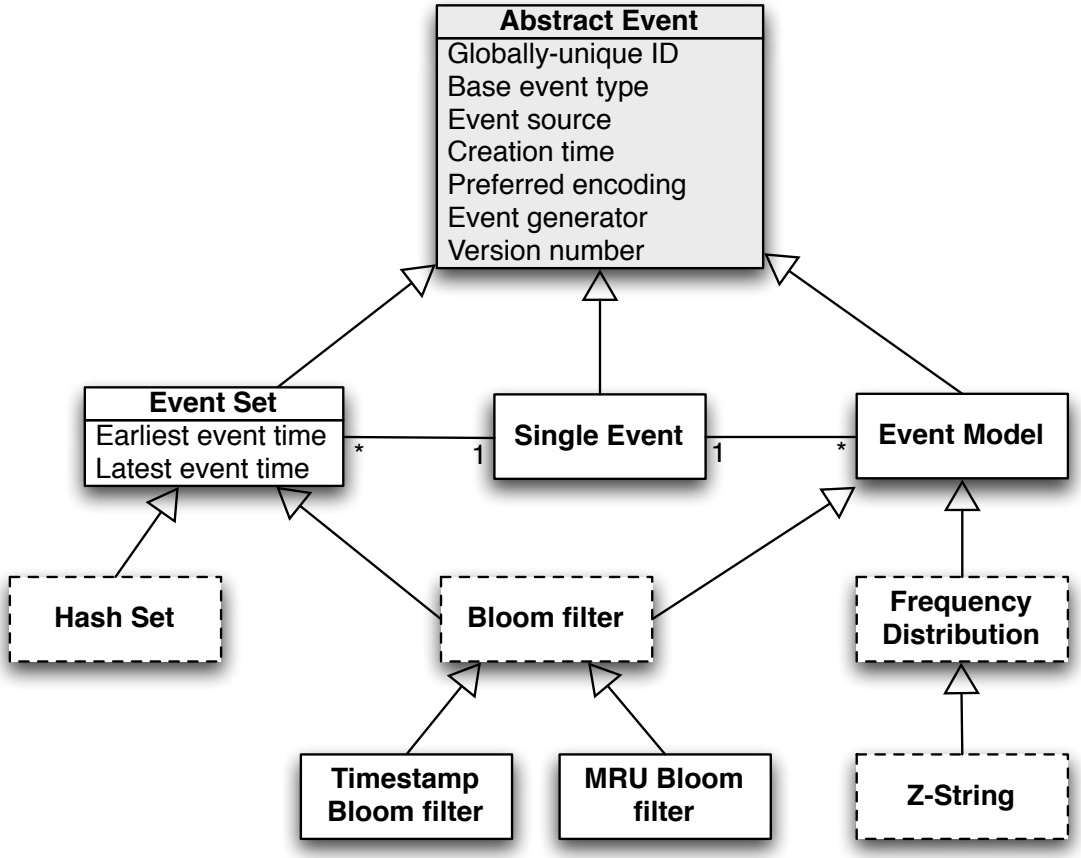

Figure 3.2: Base event type hierarchy. The shaded class represents an abstract type, while the dashed-border classes represent privacy-preserving types that may be exchanged in corroboration.

- A globally unique identifier. Psuedorandom uniform ID (UID) generators may be used here, such as [60], or the event distribution infrastructure may itself introduce semantics for identifiers. It is critical that such identifiers not be tied to the data or the source of the data.
- The base type of the event(s). This is needed to help the correlator determine if it can support the heterogeneous corroboration of a privacy-typed event against a given raw event. (For a non-privacy-typed event, this will describe the type of data contained in the event; non-privacy-typed lists and sets typically leave this blank and let the constituent events report their own base type.

- The source of the event. As discussed in section 5.3, this is typically a single identifier, but may contain different information based on disclosure restrictions.
- The timestamp(s) of the event. As a minimum, this refers to the generation time of the event itself. As implied in the introduction, there can be more, e.g., the creation time of the event, the timestamp of the earliest embedded event, and the timestamp of the latest embedded event. The latter two are useful in supporting temporal constraints for a list or set.
- A preferred encoding. This is primarily for implementation purposes, especially when standardized wire exchange formats (e.g., [35]) are used—most contain an "other data" or "miscellaneous" field which is necessary to transport privacy-transformed data structures, as most of the data remains opaque to the format itself.
- The generator of the event. Depending on privacy concerns, this may not be filled, but in general this refers to the class of event producer. This is *not* equivalent to the source of the event; the generator typically refers to the piece of software (e.g., sensor) whereas the source refers to the organizational identity of the producer.
- The version number of this event type. The model supports a primitive versioning scheme, allowing for type evolution and runtime type changing. The primary intention of this versioning is to ensure participants keep types synchronized; this is especially significant when changing the choice of privacy-preserving mechanisms based on evolving privacy requirements, in which a event distributor or correlator may reject versions deemed too old to use for safe exchange. This thesis does *not* generally address type versioning, although the model does allow for more flexible schemes; possible future versioning extensions are discussed in section

7.3.

- **Single events.** This is the core event type, representing one event instance. In addition to the baseline, raw event, the model supports at least three privacy-enabling mechanisms: sanitization (scrubbing of relevant fields, no data transformation), hashing, and frequency transforms. Sanitization is not new research, and is not considered further in this thesis.

  Hashed structures are optimal for events that need precise matches (either whole or partial). Frequency transforms, on the other hand, are particularly useful when considering large, opaque events on which aggregated statistics are effective. A more detailed discussion and comparison on these techniques is in section 5.2.

- **Event lists/sets.** A number of event exchange standards, such as the IETF intrusion-detection alert exchange standards [31, 35], allow the transmission of more than one event per communication by concatenating them into a single event, implying a logical grouping. The model discussed here supports these by treating them as a form of meta-event that can be created by packing individual events or unpacked into individual events, which are then treated as per the events described above.

- **Event models.** Last, a number of data structures, including both Bloom filters and frequency models, can *represent* multiple events without literally embedding them as per a list or set. This is particularly relevant to privacy-preservation, as the construction of a model creates a representation but destroys the underlying actual sensitive data. A model may represent something as simple as a list of individual events or can be a more comprehensive *profile* of a publisher of events over time. While a model is not colloquially considered a single *event* in time, there is no reason a model cannot be transmitted

as one; it has metadata, including a timestamp, and individual events can be correlated against a model to see if expected behavior or patterns match. The next section shows, abstractly, how models can be used in corroboration, while chapter 5 discusses design challenges, including effective temporal corroborations when having many events in a single model.

Many of these models can also be made privacy-preserving via the use of frequency distributions or Bloom filters; one such application, supporting intrusion detection in Mobile Ad-hoc NETworks (MANETs [132]), is described in section 6.6.

This typology maps nicely to a object-oriented type hierarchy as per figure 3.2. Given the ability to do type introspection, a correlator can quickly differentiate between a hash verification-based corroboration as opposed to a frequency analysis or a more generic predicate-matched rich event correlation operation, and allow or deny appropriate correlation mechanisms.

3.2 Corroboration

The corroboration model presented in this thesis is formalized as follows. Envision a set of participants $A, B, C, \ldots$ each with a set of events, i.e., $\mathcal{E}_A, \mathcal{E}_B, \mathcal{E}_C, \ldots$, that they wish to corroborate with each other. Each set of events $\mathcal{E}$ contains many individual events e, e.g., $e_{A_1}, e_{A_2}, \ldots, e_{A_n}$ for participant A.

A non-privacy-preserving corroboration *done by participant A*, $C_A(\mathcal{E}_B, \mathcal{E}_C, \ldots)$, is a set intersection with the events $\mathcal{E}_A$ with the supplied event sets. Note that the subscript notation next to C explicitly designates a *participant* to do the corroboration, whereas the parameters are event sets. In other words, here A is the *subscriber*, or recipient, of sets or lists of events *published* by others. We assume that A is privy to its own events. C essentially computes a set intersection with what has been

provided; for example, if A was the subscriber and B was the (only) publisher, we denote that as

$$C_A(\mathcal{E}_B) = \mathcal{E}_A \cap \mathcal{E}_B = \{e_{A_1}, e_{A_2}, \ldots, e_{A_n}\} \cap \{e_{B_1}, e_{B_2}, \ldots, e_{B_n}\}$$

yielding $\mathcal{E}_{A\cap B} = \{e_{A\cap B_1}, e_{A\cap B_2}, \ldots, e_{A\cap B_n}\}$.[2]

However, since each event e may contain sensitive data, we would like to develop a privacy-preserving corroborator $C'_A(\mathcal{E}'_B, \mathcal{E}'_C, \ldots)$, where $\mathcal{E}'$ either:

- Is a set of *privacy-transformed events* e'. This presumes the presence of a *privacy transform* $p(e) = e'$, and by extension, $\mathcal{E}' = \mathcal{P}(\mathcal{E}) = \{p(e_1), p(e_2), \ldots, p(e_n)\}$.
- Is a *privacy-preserving model* of events $e_1, e_2, \ldots, e_n$. This presumes the presence of a *model creation function* $\mathcal{M}(\mathcal{E})$, where $\mathcal{E}' = \mathcal{M}(\mathcal{E})$, and a *similarity metric* $\mathcal{S}(e, \mathcal{E}') \rightarrow [0, 1]$ which determines whether or not an individual event e *conforms* to the model (and returns a *score* ranging from 0 to 1, where 0 implies an event completely dissimilar and 1 completely conformant).

As the above implies, $\mathcal{E}'$ may not actually be a set of events, unlike $\mathcal{E}$; a model, while incorporating information about the corresponding set, does not have to explicitly reference each event individually. $\mathcal{E}'$ is deliberately overloaded to connote the fact that *either* representation may be published, depending on the privacy technique used, and they are distributed in the same fashion and using the same event type structure as discussed in section 3.1.

Given this pair of transforms, we can now write out a first attempt at privacy-

[2]Note that the indices i on events $e_{A\cap B_i}$ do not correspond to the events from individual sets. We could redefine the terminology as $e_{A_i\cap B_j}$, but this is not critical for our notation, and so choose the simpler approach.

preserving corroboration if $\mathcal{E}'$ is a event set:

$$C^*{}_A(\mathcal{E}'_B) = \mathcal{P}(\mathcal{E}_A) \cap \mathcal{E}'_B = \{e'_{A_1}, e'_{A_2}, \ldots, e'_{A_n}\} \cap \{e'_{B_1}, e'_{B_2}, \ldots, e'_{B_n}\} = \{e'_{A \cap B_1}, e'_{A \cap B_2}, \ldots, e'_{A \cap B_n}\}$$

While this is technically correct, and is certainly privacy-preserving, it is also not useful: we are given a set of *privacy-transformed* events in return. These events, by definition, are irreversible to the original objects, so we cannot make anything meaningful of them (apart from $|\mathcal{E}'|$). Additionally, this approach clearly does not work for a model $\mathcal{E}'$. What we *actually* want is the following, a special form of intersection that selects the raw objects from A's local event list that match the set or model:

$$C'_A(\mathcal{E}'_B) = \begin{cases} \{e_{A_i} \mid p(e_{A_i}) \in \mathcal{E}'_B\} & : \quad \mathcal{E}' \text{ is a set} \\ \{e_{A_i} \mid \mathcal{S}(e_{A_i}, \mathcal{E}') > \tau\} & : \quad \mathcal{E}' \text{ is a model} \end{cases}$$

C' is the key *privacy-preserving corroboration* operation. In the case of model exchange, τ is an empirically-set threshold. I describe appropriate thresholds for the applications discussed in chapter 6. The set of privacy and model transformation functions $p, \mathcal{M}$ used in this thesis are discussed in section 5.1.

3.3 Pluggable, Event Type-Driven Middleware

Given a standardized, type-driven event model and a corroboration scheme, I now discuss the key pieces of the software framework.

- **Type modules.** Each of the above types are translated into first-class objects. The system in chapter 6 implement these types as a collection of Java interfaces and classes. In addition, *translation facilities* are written to support conversion from one event format to another as necessary and when possible. For example,

during the transmission phase, events are converted into an instance of a standardized messaging event, which then has code to serialize itself into a compatible wire format when being sent. This translation facility is defined at the abstract level and made concrete at lower levels where appropriate. These facilities include the conversion of non-privacy-typed events to privacy-typed events.

- **Transform modules.** In the cases where types cannot be directly converted to each other (such as model creation $\mathcal{M}$), transform modules are custom-written. In particular, a number of *aggregators* are used when converting single events into lists or sets. Configuration directives dictate appropriate aggregation intervals (e.g., based on time or volume of event flow).

- **Correlation modules.** This is the heart of the framework: type-driven temporal correlation. Depending on the scenario, a custom correlator can be developed as its own standalone module, or a general rule-based correlator can be adopted. Examples of general correlators include our Event Distiller (chapter 2) or a third-party solution (see section 4.1 for examples). Third-party solutions that are unaware of how to handle privacy-preservation can be used with a retrofitting layer, as discussed in section 5.4. Retrofitting has its limitations; ideally, a privacy-aware correlator would be used.

- **Other modules.** Finally, other modules are present to enable communication in and out of the framework, including mechanisms to connect the framework to the event publish/subscribe architecture, to potential backend databases, and even to reporting/frontend modules. These are essentially middleware "glue" and are not elaborated upon further here.

These modules are tied together by a runtime middleware engine that provides module lookup, event routing and conversion facilities as needed. Ideally, this

engine should be written on a platform that supports late binding and/or dynamic libraries, to support the installation of new type versions and correlation modules during runtime. The implementations discussed in this thesis, including the Event Packager in section 2 and the Worminator engine in section 6, were written in Java, which support the runtime installation of both.

3.4 Publish/Subscribe Event Infrastructure

As stated earlier, this thesis is *not* concerned with implementing a new communication substrate. However, the notion of an event-driven publish/subscribe infrastructure remains intrinsic to the model, and in addition to the anonymity and authentication issues as previously discussed, other considerations must be taken into account.

Briefly, publish-subscribe is a communication model whereby nodes (*subscribers*) explicitly *join* a communication network, and then receive events of interest from *publishers*, which may be a subset of the subscriber pool or a completely separate pool, without explicit source or destination identifiers. Underlying nodes *route* messages by using appropriate Internet communication mechanisms, ranging from IP multicast to far more sophisticated techniques in an overlay network.

Both subscribers and publishers participate with only minimal information about the underlying network; in particular, the address of a routing node that is already in the network, possibly credentials to authenticate themselves, and optionally declared *interests*, e.g., a pattern that matches the events that they would like to receive or a predetermined "channel" in which events of interest are being discussed. Given this information, routing nodes may adopt a number of techniques to optimize communication efficiency, such as content-based routing (CBR) (discussed in section 4.2).

From the perspective of this thesis, several features of any chosen substrate are desirable. At the same time, any implementation should do its best to work without these features, so as to support generic applicability to a broad variety of applications.

- **Event type compatibility.** Ideally, the infrastructure is type-compatible with the definitions in section 3.1, as that minimizes the effort to marshal data to and from different formats, and simplifies communication workflow. JMS, for instance, supports arbitrary Java datatypes and can be easily integrated.

  However, most pub/sub infrastructures do not offer generic typing facilities. For instance, Siena [19] supports primitive-typed attribute-value pairs. As a worst-case scenario, the model should support exchange via primitive String-based datatypes, serializing more complex types into a publishable form.

- **Event latency and ordering.** Again, an ideal pub/sub infrastructure minimizes latency and guarantees ordering. However, anonymity and distributed mechanisms increase latency and may, in a P2P deployment, disrupt ordering. In order to effectively correlate, the proposed framework must support a *reordering* mechanism, which would employ a sliding window and reorder incoming events by their generation timestamps. The implementation of Event Distiller, in section 2, features such a mechanism. If these timestamps are absent, reordering becomes infeasible and correlation must be performed in a non-order-specific manner, such as set intersections.

- **Encryption** is one last optional service to be offered by a pub/sub infrastructure (optional, as privacy-preserving transforms can act as a "shared secret", but potentially important if authentication, metadata and related communication should be protected). Ideally, a pub/sub infrastructure supports distinct per-

channel permission and encryption models, enabling distinct simultaneous correlation groups and enabling scenarios where certain classes of peers may not be able to interoperate with each other. However, many CBR-based infrastructures do not support encryption beyond basic link encryption (e.g., SSL) due to the significant cost of decrypting/encrypting data at each node to make routing decisions.

Very few publish/subscribe systems offer all of the above features, plus a robust authentication, anonymization, and distribution infrastructure. Nevertheless, by layering the proposed framework on top of a third-party infrastructure, different infrastructures can be adopted based on the requirements of the collaboration environment. Notable related and applicable work in this field, including several projects done here at Columbia, are discussed in greater detail in chapter 4. A brief comparative evaluation is included there for their relative applicability to this thesis.

3.4.1 Distribution and Timestamping

As discussed briefly in the introduction, one of several timestamps can be used during the generation and distribution of an event in order to support, ideally, the global ordering of events. Depending on the event typing used, the semantics of a timestamp may vary–especially in the case of event sets or models, which can contain many individual events.

Ideally, timestamps are assigned upon event creation. This enables more precise temporal corroboration. In the case of an event model, this timestamp may act as an upper bound (i.e., the time at which all events have been inserted and the model is ready to be published) or both lower/upper timebounds may be included (i.e., individual events' timestamps are scanned and appropriately aggregated into the model's metadata).

If this timestamp is not available, then an implicit timestamp must be created. This is possible at several different points: at the point of distribution, i.e., when the event is sent to the distribution framework, or upon receipt by the subscriber of the event. If the distribution frameworks' clocks are synchronized (either in an absolute or logical [83] sense), this alternative produces similar results to having the producer timestamp the clock. The received timestamp may work sufficiently well in the case of a synchronized, low-latency publish/subscribe network. In a more distributed/P2P environment, however, this may lead to ordering errors. In the case of the model, these timestamps should represent the original event ordering as much as possible.

The last alternative is to corroborate without timestamps. This reduces the problem to one of set intersection. Having accurate timestamp data, however, enables accurate temporal constraints on correlation, which reduces state maintenance in a correlator (as described in chapter 2).

Chapter 4

Related Work

This thesis and the implementations discussed herein relate to four main subclasses of work in the software engineering and network security/intrusion detection fields.

4.1 Event Correlation

Event correlation is a difficult field to summarize, as nearly every specialization in Computer Science has its own event correlation techniques. I confine my focus here to some of the seminal event correlators in the software engineering and networking communities; some intrusion detection-specific event correlation papers are discussed in section 4.4.

- Luckham et al. describe *Rapide*, an "event-based, concurrent, object oriented language" for describing event patterns to determine event causalities and Root Cause Analysis (RCA). Rapide has been employed for a broad variety of applications; [93, 94] discusses the most important context, which is the enabling of specification and analysis of system architecture. It enables declarative programming of system and communication architecture for system monitoring and simulation. [92] is a book by the same author describing

Rapide as an event pattern language, and supporting a broader array of event-driven "Complex Event Processing".

- Yemini et al.'s work on *DECS* [181], or Distributed Event Correlation System, describes techniques for very high-speed event correlation for network and distributed application management. The "codebook approach" essentially decomposed events into "codes" in bit vectors, enabling very high-speed lookup and near real-time performance. DECS is paired with the Netmate network modeling tool [41]; they used this architecture to model a distributed database application for decentralized network problem detection (e.g., communication failure).

There are many other event correlation approaches, but virtually all of them (with a few notable exceptions, discussed in section 4.6) assume access to raw event data, and the associated rule languages assume the ability to perform predicate tests on whole events. Rapide provides a very general, flexible approach to event correlation, including time constraints, but does not include any privacy-preservation; in theory, an implementation of this thesis model could be built as a framework on top of Rapide (as it provides a general programming language), but the project is currently unmaintained [6]. Yemini et al. use a more compact and fast representation, and their transformation to bit vectors shares some similarities with the data structures used in section 5.2.2, but do not embed privacy-preserving transforms.

Event correlators have become ubiquitous enough that general open-source solutions exist, e.g., [158, 13].

4.2 Event Distribution

While this thesis does *not* focus on the network aspects of event distribution per se, it does depend on the use of a reliable, scalable, and anonymity-capable event

distribution architecture. I describe several relevant genres of distribution systems, including Content-Based Routing (CBR) and Peer-to-Peer (P2P) systems, which would be appropriate for the requirements discussed in chapter 1. Such systems can be viewed as providers for the proposed infrastructure, as the model would essentially "ride" on one or more of these systems.

Peer-to-Peer Systems

Peer-to-peer (P2P) systems are an increasingly popular decentralized communication platform for a wide variety of applications, including event-based systems. They offer resiliency where centralized systems do not, and can also be employed to provide anonymity guarantees where no one party has enough information to determine the identity of a collaborating participant. These systems fall into several categories:

- **Storage and lookup.** Chord [144] is a popular distributed lookup protocol, essentially guaranteeing a reliable assignment of hash keys to nodes in a participating network, which may choose to anonymously store data against these hash keys. It has been used for a wide variety of applications, from distributed hashtable (DHT) storage to distributed intrusion detection. Similarly, Freenet [23] pools storage space amongst a large set of nodes, and can be employed to ensure that the source cannot trivially be traced without a very large number of cooperating malicious entities in the cloud.

- **Cryptographic routing.** Onion routing [54] provides anonymity guarantees in a P2P system using efficient cryptographic techniques; essentially, the data structure is layered like an onion, with a progressive set of cryptographic keys that must be decrypted as the message is routed. At the same time, this layering provides guarantees of confidentiality and integrity via the use of

probabilistic memoryless random walks.

- **P2P Distribution/Dissemination Architectures.** Other distributed P2P infrastructures of interest include JXTA [56], Pastry [130], and Tarzan [46]. These generally do not provide direct anonymity, but could be used in conjunction with other techniques, typically as a low-level networking topology/overlay.

- **Network Scheduling.** Network scheduling discusses the problem of efficient P2P pairings between nodes in a network where participants may rapidly join and leave. Our own work [88] discussed an architecture called Whirlpool, which envisioned "rings" of nodes rotating at different velocities; snapshots of this topology would determine communications for any individual time quantum. While this provided some desirable decentralized properties, some data took a very long time (i.e., many epochs) to be fully disseminated.

Content-Based Routing

Content-based routing, or CBR, enables the subscription of events by (as its name implies) *content*, instead of requiring subscribers to join a channel or multicast/P2P group. This enables intelligent routing, as clients are able to specify their interest more precisely and let the *event infrastructure* decide whether the event need be propagated to that node. This often allows for the notion of *pushing subscriptions into the network*, enabling the event distribution system to drop or summarize events at an earlier point in their distribution path, reducing network traffic.

As a consequence of this subscription mechanism, CBR systems have intrinsic basic event correlation techniques, but also pose unique problems for privacy-preservation, since the CBR infrastructure generally expects full data access to enable subscription matching. A few seminal systems are listed here. Note that, in

theory, such CBR systems could also be P2P; however, the ones discussed here are either centralized or tree topology-based.

- *Elvin* [135] is perhaps the best known CBR publish-subscribe system; it supports single-message predicate matching, but does not correlate between events. Elvin also uses a centralized topology; while it supports multiple-server federations, these are primarily for failure robustness, not arbitrary routing.

- Carzaniga et al. [19] built *Siena*, a Scalable Internet Event Notification Architecture, with a flexible subscription language featuring predicate matching. In addition, support for "sequence subscriptions" was added to the language where predicates can be declared on a sequence of events; the subscriber then receives the sequence of notifications that matched the subscription.

- IBM's *Gryphon* [9] has the most extensive correlation facilities available for a CBR publish-subscribe solution; "event stream interpretation" [182] allows for both the semantic reduction of data, i.e., Gryphon will rewrite a stream of events as a single event, and optimistic delivery to counter the slowness of stream processing.

There are several issues with ensuring that CBR routers provide anonymity; such desiderata are discussed briefly in section 5.3.5.

Other distribution technologies

Finally, I briefly discuss other distribution methodologies that may be applicable. Use of these technologies with the model presented in this thesis is left for future work, but may pose interesting possibilities.

- Cornell's *Astrolabe* system [162] provides a different approach: it acts as a distributed hierarchical information repository, using a replicated DNS-like

infrastructure to support a number of applications. They provide an SQL-like interface as a base model, and define solutions, including management, as manipulations on top of this repository. Information is summarized towards the top of the hierarchy, and drill-down is supported to perform after-the-fact analysis. While they have developed system monitoring solutions on the Astrolabe infrastructure, their implementation benefits environments where a large number of nodes may need to know specific application-specific information and where short latencies are less critical.

4.3 Software Monitoring Middleware and Autonomic Computing

Autonomic computing, and in particular the notion of self-managing and self-healing software, is an attractive approach to reducing the time and effort costs of operating and maintaining software systems, and to increase their dependability and assurance levels [34]. Related solutions are already being promoted commercially; several major vendors sell enterprise applications that require little help from IT staff to run and maintain [57]. I briefly discuss related work to chapter 2 here.

- Several sensor and gauge technologies have been integrated into the event propagation [182] and network layers, often in hardware via SNMP. These tend to be optimized for lower-level, high-volume general-purpose packet streams. They can easily be utilized by KX, which can provide higher-level semantics to simple matches found in these lower layers. In the commercial arena, OC Systems has an analogous platform to DASADA sensors and monitors with their AProbe [106] and RootCause [107] products, while SMARTS offers their Automated Business Assurance service with "Codebook Correlation

Technology." [154] These technologies are generally noninvasive and rely on quickly matching against static or predetermined analysis, as compared to our intent to integrate with application semantics, where new success or failure rules can be introduced on the fly.

- Distributed software monitoring middleware has also played a role in enabling autonomic computing solutions. The JAGR project [15] adds self-recoverability to the open-source JBoss J2EE application server, thereby providing a middleware autonomic layer for J2EE business logic components (known as Enterprise Java Beans, or EJBs).

- Both the network and fault management communities have some autonomic behavior as well. The NESTOR project [78] takes a network-layer approach to monitoring. Additionally, JSpoon [79] is a language developed for the NESTOR project that adds "management" attributes to a network architecture. KX may benefit from JSpoon-like semantics in an attempt to enrich our behavioral models with lower-level network information. Fault management systems [143, 141] are also closely-integrated at the systems level, for telecommunications-level reliability. These systems are largely static, designed for vertical solutions, and not for complex distributed "systems of systems".

- Minsky [99] defines a formalism to define whether or not a complex, distributed system can be termed self-healing; the principal concept is that the system must have regularities. Further research might help determine if the addition of KX to an "irregular" legacy architecture can "smoothen" it and provide the regularities needed to enable such self-healing behavior.

Although autonomic computing is a growing field, and several of the techniques discussed in this thesis may be used in an autonomic computing context without loss of generality, the rest of the thesis focuses on the actual privacy transformation

and corroboration techniques, upon which such autonomic systems would function as distributed applications.

4.4 Distributed Intrusion Detection

A Distributed Intrusion Detection/Collaborative Intrusion Detection (DIDS/CIDS) system is one that employs *multiple* Network Intrusion Detection and/or Host Intrusion Detection sensors (NIDS/HIDS), often across multiple local area networks, and correlates resulting alerts to get a broader picture of Internet-based threats. I describe some of the more notable literature in DIDS below; many of them focus on exchange of data within a single organization, e.g., distributed collection and centralized correlation [145].

- *GrIDS* by Staniford et al. [139] is perhaps the earliest well-known DIDS; its name stands for a Graph-based Intrusion Detection System. The graph forms a hierarchy (tree); data is aggregated at lower-level nodes and summaries are passed up the hierarchy. A rule engine is used to determine appropriate aggregation.

- SRI's *EMERALD* (Event Monitoring Enabling Responses to Anomalous Live Disturbances) [121] is not formally a DIDS—it's technically a NIDS—but the authors point to its distributed component model as a "composable surveillance" that scales better than a centralized monitor. Components include service monitors that act as sensors, and domain/enterprise monitors that aggregate and interpret sensor information. EMERALD uses event communication mechanisms, but does not dictate a format to be used. It is primarily geared towards signature analysis.

- *Quicksand*, by Kruegel et al. [82], is close to a hybrid of the first two; it is a

completely decentralized architecture, like EMERALD, but uses a rule engine to implement a distributed pattern language (Attack Specification Language, or ASL). Quicksand is particularly designed for a large network with lots of sensors, although their experiments focused on a single departmental network.

Recently, more Internet-scale, cross-domain DIDS research has been conducted; I describe several below.

- CARDS [104] is a prototype distributed intrusion detection system that uses "attack trees", or pre-defined sequences of attack steps. CARDS decomposes global representations of distributed attacks into smaller units (called *detection tasks*) that correspond to the distributed events indicating the attacks, and then executes and coordinates the detection tasks in the places where the corresponding events are observed. While fast algorithms for signature and string matching exist, the best known are of complexity $O(n \log^3 n)$. One notable follow-up work is constructing new attack sequences to keep the signature database up to date.

- Cuppens and Miege [28, 29] discuss methods for cooperatively correlating alerts from different types of intrusion detection systems.

- Anagnostakis et. al. [5] has looked at Internet-scale correlation and detection from the context of "cooperative immunization". Their algorithm, COVERAGE, is based on a virulence model; they show simulations that demonstrate its effectiveness. The mechanics of information exchange, however, are not discussed.

- A study by Moore *et al.* [101] defines the resources needed to counter worm propagation. Amongst other important observations, they note that effective response must be done within minutes of an outbreak, and that

this may become more difficult as worms become more aggressive. Most significantly, their data suggests that content-based filtering supports a much longer response time than simple IP blacklists.

- Janakiraman et al.'s *Indra* [67] is one of the first DIDS platforms to explicitly support a P2P-based communication platform. They also stress that most existing DIDS platforms only collect data for humans, and that an autonomic system is more appropriate. Their architecture leverages the Scribe publish-subscribe protocol [131] running on top of Pastry, a generic P2P routing substrate. However, at the time of publication this architecture was theoretical—the implementation was at an early prototype stage.

- *DShield* [157], built by Ullrich, is the most active DIDS project on the Internet. It is volunteer-based; anyone can download a client and submit alerts to DShield. DShield then regularly produces "top 10"-style reports and blacklists that enable people to update their filters to common threats.

- *DOMINO* by Yegneswaran et al. [179] organizes a decentralized, heterogeneous collection of NIDS sensors into "satellite" and "axis" nodes, with satellites talking only to axis nodes and the axis nodes forming a overlay graph between which encrypted traffic is exchanged hourly. The architecture is mostly theoretical; a small experimental subset has been implemented, featuring an "active sink" honeypot to collect potentially malicious payloads. More importantly, the paper measured, using DShield alert logs, the notion of information gain, and concluded that 40-60 sites enables building summaries and/or blacklists with high degrees of confidence, and that very few IPs dominate the alerts.

There are significant differences between the work presented in this thesis and the works cited above. First, none of the above approaches are privacy-preserving,

although some of them use end-to-end encryption to prevent sniffers and man-in-the-middle attacks. Second, with the exception of DShield (and the corresponding analysis in [179]), none of the projects above offer a significant implementation or deployment to verify the hypothesis with collected data; section 6 details our approach—Worminator—and presents results collected from its deployment. Finally, Worminator's modular framework allows for the separation of event type, application, correlation engine, and the underlying communication substrate, allowing for a broader set of solutions.

4.5 Signature Generation and Exchange

Another approach is to "eliminate" the privacy-preservation problem by exchanging only "known-bad" content; this approach is particularly common in the IDS signature generation community.

- Earlybird [138], Honeycomb [80] and Autograph [72] are some of the seminal approaches to signature generation; they generally implement string-style payload comparison algorithms, including LCS, LCSeq, and Rabin fingerprints, and can be considered similar in nature to the baseline techniques discussed in section 6.5. Polygraph [103] builds on earlier work on Autograph explicitly addresses the notion of polymorphic worms using LCSeq-like techniques; the authors show that even polymorphic attacks *must* contain invariant substrings, which is promising for the approaches discussed in section 6.5.

- FLIPS [91] pairs PAYL, one of our own in-house anomaly detectors ([174], §6.3) with an Instruction Set Randomization infrastructure for zero-day worm signature generation.

- PADS [155], or "Position-Aware Distribution Signatures", seek to blend

frequency distributions and packet signature positioning.

While several of these systems attempt to sidestep privacy issues by focusing on signatures generated in an enclave, which may then be distributed, they do not offer privacy guarantees as to the generated signatures; false positives, in particular, could convey sensitive content.

Additionally, newer attacks, such as polymorphic worms, have posed challenges to pure payload-based signature generation. As a result, work has focused on building *semantic-aware* or *vulnerability*-based signatures to handle multiple (or polymorphic) attacks for the same exploit.

- Kruegel et. al. [81] use structural analysis of binary code and generate control-flow graphs to catch worm mutations.
- Shield [169] provides vulnerability-specific but exploit-generic filters based on predefined protocol-based policies.
- Vigilante [26] introduces the notion of vulnerability-specific *self-certifying alerts* that focus on filtering undesirable execution control, code execution, or function arguments, and can be exchanged via P2P systems.
- VSEF [102] builds *execution-based filters* that filter out vulnerable processor instruction-based traces.
- COVERS [85] analyzes attack-triggered memory errors on a host and develops structural memory signatures.
- Nemean [180] uses session-layer and application-protocol semantics to reduce false positives. Some of these signatures and filter descriptions may be exchangeable using our techniques.

These systems, to some extent, represent the next generation in vulnerability or exploit-targeting strategies. While the techniques in section 6.5 focus on payload-based strategies, there is no reason why these systems could be employed instead; I consider integration of our privacy-preserving techniques with such approaches in future work.

4.6 Privacy-Preserving Sanitization and Collaboration

The works referenced here are most similar in nature to the proposed work. In particular, they make privacy preservation one of the key requirements, and support it to varying degrees.

- Lincoln et al. [86] describe a privacy-preserving mechanism for sharing security alerts, and addresses several techniques to sanitize alert data, including scrubbing and hashing. They propose the use of multiple hash functions, some keyed, to build solutions that avoid dictionary attacks. They also employ multiple repositories that randomly forward alerts to each other to obfuscate event sources. There appears to be no implementation, and more seriously, no privacy evaluation of the above model, despite some small performance tests of hashing and correlation overhead.

  As opposed to Lincoln, et al., this thesis is application-agnostic; while section 6.4 discusses a similar, privacy-preserving CIDS, the infrastructure behind Worminator can be applied to other forms of intrusion detection or software fault correlation. Indeed, I describe and implement approaches to support privacy-preserving collaborative payload anomaly detection. Additionally, this thesis introduces the notion of a framework to enable scalable, heterogeneous privacy-preserving mechanisms, while [86] focuses on a fixed basket of techniques. Finally, Worminator has several significant differences to enable

practical deployment, including the use of Bloom filters, fast Bloom filter correlation techniques, and publish-subscribe infrastructures. To the best of my knowledge, the work proposed in [86] remains unimplemented, and in fact postdates much of the Worminator work. Finally, as I later show, their work, while good in concept, has several fundamental issues. I demonstrate how to resolve these and make the approaches described more practical in the context of Worminator (§6.4.7).

- Kissner just completed a thesis titled "Privacy-Preserving Distributed Information Sharing" [74]; some of the results were also published in [75]. She outlines two different privacy-preserving mechanisms: a polynomial set representation that supports not only privacy-preserving intersection, but also union and element reduction (i.e., set count difference), and a pair of *hot item* algorithms, one defining an identification mechanism and the other defining a publication mechanism. The latter two algorithms are closest to the use of Bloom filters [14] in this thesis; in fact, the HotItem algorithms use bit vectors that closely resemble Bloom filters. Her notions of data and owner privacy also closely correspond to the definition of source anonymity and data privacy in section 1.1. However, there are significant differences.

  1. The described algorithms focus primarily on sets and associated membership tests. The work proposed here takes an event-first approach, where data have timestamps closely associated with them and used in correlation. The use of time-based correlation changes the algorithmic approach significantly.
  2. The model proposed here (see section 3) assumes the *differentiability* of sources as being privacy-preserving. The work proposed by Kissner enables either owner non-privacy or complete owner privacy, but does

not directly correspond to the indirectly differentiable-but-anonymous model. The correlation techniques leveraged here require the use of differentiability, and even classification in special cases.

Most significantly, her thesis focuses on a theoretical, cryptographic approach to set operations and hot item identification; this thesis uses a set of algorithms very close to her HotItem work, but focuses on the correlation itself. An alternative version of the framework described here could be developed with her algorithms, and in fact may serve as interesting future work. As a result, I view the theses as more complementary than competing.

- Huang et al. describe *Privacy-Preserving Friends Troubleshooting Network* [63], which extends Wang et al.'s PeerPressure research [170]—a collaborative model for software configuration diagnosis—with a privacy-preserving architecture utilizing a "friend"-based neighbor approach to collaboration. The key relevant aspects of the paper include a variation of secure multi-party computation problem to "vote" on the popularity of a configuration to determine the configuration outlier, and the use of hash functions to enable secure multiparty computation (SMC) to support an unknown set of values; the relation of this thesis to SMC is discussed further in the next section. Finally, as with the previous work, they do not address temporal constraints in their correlation mechanisms.

- Xu [176] introduces the notion of "concept hierarchies" to abstract low-level concepts, along with the use of entropy, to balance the sanitization and information gain of alerts; a similar use of entropy may also be applicable here.

- The JAM project at Columbia University [147] looked at abstracting and comparing models of data for bank fraud to enable competing financial institutions

to collaborate in catching criminals without releasing sensitive information. The work in section 6.5, on the other hand, focuses on the validation of byte content detected at multiple sites as indicators of common attack information; the JAM work more closely resembles BF model intersection, which I briefly discuss in section 7.3.

4.7 Other Privacy-Preserving Computation

Statistical transformation. If the data to be correlated is of significant size, a statistical transformation of the data (such as a frequency distribution) is a convenient privacy-preserving mechanism; not only is it extremely difficult to reconstitute the original content given the distribution, but the distribution can also be transformed in various ways to make the distribution itself difficult to reconstruct.

- Wang's *PAYLoad Anomaly Detector* (PAYL [172], [174]) uses the combination of a 1-gram statistical model with a learning anomaly detector to build extremely accurate "normal" models of network packet payloads, and can effectively label anomalous traffic. Additionally, PAYL can generate a signature of ingress/egress correlated traffic, which can be represented as a substring of the original payload *or* as a privacy-preserving Z-string (Zipf distribution) that cannot be used to reconstruct the original packet(s).

- Wang's Anagram Anomaly Detector [173] evolves the PAYL work into supporting n-gram statistical models, and not only supports effective and accurate models of normal traffic and detection of anomalous traffic, but also takes steps to being robust against mimicry and other evasion techniques against anomaly detectors.

Such techniques are complementary to this thesis; in fact, section 6.5 utilizes both PAYL and Anagram as top-notch anomaly detectors that can generate payload alerts that are exchanged using Worminator's privacy-preserving mechanisms. Other statistical modelers could be applied in a similar fashion.

4.7.1 Privacy-Preserving Databases and Data Mining

There is a tremendous volume of work on various aspects of data mining and databases, including secure query mechanisms, statistical databases, secure indices, etc. I mention here some of the relevant classes of work.

- **Statistical databases** refer to query engines that may return individually restricted or perturbed results as privacy-preserving measures; Agrawal et al. [4] demonstrates that accurate aggregations can still be reconstructed via decision-tree learning. Lindell and Pinkas [87] expand on this notion and discuss mechanisms to **securely build decision trees** over the union of two otherwise private databases, enabling aggregate data mining operations. However, statistical databases and aggregate data-mining do not directly apply to precise matching applications as discussed in this thesis.

- Agrawal et al. [1] also presents mechanisms for supporting **privacy-preserving information sharing across databases**; in particular, they develop secure two-party intersection, equijoin, and cardinality algorithms. Strong privacy guarantees are accomplished by the use of commutative encryption.

  This approach—and many others—fundamentally assume two-party interaction, either based on the querier-database model or set operations on two information-sharing databases. In general, these models do not generally scale to multiparty event correlation.

- Instead of obscuring data, the **k-anonymity model** (Sweeney [152] and others) argues disclosure is not an issue as long as the data has been scrubbed of source information *and* individual sources cannot be correlated via any subquery of the raw, anonymized data. The *k* refers to the minimum number of records that must have the same value for a given column on any subquery. While this property can be maintained for statistical aggregation models, this restriction would be difficult to enforce on a generalized event correlation framework.

- In certain scenarios, the **search queries and/or indices may need to be privacy-preserving**; Bloom filters [14] play a significant role in this requirement. Bellovin and Cheswick [12] support privacy-preserving queries through a "semi-trusted" third party to accomplish this goal; Bloom filters are used as search keys. For indices, the goal is to prevent the index from revealing content that was indexed within it. Bawa et al.'s privacy-preserving index [10] uses *content vectors* constructed of Bloom filters. Finally, Goh [52] uses Bloom filters as a *per-document index* to track words until they are integrated into a *secure index*. The index itself is secured by requiring queriers to obtain key-based trapdoors for words. In all of these cases, as with information sharing, the model is two-party interaction, and the database is typically trusted; while this thesis uses Bloom filters, it relies on a multi-party, untrusted model.

  [45] introduced the notion of *counting Bloom filters*, which stores an integer value in each array cell instead of a bit value, and thereby supports deletion via a mechanism extremely similar to *insert*. This concept serves as the basis for the MRU and Timestamp Bloom filters described in section 5.2.2.

  [10] suggests the probabilistic copying of bits into a Bloom filter to introduce uncertainty and robustness against curious entities or attackers. A more extensive approach, along with a detailed evaluation, is described in section

6.4.

Additionally, most of the research in this field is more concerned with offline analysis, as opposed to near real-time event correlation, and the adopted algorithms and mechanisms differ correspondingly. [2] is more open-ended, and presents a strawman for a "Hippocratic database" design that respects privacy, regulatory, retention and other policies. Many of the approaches covered are complementary, and could be applied as an extension to the proposed framework.

4.8 Other Privacy-Preserving Techniques

- **Secure multiparty computation (SMC)** ([177, 178], etc.) is a theoretically attractive way to accomplish privacy-preservation; certain forms of correlation can be fashioned as such a computation problem. For example, a SMC approach to intrusion detection would formulate intersection as a function to be executed over every participant's input. Du et al. [40] discuss a model to transform standard computation problems to secure multiparty computations, and review, amongst other problems, the possibility of sharing intrusion detection information. However, existing SMC algorithms are considered too expensive [86] to handle large event streams such as the ones presented in this thesis.

- **Zero-Knowledge Proofs (ZKPs)** ([55, 53] and others) are somewhat related to SMC approaches. A Zero-Knowledge Proof is a theoretical approach for a party (the "prover") to prove to another (the "verifier") that an assertion is true, *without* having to reveal any information other than the assertion. This is typically accomplished by establishing multiple probability distributions in the proof structure which the verifier can use; the uncertainty as to which distribution holds interesting data makes it impossible for the verifier to

glean information. Like secure data-mining, however, ZKPs are primarily designed to be conducted between two parties; [42] proposes a model for scaling the number of participants, but this requires clever timing constraints. Moreover, like SMC, such approaches do not generally scale to the event volumes discussed in this thesis.

- [38, 21] use Bloom filters (§5.2.2) for hardware-based packet inspection and classification.

- **Code obfuscation** for introspection-enabled languages, e.g., [127] for Java, is a different form of privacy, which corresponds to the notion of retrofitting discussed in section 5.4. This work differs significantly in that it is the *data* that is being obfuscated, which poses its own challenges (e.g., dealing with inequality checks)—as compared to code, where *namespaces*, essentially syntactic sugar, are being removed without any change in functionality.

Chapter 5

Privacy Preservation

As its name implies, this chapter is focused on the privacy challenges discussed in section 1.3, and presents solutions for data privacy and anonymity. An application of these techniques, Worminator, is discussed in chapter 6.

This chapter as organized as follows. First, the problem of *data privacy* is discussed in detail, followed by a set of approaches to effectively enforce data privacy while maintaining the ability to corroborate events. Next, methods and criteria for anonymity are discussed, especially in the context of choosing appropriate event distribution systems. Finally, I discuss strategies for applying privacy to *existing, legacy* event correlation and distribution systems.

5.1 Data Privacy

The key consideration to providing data privacy is to transform data before it is published on the collaboration network, yet still allow effective corroboration. I define the key base operations required for corroboration as *insert* and *verify*: a producer needs to insert values into the collaboration network, and peers need to verify them to determine if they can corroborate what the initial producer saw. For complete data privacy, no other data retrieval mechanism should be supported (e.g.,

enumerate, get, count, etc.). In other words, a one-way data structure is ideal for this problem, which is reduced to set matching as per [73].

It is important to note, however, that such set semantics does *not* preclude partial matching. A number of incremental analyses, especially n-gram analysis (§5.1.3), are valuable in supporting more flexible matching than just entity equality—and by using just the same two set primitives. Attribute-typed events can also support partial matching, as discussed in section 5.4.

5.1.1 Techniques and Privacy Gain

As alluded to in figure 3.2, I introduce the use of two classes of privacy-preserving, one-way data structures: hash-based transforms (hash sets and Bloom filters) in section 5.2.1, and frequency-based transforms (frequency distributions and Z-Strings) in section 5.2.3. The applicability of these techniques is application and data-domain specific; several of these considerations are discussed in the above sections, and others are described in the Worminator applications in chapter 6.

For each, we are interested in characterizing the *privacy gain* against a "honest but curious" attacker accomplished by using these techniques. We focus on "honest but curious" entities because a purely malicious attacker also has other avenues for subversion, such as attacking individual hosts and reading events before they are privacy-transformed and exchanged, or introducing a large number of fake hosts that may be able to collude to gain revealing data about other participants. Both are interesting distributed computation problems in their own right and are considered outside the scope of this thesis, although the techniques in this thesis may be useful as part of a larger system used to mitigate the effect of a malicious attacker.

Measuring such privacy gain depends on a number of factors, the most important being the *knowledge of discourse*, that is, knowledge (language) of the event semantics being exchanged, which in turn depends on the forms of data being exchanged.

Given this, I devise a probabilistic model as a first-order approximation to measure *the relative privacy* of each privacy-preserving technique given participants who are either aware or unaware of the language of discourse; more precisely, I develop a probabilistic characterization of *the recovery likelihood* of the original data given the privacy-transform data.

As a footnote, it is impossible to achieve perfect data privacy if meaningful corroboration is desired. As a worst-case scenario, consider an event corroboration scenario with two possible values. A participant in this corroboration will know the published value with 100% certainty, no matter the data encoding technique used. The argument put forth here is that if an appropriate *balance* can be struck between privacy and corroboration, the value proposition of information sharing far outweighs the extremely low probability of information disclosure, and the results in section 6.5 support this argument.

One may also argue that an attacker does not need the exact original data to pierce privacy; while this may be true, determining this is highly application-specific. More detailed privacy analysis on Worminator can be seen in chapter 6. Of course, anonymity still plays a meaningful role in all scenarios; I discuss appropriate anonymity techniques in section 5.3.

Other desiderata in picking a privacy preservation technique include computation time and memory overhead; both are also discussed in each subsection.

5.1.2 Aggregate Matching

Despite the fact that hash sets and related structures only support individual set membership tests, one can leverage their "generic container" aspect and hash different raw datatypes into the same set. Not only does this make it more difficult to brute-force, it allows a primitive form of aggregate matching.

For instance, in the case of inserting events containing an IP address, an additional

strategy would be to hash not only the IP address (1.2.3.4), but the corresponding class C network address (1.2.3.0), the class B network address (1.2.0.0), and so on. Of course, this requires *a priori* knowledge of the ideal aggregates to hash. Additionally, this increases space and computation requirements, although a) redundancy amongst aggregates and b) compression can be used to significantly ameliorate storage overhead.

Syntactically heterogeneous event types can also be mixed, e.g., IP network addresses versus network payloads, so long as a good set of hash functions is used and can easily distinguish between them.

5.1.3 Incremental/N-gram Analysis

Despite the aggregation model suggested above, the model introduced in section 3.2 *still* is largely restricted to matching discrete events, since events e are transformed into privacy-enabled events e' and cannot be generally transformed back. However, by treating an individual event as a *set* unto itself, we can reevaluate the problem as ensuring a majority of event features are present, as opposed to the entire event being identical. There are a number of techniques to accomplish this.

Feature-based incremental analysis. Given an event e with features $e_\alpha, e_\beta, e_\gamma, \ldots$, I define $\mathcal{E}^e$ as a set of events where each event is composed of an individual feature in e.[1] $\mathcal{E}'^e$ is then a privacy-transformed version of that set, or rather, of e piecewise. Given two or more events $e_1, e_2, \ldots, e_n$, we now transform the corresponding sets into $\mathcal{E}'^{e_1}, \mathcal{E}'^{e_2}, \ldots, \mathcal{E}'^{e_n}$. These sets are now distributed to other participants.

At a recipient A, the same corroboration algorithm $C'^{e_i}_A(\mathcal{E}'^{e_i}_B, \ldots)$ can be applied for each set $\mathcal{E}^{e_i}$, corresponding to event e_i, yielding an intersected set of features. Additionally, $|C'^{e_i}_A|$, the size of the intersected set, can be computed, scored and/or

[1] The term e or terms e_i are superscripted to prevent confusion with the participant designator, which is subscripted in my notation.

thresholded, e.g., $|C'^{e_i}_A| \div |\mathcal{E}^{e_i}| \stackrel{?}{>} \tau$, where τ is once again empirically set.

Feature determination is highly dependent on event semantics. For example, in the case where attribute-valued events are used, a feature may be a string concatenation of the attribute and value types. For XML, one may choose to process all individual XPaths [166] of an XML message as distinct features. Finally, features can be segregated opaquely, e.g., every n bytes. N-gram analysis is a variation of this technique and is discussed below.

The one significant challenge to the above techniques is increased complexity; instead of searching a set of events from a remote participant, the local corroborator may potentially have to search many sets, transforming lookups from $O(1)$ to $O(n)$. Alternatively, the sets themselves can be nested, rearranged or flattened to combine separate event semantics and provide computational speedups. Nesting is largely a convenience mechanism to reduce the amount of metadata transmitted per set of events. Rearranging transmits sets of *features* instead of sets of events, allowing aggregate frequency analysis on the prevalence of a particular feature. Flattening, which essentially tosses out event boundaries and views the problem as a collection of features that are to be correlated, is particularly useful in building models used with n-gram analysis. Alternatively, multiple types of sets can be transmitted; the flattened form can be used for quick lookup (e.g., look for an arbitrary feature f to be present in this set before we search for it).

N-gram incremental analysis. In the case of opaque events, e.g., binary data for which no wire format exists or is readily available, feature abstraction is not necessarily tractable. Nevertheless, one may want to corroborate such events to find commonalities, such as network payloads (§6.5). In this case, *n-gram analysis* is used to extract features. N-grams entail a sliding window of n bytes over the event data in question; each of the resulting entities—an individual n-gram—are then viewed as a feature that is inserted into the (privacy-preserving) structure in

question. See figure 5.1. The set of n-grams in the data structure can be either kept sorted by feature (i.e., enabling frequency analysis) or flattened (i.e., enabling binary analysis). The latter allows for fairly efficient space coding via a Bloom filter, as discussed in section 6.5, and produces surprisingly good results (specifically, for network payload events).

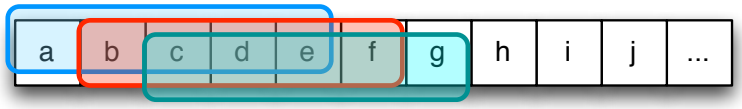

Figure 5.1: 5-grams computed over opaque data.
The first three 5-grams are shown. There are $b-4$ distinct 5-grams in this data, where b is the number of bytes.

The resulting privacy-enhanced event set is equivalent to a model over the original event(s), and the similarity metric of an event e with k n-grams against model $\mathcal{E}'$ may be computed as

$$\mathcal{S}(e,\mathcal{E}') = \begin{cases} \dfrac{\sum_{i=0}^{k} f(g_i)}{k} & : \quad \mathcal{E}' \text{ is frequency-modeled} \\ \dfrac{\sum_{i=0}^{k} \mathcal{F}(g_i)}{k} & : \quad \mathcal{E}' \text{ is binary-modeled} \end{cases} \tag{5.1}$$

where g_i is the i^{th} n-gram in event e, $f(g_i)$ is the normalized frequency of n-gram g_i in $\mathcal{E}'$, $f(g) \in [0,1]$, and $\mathcal{F}(g_i)$ is a binary function returning 1 if n-gram g_i is in $\mathcal{E}'$ and 0 if not.

In fact, many types of events can be broken down into n-grams and inserted into a *single* model, which can then be seen as a *characterization* of event flow. Given such a model, not only can similar events be corroborated at other sites (given appropriate thresholding), but *new* events can be checked to see how closely they fit the characterization of event flows. Detection of events which differ, or which are *anomalous*, is a key mechanism used in network traffic anomaly detection, as I

discuss in chapter 6.

One key consideration: the values of n, defining the size of the n-gram, have to be predetermined before a n-gram corroborative distribution is started. Multiple n can be inserted into the same model if desired. Automatic determination of optimal n-gram sizes remains an open research topic.

5.1.4 Temporal Corroboration with Models

Temporal correlation for arbitrary events is a well-researched and implemented topic; the Event Packager and Distiller tools in chapter 2 represent one such implementation. However, temporal corroboration becomes significantly more difficult when privacy-preserving models of events are exchanged. Since models are not discrete collections of events, they must be kept "in full" if corresponding local events have not yet been generated. Scans through these models as new events are generated then becomes a (rather expensive) linear-time operation. For example, the alerts generated in Worminator may be frequently distributed to peers—on the order of several per minute—leading to a very large cache of Bloom filters.

This differs significantly from raw events or even hashed privacy-preserving sets, as these can be either traditionally indexed to provide rapid lookup *or* recognized through privacy-transformed event subscriptions (§5.4), enabling corroborators to keep state. In this section, I discuss several techniques to accomplish nearly equivalent functionality for models—primarily Bloom filters, but frequency models as well.

Naïve approach. A naïve approach to solving this problem is to simply keep *merging* models as they arrive. This reduces lookup time from $O(n)$ to $O(1)$. However, repeated merges end up saturating models with old data, increasing false positive rates and making them largely useless for meaningful corroboration.

Several techniques can be used to avoid saturation, as discussed in [59, 21]. One

is a *cold cache*, whereby the model is emptied when it crosses a certain threshold of fullness, or *double buffering*, which allows a staging empty operation. Double buffering employs a second model that is initially empty, but is increasingly filled as the first becomes too full or when a predetermined time duration passes. Once the first one has crossed the appropriate threshold, the first one is emptied *and the role of the two filters are switched*, allowing for a graceful emptying operation. However, neither of these enable effective timestamp-based constraints.

Expiration. The next level of timestamp support is to expire all models that were created before some time t, possibly based on the longest timebound constraint in corroboration rules, i.e., expire events which are too old to corroborate against any rules. This can be accomplished in one of several ways:

- Associate a timestamp with each model, and garbage collect models whose timestamps cross a threshold. Each model is kept distinct. While this avoids ever-growing lists of models to corroborate against, it does little to reduce the linear-time component of comparing events to models. On the other hand, it works with essentially all model types, including frequency-based models.

- Associate timestamps with each model, cache a merged model, but also remove entries from the merged model as individual models expire. This assumes a model from which items can be removed, such as a *counting Bloom filter* [45].

- Build a merged model which keeps the last timestamp per entry. This is not practical for a frequency model, but I describe a variant of a Bloom filter, an *MRU Bloom Filter*, which supports this operation, in section 5.2.2. This Bloom filter supports aging without needing to store constituent event models, but with the cost of losing individual timestamp information.

Timestamp range support. Expiration still does not accommodate range or historical queries, i.e., "was event e seen by entity E between t_1 and t_2?". Such range queries become significant when coordinating a large number of events, especially because sources may not publish at any coordinated time intervals. As previously mentioned, however, standard techniques to support temporal corroboration do not work with a opaque model. Here, I discuss two ways to support range lookups in a faster-than-linear time.

First, a *tree-based index* can be created over the set of models for which lookups will occur. The concept is vaguely similar to a B+-tree, with the original models occupying the leaves of the tree, but intermediate nodes differ significantly. Instead of having a set of discrete items in intermediate levels, each with associated references to lower levels, each intermediate node consists of a merged model containing all of its children's data, the range of timestamps covered by its children, and the associated list of children. See figure 5.2.

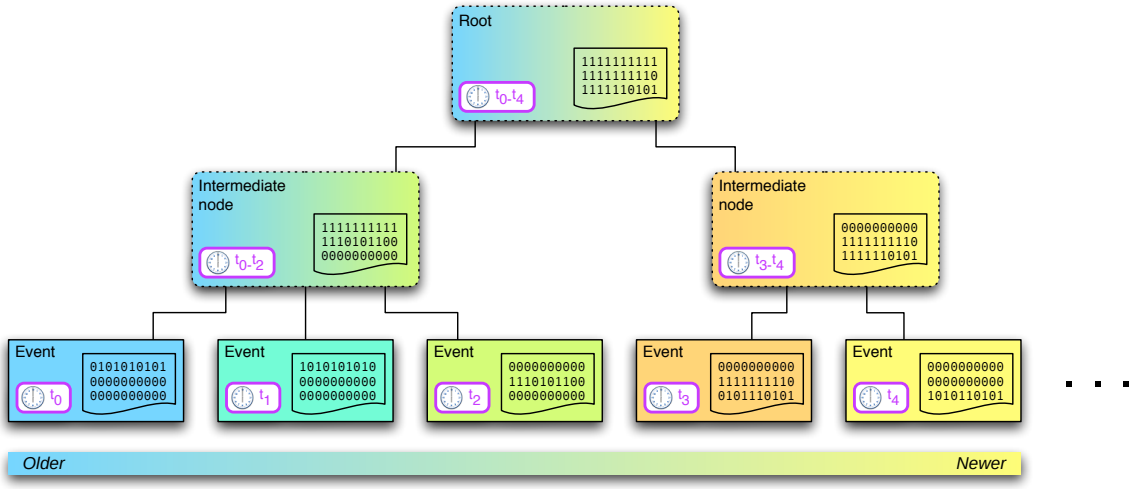

Figure 5.2: Temporal Tree Index of Privacy-Preserving Models.

Leaves are populated as remote event models are received, from left-to-right, and are destroyed in the same fashion during garbage collection, as with a queue. Intermediate nodes, like with B+ trees, contain a variable-length child list; new

nodes are created when more nodes appear on the level below it that do not fit in the last existing node on that level, and are deleted when their children no longer exist. Searches are accomplished by starting at the root, checking to see if the node under consideration contains the item in question and fulfills the time constraints. If so, *all* of its children are examined in the same way. The process repeats until the original leaf nodes are reached and validated, which suggests a hit, or until a node is reached which suggests no such event existed at a remote site within the specified time period.

This approach, assuming the average intermediate node contains more than one child, but still has $O(1)$ children (i.e., proportional to the number of nodes), and is clearly logarithmic in both size overhead and computation time. This temporal corroboration tree structure is evaluated more closely in chapter 6.

The second possibility is to build a data structure that stores a *collection of timestamps* per privacy-transformed "entry", such as individual bits of a Bloom filter; each bit can be expanded into a set of timestamps. This technique is very memory-expensive but supports very fast lookup; if necessary, the *timestamp Bloom filter* can support direct disk lookups. This technique is discussed in greater detail in section 5.2.2.

Note that these techniques are intended to be used locally, in order to build efficient structures for corroboration as remote events arrive. Most of the structures created are too memory-intensive for frequent distribution, and some of them will sacrifice privacy. For example, an MRU Bloom filter can be distributed to enable individual event timestamp dissemination. However, the timestamps can be used to more rapidly discern individual entries, reducing the brute-force search space of the problem. One can round timestamps to reduce differentiation, but at the loss of time resolution; and, in general, a MRU structure only contains a subset of the total timestamps that may have been inserted in the model at a significant size cost.

Model clustering. Frequency models, in particular, are not amenable to building intermediate data structures, as the relative frequency information is lost. However, frequency models (and, potentially, Bloom filters) can potentially be clustered into groups of similar events to reduce the linear scanning overhead. Doing this effectively requires a very fast model comparison algorithm and significant clustering. Model clustering is an interesting problem in its own right. One algorithm, using Z-Strings, is discussed in section 5.2.4.

5.1.5 Model Combination and Comparison

Models also offer the opportunity to be opaquely *combined* or *compared*, while maintaining privacy. This is usually a linear-time operation in length of the model, and the different forms of combination/comparison vary with the encoding mechanism.

There are a variety of applications that can benefit from these; the main application explored in this thesis is that of building *network traffic content* models. These models can be used not only to corroborate individual events, e.g., classify new traffic based on old traffic patterns, but also to compare different traffic flows, e.g., compare models directly against each other and see if they are similar or different. Such model comparison can be used as a trust model; similar models may imply more trust, as traffic flows are similar, as opposed to distinct traffic flows which may suggest incompatibility or malicious intent. I discuss this concept further in section 6.6.

5.1.6 Varying Privacy Considerations

If a lower level of data privacy is required, less obfuscating solutions can be used. For example, some sources are only concerned with obfuscating *internal* information,

while external data can be transmitted verbatim. The inclusion of raw events alongside hashed events does not pose a significant challenge for corroboration, as the system described here automatically utilizes event polymorphism and corroborates accordingly based on incoming event types, and can support more general event correlation when raw events are used.

One potentially appropriate mechanism where limited corroboration may be extended to more generalized correlation is to establish policies releasing underlying event data *if* it has been corroborated by a sufficient number of peers; the fundamental idea is that this event is no longer a secret, so obfuscation is no longer necessary. This technique proves useful if only a fraction k of the participants can corroborate the event; the remaining $1 - k$ fraction of participants remain in the dark, as they cannot test the appropriate bits of the Bloom filter. (For the purposes of this thesis, the value of k is empirically determined, but one could envision a policy-driven or an information-theoretic approach to determining the optimal fraction of participants.)

By sending out the raw event itself, remaining unaware participants can include the event's information in their correlation scenarios. This model proves especially useful in Worminator, where I define the notion of a privacy-preserving "watchlist" vs. a more explicit "warnlist" of correlated suspicious sources.

5.2 Privacy-Preservation Techniques and Transforms

5.2.1 Hashing

The use of hashes/one-way cryptographic functions, i.e., $p(e) = h(e)$, are a baseline mechanism to support insert/verify semantics. "Public" cryptographic hashing, such as SHA-1 [105], works especially well—it is not computationally feasible to reverse the hash, and the likelihood of collisions is extremely low. Other perfect

hash functions, like H_3 [124], can also be used. H_3 has some desirable properties, such as support for incremental hashes, as discussed in 5.2.2.

Privacy gains based on hashing can differ significantly based on knowledge of discourse. A attacker who is not aware of the correlation application would have essentially no ability to determine the content. More relevant is an corroboration participant; in this case, the recovery likelihood ranges from $1/A^n$, where A is the size of the event's alphabet and n is the number of bytes in the event, to $1/L$, where L is the size of the event's language. In the case of a small language, brute-force attacks can be a problem. A number of techniques can be adopted here, including mapping to increase the size of the language, mixing public and private hashes [86] to provide plausible deniability, or adopting cryptographic techniques designed for secure set intersection [73]; the choice is often application-specific. Chapter 6 describes several strategies.

The computation and storage overhead of an individual hash are both proportional to the size of an event; compared to a large event, hashing is relatively small but inefficient, whereas for a small event, hashing is relatively large but efficient.

These hashes can be transported as a set of items, e.g., a hash set can be made into an event. Hash sets are very fast, but can be very memory-inefficient, especially when many small items are inserted, since the table contains the original hash values, and given a large alphabet needs nontrivial amounts of memory space to avoid collisions. Such a scenario becomes likely when incremental event analysis, such as n-gram analysis, is used; in fact, the hash table's memory overhead renders this technique moot. A hash list uses slightly less memory but at an extreme access cost, and so is not considered. Bloom filters, on the other hand, ameliorate this problem significantly by essentially implementing a form of lossy compression, yet while remaining very fast and providing several desirable properties on their own. Each of these are discussed in the following subsections. The implementation in

chapter 6 focuses primarily on Bloom filters as the preferred hashing technique.

5.2.2 Bloom Filters

The Bloom filter is not a new data structure [14], but its adoption to privacy-preservation is recent (i.e., around the timeframe of the early work done in this thesis) and its application to intrusion detection novel; see section 4.7 for related work.

As seen in figure 5.3, a Bloom filter is essentially a bit array of n bits, where any individual bit i is set if the hash of an input value, `mod` n, is i. A Bloom filter contains no false negatives, but may contain false positives if collisions occur. The false positive rate can be ameliorated by avoiding saturation:

- Size the bit array appropriately. In particular, Bloom filters are prone to saturation and corresponding false positives, especially after approximately 50% of bits in the Bloom filter are set [14].

- Use multiple perfect hash functions. An entry is only considered present in a Bloom filter *if* all of the corresponding bits to every hash function are set. Given hash functions that avoid significant numbers of collisions, multiple hash functions reduce false positive rates significantly.

- Leverage data diversity, as gained by participation by many collaborators. Given $B_{1...n}$, i.e., Bloom filters from n peers, the likelihood that the *same* hash collisions occur at every peer decreases as n increases, i.e., the "global" view of corroborated events filters out the "local" noise from any individual Bloom filter.

I show results of false positive rates and amelioration techniques in chapter 6.

The Bloom filter itself is treated as a *privacy-preserving model*, where $\mathcal{M}$ entails the *insertion* of items into the Bloom filter. $\mathcal{S}$, or *verification*, returns a binary response,

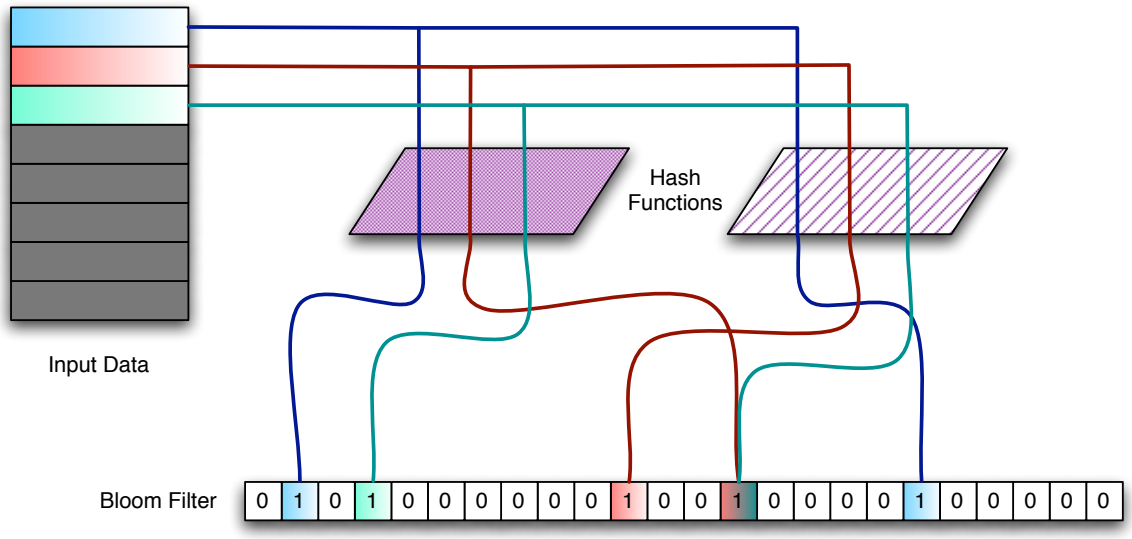

Figure 5.3: Bloom filter.
A Bloom filter with three items inserted, using two hash functions. Two of the entries collide in one cell, but this does not affect the verify function thanks to the use of different hash functions.

either 0 (not present) or 1 (present), although false positives can return a 1 when the item was never inserted. Both insertion and verification are similar to a hash set—$O(1)$, assuming a standard input size.

As with a hash set, a Bloom filter acts as a convenient one-way data structure that can contain many items; however, it generally is orders-of-magnitude smaller. For instance, given a 128-bit hash, a hash set would use 16 bytes to store just the hash value, let alone allocate references or memory to that entry (say, 4 bytes). Compare that 160 bit-per-entry overhead with the n_H bits used per entry, where n_H is the number of hash functions used, and there is the possibility for significant space savings. As the Worminator application will show, 3 hash functions can often be sufficient to accurately represent complex data, which represents a memory savings of almost two orders-of-magnitude.

The choice of hash functions used also essentially acts as a symmetric key/shared secret encrypting the contents of the Bloom filter amongst all participants, obfuscat-

ing it to outsiders and reducing the need for wire encryption, although corroboration itself obviously does not rely on keeping the choice of hash functions secret.

The privacy gain of a Bloom filter is at worst equivalent to hashes (e.g., if exactly one item is hashed into a Bloom filter), and at best is considerably better, as bits are reused amongst many entries, making it difficult to come up with characterizations of size or the correctness of brute-force examinations. As implied, the application chosen again has a significant effect on the privacy gain.

Experiments and corresponding results for Bloom filter corroboration false positive rates are presented in sections 6.4 and 6.5, corresponding to IP alert and payload corroboration. In general, Bloom filters, given the use of multiple hash functions and multiple corroboration partners, perform very well with few false positives, yet provide effective privacy.

Intersecting/Merging BF Models

Bloom filters of the same size and using the same hash functions also have the ability to be intersected via simple, yet privacy-preserving, bitwise AND or OR operations, corresponding to Bloom filter intersection and union. This enables two or more Bloom filters to be semantically combined, depending on the corroboration problem, in $O(n)$ time, without needing access to the original data in the Bloom filter.

This property enables a practical method for exchanging and comparing Bloom filters—in addition to corroborating local events with received models, a similarity metric can be defined between the Bloom filters/models *themselves*,

$$\mathcal{S}_A(\mathcal{E}'_B) = \frac{|\mathcal{E}'_A \,\&\, \mathcal{E}'_B|}{|\mathcal{E}'_A|}$$

where & is the bitwise AND operation and $|\mathcal{E}'|$ refers to the cardinality of a Bloom

filter, that is, the number of bits *set* in the BF. As the above definition implies, this is an asymmetric metric, and can be computed differently for each participant.

The notion of model combination and comparison are novel in an Intrusion Detection context; I discuss early research in chapter 6, and leave a more general treatise to future work.

Temporal Corroboration and Bloom Filters

As discussed in section 5.1.4, a number of model-based techniques can be adopted to deal with the inflow of a large number of Bloom filters over time. Here, Bloom filter-specific techniques are discussed in addition to the general algorithms previously described.

At the simplest level, we leverage the union operation over Bloom filters to avoid a large number of BF comparisons. This naïve approach, while spacewise and computationally efficient, fares poorly with time for several reasons. First, there is no way to expire content from a normal Bloom filter, since an individual bit may be reused for many entries. Given this lack of expiration, saturation becomes a severe issue, since entries corresponding to old data are not removed. Alternatively, a counting Bloom filter can be used, which uses more than 1 bit per cell, to enable removal; in this scenario, received Bloom filters are kept and this structure is kept as a lookup cache, and expiration involves the subtraction of a old Bloom filter from the counting Bloom filter cache. However, like cold caching and double buffering, temporal representational power is still limited for more precise time ranges.

A more sophisticated solution involves an aging Bloom filter; the **Most-Recently-Used (MRU) Bloom filter** is a simple variation on a Bloom filter that uses a similar concept to counting Bloom filters, but instead of storing an integer representing the number of events, a timestamp is inserted. While this increases memory requirements by an order of magnitude, only one is instantiated at each site.

Moreover, timestamp requirements[2] can be reduced, by more than half, by offsetting from a more recent start time and using coarser granularity. Given this arrangement, insert now updates the appropriate cell(s) with the latest timestamp. Union, in turn, is also modified to store the latest timestamp for each corresponding cell in the incoming BF; both insert and union maintain the same order running time, $O(1)$ and $O(n)$, respectively. Verify, which now tests for the presence of a timestamp as opposed to a on-bit, remains $O(1)$, and an expire operation is added, which removes bits that are older than some time t via a linear $O(n)$ scan time. Finally, an optional linear-time operation is *generate*, which generates a "regular Bloom filter as of time t" by scanning the timestamp values and extracting those bits who fit the threshold. The latter operation may be useful if the *union*ed Bloom filter needs to be stored for historical purposes, or transmitted to others in a space-saving and privacy-preserving fashion.

Finally, instead of storing *one* timestamp, many timestamps can be stored per cell; this gives rise to the **timestamp Bloom filter**, or TSBF, as seen in figure 5.4. In order to support effective lookup of these timestamps, we replace the one timestamp per cell in the MRU Bloom filter with a reference to a range-capable data structure, e.g., balanced binary search tree. Union and insert are modified appropriately to manipulate this data structure. The computation costs increase; given n bits and an average of m timestamps per bit, *insert* and *remove* become $O(\lg m)$, *union* (with incoming non-timestamp BFs) is $O(n \lg m)$, and *expire* and *generate* become $O(nm)$. Two *verify* operations exist—one with a range lookup and another without—and their computation times range from $O(1)$ to $O(\lg m)$. (Interestingly, the more expensive verify operation can be far more accurate than the regular verify, as it can distinguish between bit collisions by checking for identical timestamps between the n hash functions used for each entry, and failing if a common timestamp is not

[2] A typical UNIX timestamp is 64 bits.

found amongst them.)

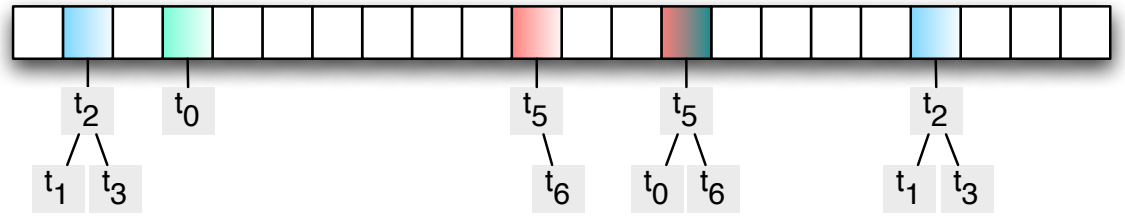

Figure 5.4: Timestamp Bloom filter.
This TSBF corresponds with the Bloom filter in figure 5.3, with differing number of timestamps per set entry.

Most of these computation costs are mitigable because the average value of m is significantly smaller than that of n, and a reasonable expiration policy keeps the value of m manageable. However, the memory overhead for a timestamp BF is significant. Instead of storing 1 bit per cell, many bits may be allocated; for example, given an average m of 5, a memory allocation of $160n$ bits or more, assuming 32-bit timestamps, is very possible. For $n = 2^{20}$ regular Bloom filter, capable of reliably storing approximately 500,000 entries with one hash function, the timestamp BF with $m = 5$ would be roughly 20MB in size. Once again, this is ameliorated by the fact that only one TSBF needs instantiation. In cases where such overheads may be too large, the TSBF can be adapted to disk-based storage, where a smaller in-memory BF (either a counting or MRU BF) would be used as a first-pass cache to see if an item exists; a corresponding disk lookup would then be implemented to determine whether temporal constraints are satisfied.

BF Models and Incremental Analysis Optimization

Bloom filters are ideal for the incremental analysis approaches discussed in section 5.1.3. Not only do Bloom filters represent entries in very few bits (typically equivalent to the number of hash functions used), enabling the storage of a large number of features. Additionally, a single Bloom filter can store a *mix* of different feature sets;

for example, a single Bloom filter can contain 3-gram, 4-gram, and 5-gram "features" of a original packet payload.

As previously mentioned, the overhead of inserting or verifying a single item into a Bloom filter is O(1) per item inserted in the Bloom filter. However, the constant-time overhead to process new entries can become significant when inserting or checking many types of n-grams over a large population of packets, as repeated computation is done when hashing different length n-grams over the same data into the Bloom filter. Couple this with the fact that Bloom filters need good hash function, and hashing can form the majority of computation overhead when processing a collection of events into a model.

To reduce this computation overhead, a *cumulative* universal hash function can be used with a Bloom filter. A cumulative hash function h^* fulfills the requirement that $h^*(c(a,b)) = d(h^*(a), h^*(b))$, where a and b are data (n-grams or fragments of them), c is a (bitwise) concatenation function, and d is a composition function. Given such a hash function, we can avoid re-computing hashes when sliding the n-gram window and when using different window sizes, e.g., if we've hashed a 5-gram and need to generate the hash of a 7-gram, we can just hash the additional 2 grams and combine that with the 5-gram hash value. A class of universal hash functions known as H_3 [124] uses XOR as the composition function, which is very fast and lends itself well to our applications.

Compressed and Cascading Bloom filters

One of the challenges with Bloom filters alluded to earlier is its reliance on a predefined size, which implies some *a priori* insight into the quantity of data to be inserted into the BF. If such data is not available, there are essentially two choices: assume the worst-case and instantiate a very large, potentially very empty Bloom filter, or to create a smaller Bloom filter, possibly at the risk of saturation (and

increased false positives), and employ techniques to avoid that saturation scenario.

Given a large Bloom filter, we can compress it to make transmission sizes manageable. LZW and similar off-the-shelf techniques can be used, and naturally perform well in the case of a sparse BF. Alternatively, a more comprehensive solution, like [100], can be adopted.

However, large Bloom filters still require significant amounts of space in memory when manipulating the BF; in addition, there is the potential of significant CPU overhead in repeatedly compressing and decompressing messages. These can be ameliorated by choosing a smaller default Bloom filter size, at the cost of risking saturation (i.e., 50%+ of the BF is full). Since saturation is also undesirable, *cascading* can be employed. Simply, additional Bloom filters of the same size are instantiated when the previous Bloom filter crosses a fullness threshold. Then, when checking, subsequent cascaded BFs are checked *if* the previous BF did not successfully verify the object. As long as only a few cascading BFs are present, this does not measurably change the $O(1)$ computation overhead of a BF. At the same time, this reduces the usefulness of model merging and other techniques, including several of the timestamp correlators, as the models must have the same number of bits and saturation becomes much more likely.

5.2.3 Frequency Transforms

The other major class of privacy transforms explored in this thesis are frequency transforms. Typically, this involves segmentation of the input event data and compilation of a frequency histogram model of the segments for the events in question. Given sufficiently small segments, the original event cannot be reconstructed. Frequency transforms dovetail particularly well with incremental/n-gram analysis, as those increments become the segments for which histograms are computed.

In order to implement $\mathcal{M}(\mathcal{E})$, the model creation function, events $e_1, e_2, \ldots, e_n$

are broken into segments $s_1^{e_1}, s_2^{e_1}, \ldots, s_{m_1}^{e_1}, s_1^{e_2}, s_2^{e_2}, \ldots, s_{m_2}^{e_2}, \ldots, s_1^{e_n}, s_2^{e_n}, \ldots, s_{m_n}^{e_n}$. These segments are *not* necessarily all unique, and so they are gathered appropriately into $s_1, s_2, \ldots, s_k$, where k is the total unique number of segments and each segment s_i has a *occurrence frequency* amongst all events $e_1, \ldots, e_n$. (Note that a segment may occur many times in any individual event; we are more concerned with the total occurrence frequency over the set $\mathcal{E}$.) These scores are normalized, i.e., $f(s_i) \in [0, 1], \sum_i f(s_i) = 1$ where $f(s_i)$ is the frequency of segment s_i amongst all segments, and stored against index i.

Computing the similarity metric $\mathcal{S}(e, \mathcal{E}')$ can be done in one of several different ways, depending on segment size and the number of segments. For small and relatively few distinct segments, e.g., the 1-gram distribution of byte payloads, we can use a statistical distance metric, i.e., $\mathcal{S}(e, \mathcal{E}') = D(\mathcal{M}(e), \mathcal{E}')$. One such statistical metric is the *simplified Mahalanobis distance* [174]

$$D'_{Mah}(x, \mu) = \sum_{i=0}^{n-1} (|x_i - \mu_i| / (\sigma_i + \alpha)), \tag{5.2}$$

where σ_i is the standard deviation, and α is a smoothing factor to avoid dividing by zero (with higher α values implying less of a confidence about the model).

However, even the simplified Mahalanobis distance is expensive for larger segments, such as higher-order n-grams, as computations are done over very sparse event vectors. Instead, we use the more approximate similarity metric defined in equation 5.1, which looks for the frequency prevalence of each n-gram g_i amongst all n-grams k in an arbitrary event e.

The memory overhead of a frequency transform varies based on how much data is inserted into the model. Given double-precision frequencies per segment, a frequency transform could use 64 bits per segment; for the 1-gram payload distributions in chapter 6, one frequency model would be 2KB. This is large if the

model is used for a single event with relatively few features; if event rates are high, combining events may be desirable.

The last important consideration is that frequency transforms are not a useful technique where *feature precision* is required, i.e., the ability to determine if a particular feature X corroborates between an incoming event and the model. As a consequence, a frequency model is not a very good privacy-preserving transform when the event contains little data or data that is not easily separable into a large number of features, e.g., if it is very tightly structured in the first place. On the other hand, events that are largely opaque can be arbitrarily broken into small features, which in turn provide sufficient data for a rich frequency distribution.

As this implies, selection of the appropriate mechanism is highly application-dependent. Given appropriate feature selection, frequency models can be highly accurate in corroboration, as shown in both section 6.5 and in [171]. However, they *are* susceptible to mimicry attack, a form of masquerading that is based on modeling malicious payloads as similar to legitimate ones as possible, as described in 5.5.

The privacy gain accomplished via a frequency distribution depends on the feature size, as the problem is reduced to a permutation problem. Given sufficiently small and diverse features, the number of permutations grows large and makes events transformed into frequency distributions nearly unrecoverable.

Frequency Model Comparison and Combination

Frequency models can, of course, be compared against each other as opposed to just purely against events. The overarching consideration here is to make such comparisons fast to enable rapid determination of similar models, and for our applications, we use the *Manhattan distance* metric

$$D_{Man} = \sum_{i=0}^{n-1} |x_i - y_i|, \tag{5.3}$$

which is reasonably accurate for our applications and requires only linear time in the length of the model (and is essentially constant-time for the short models used in chapter 6).

Given sufficiently similar models, there are various applications of *merging* or *combining* models, which can be done via a straightforward averaging of the features in question. Optionally, the original non-normalized frequencies may be stored to provide a more accurate averaging (summing) of the individual models in question. We describe one technique and application of model merging in section 6.6.

Effective Temporal Corroboration via Clustering

As implied in section 5.1.4, effective temporal corroboration is difficult with frequency models—averaging dissimilar models does *not* have the same effect as ORing two Bloom filters. In fact, merging many frequency models ultimately can cause the model to become effectively uniform, eliminating its utility in corroboration. At the same time, a linear scan through all collected remote models is not pragmatic or generally scalable.

As a compromise, it is preferable to, at least, *reduce* the number of models that must be scanned through, preferably to sublinear levels, when doing temporal corroboration. Model combination *is* feasible when combining similar models, so a clustering algorithm may be adopted. However, the cost of this algorithm must be made as low as possible to accommodate the continuous real-time influx of new remote frequency models. One can use Manhattan distance, which is reasonably effective. Another, faster algorithm to cluster similar models is to use a portion of the Z-String representation as a cluster key. This technique is described in the next section, once Z-Strings are introduced.

5.2.4 Z-Strings

While frequency models introduce a level of privacy preservation, they are large and they still do release aggregate frequency data. Such data is not revealing when selecting extremely small features with a correspondingly large event, e.g., 1-grams over network traffic, but can be more of a concern when the features are larger or events are significantly smaller. In either case, a more limited representation of a frequency distribution, termed *Z-String* [174], can be used.

A Z-String is essentially a list of the feature vectors from a histogram showing the *Zipf distribution* of a given frequency distribution. In other words, the frequencies are arranged from most-to-least, and the corresponding feature list is kept, while the actual frequencies are thrown away. This is *extremely* small; the largest possible Z-String is the size of the alphabet, and given a sufficiently small feature set, that is generally small in-and-of-itself. Additionally, all zero-occurring members of the alphabet are omitted.

In terms of privacy guarantees, a Z-String is essentially a permutation of a subset of all possible features in the corroboration problem. Assuming a single feature instance is not privacy-piercing, this transformation is very effective. A precise characterization for network traffic is discussed in chapter 6.

Comparison, Accuracy and Clustering

As the Z-String is just a permutation of features, comparison may be difficult, especially in the "tail" of the Z-String where individual features may be ordered randomly. As a result, any practical Z-String comparison must look primarily at the heavy-hitters, i.e., the beginning of the Z-String, instead of the later characters, with a much flatter and lower frequency distribution. We can "tag" the Z-String with a delimiter to separate the frequent from the not-so-frequent entries to enable accurate comparison. Given this specification of the heavy hitters in the Z-String, string comparisons can be employed, e.g., longest common subsequence or edit

distance. Experiments using both are described in section 6.5.

The major downside to the use of Z-Strings is its coarseness; given its very approximate characterization of events, it is easy to construct multiple classes of data that have the same Z-String. As such, its use as a malicious classifier is limited, as it will naturally generate false positives. An analysis of false positive rates for Z-Strings (and with other privacy transformations) with respect to network packet payload data can be seen in section 6.5.4; in particular, figure 6.85 shows that Z-Strings perform poorly for precise payload corroboration.

However, Z-Strings composed of the heavy-hitters *can* be useful as a clustering tag on frequency models to enable sublinear temporal corroboration. Computing the Z-String itself is straightforward, and string comparisons on relatively short Z-Strings is also very fast. Section 6.5.5 describes such a clustering algorithm and presents corresponding results that demonstrate that, while not perfect, Z-Strings are particularly useful in such situations, especially when the clusters do not need to be perfect, i.e., detection can still effectively be done even when some percentage of payloads are separated across multiple clusters.

5.3 Anonymity and Publish-Subscribe Distribution

While data privacy removes personally identifiable information from the event data itself, it still does not eliminate connection-level metadata; for example, in the case of directed point-to-point communications, one can establish an event source. There are circumstances in which anonymity must be kept even amongst this communication; for example, one may guess the language of discourse of the data based on the source. Therefore, anonymity guarantees are required. For the purposes of this thesis, I propose the integration of several techniques into the framework that accomplishes just that.

Before further elaboration, it is important to note that "anonymity" does not clearly connote one concept. In this thesis, I focus on the problem of anonymity amongst known participants, i.e., ensuring an inability to determine the source of a given event in a well-defined corroboration group. I do not consider true anonymity, as that raises a host of significant, possibly unsolvable, challenges; most notably, untraceable dishonest participants can render a corroboration group meaningless via noise and misdirection. Note that "known participants" does not imply that all participants must know the identity of the others; as I discuss shortly, that information can be kept confidential via a trusted party who is willing to vouch for (authenticate) the participant.

Given this definition, one can define four different levels of anonymity: non-anonymous, categorizable, differentiable, and nondifferentiable.

Differentiable refers to the notion that while sources (and events produced by sources) cannot be traced back to the original producer, a distinction between sources can be made. This enables techniques such as thresholding on the number of corroborating peers. In addition, differentiability enables cardinality, e.g., one can "count" the number of peers that detect a given event of interest; this is extremely useful for thresholding.

Categorizable is a slightly more revealing variation on differentiability, where a source may be unknown but can be differentiated *and* categorized into a certain class of sources; for example, a distinction can be made between friends, peers, or unknown people. An event coming from a friend may be valued more highly than an event coming from a truly unknown person. Given a sufficiently large body of friends, an individual friend's privacy is preserved, while providing valuable information for corroboration. This can be particularly useful for Worminator, where different corroboration scenarios may include different genres of organizations, e.g., banks vs. academic institutions, and this metadata may be significant in determining

the threat of a common attack source.

Nondifferentiable, on the other hand, assumes a corroboration group that has "known" participants but events are received completely anonymously, i.e., all source information has been stripped from the event, leaving nothing but data. This scenario, while the most anonymous, poses a number of difficulties for practical corroboration. First, it becomes extremely difficult to differentiate between one party publishing two instances of an alert versus two different parties publishing identical alerts. Second, a misbehaving party can further this by flooding the network with duplicate or bogus messages, and much like the true anonymity scenario described above, can do so with impunity.

This thesis focuses on differentiable and categorizable anonymity. Nondifferentiable anonymity can be deployed using the implementations discussed, but with the caveat that the participants must be "known honest", and with a poorer resolution on the identity of alerts, raising the possibility of more granular corroboration. As the architecture in chapter 6 is specifically geared towards effective cross-site (cross-participant) corroboration, I do not consider this further.

To support these forms of anonymity, I adopt several components: a trusted party responsible for authentication, the use of message signing for differentiability, and publish-subscribe event infrastructure for distribution. As previously mentioned, this thesis does *not* seek to implement a pub-sub event distribution infrastructure from scratch, but the presence of one is crucial to ensure the anonymity of the data-privacy-enhanced events exchanged.

It is also important to note that the framework described in 3 allows all of these elements to be plugged in modularly, based on the requirements of the actual application; moreover, the implementations discussed in chapters 2 and 6 were designed to support different publish-subscribe architectures, and the current implementations of both use "reference systems" (Siena [20], JMS [151]) at the

distribution level. Siena is decentralized but is hierarchical, while JMS is an open standard that is largely implementation-dependent (although most implementations are centralized).

5.3.1 Event Model

Based on the model proposed in chapter 3, combined with the notion of *typed* events, a correlator can seamlessly support heterogeneous types of events. This heterogeneity is leveraged alongside an anonymity-capable event distribution infrastructure, as described in a following section, to map the different levels of anonymity to our model via the use of a uniform source identifier field.

- For completely anonymous sources, the source identifier is left blank. This is a degenerate condition and can be considered a special case of the other scenarios, and is not explored further in this thesis.

- Differentiably anonymous sources use a randomized-but-consistent string when producing events—either a universally unique identifier (such as UUID [60]), or an authority-assigned unique identifier. This identifier is created at the beginning of a correlation task and is not changed for the duration of the task. Optionally, producers may choose to change the identifier between each correlation task in a long collaboration to further obscure their identity.

- Categorizably anonymous sources use a combination of a *category identifier* (e.g., SIC codes for corporate sources [109]) along with a unique identifier.

- Finally, non-anonymous sources would simply use a descriptive string in the source identifier, based on a pre-agreed standard naming scheme (e.g., hostname of the sensor).

5.3.2 Authentication

As implied earlier, a trusted third party (TTP)'s primary responsibility is to authenticate participants. A publish/subscribe router *may* either coexist with the authenticator, or be distributed, where the TTP is used for authentication but events are then anonymously routed through a publish-subscribe cloud.

Authentication can be accomplished in one of several ways; if the TTP is responsible for message distribution, it may act as a direct authenticator and can enforce access control on the publish-subscribe network. If a peer-to-peer pub/sub system is adopted, then the TTP may simply be a key distribution authority that, upon verification of a corroborating peer, distributes keys to enable message signatures that other corroborating peers can trust come from a "known" source, i.e., given a message and a corresponding signature, the signature can be verified as having been generated by a TTP-issued key.

Note that a key signed by a TTP is *not* a guarantee of innocuous behavior; this architecture is designed to be robust in the case of known, but misbehaving, entities.

5.3.3 Malicious TTPs

One significant challenge exists with this authentication scheme—malicious TTPs, who can do one of several malicious acts: reveal the privacy of contributors; exchange or corrupt identities of contributors, invalidating differentiability; and, in the case of centralized event distribution, either corrupt contributors' data or feed invalid data. The worst-case scenario enables the authority to disrupt event correlation at each of the nodes. It is important to note that the malicious authority *still would not have access to the underlying data*, as the data is encoded in a fashion to preserve data privacy not only amongst collaborating peers but for the authority itself. As for data corruption, this can be mitigated by using a decentralized routing

mechanism for publication, e.g., P2P, since the TTP is no longer involved in the data exchange.

Unfortunately, corrupted or compromised identities remain a fundamental problem. Even true anonymity, which prevents identity compromise, poses its own problems, as the problem of determining identities is not generally solvable without such an authority *or* significant computational expense [39], so while a completely decentralized model where *both* anonymized routing and authentication may be possible, this architecture does not address that scenario. Among two architectures mentioned in the related work, [73] uses a TTP for group signature management, and [63] uses a "friendship" trust model. Neither directly enables anonymous differentiability, which is necessary in our correlation scenarios. Fortunately, many applications adopt well to the TTP approach; Worminator illustrates one such application and I discuss such "natural" TTPs in chapter 6.

5.3.4 Anonymity-Supporting Distribution Architectures

There are several desiderata that help differentiate the use of a central node, a cluster of nodes, or a decentralized, peer-to-peer collection of nodes in event distribution; the key considerations here are speed, resiliency, and anonymity guarantees.

- Distribution performance varies hugely, and is dependent on network topology, the distribution of nodes, the number of publishers and subscribers, and the overlap of subscribers' interest. *If* network bandwidth is not saturated and the network has optimal latency, centralized systems generally provide the best performance, as both publishers and subscribers communicate with only one other node, eliminating extra replication. On the other hand, distributed peer-to-peer systems frequently replicate events as part of event routing, but are robust to bandwidth saturation and network partitioning. Several other cluster-of-nodes systems, such as hierarchical event distribution, can offer

good performance under both ideal and less ideal network conditions, but incur privacy issues and still generally fail during severe network partitioning.

- Centralized systems are not resilient to network or server-side faults, although they are not reliant on client reliability. P2P systems are extremely resilient to network failures, although portions of the P2P network may be segmented in certain situations; still, limited corroboration may be preferable over no corroboration at all. Additionally, a central distribution node can be a target for DDoS or subversion attacks. Clusters vary widely based on the type and distribution of the cluster.

- Lastly, a trusted centralized distribution node provides the best anonymity guarantee, as it can reliably guarantee that its communicating peers are authentic, and can appropriately sanitize or transform network connection information to prevent source determination. On the other hand, if this trust is violated (e.g., a malicious TTP), that centralized node can wreak significant havoc, as it has relevant information for every publisher.

  A pre-established cluster of nodes can approach this level of privacy, as long as a honest TTP is present to "vouch" for the authenticity of the cluster. The cluster also provides limited resilience against corruption against a few distribution nodes, although sufficient subversion, resulting in a significant fraction of nodes as being untrustworthy, can pierce privacy by colluding on data replicated between the nodes.

  The anonymity of a P2P distributed system varies greatly. If nodes communicate only directly to each other, i.e., the publisher publishes to a subscriber by opening a network connection to it, anonymity is difficult—in fact, a node can easily build a table of signatures and the corresponding network nodes. Therefore, adoption of a less-deterministic publication path, such as

onion routing or memoryless random-walk routing (see section 4.2), is useful; determination of the source then requires a large number (greater than 50%) of malicious entities in the P2P cloud.

5.3.5 Routing Options: Channel, Content-Based, Destination-Based

The discussion of distributions so far does not consider the routing mechanism employed in our publish-subscribe architecture, both in determining the correct target entities and choosing optimal routing strategies to reduce latency.

As source-based routing is impractical due to our desire for source anonymity, there are at least three different routing algorithms that can be chosen. (Broadcast is always a routing option, but is essentially a baseline and is not considered further.)

- Route events based on a predefined *channel*. The channel can be implemented as a tag on events entering the distribution infrastructure, *or* the distribution infrastructure can employ different node(s)/overlays for different channels. This approach is the most straightforward and lends itself to corroboration applications where a large number of peers are exchanging information for the same purpose. This is the core routing mechanism used for Worminator; the goal there is to disseminate as much information as possible in as short a time as possible, and minimizing routing decisions to channels simplifies this problem.

- Route events based on content-based interest. A particular subscriber may declare their interest to corroborate event "X", and is not particularly interested in receiving events of other types. This reduces network communication by pushing much of the corroboration problem into the network. However, building the routing infrastructure is challenging as the events are privacy-preserving in the first place; the subscriber must declare their preference in a

matching privacy-preserving form (e.g., hash or Bloom filter), and then the distributed event distribution infrastructure must push these subscriptions to appropriate nodes. Our discussion on the implementation challenges and strategies for building efficient Privacy-Preserving Content Based Routers is considered outside the scope of this thesis; see [59].

- One last routing mechanism is particularly geared towards distributed P2P networks, and implements publish-subscribe in an entirely different mechanism: the published material itself is used as metadata to make *destination-based* routing decisions. For example, a Bloom filter may be used as a routing key to determine the target node in a distributed hash table. This is particularly useful for systems where channel-based or CBR-based routing decisions are practical, and avoids the problematic approach of broadcasting, which does not scale to large distributed networks. See [59] for further discussion on this approach. Another question of relevance in P2P networks is determining optimal *network schedules* for exchange amongst many peers of nodes; our work in [88] provides some early strategies for effective event dissemination. Work on decentralized exchange is ongoing orthogonal to this thesis.

5.4 Retrofitting Privacy onto Legacy Event Systems

The material in this chapter has been implemented across several different Intrusion Detection System (IDS) applications, which are described in chapter 6. However, many existing Internet-scale event distribution and correlation systems are currently deployed, and it is not feasible to assume that they will all be adapted to support privacy-preserving corroboration.

In order to work with these systems, therefore, privacy must be *retrofitted* onto them. This can be accomplished by implementing the model from chapter 3 as

a lightweight translation layer running on top of legacy event distribution and correlation systems. I first describe techniques to transform individual events for a variety of common event formats, and then briefly discuss strategies and challenges when retrofitting both distributors and correlators.

5.4.1 Rewriting Events

Many legacy event processing systems, including many of the ones described in chapter 4, frequently take one of two underlying event formats: collections of attribute-value pairs *or* a hierarchical tree of attributes, values and data, like XML. Here, I describe three different transformations: first, the baseline technique of transforming an entire event opaquely, followed by more specific strategies for both attribute-value and tree-structured events. Many other event types can be transformed using similar techniques.

Equality Matching

The first option is to treat an event as a datum which must be matched precisely during correlation. In this case, the problem is straightforward; the event is privacy-transformed and distributed, and other peers can similarly hash and distribute their events. A translation tool can be used to transform event matches into Bloom filters, frequency distributions, etc.

The advantage of this approach is that it will work with *any* event semantic, no matter the structure. While meaningful data cannot be extracted (which attributes about the event matched, etc.), it still follows the corroboration model. However, this is a very coarse-grained approach, since *all* attribute/value pairs or other data in the event must be identical.

Attribute-Value Pair Matching

For systems that treat events as collections of attribute-value pairs, i.e., $e = \{(a_1, v_1), (a_2, v_2), \ldots, (a_n, v_n)\}$, the event can be processed in one of several ways, with differing privacy guarantees while supporting retrofitting.

- Transform only the values and corresponding subscriptions, i.e., $e' = \{(a_1, p(v_1)), (a_2, p(v_2)), \ldots, (a_n, p(v_n))\}$. The advantage of this approach is that it's straightforward to determine which attributes match, and the values are typically unnecessary, as they're already embedded in the subscription. The privacy gain varies based on the language of discourse; given a sufficiently large distribution of possible values for $v_1, v_2, \ldots, v_n$, this may be sufficient. On the other hand, the presence of attribute names may give information away—especially attributes that were not present in the subscription.

- Transform both the attribute name and corresponding value pair for published events and corresponding subscriptions: $e' = \{(p(a_1), p(v_1)), (p(a_2), p(v_2)), \ldots, (p(a_n), p(v_n))\}$. This offers some more privacy than the previous approach, as only the presence of attributes the subscriber is aware of are revealed, and only matching values are revealed. All other attributes and values remain private.

- Privacy-transform the *concatenation* of the attribute value pair, i.e., $e' = \{p(a_1v_1), p(a_2v_2), \ldots, p(a_nv_n)\}$. Here, attributes are only discovered if *both* the attribute and value of any given pair match. Additionally, this produces a collection of single entities, not of attribute-value pairs. These entities can be transmitted as a set of attributes, i.e., $e' = \{(p(a_1v_1), \text{null}), (p(a_2v_2), \text{null}), \ldots, (p(a_nv_n), \text{null})\}$, or via a retrofitted set-based structure, such as a Bloom filter. Optionally, we can *also* hash the attributes by themselves to allow for presence testing in lieu of an exact attribute match.

While the above techniques allow for equality matching over individual attributes, this *still* does not allow numerical or timestamp inequality matching, e.g., "Corroborate temperatures greater than 60". However, this problem can be fixed by using the first or second technique and adopting a numerical privacy transform that supports inequalities, such as [3]. The third technique is still limited to equality matches, but it does provide the greatest privacy of the three, and unlike the first two can be used with the various set-based structures discussed in this chapter.

Hierarchical (Tree) Matching

The third subcategory represents hierarchically-structured events that essentially form trees that are composed of nodes, attribute-value lists associated with each node, and references to children. Nodes may also have a name and associated data. The canonical example of such a structure is XML [165], although the techniques described here can be generalized to other tree-based events.

For such events, the techniques used with events composed solely of attribute-value pairs can be adapted to such an approach.

- Walk through the tree and only transform the values and associated data, keeping the rest of the event structure intact. Once again, this approach reveals metadata that is implicitly contained within the tree structure or attribute names.

- Walk through the tree and transform *all* literals while keeping the underlying structure of the tree intact. For example, the XML message <temperature value="60" /> may become <feahijo ajife="33201" />.

- "Flatten" the hierarchical structure and, for each attribute-value pair found, concatenate the path of the containing node and the attribute name, and store it as a collection of privacy-transformed attribute-value pairs as per the second

strategy in the previous subsection. This assumes that a path can be clearly denoted; in XML, the XPath expression syntax [166] is a natural vehicle for expressing both the path and the attribute's name to which a value is mapped.

- Once again, flatten the hierarchical structure; this time, the path/attribute name is concatenated with the attribute value. Once again, this leads to a result set that can be embedded as a collection of attribute names or in a set-based data structure, and generally preserves more privacy than the previous approach at the cost of losing the ability to do inequality matches.

As with the attribute-value pairs, the last technique can be used to insert items into a set-based data structure, like a Bloom filter. The privacy guarantees are also very similar, although retaining the hierarchical structure may yield some more information, while flattening it may make it more difficult to brute-force possibilities.

5.4.2 Retrofitting Event Distribution Systems

As discussed in section 5.3.5, several different distribution systems can be used by our model. The most straightforward system, channel-based event distribution, puts all the logic in the actual destinations and naturally adapts to privacy-preservation; Worminator was designed and built as a translation and corroboration layer designed to run on top of legacy channel-based event providers, such as JMS.

Legacy content-based routers are significantly more difficult to adapt in an effective fashion. While CBR systems can be used by treating them as a channel-based system and using non-privacy-transformed event metadata as establishment of implicit communication channels, this defeats the purpose of content-based interest routing. Instead, a simple solution is adopted: the *subscriptions* used in the CBR system are *also* privacy-preserved, in a similar fashion, i.e., $\mathbb{S}' = \mathcal{P}(\mathbb{S})$. The

privacy-enabled subscriptions $\mathbb{S}'$ are then pushed into the CBR network as ordinary subscriptions would be.

If a subscription is envisioned as a collection of attribute names, comparison operators, and values $\mathbb{S} = \{(a_1, o_1, v_1), (a_2, o_2, v_2), \ldots, (a_n, o_n, v_n)\}$—and many of the CBR systems discussed in section 4.2 follow this model—$\mathbb{S}' = \{(p(a_1), o_1, p(v_1)), (p(a_2), o_2, p(v_2)), \ldots, (p(a_n), o_n, p(v_n))\}$ can be literally used as a subscription. Assuming the use of range-compatible transforms as discussed above, the set of operators $o = \{=, <, >, \leq, \geq\}$, which covers many of the basic predicates that CBR routers use. (It is important to note that, in-and-of-itself, partial matches such a "in" e are not trivial to support.) These privacy-transformed subscriptions correspond to the attribute-value pair transforms described above, where the attribute names and values are kept distinct. Similarly, XPath matching can be still be accomplished against flattened XML events. Of course, concatenated events only support $o = \{=\}$.

5.4.3 Retrofitting Event Correlators

Event correlators, such as the Event Distiller in section 2.4, support varying amounts of correlation capability. Based on the system being used, one of several different strategies can be adopted.

First, correlation rules can be rewritten much like CBR subscriptions. This supports entire-event matching, attribute/value pairs, and hierarchical events; it does *not* directly support set-based transforms. Temporal correlation can be supported via explicitly-published timestamps or implicit received timestamps. The same predicate matching operators are supported. Upon a successful correlation, however, the correlator cannot report on the original values that were used to create a match.

Alternatively, a two-phase correlation mechanism can be adopted. The first phase would employ a *dedicated* corroborator whose job is to perform temporally

constrained corroboration against incoming privacy-preserved events, and to update *local* event data with corroboration results. For example, a field may be added to the local event attribute-value list specifying the number of peers who have validated a particular event. The events are replicated and published accordingly into the third-party correlator, which can execute rules based on these "local" events. Original values can easily be re-reported, but only for equality matches. Finally, an optional transform could then be written to take generated resulting events and make them privacy-preserving before the results are transmitted.

Both the first and second approaches are compatible with the original Event Distiller. Wildcard binding works, although the wildcard value cannot be meaningfully reported using the first approach.

5.5 Potential Attacks

Like any other distributed protocol, there are several attack venues against the techniques described in previous sections. Here, I briefly discuss several classes of these attacks and mitigation strategies. Worminator, itself, implements sufficient privacy to effectively deal with "honest-but-curious" participants, and can be extended to handle the malicious approaches described here. This is not intended to be a comprehensive list; moreover, an implementation of these mitigation strategies is outside the scope of this thesis, and is left for future work (see section 7.3.2).

5.5.1 Pollution

The simplest form of attack against such a system is to have a malicious entity report large numbers of fake entries in the hope of confusing other peers who receive its messages. This does not require collusion or other forms of subversion against the network.

It is important to point out that the "pollution" problem is common to any DIDS; the key difference here is that the privacy-preserving properties of exchange make it trickier to identify the polluter. However, there are several strategies that one can still adopt. First, while the model presumes anonymity, it does *not* assume non-differentiability; in fact, it was observed earlier that differentiability is a key requirement for a single node to avoid masquerading as multiple nodes. As a result, while the actual identity of the polluter may not be known, the anonymized identity can still be used to judge, and if necessary, discredit the polluter. Such a discrediting can be done either at an individual site, i.e., a site may notice that one alert contributor's data distribution is significantly different from others—or a distributed voting algorithm can be used.

Finally, it must also be observed that the most effective defense against pollution is a prevalence of legitimate participants. As the results will show in chapter 6, the approach is extremely resilient to noise, especially as the number of collaborators increases.

5.5.2 Watermarking

Arguably, the most difficult attack vector to defend against is that of *watermarking*, where, if collaborators are known to each other, enables a single participant to pierce the veil of anonymous-but-differentiable by triggering events at peers' sites in a coordinated fashion.

For example, a malicious insider could execute portscans from a known source to a specific peer known to be participating in a collaboration network, and can then wait for corroborating evidence to come from that peer ("watermarking"). While the event would be tagged with a unique identifier, the insider could scan the event, and if the known source is found, draw an association.

This is fundamentally a difficult problem while maintaining differentiable

anonymity, as the fundamental goal is to enable differentiated alert source determination. The only truly general solution may be to eliminate differentiability, but as discussed before, this has other negative consequences. Another strategy would be to keep differentiable anonymity, but to *mask* it and only use it when necessary (i.e., during selective verification). This can be accomplished by using, for example, an Onion routing approach [54]—where the original identity is derivable, but not without the cooperation of many peers. An alternative approach would be to have nodes aggregate and republish data that others have given it. However, this can significantly increase the amount of data exchanged and has implications for determining an appropriate consensus on a candidate suspect.

5.5.3 Collusion

The techniques described in this chapter assume that each participant, while possibly curious, does not collaborate in a potentially-malicious attempt to glean additional data. If they do, there is the potential for several attacks, most notably those of identity detection, brute-forcing, and significant pollution.

Identity detection may be possible via collusion *if* a significant percentage of the nodes collude *and* the set of participants is known, by reducing the probability of a bad guess (i.e., since the colluders are presumably aware of their mutual identities). Brute-forcing is rendered more possible as the ability to correlate alerts between two sites reduces the likelihood of false positives providing a reasonable "plausible deniability". Finally, the most serious aspect of collusion is cooperative pollution; peers that receive such cooperative polluted messages will rank matching entries with a higher threat metric, even if they are not of concern. In the worst-case scenario, this provides an effective denial-of-service as nodes may choose not to communicate with legitimate endpoints, because polluted alerts have confused sites' defense mechanisms.

There is no perfect solution for collusion detection, although some form of distributed verification and trust may be useful in solving this problem. For instance, legitimate nodes may choose to sprinkle in a few "fake alerts" to see if polluters decide to mimic them in order to confuse the global threat picture. However, in order to resolve such a dispute, a central authority, similar to the ISACs described in chapter 6, may be necessary.

5.5.4 Mimicry

All of the techniques described in this chapter are designed to produce, essentially, differentiable transformations of data in a manner that the original data cannot be recovered. The simplest approach, hashing, is brittle with respect to the data that is being hashed, i.e., if one bit of that data is changed, a hash value changes significantly. Incremental techniques, such as N-gram analysis, work around this problem by breaking the input datum into a set of features which are then transformed, and looking for a *prevalence* of matching N-grams.

However, this looser prevalence requirement makes for an interesting problem: what if a datum can be composed to look like another datum by matching the prevalence requirement while being something significantly different? It turns out that this is not a theoretical question; [168, 77] studied this problem for different domains. Ultimately, the severity of the problem depends on the application at hand; for IP-based collaboration as described in section 6.4, this simply devolves to the noise problem, as the features are fundamentally brittle and mimicry attacks can be treated as pure noise.

For payload analysis (section 6.5), however, the problem is more significant. Mitigation strategies can include moving the corroboration problem to a domain where mimicry is hard; for example, given network traffic, perhaps corroboration can be performed at a higher level (e.g., application state) which focus on the

effect of traffic, as opposed to the traffic itself. A number of strategies specific to payload-based corroboration are discussed in future work.

5.6 Summary

This chapter introduced and comprehensively covered a diverse array of methodologies for privacy-preservation, discussed requirements and desiderata to maintain anonymity, discussed how existing event systems can participate in such a system, and described several potential attack vectors for the techniques. The next chapter introduces an implementation, Worminator, that uses these techniques. Results are presented for both individual feasibility studies (e.g., the effectiveness of a timestamp Bloom filter) and overall performance for real-world scenarios.

Chapter 6

Privacy and Intrusion Detection

As alluded to in chapter 5, many of the techniques presented have different levels of utility, based on the application. In this chapter, I discuss each of these methodologies and techniques in the context of a comprehensive set of Intrusion Detection applications, the most important of which is Worminator, a ground-up rewrite of the XUES platform with privacy mechanisms for the purposes of *Collaborative Intrusion Detection*.

Collaborative Intrusion Detection proposes the *sharing* of intrusion alerts and associated models in order to get a global view on network/Internet threats. As such, it fundamentally runs into significant privacy issues, as participants may be unwilling to release sensitive information pertaining to their network topology or transmitted data. Worminator strikes a balance between the privacy requirements of participating organizations and effective Intrusion Detection alert corroboration. It has broad applications, ranging from sharing simple lists of sources, to automatic malicious payload detection and signature generation, to even traffic modeling in mobile ad-hoc networks (MANETs). It is not itself a sensor; instead, Worminator is capable of using different third-party sensors, including both commercial-off-the-shelf (COTS) and in-house research tools, such as the Columbia IDS PAYL [174, 172]

and Anagram [173] anomaly detectors.

This chapter is organized as follows. First, a detailed overview and motivation for collaborative intrusion detection is discussed. Next, I briefly describe each of the different underlying sensors used, including both of our in-house anomaly detectors, and how they particularly dovetail with the privacy techniques described earlier. Two specific collaboration domains—packet header-based misuse detection and payload-based anomaly detection—are then designed and validated in a privacy-preserving manner. Both quantitative results and empirical analyses are presented, demonstrating the value of corroboration for alert accuracy as well as the privacy guarantees afforded by this approach. Finally, a new application of collaborative traffic *model* exchange for new network topologies, such as MANETs, is briefly discussed. The chapter concludes with a brief look at other potential applications for the developed Worminator technology.

6.1 Collaborative Intrusion Detection

Internet-based attacks, ranging from distributed denial-of-service attacks to trojan, worm, virus, and direct attacks have been growing in frequency, size, and threat [153]. Of particular concern are the growing class of *zero-day attacks*, which refer to successful exploits of vulnerabilities the same day, if not before, they are disclosed by the software vendor or security community. It is hypothesized that these and other rapid vulnerability exploits are often preceded by difficult-to-detect *stealthy scans* to determine hosts vulnerable to specific exploit vectors. Such stealthy scans are generally accomplished by using many probes, distributing scan patterns across the Internet so that any given target is hit by a particular source at very low rates.

Intrusion detection systems (IDSes) are often designed to detect such scan attempts, but as most current commercial deployments are typically constrained

within one administrative domain, they lack information about global scanning patterns and have little chance to detect such behavior—an individual COTS IDS sensor either misses broad, slow scans (as they require too much state), or if their sensitivity is turned up, produce far too many alerts, making it infeasible for a system administrator or an automated response mechanism to respond in an appropriate fashion. Such scans can be used to build hitlists, which in turn can be employed by rapid "Warhol" worms [140], slow surreptitious worms, *and* targeted attacks.

In general, intrusion detection systems face the very real threat of information loss from the sheer rate of available information. Schaelicke, et. al. [134] are decidedly pessimistic about the ability of relatively powerful commodity hardware and network links to absorb peak alert loads, noting that an IDS is effectively neutralized by the loss of alert data resulting from a database unable to keep up with incoming network data. DShield reports about 10 million alert records added daily. Figure 6.1 shows the increase in contributed data per month between January 2002 and May 2003.

Therefore, the ability to rapidly and correctly identify, rank, and react to both active attacks *and* stealthy probes is of importance. The exchange of alert data between administrative domains can effectively supplement the knowledge gained from local sensors. Global information can aid organizations in ranking and addressing threats that they do perceive and in alerting organizations to threats they would not otherwise have recognized. We term this *Collaborative Intrusion Detection*, or CIDS. CIDS is an evolution from *Distributed Intrusion Detection*, or DIDS, which also employed multiple sensors but did not distinguish between organizations, instead uniformly viewing them as a set of nodes across a single or many different networks.

CIDS, on the other hand, leverages the *data diversity* gained through multiple sensors at different sites to better rank and understand the meaning of common scans

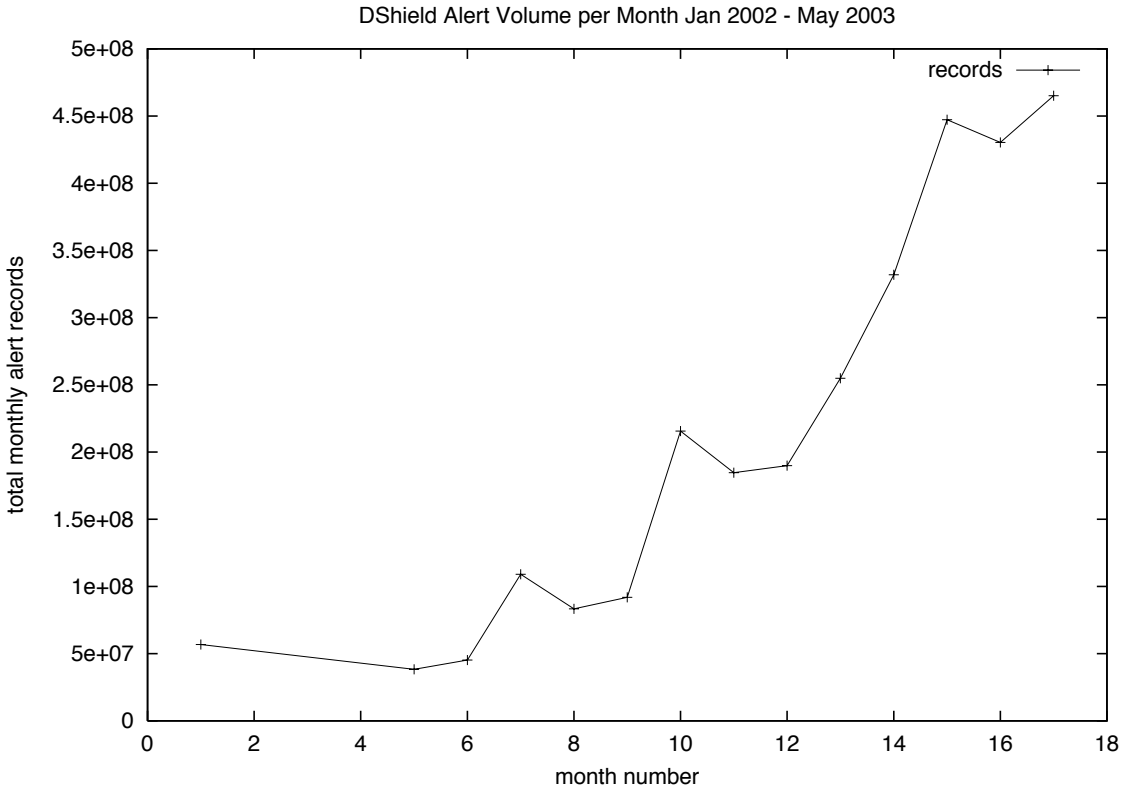

Figure 6.1: DShield monthly alert record contributions. The graph is not cumulative, but rather shows the rapid increase in contributed alert information per month as DShield grew in popularity.

and attacks. In particular, with sufficient global scan data from a CIDS, it should be possible to create a *profile* of scanner behavior to help differentiate benign-and-curious scanners, worm-infested machines, and actively malicious entities planning their next attack. However, potential contributors, both academic and commercial, are unwilling to release alerts, for fear of revealing their network topology, open services, or even the distribution of source addresses or communications of peers that regularly communicate with them. This concern is what motivates privacy-preservation, and leads to the main hypotheses of the Worminator project.

6.1.1 Hypotheses

This chapter explores three key hypotheses connected to privacy-preservation and intrusion detection.

> A privacy-preserving architecture enables:
>
> 1. The participation of a broad group of contributors to detect both slow, stealthy scans that act as precursors of attacks, as well as traffic information necessary to automatically build defenses for actively ongoing attacks, while ignoring false positives;
> 2. The ability of contributors to exchange *vulnerability-specific* information to enable effective signature generation to help defend against very broad exploits;
> 3. The ability of ad-hoc communication participants to determine each other's communication profiles, and develop a trust model to determine exchange, without revealing intended future communication.

These hypotheses are explored in great detail in sections 6.4, 6.5, and 6.6, respectively, after Worminator's requirements and a technical overview are discussed in the next sections.

6.1.2 Requirements

From the discussion above, a set of requirements can be derived to guide the design of the system. These requirements can be viewed as a specialization of the requirements originally discussed in section 1.3.

- The exchange of alert information must not leak potentially sensitive data, and should be robust to "curious" participants, yet should exchange enough data for meaningful corroboration.
- The system should not involuntarily reveal the identity of participants, but should allow the ability to distinguish between participants to help accurately measure threats.

- Large alert rates may hide stealthy activity; any reasonable solution must deal with or reduce the effects of these rates, including the use of temporal constraints.

- The system must scale to handle *active* alert flows for anything from slow, surreptitious scans to active worm attacks.

- The system should handle a heterogeneous set of intrusion detection sensors to cover different applications and collaboration scenarios. Ideally, the system should be sensor-agnostic, minimizing the amount of work needed to integrate any individual sensor.

In the next section, I describe Worminator, our implementation of a CIDS that aims to fulfill these requirements with the inclusion of privacy-preserving mechanisms.

6.2 Worminator Overview

A ground-up rewrite of the XUES platform, Worminator adds support for privacy-enabled data structures and transforms to take intrusion alerts generated by underlying sensors and embed them in Bloom filters or other hash-based structures. Like EP, Worminator is fully modular and supports heterogeneous data types, sensors and communication networks, and supports near real-time event processing and correlation/corroboration.

Preserving Privacy

While IDS alerts themselves could be distributed, there are two substantial disadvantages to doing this: first, organizations may have privacy policies or concerns about sharing detailed IP data, some of which might uncover who they normally

communicate with. Second, these alert files grow rapidly given substantial traffic. While parameters may be tweaked to reduce potential noise, a preferable solution would be to encode the relevant information in a compact yet useful manner.

Worminator provides the ability to represent alerts via one of several compact formats, including hash sets, Bloom filters, frequency distributions, Z-Strings, and n-gram signatures, as described in forthcoming sections. Since these are compact, one-way data structures, we get several benefits:

- *Compactness:* Most of these alert representations are extremely compact; for example, a Bloom filter smaller than 10k bits in size is still able to accurately verify thousands of entries.

- *Resiliency:* Worminator's support of decentralized corroboration enables effective alert-sharing even in the case of worm storms or network segmentation due to failures.[1] Several of Worminator's data structures are also resilient when scaling: for example, smaller Bloom filters saturate quicker, and start producing false positives, but do not produce false negatives. Moreover, corroboration against multiple alert lists can effectively ameliorate these false positives.

- *Security:* By utilizing a one-way data structure, organizations can correlate watchlists or models of behavior without releasing actual IP or payload data, satisfying privacy needs while being able to participate. We demonstrate that the above-mentioned data structures do indeed effectively corroborate while maintaining extremely good privacy metrics.

[1]This is predicated on the use of an effective distributed event dissemination infrastructure; Worminator itself is completely decentralized, and so does not impose extra conditions on such a dissemination platform.

6.2.1 Architecture

First, the construction of privacy-preserving transforms by Worminator is employed to protect the confidentiality of the data being exchanged between domains. Since information is also compacted by the Bloom filter, correlation between peers becomes extremely cost-effective in terms of bandwidth and processing power.

A key motivating factor for organizations to join a collaboration group in performing distributed intrusion detection is that participants can implement fast mitigation strategies against threats they otherwise would not have known about. For example, DOMINO [179] illustrates the advantage of small blacklists (around 40 entries) that retain their efficacy even when data is fairly stale. Other responses include the content filtering strategies proposed by Moore *et al.* [101], prosecution, or military action.

One of the most important decisions we make is to employ the use of "watchlists," or lists of IP addresses or payload exploits suspected of subversive behavior. The task of the distributed detection system is not to analyze the network or host events of other domains, but rather to correlate summaries of alerts to identify attackers. Therefore, watchlists encapsulate the appropriate information to exchange.

The goal, in the case of alert and suspect payload sharing, is to enable sites to maintain a secure *watchlist* of alerts seen locally and from other sites, and to generate a *warnlist* of significant threats if they have been correlated as having been seen at multiple sites. This warnlist can then be reported to network administrators or could be directly mapped to firewall or signature scanner rules to prevent impending attacks. Depending on privacy policies, these local warnlists may also be explicitly replicated to other sites to enable a fast global-scale response.

For alert exchange, *watchlists* consisting of encoded Bloom filters, hash sets, etc. are created and exchanged between peers' sites to corroborate local data.[2] A

[2] As previously implied, raw alert data can be collected if participants are willing.

corroborator runs as a Worminator module on each site, and can perform decentralized privacy-preserving corroboration using temporal corroboration structures, as discussed in 5.2.2. Corroborated alerts can then be either reported via a website, as feedback into the IDS system as additional metadata to determine threat levels, or as a *warnlist* published back to the community. The latter may optionally be published *without* privacy-preserving mechanisms to enable peers that have not corroborated to take proactive measures against a correlated threat.

6.2.2 Implementation and Deployment

Worminator is written in about 20,000 lines of Java and Python code, and leverages a number of J2EE (Java 2 Enterprise Edition) providers, including the PostgreSQL JDBC provider for querying databases, the Apache Tomcat JSP/Servlet container for the UI, and the JBossMQ JMS (Java Message Service [151]) provider as a publish/subscribe infrastructure to communicate events. Worminator, like XUES, is completely pluggable and supports different sensor and alert types, correlators, and communication frameworks. Just as importantly, it supports *continuous validation*; alerts from sensor(s) are exchanged immediately, and correlation runs real-time to glean data as soon as possible to help prepare defenses against pending attacks or fast-moving worms.

Watchlists and warnlists are communicated using the IETF IDMEF draft standard (Intrusion Detection Message Exchange Format [35]). The configuration to determine alert communication and workflow is written in an XML rule file and is site-specific.

Deployment

Worminator was successfully deployed for IP-based alert exchange at three academic and two commercial sites over a six-month period. Deployment discussions with other organizations have been ongoing; regulatory compliance and legal issues

make deployment an extremely slow, unpredictable process.

We have also been in communication with various ISAC (Information Sharing Alert Coordination) organizations, set up to enable exchange of Internet threats, especially the REN-ISAC (Research and Education Network ISAC [126]). An ISAC forms a natural authenticating and anonymization authority, and future participation with the REN-ISAC is possible, and may help encourage more participants to join the Worminator effort in the future.

Finally, the performance benchmarks discussed in this chapter were tested on a dual-processor 3GHz Xeon system with 4GB of RAM, running Java SE 5.[3]

6.3 Sensors

Intrusion Detection sensors typically partition into two groups: *misuse detection*, which uses a predefined corpus of rules to determine scanning and suspicious behavior, and *anomaly detection*, which builds "models" of normal behavior and flags suspicious behavior as those that differ significantly from this norm.

Worminator has been tested with both categories of sensors; in fact, the modeling techniques in the two anomaly detection sensors closely resemble several of the privacy-preserving data structures discussed in chapter 5, and thus naturally scale to the applications described here. It is important to note, however, that the algorithms described are not bound to these sensors, per se, and they may be usable with other host- or network-based misuse and anomaly sensors.

6.3.1 Misuse Detection

As mentioned, misuse detectors are essentially rule-based correlators that build alerts from incoming traffic based on a static rulebase, looking at packets for content

[3]Only one processor was used, as out-of-the-box Java uses one CPU per process.

patterns (e.g., matching a known signature) or distribution patterns (e.g., a certain volume of packets are received in a certain timeframe from the same source). Outputted results usually include a pair of (*IP*, *port*) tuples representing the source and the target, as well as other metadata, such as alert time, packet length, possibly a score, etc. From this, features can be chosen for exchange.

The classic misuse detectors in the IDS community, Snort [129] and Bro [118], can be used with Worminator, but they are ill-suited for long-term scan detection. Instead, the main COTS IDS sensor used in Worminator was [128]. It is a *surveillance detector*, a form of misuse detector that maintains a significant amount of state to detect long-term scan attempts. Worminator read alerts from the underlying sensor, extracted appropriate features, packaged them into Bloom filters, and exchanged them with peers to determine sources of interest, targeted services, and potentially exploit vectors.

6.3.2 Anomaly Detection

Worminator can theoretically be used with virtually any anomaly detector that publishes alerts with discrete features. For the purposes of the payload-based experiments described in sections 6.5 and 6.6, I opted to work with two exemplary anomaly detectors developed in the Columbia IDS lab: PAYL, which implements anomaly detection based on frequency-based 1-gram modeling, and Anagram, which uses binary-based mixtures of higher order n-gram modeling ($n > 1$). Both sensors train on *normal* unencrypted content flows and employ service-specific models to test for suspicious traffic.[4] Alerts are generated on traffic sufficiently deviant from normal; it is these alerts that we wish to share with other sites to resolve false positives from true zero-day attacks. The reader is encouraged to refer

[4] Anagram utilizes other information and is semi-supervised, i.e., it uses tools and heuristics to ensure the training data is sufficiently devoid of anomalous n-grams in order to minimize false negatives.

to [174, 172, 173] for detailed descriptions of the aforementioned sensors.[5]

PAYL: 1-gram frequency modeling

PAYL, or PAYLoad anomaly detection, was the first significant packet payload-based anomaly detector that was developed at the Columbia IDS lab, and one of the first in the community [174, 172]. PAYL's models are 1-gram byte frequency distributions conditioned on packet length and targeted service; data shows that network traffic differs considerably given these two features. During the training phase, incoming packets on a given port are frequency analyzed, and the distributions are clustered together based on the payloads. This clustering process results in a number of centroids that characterize the traffic for the chosen port and packet length. By considering all the centroids obtained for different port and length, we obtain a PAYL model.

Incoming packets are then compared against this model in the detection phase to check for anomalies; traffic is classified as normal or malicious by computing the Mahalanobis distance between the distribution of the candidate packets and the frequency model. A larger distance means bigger deviation from the model and a more abnormal packet; thresholding differentiates normal from malicious traffic. It is also possible to check bidirectional traffic in the same manner and to detect worms by performing ingress/egress correlation.

A *raw* PAYL alert typically contains metadata, including the source and target IP/port pair, payload length, and score (distance from model). Additionally, the suspicious packet may be included in its alert. While the payloads can be shared, they significantly increase alert sizes and run into privacy issues, especially for misclassified traffic, i.e., false positives. While PAYL's false positive rates have been

[5]Both PAYL and Anagram were primarily written by my colleague, Ke Wang, and form a significant part of her thesis [171]. Parts of Anagram use Worminator libraries written by myself, most notably the Bloom filter implementation and its incremental multiple n-gram analysis techniques.

determined to be very low [172], the notion of transmitting any raw payload inhibits collaboration among defensive sites.

Anagram: n-gram binary modeling

Anagram [173], the second payload-based anomaly detector developed at Columbia IDS, uses binary-based high order n-gram modeling, i.e., it simply models the *presence* or *absence* of particular n-grams as opposed to their relative frequency of appearance. Compared to 1-gram, higher order n-grams are better at modeling sequential content information in packets, and thus it is capable of detecting significant anomalous byte sequences and their location within a packet. To avoid significant memory overhead associated with n-gram frequency distributions, only a binary (yes/no) statistic is kept for each possible gram. Scoring is accomplished by counting the percentage of not-seen-before (i.e., unusual) n-grams out of the total n-grams in the packet, and thresholding is again applied to differentiate traffic.

Surprisingly, analysis in [173] shows that binary-based modeling produces extremely good results; it turns out the additional data representation of frequency-based modeling is less advantageous when the space of potential grams grows significantly (e.g., the likelihood of having significant frequency information for distinct 5-grams, or 256^5 grams, is significantly smaller than for the 256 distinct 1-gram), and the representational power of higher-order n-grams effectively offsets the loss of frequency information. However, even though binary-based modeling significantly reduces space overhead, there is still a significant number of possible n-grams as n increases, and a typical hash set structure uses at least 4 bytes per entry. Since only the binary set property is needed, we can use Bloom filters (§5.2.2) as natural data structures for an Anagram model, reducing data requirements by an order of magnitude.

The structure of a raw Anagram alert is similar to that of a raw PAYL alert.

6.4 IP-Based Collaboration and Scan Detection

The first application of Worminator is to enable the exchange of IP header-based alerts. These alerts, typically generated by a local misuse or anomaly sensor, consist of source and destination endpoints associated with some metadata (perhaps information on the scan/attack vector, the payload length, time of the scan/attack, etc.) It is important to note that the sensor itself is not constrained to IP headers; rather, in this environment the sensor does not reveal payload information in its alerts.

This model is the one used by many traditional network-based IDS sensors, including Snort [129] and Bro [118]. Enabling the exchange of alerts enables a global *watchlist*, i.e., information about the sources of a scan or attack. These watchlists are privacy-preserving, are exchanged with collaboration peers, and verified against each peer's local database of alerts. Corroborated IPs that exist in the local database *plus* one or more peers are then published into a *warnlist*, which may optionally be distributed without privacy preservation in order to enable *all* nodes to fortify defenses against a common scanner/attacker.

By building and optionally exchanging these warnlists, participating nodes can get *longitudinal* attack information across IP space, and can choose to rank threats based on their perceived risk. These longitudes can vary based on a number of factors, including time, target classes, targeted services, etc. For example, a common source scanning banks in particular may be viewed as a more serious threat, as opposed to a source exhibiting worm-like random scanning behavior. Worminator does *not* rank these threats automatically, but rather facilitates the exchange of the watchlist and generation of the corresponding warnlist in order to make such rankings possible either by a human or automated analysis tool.

The rest of this section is organized as follows. First, a precise description of the corroboration in this scenario is made. Efficient ways of supporting the corroboration

are then discussed, including from computation, storage, accuracy, and privacy perspectives. Finally, data collected from a test deployment of Worminator amongst 5 sites is analyzed to demonstrate the empirical observations that can be made with such information; these observations form the basis of a longitudinal study of common threats over space and time.

6.4.1 Corroboration Methodology

A set of participants $A, B, C, \ldots$ each contain an IDS sensor, e.g. $\mathcal{I}_A, \mathcal{I}_B, \mathcal{I}_C, \ldots$ that produces alerts $\mathcal{E}_A, \mathcal{E}_B, \mathcal{E}_C, \ldots$, possibly at different times, e.g., we may have a collection of events $e_{At_1}, e_{Bt_2}, \ldots$. While each sensor produces alerts asynchronously, a common (global) clock is assumed. If a site were to have multiple sensors $\mathcal{I}_{A_1}, \mathcal{I}_{A_2}, \ldots$, we remap each of those sensors as appearing from different sites, i.e., $A_1 \rightarrow B, A_2 \rightarrow C$, etc. This methodology does *not* impose that all sensors must therefore be viewed as a flat group; if desired, the corroboration described here can be done hierarchically, where a site may choose to aggregate data from individual sensors and publish them as a single source. However, a global hierarchy is not maintained; maintaining such a hierarchy makes anonymous differentiability more complex—both from a privacy and anonymity-piercing perspective *and* from a Sybil attack-like [39] poisoning perspective—and so I leave such a characterization to future work.

An individual alert e has features $e_\alpha, e_\beta, \ldots$; for the purposes of this experiment, we choose and extract three specific features to be corroborated.

- The *source* e_s, representing the source identifier in the IDS alert. Typically, this is an Internet-accessible IP address. The presumption is that this IP address is remote to the set of participants, although the system does not intrinsically require this.

- The *destination service* e_d, representing the service aspect of the destination endpoint. Typically, this is mapped to the destination port. Destination IPs are not kept as they are less useful when corroborating *across* participants; it is fundamentally assumed that each participant has only a complete view of *their* traffic. If they were to have access to a more global perspective, this corroboration application would be unnecessary. (One special case is that of a corroboration scenario when one of the participants, or a node on the participant's network, is themselves an attacker; in this case, corroborating destination addresses may enable quicker detection. However, I leave this to future work; amongst other challenges, participants are typically unwilling to report on their nodes' activities directly.)

- The *time* e_t, representing the timestamp as written by the IDS system. This may vary; some IDS systems produce multiple timestamps, corresponding to the detection window as well as the publication time. This system is capable of using any of these, as well as an implicit alert timestamp that is generated by the receipt of an alert from a remote site. More specific timestamps may enable a higher-precision corroboration, but as timestamps must be transmitted unencoded, this may not always be possible.

Given these features, we design a set of alerts $\mathcal{A} = a_1, \ldots, a_n$, each alert $a_i = \{e_s, e_d, e_t\}$, and then *privacy-transform* them via one of two mechanisms. (Frequency models are *not* considered for this application, as they are ill-suited to support entity matching.)

- A privacy-enabled alert set $\mathcal{A}' = \mathcal{P}(A) = \{p(a_1), p(a_2), \ldots, p(a_n)\}$, where each privacy-transformed event $a' = \{p(\{e_s, e_d\}), e_t\}$. That is, a privacy transform is applied to the source/destination tuple and is then associated with the timestamp in question.

- A privacy-enabled alert model $\mathcal{A}' = \mathcal{M}(A)$, via the creation of a Bloom filter $\mathcal{A}' = \mathcal{B}(\{e_{s_0}, e_{d_0}\}, \{e_{s_1}, e_{d_1}\}, \ldots, \{e_{s_n}, e_{d_n}\})$. Once again, the timestamp itself is not inserted in the Bloom filter. The aggregate Bloom filter is assigned a pair of timestamps $\mathcal{B}_{t_0} = e_{t_0}$, $\mathcal{B}_{t_n} = e_{t_n}$ as metadata, $t_0 \leq t_1 \leq \cdots \leq t_n$.

The resulting set/model $\mathcal{A}'$ is the participant's *watchlist*, and is published to peers. Note that the set/model essentially acts as a "snapshot" of the alerts received between the last publication t_l and the current time t_c. This publication schedule is flexible; if more frequent exchanges are desired, the window $t_w = t_c - t_l$ can be reduced, creating smaller sets/models that are published more frequently. This window size also effects the granularity of the timestamps for a Bloom filter-based model, i.e., typically $\mathcal{B}_{t_n} \leq t_c$, $\mathcal{B}_{t_0} \leq t_l$ and $\mathcal{B}_{t_n} - \mathcal{B}_{t_0} \leq t_w$. (In theory, if a participant wishes, they could retransmit earlier events, which would produce a correspondingly larger window; this case is not considered.) Additionally, the features published are also flexible; if desired, any of the features can be removed, e.g. watchlists consisting only of source IPs can be exchanged. This allows for threat-independent corroboration of suspicious sources, and a few of the later aggregate experiments assume source IP-only corroboration.

Upon receipt of one or more watchlists, a *warnlist* $\mathcal{W}$ can be generated via the corroboration function $\mathcal{W} = C'$ as described in section 3.2, with one significant difference: as Bloom filters are binary-based models, the threshold function τ is irrelevant; instead, a simple binary thresholding is done to determine whether entries do corroborate against the BF.

For example, $\mathcal{W}_{AB} = C'_A(\mathcal{A}'_B)$ represents the set of alerts at participant A that match entries in B's watchlist. (As would be expected in corroboration, $\mathcal{W}_{AB} = \mathcal{W}_{BA}$.) Multiple watchlists can be corroborated serially or in parallel, e.g., $C'_{C'_A(\mathcal{A}'_B)}(\mathcal{A}'_C)$ or $C'_A(\mathcal{A}'_B \cap \mathcal{A}'_C)$ given watchlists from B and C. While the latter is generally more efficient, it presumes the simultaneous availability of both instances of $\mathcal{A}'$, whereas

the serial corroboration can be done progressively.

As previously mentioned, participants may choose to distribute a warnlist $\mathcal{W}$ without privacy transformation. Consider the three-participant scenario A, B, C, where a given source s_i scans both B and C on a particular service d_j. B and C will build local alerts and will distribute the watchlists $\mathcal{A}'_B, \mathcal{A}'_C$, both of which contain $e'_x = p(\{s_i, d_j\})$. However, until A witnesses activity from s_i, it will not be able to corroborate against e'_x. The situation remains the same if 100 participants are present in the collaboration network and 50 of them observe some behavior of interest. One can argue that, given corroboration, source s_i no longer represents "private" information; it has been observed elsewhere. An unencrypted warnlist $\mathcal{W}_{BC}$, $e_x \in \mathcal{W}_{BC}$, gives A the opportunity to prepare for potentially suspicious behavior from s_i—which may be important when attempting to, for example, stem a worm attack. Finally, such warnlists may be publishable to other distributed systems, such as DShield, as a public service. Note that Worminator does *not* mandate such a system; it is flexible to information disclosure policies. However, Worminator *does* support this form of collaboration in conjunction with privacy-enhanced watchlists.

Finally, given one or more warnlists $\mathcal{W}_{XY}$ or even $\mathcal{W}*$, i.e., a warnlist composed of all warnlists published by participants, various defensive strategies can be adopted. At the simplest level, the warnlist can be used to alert system administrator(s). They can also be used to drive firewall and access-control rules. More sophisticated strategies can be used, such as thresholding or threat ranking based on the number of participants that corroborate a particular alert. Worminator is compatible with such systems, so long as they are decentralized. (Centralized response and mitigation systems can be adapted, but then effectively require the publication of non-private warnlists or summaries).

6.4.2 Evaluation and Test Data

The following subsections evaluate the above methodology using a number of criteria, including performance, space and transmission requirements, accuracy, privacy gain, and results gathered. In order to complete these evaluations, data was collected as part of the deployment discussed in section 6.2.2. Raw data was collected where possible to enable a comparative evaluation between raw correlation and various forms of privacy-preserving corroboration.

Approximately one year of data (March 2005–March 2006) was collected from 5 different sensors, three of them academic (including Columbia) and two of them commercial (organizations in New York and in Washington DC); the statistics are shown in table 6.1. Note that the total row is only a strict total for the number of alerts, as IPs and IP/port pairs overlap by definition when considered across sites. Additionally, the COTS sensor used with Worminator was capable of detecting multiple targeted ports per a single alert. Unfortunately, for technical and legal reasons, we were not able to run all of these sensors together for the entire experimental period, although several subsets did collaborate for long periods of time.

Site	Time (days)	# Alerts	# Distinct IPs	# Distinct IP/port pairs
Academic 1	314.87	3919604	86108	4576155
Academic 2	28.53	823631	28838	844288
Academic 3	164.56	2811553	45255	3605271
Commercial 1	242.52	923482	119675	325283
Commercial 2	373.68	543979	60585	378062
"Total"		9022249	322391	9669162

Table 6.1: Statistics on Collected IP-based data.

The above data immediately suggests that academic networks, which are frequently more "open" and may carry a much broader variety of content, generate a higher volume of alerts than the commercial sites. The corresponding alert rates are discussed in greater detail in the next subsection, while more distinguishing

characteristics are discussed in the analysis in section 6.4.8.

6.4.3 Performance and Scalability

In order to be useful, this system must be able to function in near real-time, i.e., at the same pace as or faster than collaborating IDS sensors. The feasibility of matching alert rates is examined for both the set-exchange (hash set) and model (Bloom filter) scenarios.

Alert Rates

Table 6.2 shows the corresponding alert rates, given the data in table 6.1, for each of the collaborating sensors in the aforementioned experiment.

Site	Alerts/Min.	Alerts/Sec.
Academic 1	8.64	0.3341
Academic 2	20.04	0.3341
Academic 3	11.86	0.1977
Commercial 1	2.64	0.0440
Commercial 2	1.01	0.0168

Table 6.2: Alert Rates of Participating Worminator Sites.

These alert rates are reasonably manageable, but the question is more of scalability: ideally, any such corroboration system should be able to handle thousands of such participants, i.e., thousands of alerts per second being distributed amongst participants. Distribution issues aside, two key questions remain:

- Do privacy transforms measurably affect the publication rate at individual nodes?
- As thousands of such alerts are distributed, can the recipient site corroborate them sufficiently quickly?

The next subsections explore these questions. First, I measure the effect of corroboration on alert rates. Second, the computation and space overheads of privacy-preservation are measured. Finally, I address the challenge of providing efficient temporal corroboration and the corresponding computation and memory overheads.

Alert Reduction

To some extent, alert reduction is one of the most significant features of Worminator—unless the number of alerts is sufficiently reduced through corroboration, Worminator does not help the individual site's IDS operator. In order to characterize Worminator's potential in reducing alert rates via corroboration, the aforementioned datasets were first intersected using raw data and using (IP, port) tuples. This corroboration was done 2-, 3- and 4-way; the results are shown in figures 6.2–6.4. *N*-way corroboration is defined as an AND function, i.e., the set of alerts that appear in *all* of the specified sites. "OR" corroboration is less useful, as a 3-way OR corroboration would simply be a pair of two 2-way corroborations. Since the scales of the graphs vary widely, figure 6.5 is also included; it charts the maxima of each of the figures. (This "scale figure" is also used in several of the subsequent figure sets where the Y-axis ranges differ widely.)

Each of the three figures have two Y axes; the left Y axis, corresponding to the bar values, shows the absolute number of corroborated alerts, while the right Y axis, corresponding to the superimposed line, shows the number of corroborated alerts as a *percentage* of the first site's total tuple set, i.e., the number of distinct IP-port tuples. Note that the X axis therefore features *N* copies of each unique *N*-way corroboration so that the percentages can be shown over all cases (and specifically in the case of very asymmetric original alert sets).

As the results show, corroboration leads to a dramatic decrease in the number

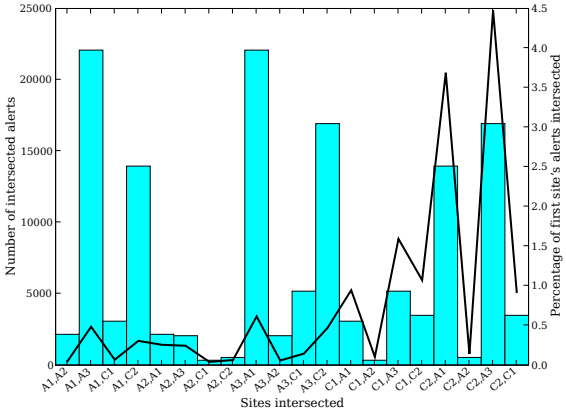

Figure 6.2: Alert rates, 2-way corroboration.

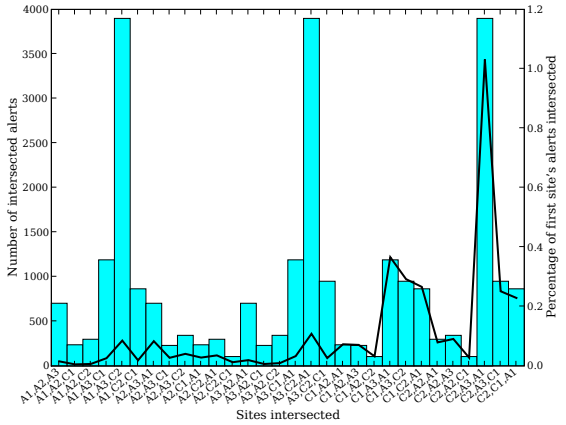

Figure 6.3: Alert rates, 3-way corroboration.

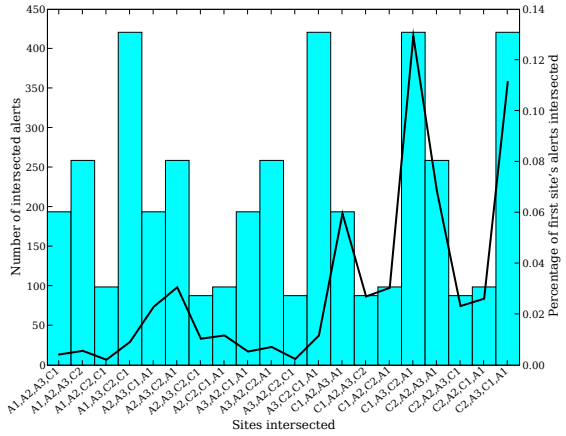

Figure 6.4: Alert rates, 4-way corroboration.

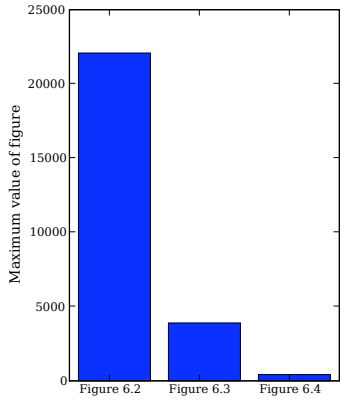

Figure 6.5: Alert rates, relative scale.

of alerts selected. Simple two-way corroboration leads to a 95%+ reduction in the number of unique IP-port tuples; four-way corroboration leads to a fraction of one percent of the original tuples. As figure 6.5 shows, each additional site yields approximately another order-of-magnitude reduction. This does not mean to imply that *all* of the other alerts should be ignored, but rather that fewer cross-site scanners exist and that they can be easily gleaned from the raw data via corroboration. Moreover, it's very unlikely that legitimate traffic is common to these unrelated sites—especially across longitudes (e.g., academia vs. commercial). Further analysis and insight on cross-site scanner behavior is discussed in a later subsection, but this simple analysis validates our approach to alert reduction.

Hashing Computation

The prime computation overhead for hash-based privacy transforms is the computation of the hash itself. For ideal privacy gain, a perfect/cryptographic hash function is preferred; such functions, like SHA-1 or H_3, are usually linear in the length of the datum to be hashed. For fairly constant-length data, such as IP addresses, this is essentially equivalent to $O(1)$. Nevertheless, practical results are useful to get a better idea of the actual constant-time overhead.

Figure 6.6 shows a performance comparison between the use of no hash functions (i.e., copying a given datum in memory) versus the use of three different hashes with varying numbers of IP/port alerts: the cryptographic hash function SHA-1 and two variants on the hash function H_3. The IP/port alert tuples are IPv4-based, i.e., 6 bytes, and were randomly sampled from the aforementioned data.

As the results show, computation overhead of the functions in question increase linearly as the number of alerts increase. The SHA-1 implementation used here is the (presumably optimized) Sun reference implementation built into Java. The two variants on H_3 reflect a simple optimization; the "cached" H_3 implementation buffers the hash values for all possible individual *bytes* in the input stream, instead of just storing the hash values per bit. Bit manipulation in many 3rd-generation languages, including Java, is cumbersome, and this removes a significant computation overhead incurred in repeatedly shifting existing bits as each additional bit is processed. The cost of the cached version is an increased use of memory to store the cache table; $4nm$ additional bytes of memory are used, where n corresponds to the number of bytes of each possible input while m corresponds to the number of bytes of hash output desired. It is important to note the hashes themselves are *not* affected.

Overall, the computation overhead is very small; in the case of one million alerts, hashing imposes approximately a 3-to-4 second overhead total over the underlying cost of copying events in memory. This translates to approximately a 3-4

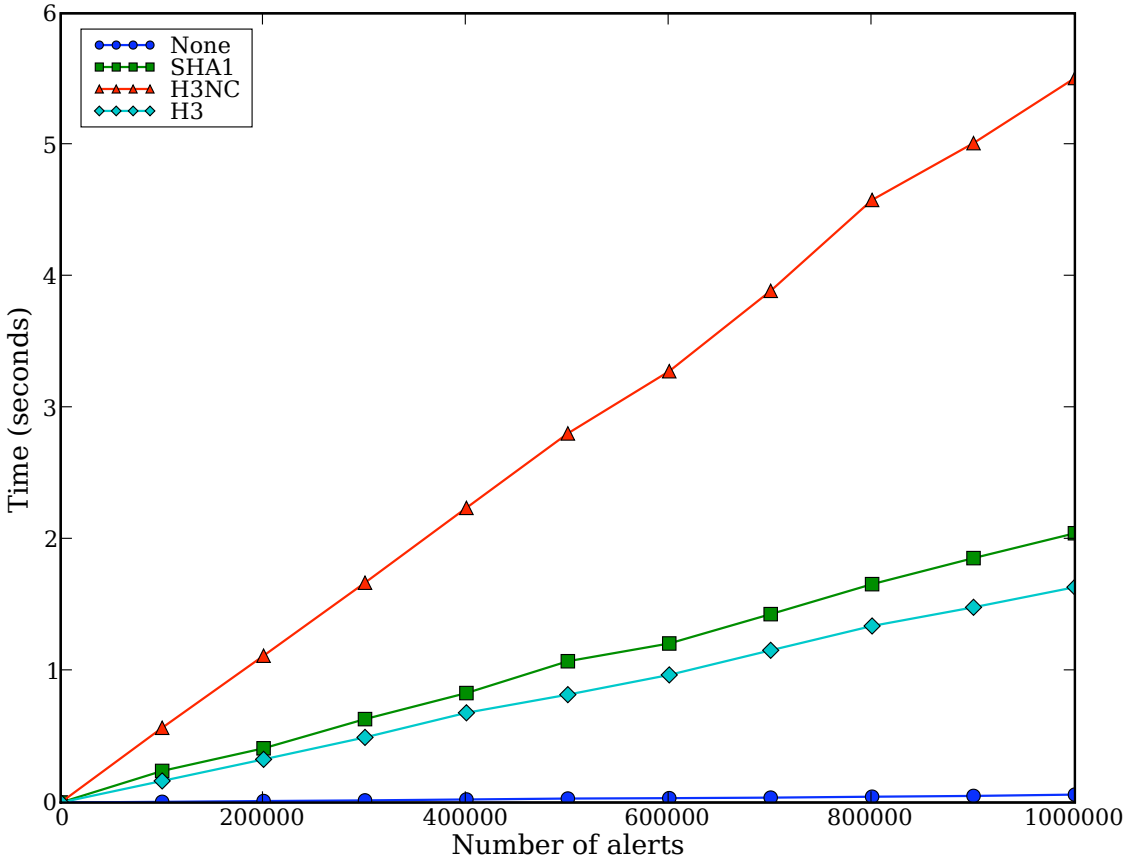

Figure 6.6: Performance Comparison of Hash Functions for IP/Port Values.

microsecond overhead per datum, as shown in figure 6.7. While this is not strictly negligible, hashing does not pose a problem for privacy transforms. If necessary, further optimizations can be done by implementing these functions in a native language or caching more commonly-hashed values.

Bloom Filter Computation

Bloom filters, being hash-based data structures, have correspondingly similar computation overheads to standard hashes, and so most of the analysis in the previous subsection applies here. Once again, BFs are $O(1)$ per datum inserted. However, the constant-time overhead can vary, due to two main factors:

1. A Bloom filter can be significantly smaller than an equivalent collection of

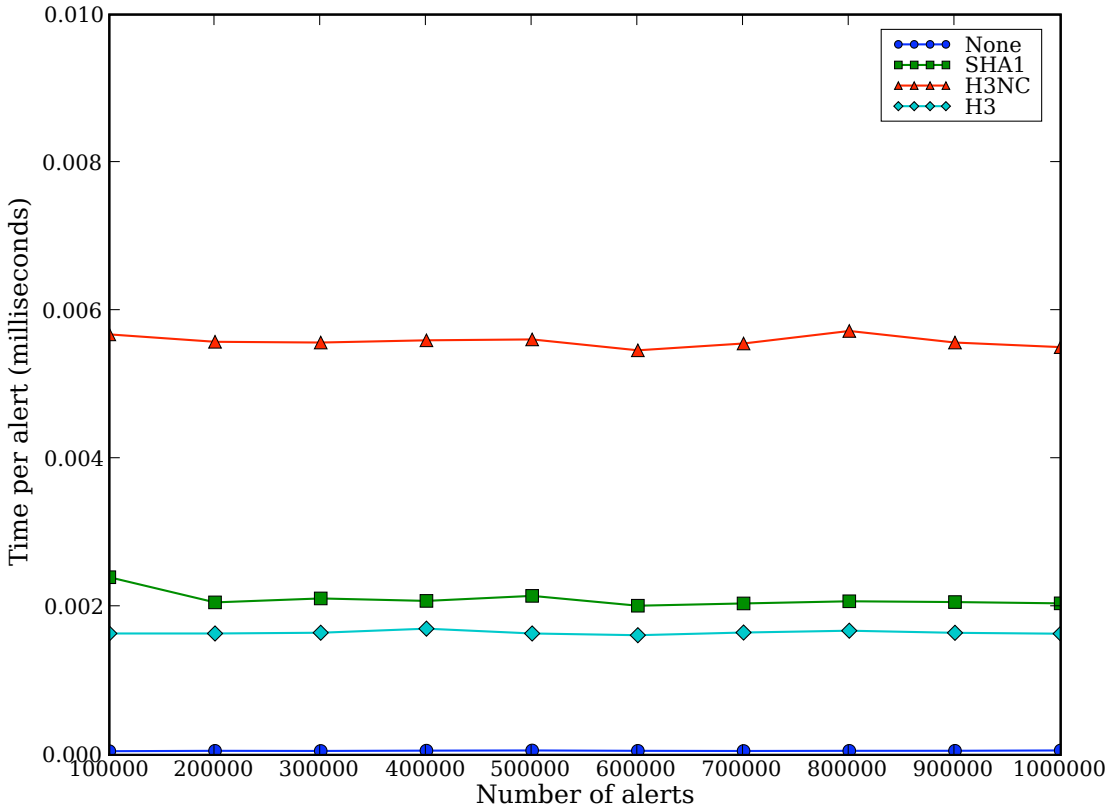

Figure 6.7: Performance Comparison of Hash Functions for IP/Port Values per Alert.

hashes, and in general is a contiguous block of memory, so memory access overhead is reduced. (A more precise characterization of Bloom filter memory overheads can be found in the next section.)

2. Due to its size and nature, a Bloom filter minimizes hash collisions (false positives) using a significantly different mechanism than a typical perfect hash. Typical perfect hashes, like SHA-1 and H_3, ensure a minimum of collisions by producing long, unique hash strings. Since a Bloom filter throws away the hash output, there is no incentive to produce such a long hash value. Instead, the size of the hash value produced for a BF is on-the-order of the number of bits used to represent a single entry, which for a variable-output hash function can take significantly less work.

To counter the corresponding increase in collision rate, *multiple* hash functions are used—typically, three or more are used. The optimal number of hash functions is a function of the data and the corroboration scenario, and this is discussed in greater detail in upcoming sections.

In order to better quantify the practical differences in overhead, I replicate the experiment highlighted in figure 6.7 here, but with two additional Bloom filter metrics. In figure 6.8, "BF-20" represents a 2^{20}-entry Bloom filter ($\approx$ 1 million entries), while "BF-24" represents a 2^{24}-entry Bloom filter ($\approx$ 16 million entries). Both use three H_3 hash functions.

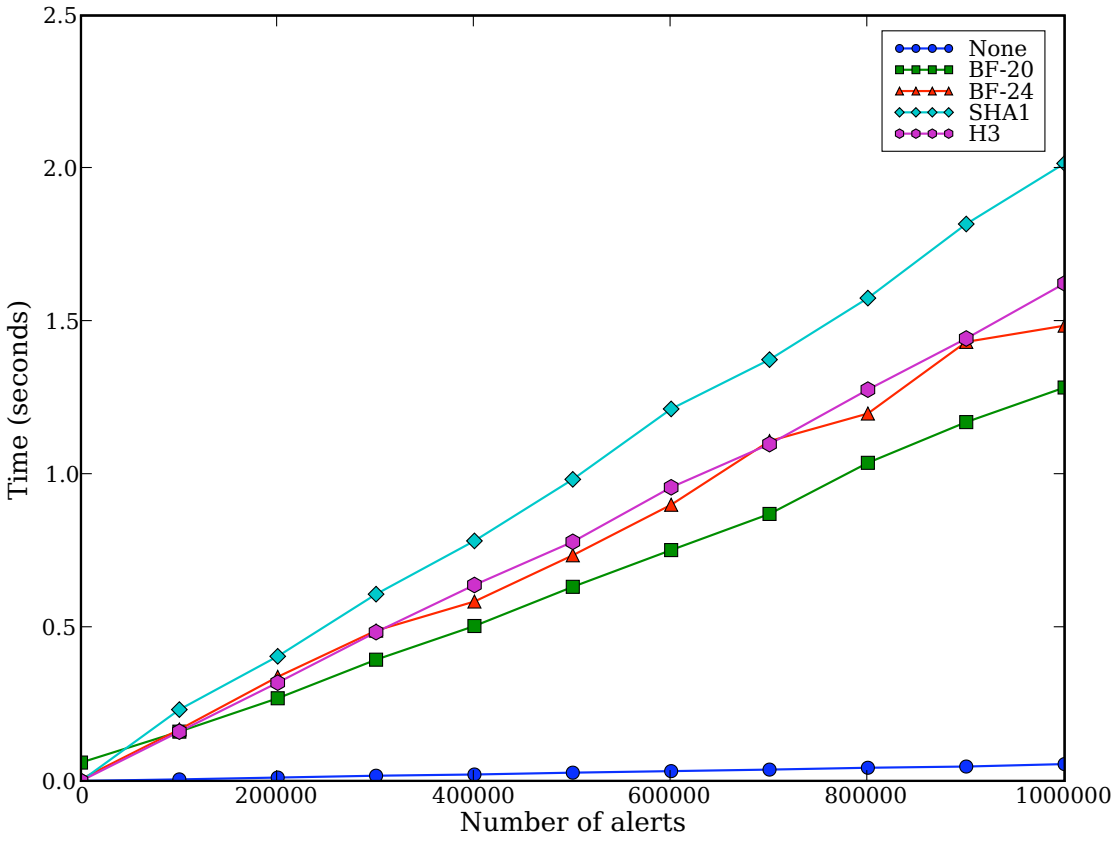

Figure 6.8: Performance Comparison of Bloom Filters and Hash Functions for IP/Port Values.

Figure 6.8 shows that Bloom filters perform well with respect to collections of hashes. Unsurprisingly, a 2^{20}-entry BF has a slightly lower computational overhead,

due to both a smaller memory footprint and less required hash output. However, the 2^{20}-entry BF would be saturated with one million alerts and would not produce satisfactory results; as we will see, 2^{24} is ideal for this purpose, and it still performs on par with a single H_3 hash function.

6.4.4 Space and Transmission Requirements

The space overhead of the two privacy-enabled data structures described above differs significantly, given the small size of an IP/port tuple.

Hashes are comparatively expensive: SHA-1, for instance, outputs 160 bits per entry, which represents roughly a 25-fold increase in memory requirements. At the same time, 28 bytes per alert (including a `long` timestamp) is not that large; given .33 alerts/second (the highest aforementioned alert rate), this represents approximately 9.25 bytes/sec per node. This can scale well; 1,000 nodes would use an aggregate bandwidth of less than 10KBps. If more space efficiency is needed, fewer bits of hash can be kept, albeit inducing a cost on corroboration accuracy, as discussed in the next section.

Of course, typical sensor alert rates play a significant difference, and vary based on the sensor used. For additional potential space savings, a Bloom filter can be used. First, a Bloom filter can theoretically represent the presence of an individual entry using a few *bits*, i.e., it acts as a compression mechanism upon the original alert. The one challenge with the use of Bloom filters for effective compression are sparse Bloom filters, which is common when alerts are being exchanged frequently. This can be ameliorated somewhat by using a typical compression algorithm on the Bloom filter.

For example, consider a Bloom filter with a 2^{20} index size, i.e., ≈ 1 million bits; it occupies 128KB of space, uncompressed. Given 5 hash functions per entry[6],

[6] As we show in the next section, ≥ 5 hash functions typically provide low false positive rates

approximately 100,000 entries can be stored before the Bloom filter is saturated, i.e., $\geq 50\%$ of the Bloom filter is filled. The uncompressed storage space is therefore 1.31 bytes per entry. A small experiment was created using varying numbers of entries and hash functions to demonstrate the effective space requirements imposed by a Bloom filter; the results are shown in table 6.3.

# entries inserted	Uncompressed (5 or 10 hash fns)		Compressed (5 hash fns)		Compressed (10 hash fns)	
	Size	Per Alert	Size	Per Alert	Size	Per Alert
1	131072	131072.00	182	182.00	189	189.00
2	131072	65536.00	184	92.00	198	99.00
10	131072	13107.20	284	28.40	312	31.20
100	131072	1310.72	1177	11.77	1467	14.67
1000	131072	131.07	5098	5.10	8507	8.51
10000	131072	13.11	26109	2.61	41048	4.10
100000	131072	1.31	109714	1.10	119955	1.20

Table 6.3: Bloom filter sizes; all sizes are in bytes.

As the results show, the Bloom filter produces better size results than hash sets, except for very sparse Bloom filters. When used to encode more alerts, Bloom filters can be far more efficient than transmitting the original alert. However, even sparse compressed Bloom filters are useable; given the same alert rate and node configuration as before, the worst-case scenario of transmitting one-entry, 5-hash BFs would consume 60KBps of bandwidth. It's also worth noting that the number of hash functions used does not significantly affect size, and as such choosing the right number of hash functions is more an issue in optimizing false positive rates and computation time.

6.4.5 Corroboration Accuracy

The most important criterion, of course, is that of accuracy; privacy-preserving transforms are only useful if alerts can be reliably corroborated. Both hashes

when corroborating.

and Bloom filters offer 100% true positives; corroboration accuracy, therefore, is predicated on effectively minimizing false positives.

The two techniques were tested with a subset of the aforementioned dataset. To ensure that a sufficient number of alerts were gathered from each site *and* enough data was retained for practical 2-, 3- and 4-way corroboration, all of 3-way alerts were kept and mixed in with other alerts selected at random. A maximum of 600,000 alerts were kept per site (so as to readily allow a 2^{24}-bit Bloom filter to store all of the alerts without saturation noise).

Hash Functions

As a baseline, false positive rates were tested using the various hash functions discussed earlier. Unsurprisingly, FP rates are extremely low. Figures 6.9–6.11 show false positive rates for H_3 hashes, ranging from 8-bit to 48-bit hash values when performing 2-, 3- and 4-way corroboration. No results are shown beyond 48 bits as zero false positives were recorded for both 40- and 48-bit hashes, strongly implying that additional bits were unnecessary for this dataset.

As can be seen in all three figures, the false positive rates rapidly converge to zero; as mentioned earlier, 40 bits is sufficient for this dataset. However, the FP rates for smaller hashes varies; in particular, as the number of corroborating sites increase, the percentage of alerts that are false positive *decreases*. This is not due to an increase in the number of true positives—in fact, they decrease by orders-of-magnitude, as discussed earlier. Instead, the number of false positives decreases even more dramatically. While smaller hash values cause an increase in the number of collisions at any given site, corroborating between sites actually eliminates coincidental collisions, since the likelihood the *same* collision was encountered at multiple sites decreases.

This key insight gives us a number of attractive properties: first, it enables

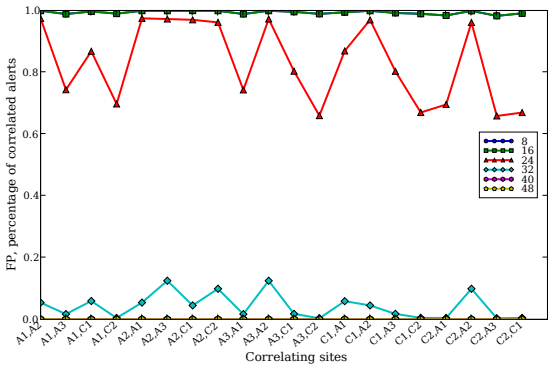

Figure 6.9: Hash function false positive rate, 2-way corroboration.

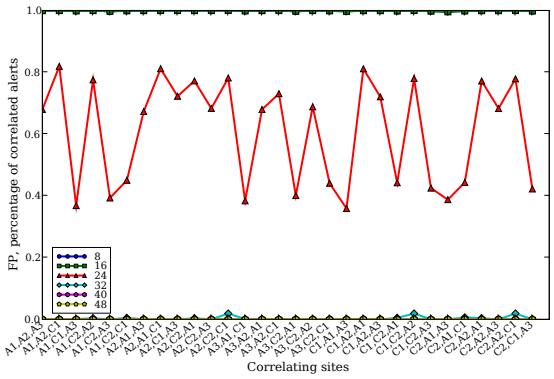

Figure 6.10: Hash function false positive rate, 3-way corroboration.

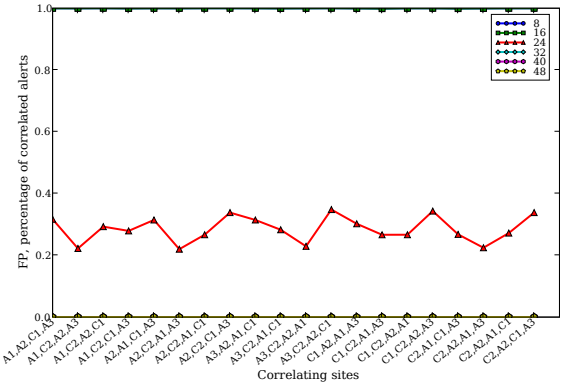

Figure 6.11: Hash function false positive rate, 4-way corroboration.

us to use smaller hash values than otherwise necessary, further ameliorating the space requirements discussed previously. Second, it provides insight as to how to minimize the false positive rates of Bloom filters, which are ordinarily far more prone to collisions; results when doing N-way corroboration with Bloom filters are discussed in the next subsection.

Bloom Filters

A similar analysis is done with Bloom filters, but with significantly different results. Figures 6.12–6.14 show false positive rates for 2-, 3- and 4-way corroboration using 2^{24}-entry Bloom filters (e.g., large enough to avoid false positives due to saturation) with H_3 hash functions. The same alert set and graphing technique was used as in

the previous experiment.

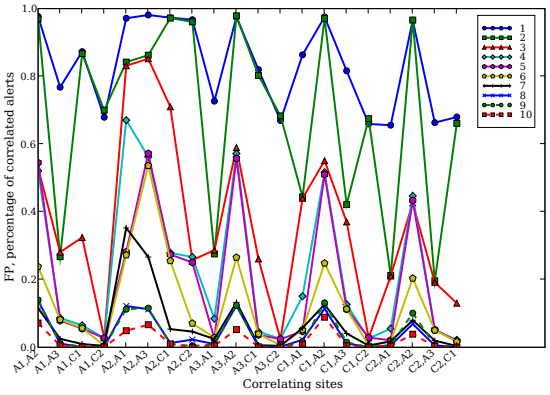

Figure 6.12: Bloom filter false positive rate, 2-way corroboration.

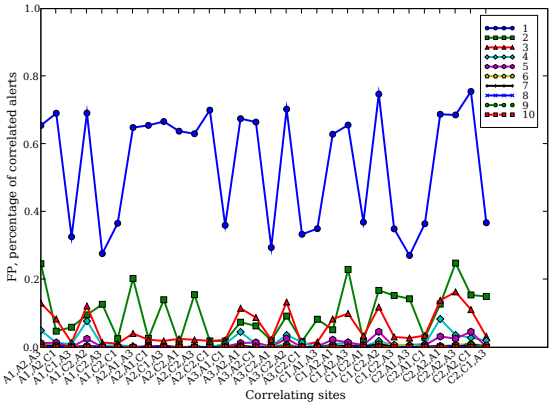

Figure 6.13: Bloom filter false positive rate, 3-way corroboration.

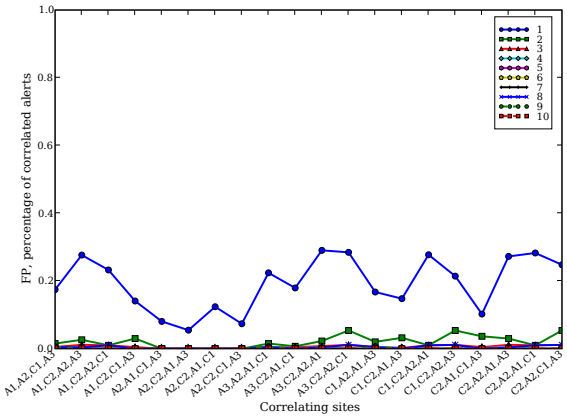

Figure 6.14: Bloom filter false positive rate, 4-way corroboration.

As the results show, there is a significant difference in false positive convergence when compared to discrete hashes. When considering 2-way corroboration, a zero false positive rate never occurs for all pairs; even with the use of *10* hash functions, two academic sites still report just under a 10% false positive rate.[7]

How can Bloom filters best be utilized, then? Arbitrarily increasing the number of hash functions past 10 is not a solution; at a certain point, the likelihood of saturation and collisions increases significantly. Additionally, there is a multiplicative overhead

[7] If *both* the number of hash functions and Bloom filter size (to avoid saturation) are increased, this FP rate will asymptotically approach zero, but this becomes increasingly inefficient.

for the extra computation required. Instead, as with hashes, increasing the number of collaborators helps significantly, making error-free corroboration possible. Given a 3-way corroboration, 6 hash functions are sufficient to obtain a near-zero false positive rate (near 0.3%, to be precise; 7 yielded a maximum false-positive rate of exactly 0%). The requirement is further reduced when four sites are corroborated; as shown, just 3 hash functions are sufficient. As is shown in the next section, reducing the number of hash functions not only reduces space requirements, but enhances our privacy gain significantly.

6.4.6 Temporal Corroboration

As has been demonstrated, effective corroboration is possible given appropriate sets of Bloom filters; however, in a distributed environment, many such Bloom filters may be exchanged before desirable data can be extracted. More significantly, it becomes difficult to *search* through the history of received Bloom filters when a new local alert is generated (i.e., to effectively corroborate when others may transmit an alert of interest before it is locally generated); a linear-order search in the number of Bloom filters is impractical. Two data structures were proposed to solve this problem in chapter 5, and in this section I evaluate their effectiveness in supporting temporal corroboration.

Corroboration Accuracy

In terms of corroboration ability, these structures are equal to or better than that of an ordinary Bloom filter. Both the *MRU Bloom Filter* and *Timestamp Bloom Filter* (heretoforth referred to as MRU BF and TSBF, respectively) support "flattening", whereby the timestamps are reduced down to single bits, where "1" indicates the presence of a timestamp and "0" indicates the lack of one. This flattened data

structure is equivalent to an ordinary Bloom filter, and so the above analysis applies.[8]

Performance and Memory Overhead

However, the most important analysis here is that of performance and memory overhead; these must be tractable in order to encourage their use on each site. Ideally, each site maintains one such Bloom filter, and progressively *merges* in received Bloom filters. Additionally, a site may choose to *expire* old entries to prevent Bloom filter saturation and to allow for the incremental evolution of alerts being exchanged. In order to determine both the experimental computation and memory overhead of these operations and of the resulting BFs, I devised a simulated experiment using the dataset from section 6.4.5 to resemble a real-world exchange scenario.

More precisely, 5 virtual sites exchange Bloom filters every 30 seconds; each Bloom filter contains 1,000 IP-based alerts, i.e., 1,000 (source IP, destination port) tuples.[9] The size of these Bloom filters varied from 16-bit (2^{16}) to 24-bit (2^{24}) to test different saturation levels and memory overheads; in each, three hash functions are used. This scenario was run for 1,800 simulated "seconds", with an expiry threshold of 900 seconds, i.e., after 900 seconds, all bits older than 900 seconds would be removed after the receipt of *each* remote Bloom filter. Figures 6.15–6.21 show the results along four different metrics with three different BF sizes: merge computation, expiry computation, uncompressed space use, and compressed space use.

As alluded to earlier, figures 6.15–6.17 represent the computation overhead for merge. In each, six lines are shown. Two data plots are shown for the MRU BF and TSBF. Linear regressions are also shown for each to show computation trends.

[8] In the case of a TSBF, it is *also* possible to reduce the false positive rate slightly—by not flattening the BF, and upon a check, ensuring that the n hash functions used yield bits that all have the presence of the same timestamp. This strategy does not work with the MRU BF, as individual timestamps may be overwritten when bits are shared. I do not explore this further here, as the flattening approach works well and existing BFs provide a very good lower bound for false positive rates.

[9] The number of alerts is purposely high to challenge the temporal data structures being evaluated. In reality, these Bloom filters could be far sparser.

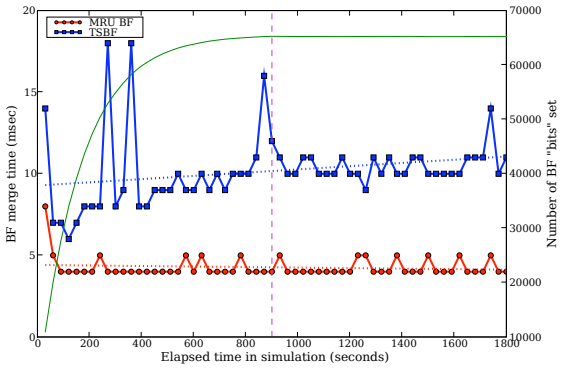

Figure 6.15: Merge performance, 16-bit BFs.

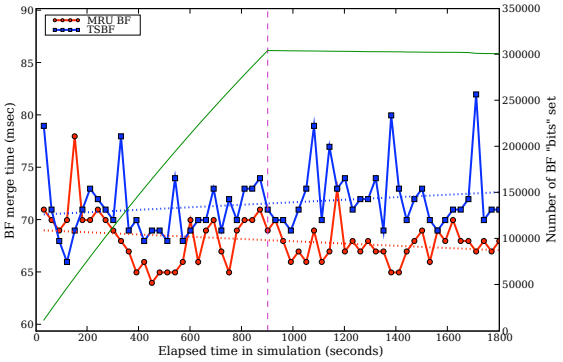

Figure 6.16: Merge performance, 20-bit BFs.

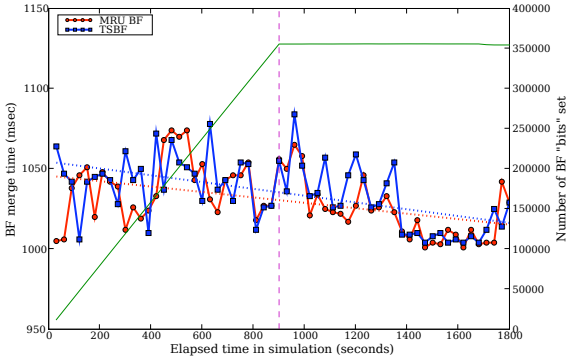

Figure 6.17: Merge performance, 24-bit BFs.

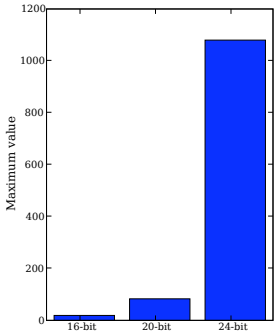

Figure 6.18: Merge performance scale.

A separate, unmarked, solid line represents the equivalent number of bits set in a flattened Bloom filter, and ranges from complete saturation to unsaturated but relatively constant thanks to expiry. Finally, a vertical dashed line represents the 900-second mark, where the expiry policy started taking effect (although, for sanity's sake, its computation overhead is shown on a separate set of graphs).

In general, both 16-bit and 20-bit temporal BFs are very fast, supporting a merge time well under 100ms. The 24-bit BF, which is significantly larger, has a greater overhead; however, the performance is still reasonable given the number of bits, hovering around 1000ms–1100ms per incoming BF even as the structures saturate. Depending on the number of Bloom filters being distributed, various strategies, including aggregation, can be used to ameliorate this computation overhead.

A close examination of the results shows surprisingly close results between MRU and TSBFs; moreover, the larger difference accentuated in figure 6.15 must be

viewed in context of the small scale of the y-axis. That said, figure 6.15 shows a clear increasing computation trend for TSBFs, while MRU BFs, representing a fixed vector, remain consistent. Note that the equivalent Bloom filter is essentially saturated around the 800 second mark in the simulation, so increasing TSBF computation can be viewed in the context of its increasing tree depth; this hypothesis is validated by figure 6.19. The 20-bit and 24-bit BFs, on the other hand, are never saturated, and so the depth increases much more slowly. This is most noticeable in the 24-bit case, where the average depth (figure 6.21) is asymptomatically close to zero, and where the corresponding computation overhead in figure 6.17 actually *decreases*. This decreasing trend is particularly noticeable after the 900-second mark where expiry takes effect, and suggests that enough of the TSBF is now cached in a CPU cache to allow performance metrics on par with MRU BFs. This surprising result refutes the intuition that the TSBF, with its additional data structure overhead, must always be slower than a MRU BF, although it will clearly suffer a performance penalty if the tree depth grows significantly.

Figures 6.23–6.25, representing expiry computation overhead, are presented identically to the previous figures, but represent the computation overhead required for expiry as opposed to merge. Correspondingly, both lines sit on the y-axis until the expiry policy takes effect. It is here that the MRU BF and TSBF differ most significantly; MRU BFs simply need to change values, while TSBFs are more complex data structures, requiring tree node destruction. Still, the difference is not very significant, and both data structures support expiry in under 200ms even in the 24-bit case.

Figures 6.27–6.29 show uncompressed memory usage, while 6.31–6.33 show compressed memory usage, both for MRU BFs and TSBFs ranging from 16-bit to 24-bit. As is to be expected, both structures use up a significant amount of memory. However, the comparative memory use, unintuitively, differs wildly based on the

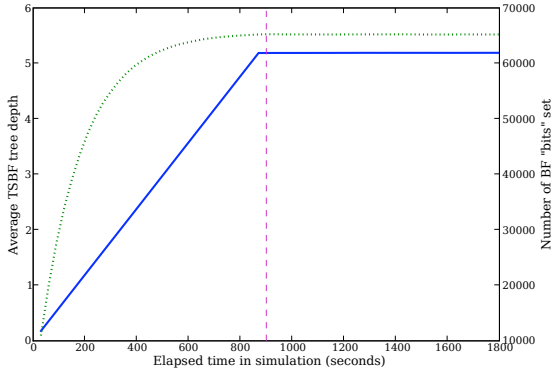

Figure 6.19: TSBF average depth, 16-bit.

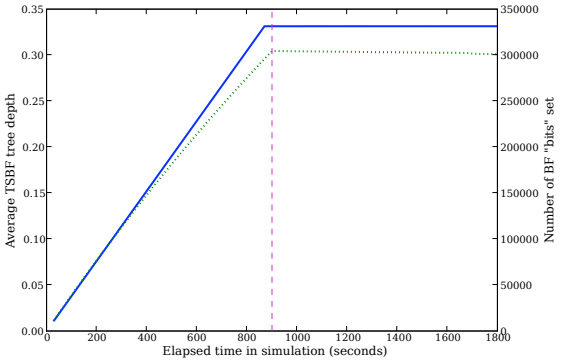

Figure 6.20: TSBF average depth, 20-bit.

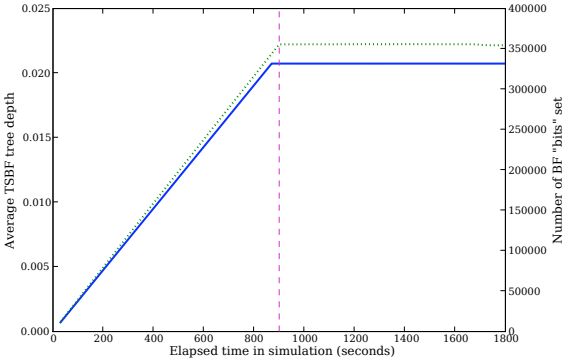

Figure 6.21: TSBF average depth, 24-bit.

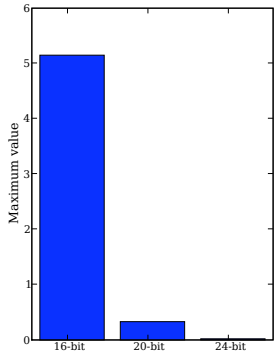

Figure 6.22: TSBF average depth scale.

size of the BFs being used. As is expected, the TSBF takes significantly more memory than the MRU BF in the 16-bit case—on the order of 10 times more, and growing until expiry kicks in.

However, in the 20-bit and 24-bit case, the TSBF is actually smaller, and *significantly* so in the 24-bit case. At first glance, this makes no sense. However, it is likely due to the uniform use of *long*-valued, or 64-bit, fields in the case of the MRU BF; the TSBF, on the other hand, is initially composed entirely of null 32-bit memory references, and only allocates memory to store timestamps as necessary. If the TSBF is relatively sparse, as is the case when 24-bits are used, its memory usage never becomes that significant. It would also possible to reduce MRU BF memory usage in a similar fashion if desired, at the cost of greater computation/expiry overhead and memory use when saturated. Choice of an appropriate strategy, therefore, will depend on the saturation and expiry patterns as events are collected into the BFs.

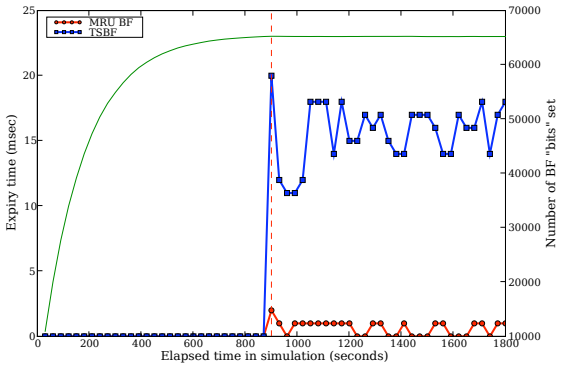

Figure 6.23: Expiry performance, 16-bit BFs.

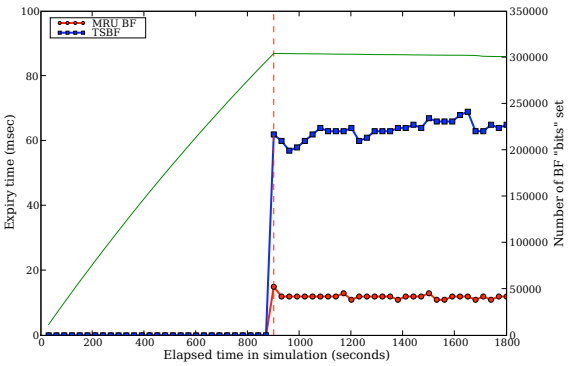

Figure 6.24: Expiry performance, 20-bit BFs.

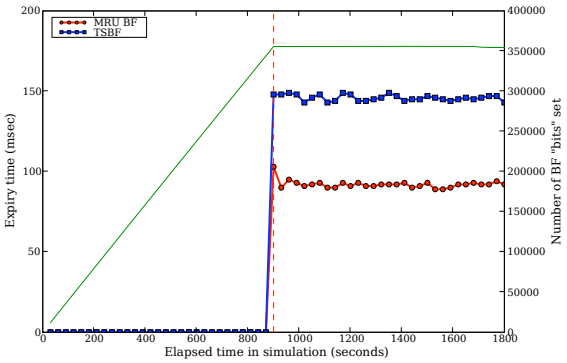

Figure 6.25: Expiry performance, 24-bit BFs.

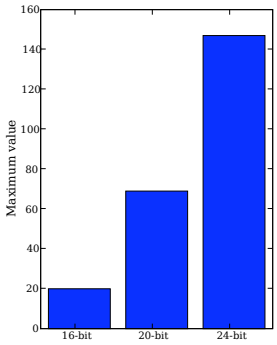

Figure 6.26: Expiry performance scale.

This intuition is demonstrated, to some extent, by the compressed size results. Here, the disparity between TSBFs and MRU BFs are significantly diminished in the 20-bit and 24-bit case, as the sparseness of relevant data in the MRU BF is efficiently compressed away. Nevertheless, the TSBF remains smaller in the 24-bit case. This suggests that the TSBF is not just a theoretical data structure that grows too fast for effective use; instead, it is capable of supporting in-memory corroboration with reasonable efficiency.

6.4.7 Privacy Gain

The key privacy consideration when exchanging IP/port pairs is the possibility that a curious remote peer may be able to brute-force the contents of the hash set or Bloom filter and regain the original data, thereby violating Worminator's utility. This is largely possible with Worminator primarily because the space to be brute-forced

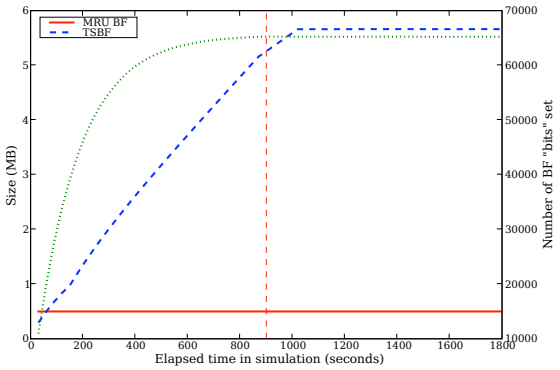

Figure 6.27: Space overhead, 16-bit BFs.

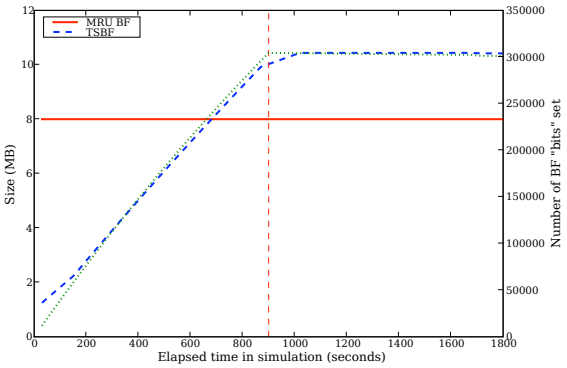

Figure 6.28: Space overhead, 20-bit BFs.

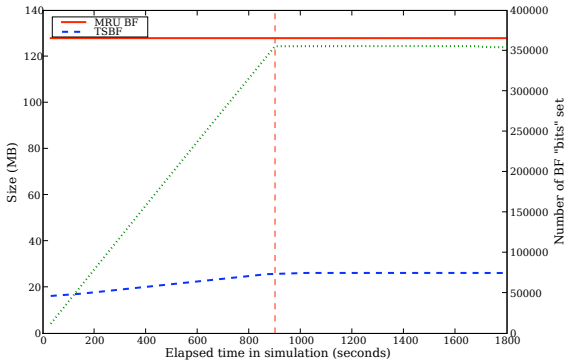

Figure 6.29: Space overhead, 24-bit BFs.

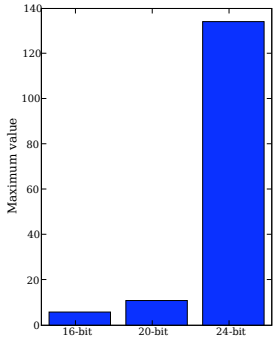

Figure 6.30: Space overhead scale.

is relatively small, i.e., $O(256^6)$. As is shown in section 6.5, brute-forcing rapidly becomes intractable as the discourse size increases.

Ideally, therefore, the goal should be to use a data structure that is not practically brute-forceable, but with which one can effectively corroborate. Both hash functions and Bloom filters have been shown to corroborate effectively as per the last subsection. As for brute-forcing, figures 6.35 and 6.36 show the result when the hash sets and Bloom filters discussed in the previous section are partially brute-forced for varying numbers of bits and hash functions, respectively. The subset that was brute-forced was the space of all IP addresses associated with port 80, i.e., $a_i = *, 80$ (note that the timestamps are not of consequence here, as they are not hashed). This limitation does not affect the validity of the approach; the computation required, while less, still captures the difficulty (or ease) in brute-forcing BFs and hashes. Additionally, limiting the scope may resemble a realistic curious participant; by querying popular

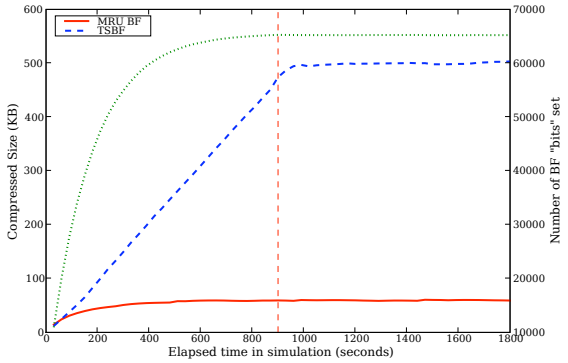

Figure 6.31: Space overhead, compressed 16-bit BFs.

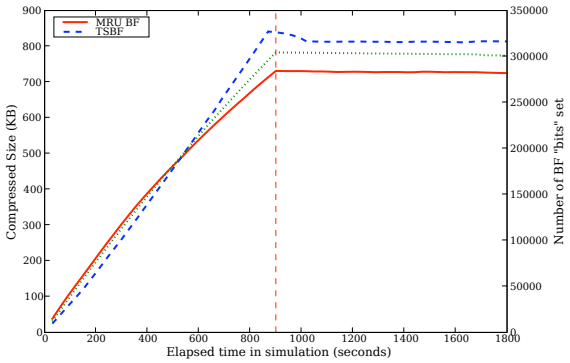

Figure 6.32: Space overhead, compressed 20-bit BFs.

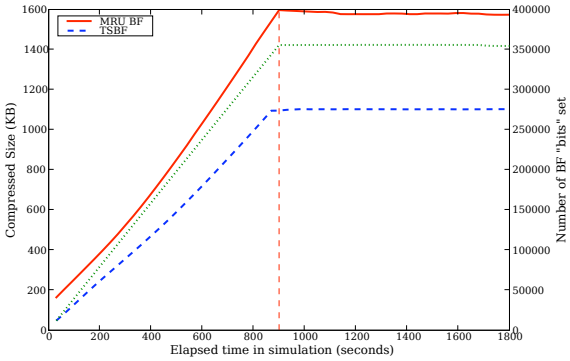

Figure 6.33: Space overhead, compressed 24-bit BFs.

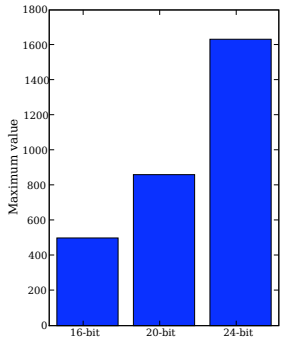

Figure 6.34: Space overhead, compressed scale.

services, they can try and reduce the search space, while focusing on the most relevant results.

The two figures show drastically different results. The false positive rate of hash functions initially start at 100%, but converge very quickly; in fact, when 48 or more bits are used, the FP rate is essentially zero. Bloom filters, on the other hand, converge far more slowly. In fact, with 10 hash functions, only site C1 witnesses a drop below a 50% false positive rate. This remarkable result is likely due to the fact that any bit i may be used by more than one possible datum; in fact, the simple observation that $256^6 >> 2^{24}$ (for the 24-bit Bloom filter) suggests that a large number of bits are in fact shared. This, then, gives us our desirable property: any individual Bloom filter by itself cannot be brute-forced meaningfully, *but* may be effectively corroborated. An attacker or curious participant would then find the most useful results by corroborating alerts of their own with one or more participants' Bloom

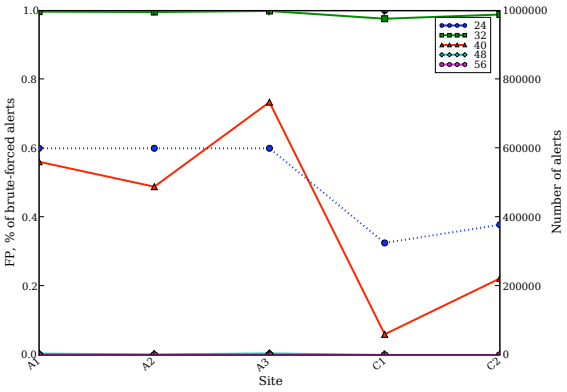

Figure 6.35: Hash set brute-force results.

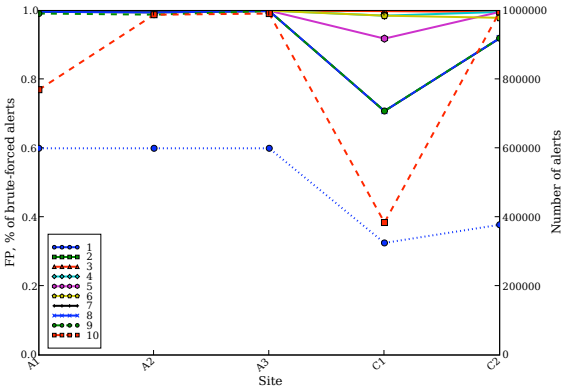

Figure 6.36: Bloom filter brute-force results.

filters; this, however, is exactly what Worminator is designed to do, and the attacker has not gained any *extra* information.

One may observe that exchanging hash sets or Bloom filters with up to 600,000 alerts is not particularly meaningful, as nodes usually exchange just a few alerts during any one communication, but do so frequently over a period of time. Ensuring the privacy of the *sparse* hash set or Bloom filter, therefore, is of equal concern; the correlation accuracy and brute-forceability of this scenario is discussed in the next section.

In the meantime, we can conclude that Bloom filters accomplish the desired privacy-preserving goals, while hashes are significantly less useful. This does not discount the utility of hashes in the general case—in particular, hashes can be extremely useful in providing accurate results when the size of discourse is significantly larger—but they do not provide adequate privacy in-and-of-themselves for exchanging IP watchlists. [86] proposes an approach by which hashes of IP information may be exchanged without violating privacy; I show shortly that their approach does *not* generally work, and propose another technique by which they can be made to work.

Sparse Hashsets and Bloom Filters

In order to test a more realistic *continuous* corroboration scenario, the following experiment was devised: for each site, 20 of the 4-way-correlating alerts were chosen, privacy-transformed into hash sets and Bloom filters, and exchanged. These 20-alert exchanges were then corroborated against the much larger alert set as used in the previous two subsections. The results for corroboration accuracy are shown in figures 6.37–6.38 for hash functions, and 6.39–6.40 for Bloom filters; brute-force results are then shown in 6.41 and 6.42.

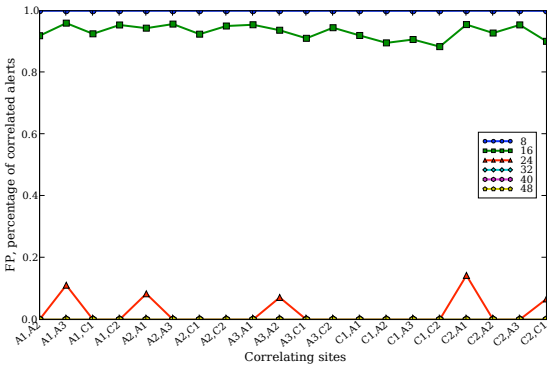

Figure 6.37: Sparse hash set false positive rate, 2-way corroboration.

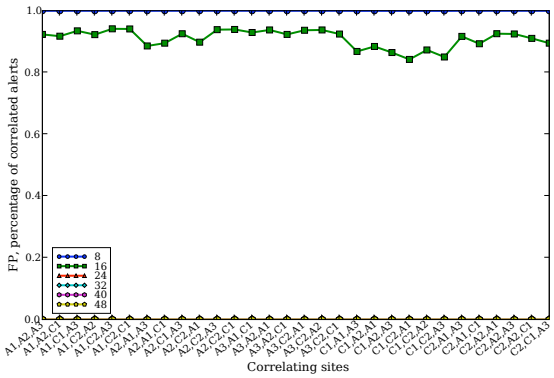

Figure 6.38: Sparse hash set false positive rate, 3-way corroboration.

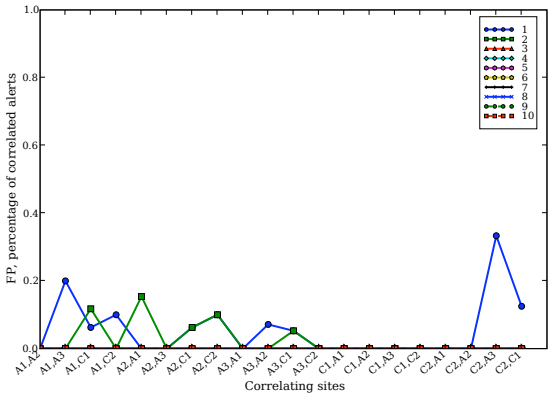

Figure 6.39: Sparse BF false positive rate, 2-way corroboration.

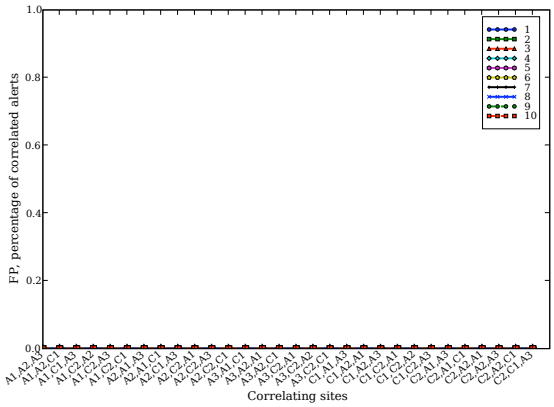

Figure 6.40: Sparse BF false positive rate, 3-way corroboration; all score at 0%.

Unsurprisingly, fewer false positives can be found when a smaller number of alerts are being corroborated. Unfortunately, however, fewer false positives are also encountered during a brute force analysis of the sparse hashset or Bloom filter. A

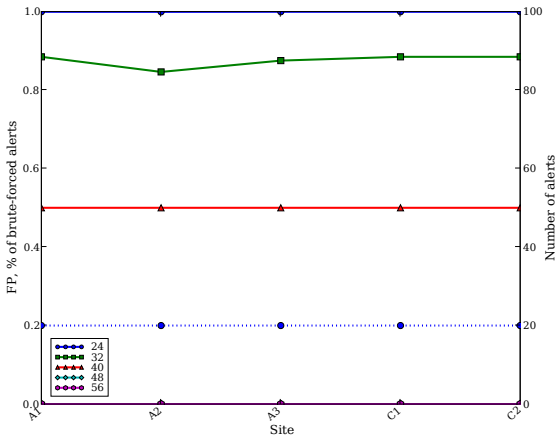

Figure 6.41: Sparse hash set brute-force results.

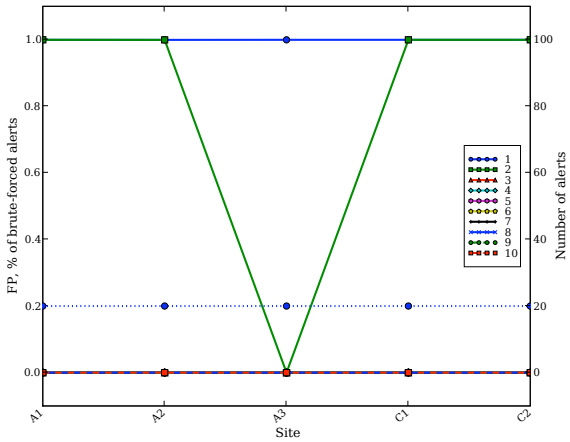

Figure 6.42: Sparse BF brute-force results. (The y-axis is extended slightly for visibility at 0% and 100%.)

48-bit hash or a Bloom filter using three or more hash functions produce zero false positives during a brute force, thereby eliminating the privacy gain discussed in the last section.

Since it is the sparseness of the data exchanged that is enabling the possibility of an effective brute force, a simple adaptation can be made to try and improve the privacy analysis: transmit *noise* along with the sparse data structure. Ideally, such noise would correlate out when comparing multiple sets/BFs, but would render a brute-forcing meaningless. Noisy Bloom filters are discussed in the next subsection, while noisy hash sets are discussed in the following section.

Noisy Sparse Bloom Filters

The analysis shown in figures 6.14 and 6.36 implies that a non-sparse Bloom filter naturally generates many false positives when brute-forced, but that these false positives drop out nicely when multiple sites corroborate data. In order to test this with the sparse Bloom filter, a Bloom filter is created with a percentage of bits randomly set. (An alternative technique for introducing noise would be to use a different hash function, random inputs, etc. However, one can observe that these other forms of noise would yield similar results, assuming the other noise technique

generates follows a near-uniform distribution.)

Two different noise levels were tested: 5% and 10%, e.g., 10% of the total *size* of the Bloom filter (as opposed to the number of occupied bits). The correlation results for a 5% noisy BF are shown in figures 6.43–6.45, while the results when using a 10% noisy BF are shown in figures 6.47–6.49; the resulting brute-force false positive results are shown in 6.46 and 6.50, respectively.

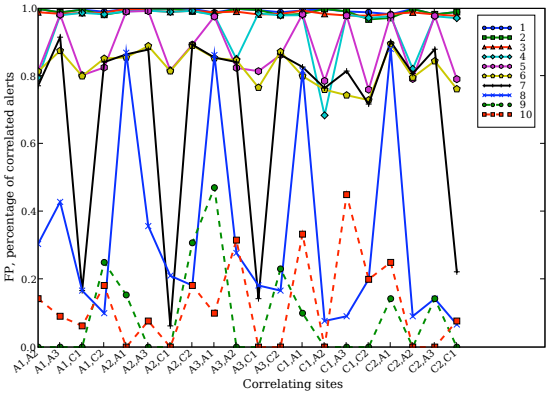

Figure 6.43: Sparse BF (5% noise) false positive rate, 2-way corroboration.

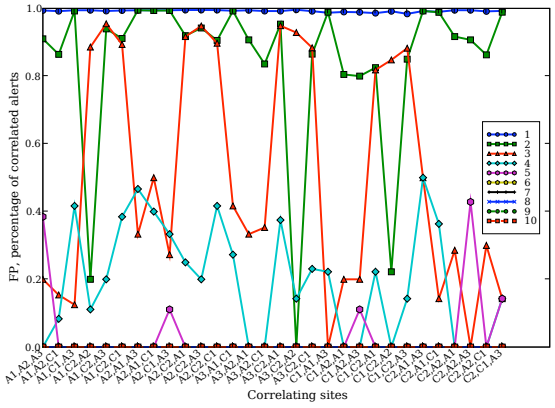

Figure 6.44: Sparse BF (5% noise) false positive rate, 3-way corroboration.

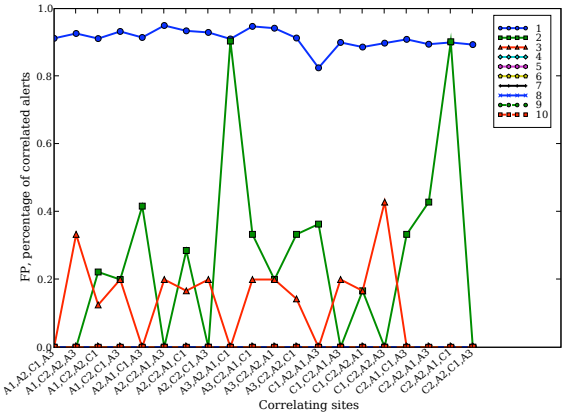

Figure 6.45: Sparse BF (5% noise) false positive rate, 4-way corroboration.

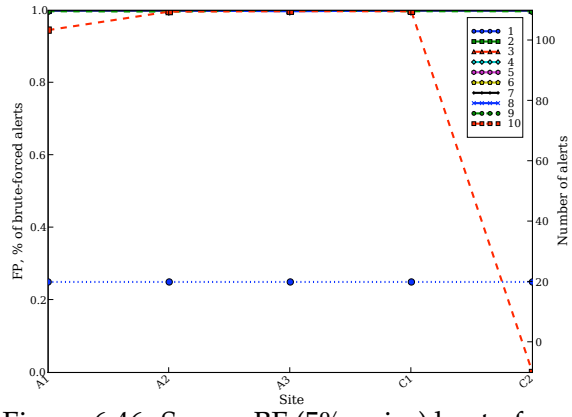

Figure 6.46: Sparse BF (5% noise) brute-force results.

The results are remarkable; effective 4-way corroboration can be accomplished with both noise values; 4 hash functions are sufficient at the 5% level, while 5 hash functions produce zero false positives at the 10% level. More importantly, all but 10 hash functions produce near-100% false positives when 5% noise is used,

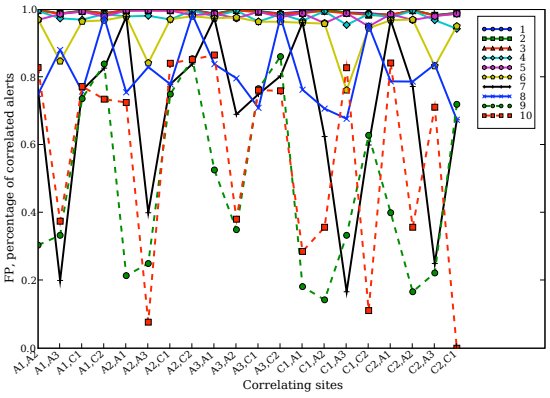

Figure 6.47: Sparse BF (10% noise) false positive rate, 2-way corroboration.

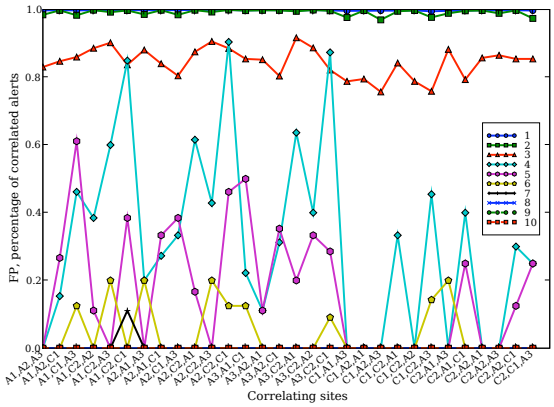

Figure 6.48: Sparse BF (10% noise) false positive rate, 3-way corroboration.

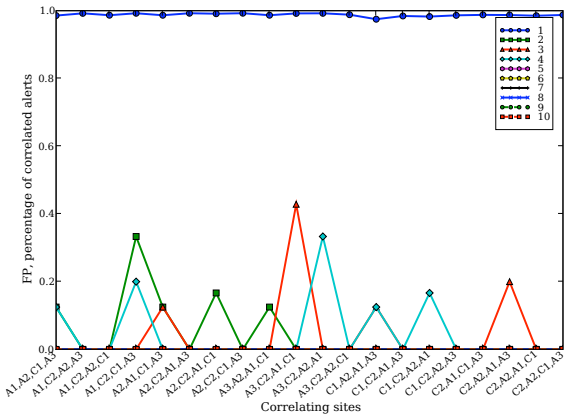

Figure 6.49: Sparse BF (10% noise) false positive rate, 4-way corroboration.

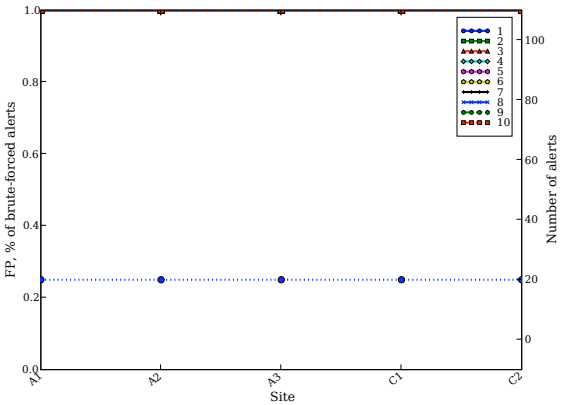

Figure 6.50: Sparse BF (10% noise) brute-force results. (No matter how many hash functions are used, all yield ~ 100% FPs.)

while not even 10 hash functions are able to effectively remove false positives when brute-forcing a 10% noise BF. While these are empirical and do not necessarily suggest the appropriate noise thresholds to use for effective corroboration in all situations, they demonstrate that a small amount of noise provides effective privacy, and does not significantly affect corroboration when enough sites are involved.

Noisy Hashsets

One of the attractive aspects of introducing noise in a Bloom filter is that uniformly distributing noise across the space of possible "hash values" is straightforward; bits are correspondingly set as desired. Unfortunately, this is not possible with hash

sets consisting of discrete entries. [86] suggested inserting *two* sets of IPs into the hash set to provide effective misdirection: *remote* IPs are inserted using a standard, known and shared hash, while *private* IPs are inserted using a HMAC-keyed hash, where the HMAC private key is not disclosed to partners. This technique was only described and not tested, but was hypothesized as making the brute-forced results meaningless, as a curious entity would not know for sure if the IP addressed brute-forced was legitimate or a coincidental side-effect of a keyed hash.

Here, a variation of the proposed algorithm was implemented and tested for its effectiveness. First, observe that HMAC-hashed entries are effectively pure noise to remote peers, and entries composed of random bits can be substituted for the HMAC entries with no loss of generality. This observation was employed to provide a similar "noise effect" as with Bloom filters. More precisely, during the insertion of a hashed IP value, an extra entry, composed of an entirely random bit sequence, is probabilistically added as well. This technique was tested at 50% and 100%, i.e., for approximately every two entries and one entry, respectively, an extra random-noise item was added to the bit hash. The resulting hash sets were tested for both corroboration and privacy effectiveness; the results are shown in figures 6.51–6.58.

Unsurprisingly, the results are only marginally better than those shown in the earlier sparse corroboration example. Despite the noise, 32 bits (the minimum needed to accomplish zero false positives when corroborating) still yields no false positives when brute-forcing. This clearly demonstrates that the aforementioned-proposed approach is not a feasible solution, at least for IP-based hashing where the domain of the input dataset is far smaller than the domain of the output dataset.

Instead, a better technique would be to randomly permute the *input* to the hash functions, i.e., the IP addresses. For example, some percentage of legitimate IP addresses could be mutated and reinserted. These would certainly affect the

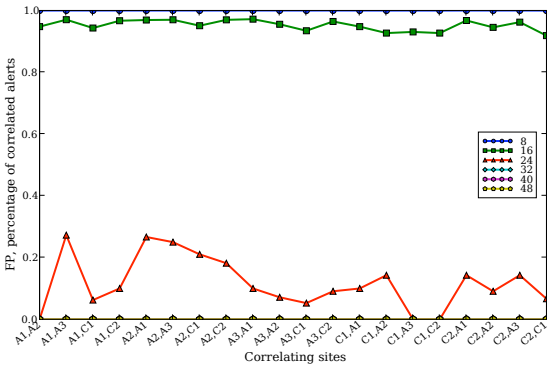

Figure 6.51: Sparse BF (50% noise) false positive rate, 2-way corroboration.

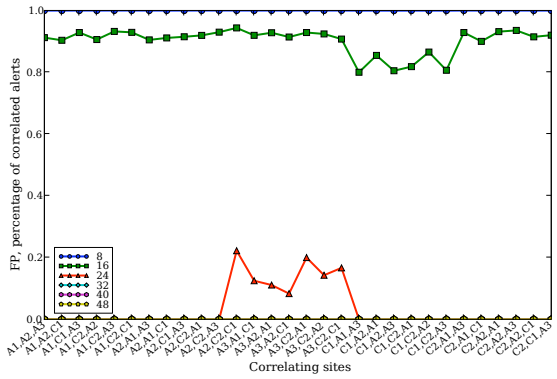

Figure 6.52: Sparse BF (50% noise) false positive rate, 3-way corroboration.

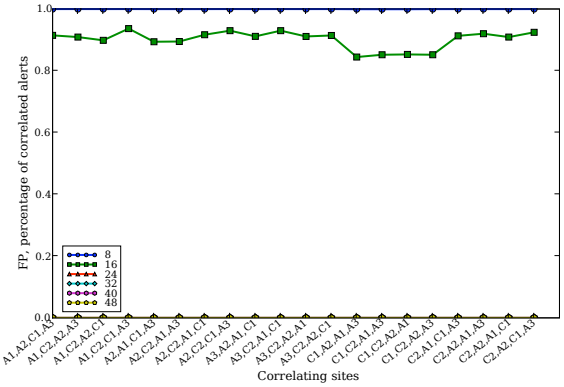

Figure 6.53: Sparse hash set (50% noise) false positive rate, 4-way corroboration.

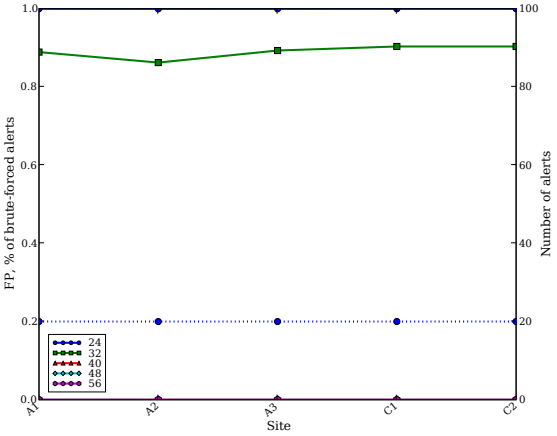

Figure 6.54: Sparse hash set (50% noise) brute force results.

results of a brute-force search; more importantly, what's the effect on corroboration accuracy? Figures 6.59–6.62 show the results for both corroboration and brute-forcing when this "new" technique is used with 100% noise, i.e., for every legitimate alert, a false one is also generated.

The brute-force results are as expected; the worst-case FP rate is round 50%, which corresponds to the number of false entries inserted. The more interesting results are those of corroboration; there is little-to-no effect by inserting these noisy entries. In fact, comparison against the previous set of figures show no qualitative difference at all. If a higher brute force FP rate is desired, it should be straightforward to increase the noise rate to 200%, and it is hypothesized that this will *still* not measurably effect corroboration effectiveness (although it will significantly increase

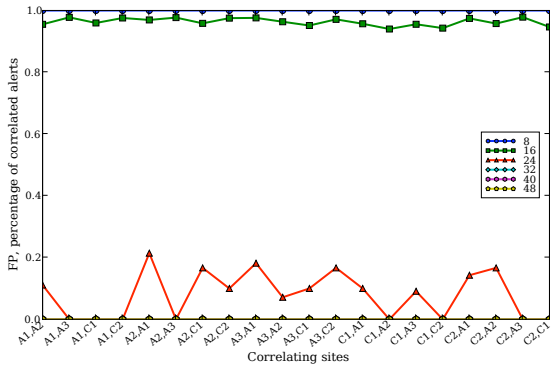

Figure 6.55: Sparse BF (100% noise) false positive rate, 2-way corroboration.

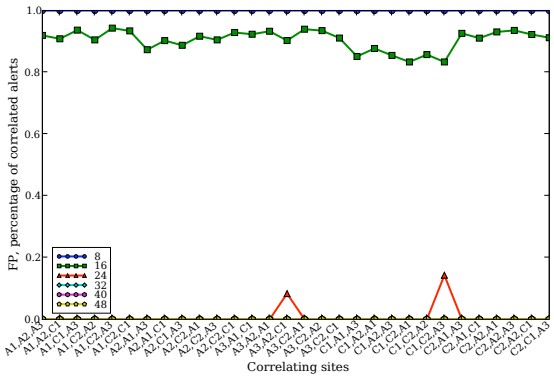

Figure 6.56: Sparse BF (100% noise) false positive rate, 3-way corroboration.

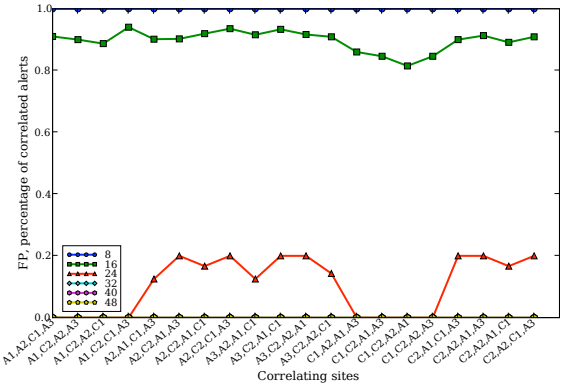

Figure 6.57: Sparse hash set (100% noise) false positive rate, 4-way corroboration.

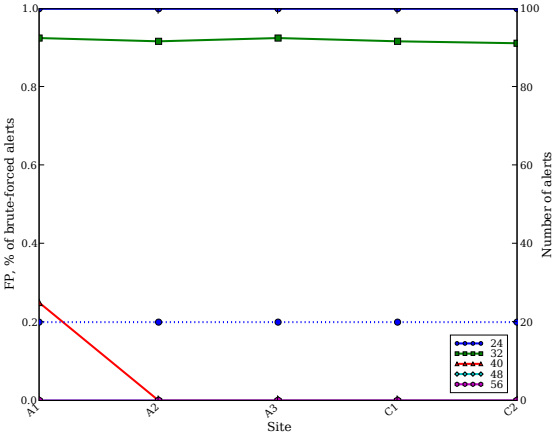

Figure 6.58: Sparse hash set (100% noise) brute force results.

storage/transmission overheads).

Temporal Corroborators

In terms of privacy gain, the MRU BF and TSBF have the potential to yield significantly more data than a regular Bloom filter, especially if each inserted timestamp is unique, as this is capable of significantly reducing false positives. Combined with the fact that the MRU BF and TSBFs use far more memory than regular BFs, this makes them unattractive for exchange. In the Worminator model, these structures are best used for local corroboration, i.e., they provide a structure that allows for effective back-lookups. A single instance is sufficient for such a model, which keeps memory requirements down, and the lack of distribution avoids

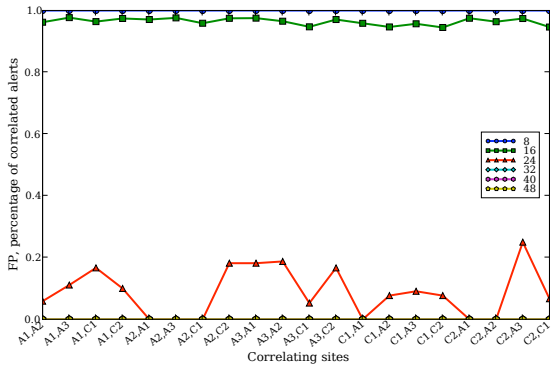

Figure 6.59: Sparse BF (100% "new" noise) false positive rate, 2-way corroboration.

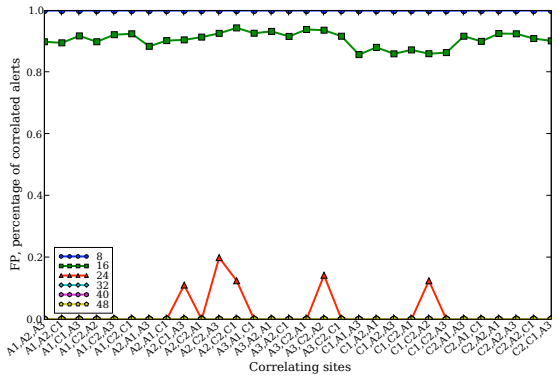

Figure 6.60: Sparse BF (100% "new" noise) false positive rate, 3-way corroboration.

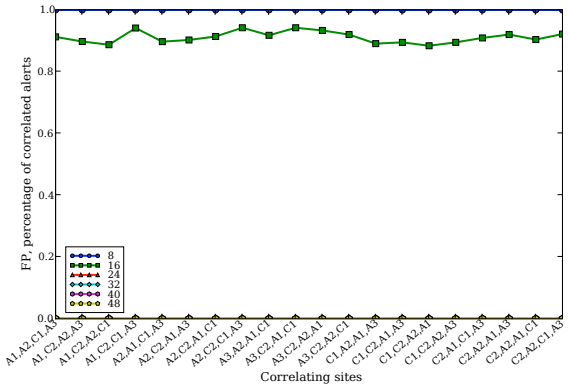

Figure 6.61: Sparse hash set (100% "new" noise) false positive rate, 4-way corroboration.

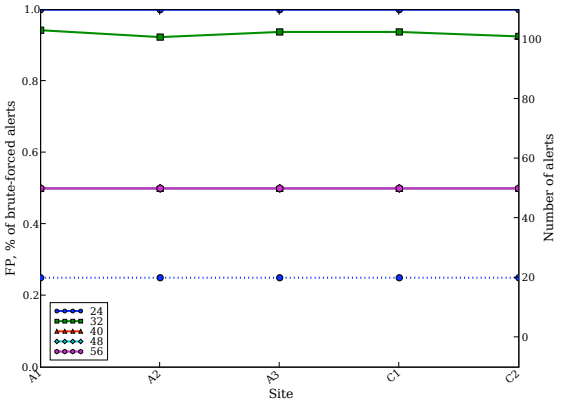

Figure 6.62: Sparse hash set (100% "new" noise) brute force results.

serious privacy implications. Both the MRU BF and TSBF allow fast generation of a "flat" Bloom filter, which may then be exchanged as desired with the same privacy guarantees as described above.

6.4.8 Longitudinal Study of Scan Behavior

This study of stealthy scan behavior is designed to demonstrate the proposed Worminator hypothesis, that collaborative intrusion detection not only enables detection of *worm spread* but also scanning behavior as precursors to an attack. There are three key *longitudes* for analysis:

1. **Over time**: as it is difficult to determine ahead of time when a widespread worm attack will occur, the goal here is not necessarily to correlate certain

scan behavior with particular attack behavior[10], but to identify long-term scanners who try to "fly under the radar" by throttling their scanning rates at any individual site to as little as a few scans per day per IP. One can define a measure of *stealthiness* by looking at scan time windows, both on a local (single-site) and on a global (many-site) basis.

2. **Over geographical and network space**: a clever attacker is unlikely to focus all their scanning efforts from a single source; instead, the current trend is to spread scan efforts over a broad range of sources, and to leverage those sources as a proxy to mask scanning behavior. Botnets ([97], [122]) are also becoming an increasingly common tool for scans. By leveraging collaboration, the goal is to observe a wider destination space to ease detection of broader networks of coordinated scanners.

3. **By target**: one key form of anonymity described in this thesis is that of *anonymous but categorizable*. This allows for the exploration of *targeted scans*, e.g., sources that scan particular categories of networks but skip others. Here, we compute aggregate statistics on the popularity of commercial vs. academic targets, etc. As of the writing of this thesis, only two commercial sites have been deployed, so the results of this experiment are limited.

Scan Lengths and Stealthiness

As the Worminator system was design to observe long-term scanning behavior, the first question of relevance is the actual scan behavior of sources, especially those who are observed at multiple sites. Table 6.4 shows aggregate scan length results for sources (site/IPs, not site/IP/destination tuples) appearing at exactly 1 through

[10]Correlating scan and attack behavior is difficult, unless known attacks occur during the scan period. While we collected much interesting information, major Internet-scale attacks did not occur during the Worminator experiment, and so conclusions are not predicated on such a correlation.

5 sites, in days, e.g. source IPs seen at four sites were observed, on average, for a period of about 30 days.

# Sites	# (Site,IPs)	Max	Avg	StDev
1	307050	373.57	7.14	33.12
2	22250	372.60	10.86	36.98
3	10074	373.64	17.20	47.01
4	3228	373.65	29.86	60.09
5	245	373.49	70.77	102.15

Table 6.4: Maximum and average scan lengths for 1–5 sites, by source IP/site, in days.

The conclusion is clear: sources which are observed at multiple sites tend to scan for longer periods. The most likely explanation for this behavior at the small scale is the elimination of false positives; source IPs that are seen at two or three sites often eliminate the local false alerts that IDSes typically observe. On the other hand, the dramatic increase in average for 5 sites is interesting. Figure 6.63 shows a time plot of the scan lengths for sources that scanned 5 sites.

This suggests that, indeed, many 5-site-scanners were long-term, and that the high standard deviation is primarily due to the limited length of the experiment (and the fact that not all sites were up for extended periods of time). Unfortunately, conclusive results cannot be drawn from the small sample set, but still, the noticeably higher average scan time suggests that many of these sources are long-term broad scanners—and that corroboration helps to identify them.

These results do not take *volume* into account, however. In particular, if a scanner happens to be a machine that aggressively scans all of its targets, that's more easily detectable without corroboration. Of greater concern are scanning sources that only generate a *few* alerts at each site over a long period of time. These scanners essentially fly under the radar by hiding behind all the noise generated at individual sites. By corroborating and looking for the slowest scanners over long periods of time, we can find, without difficulty, entities who are looking to do significant

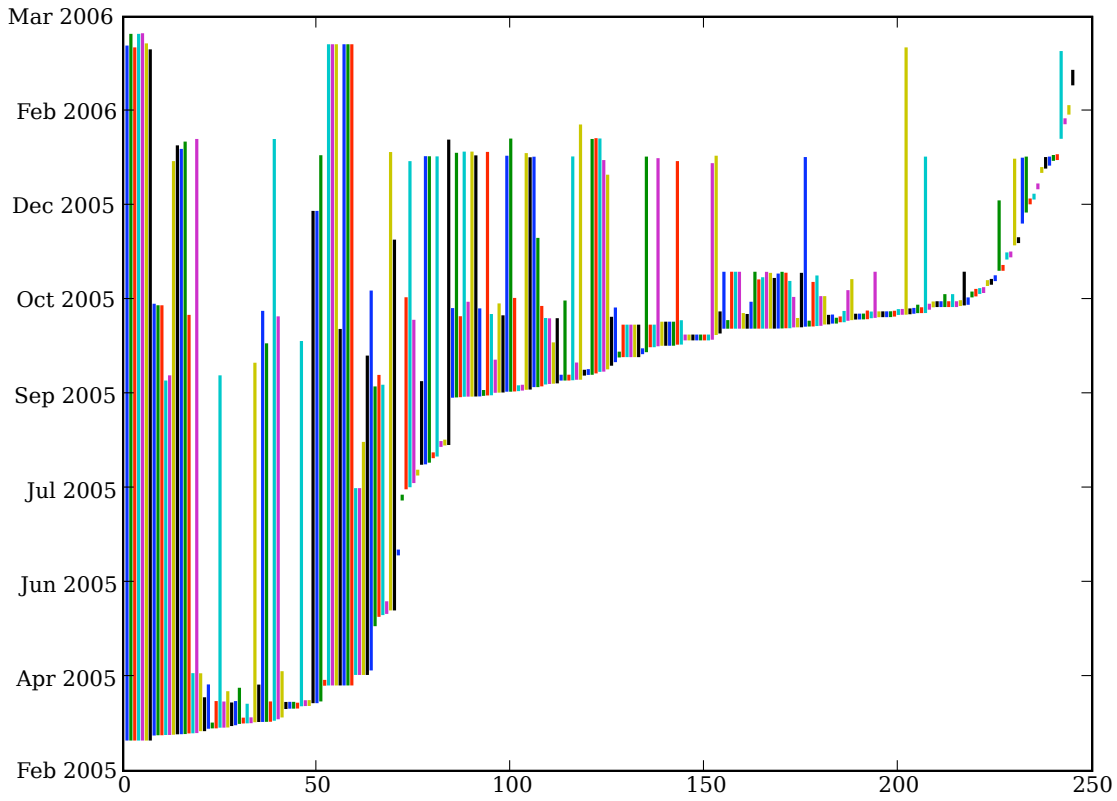

Figure 6.63: Scan length distribution, 5-site scanners.

machine mapping.

To do this, we define a "stealthiness" metric St for any arbitrary source s_i, total scanning time t_i and number of alerts $|a|_i$:

$$St(s_i) = \frac{|a|_i}{t_i}.$$

Low stealthiness levels amongst scanners at multiple sites is of particular interest; tables 6.5 and 6.6 show the top-10 stealthiest scanners detected across 4 and 5 sites, respectively.

As can be observed, there are scanners that issue only a few scans per site over the course of a year. Even more interesting are the italicized entries—these are scanners from the same subnet! A quick lookup on that /24 yields the results in table 6.7. ev1 is a major ISP in the United States, and this may have been the IP space of

Source IP	Scan Length (days)	# Alerts	*St*
61.185.246.34	257.73	7	3.144e-07
207.218.223.98	302.96	9	3.438e-07
61.129.45.54	302.12	10	3.831e-07
207.218.223.91	270.71	9	3.848e-07
207.218.223.89	271.16	11	4.695e-07
207.218.223.93	301.50	13	4.990e-07
66.150.8.18	199.92	10	5.789e-07
62.189.244.254	287.28	17	6.849e-07
61.172.250.90	234.36	14	6.914e-07
206.253.195.10	293.14	19	7.502e-07

Table 6.5: Top 10 stealthy scanners detected at 4 sites.

Source IP	Scan Length (days)	# Alerts	*St*
207.218.223.92	300.14	12	4.628e-07
207.218.223.103	302.52	17	6.504e-07
69.7.175.21	293.50	41	1.617e-06
69.25.27.10	225.52	33	1.694e-06
161.170.254.232	299.29	51	1.972e-06
219.148.119.199	227.03	45	2.294e-06
66.151.55.10	303.12	62	2.367e-06
62.73.174.150	338.39	90	3.078e-06
64.41.241.171	338.39	90	3.078e-06
64.56.168.66	338.39	96	3.283e-06

Table 6.6: Top 10 stealthy scanners detected at 5 sites.

one "customer" (be it a legitimate customer whose machines were subverted against their knowledge, or an illegitimate customer using the machines as a scanning source). The likelihood that these hosts were legitimately present at 5 disparate sites is extremely unlikely, especially since several of the sites have absolutely no relationship with each other (excepting this study; however, no Columbia IPs are listed above).

Further discussion about subnet analysis can be found later in this subsection.

Breadth and (Loud) Volume

Source IP	#sites	#alerts	Scan len	Hostname
207.218.223.92	5	12	300.14	ivhou-207-218-223-92.ev1servers.net
207.218.223.103	5	17	302.52	ivhou-207-218-223-103.ev1servers.net
207.218.223.89	4	11	271.16	ivhou-207-218-223-89.ev1servers.net
207.218.223.91	4	9	270.71	ivhou-207-218-223-91.ev1servers.net
207.218.223.93	4	13	301.50	ivhou-207-218-223-93.ev1servers.net
207.218.223.98	4	9	302.96	ivhou-207-218-223-98.ev1servers.net
207.218.223.94	3	10	300.44	ivhou-207-218-223-94.ev1servers.net
207.218.223.95	3	8	301.51	ivhou-207-218-223-95.ev1servers.net
207.218.223.97	3	8	63.06	ivhou-207-218-223-97.ev1servers.net
207.218.223.99	3	10	271.10	ivhou-207-218-223-99.ev1servers.net
207.218.223.102	3	10	297.12	ivhou-207-218-223-102.ev1servers.net
207.218.223.90	2	9	20.04	ivhou-207-218-223-90.ev1servers.net
207.218.223.101	2	5	270.55	ivhou-207-218-223-101.ev1servers.net
207.218.223.100	1	1	3.99	ivhou-207-218-223-100.ev1servers.net
207.218.223.132	1	4	2.12	ns1.rackshack.net
207.218.223.162	1	6	1.05	ns2.rackshack.net

Table 6.7: Subnet search results for 207.218.223.0/24.

As a counterpart to the previous subsection, Worminator should also ideally be able to identify the *noisy* sources—to enable, for example, evidence of an active attack. There are various ways to establish a noisy source, including: the aforementioned stealthiness metric can be used to determine the *least* stealthy source; the number of alerts generated by the IDS may also serve as an indicator, regardless of scan length; and the number of sites a source appears at. Figures 6.8–6.11 show the *noisiest* sources at 4 and 5 sites using the stealthiness and alert count metric, respectively.

The results shown here, especially the ones by stealthiness, are remarkable; for example, the top noisiest source issued 331 alerts scattered amongst 5 sites over the space of two days; a quick port analysis yields that all of these were to port 22 (ssh), suggesting a brute-force password attack against ssh servers. Moreover, a number of IPs in 61.152.* appear in the top 10 by *both* noise metrics. Of particular note was 61.152.158.109, which generated nearly 2,000 alerts over the stretch of three months at all five sources. A quick port analysis yields that these scan alerts were distributed across ports 1026–1030, which is indicative of a Windows Messenger

Source IP	Scan Length	# Alerts	*St*
61.143.210.244	0.07	137	2.224e-02
64.246.36.36	0.03	58	2.118e-02
58.215.64.204	0.03	45	2.022e-02
211.100.17.210	0.03	42	1.829e-02
202.103.178.214	0.02	33	1.817e-02
220.166.63.45	0.22	318	1.662e-02
211.76.177.154	0.13	174	1.567e-02
58.215.65.43	0.05	64	1.512e-02
222.191.251.92	0.09	111	1.488e-02
219.149.86.90	0.03	33	1.442e-02

Table 6.8: Top 10 noisy scanners by stealthiness, 4 sites.

Source IP	Scan Length	# Alerts	*St*
149.205.192.85	1.85	331	2.069e-03
61.141.32.80	11.08	484	5.055e-04
61.152.158.109	93.84	1920	2.368e-04
212.176.49.56	302.58	3697	1.414e-04
81.74.106.18	42.45	504	1.374e-04
162.40.95.86	48.64	456	1.085e-04
24.164.180.228	142.62	1300	1.055e-04
69.40.165.231	33.58	282	9.720e-05
207.67.25.104	61.45	290	5.462e-05
69.133.97.207	364.95	1654	5.246e-05

Table 6.9: Top 10 noisy scanners by stealthiness, 5 sites.

spammer. (As mentioned before, active large-scale worm attacks were not observed during this period, but one can construe a UDP spammer as an attacker, as scanning behavior will likely be similar.)

Given such metrics, a simple thresholding may enable automatic response with high confidence, which is ultimately what is desired during an actual attack. Therefore, in addition to determining stealthy *scanners*, we can also identify active *attackers*, to enable a comprehensive two-pronged approach.

Geographic Analysis

Source IP	Scan length	#alerts
12.130.50.213	373.49	5978
12.130.50.214	373.47	5589
61.152.91.69	56.19	3498
61.152.239.68	321.26	3311
219.138.199.170	47.73	2769
218.30.114.214	259.45	2436
70.86.131.171	48.94	1989
67.182.20.245	315.31	1869
68.114.241.56	307.77	1865
66.38.27.13	369.60	1853

Table 6.10: Top 10 noisy scanners by # alerts, 4 sites.

Source IP	Scan length	#alerts
212.176.49.56	302.58	3697
61.152.158.109	93.84	1920
69.133.97.207	364.95	1654
24.164.180.228	142.62	1300
199.97.98.40	368.12	1024
128.9.160.82	373.18	1017
128.9.160.251	373.49	1016
82.77.62.33	366.00	1012
128.9.160.83	373.18	1011
81.74.106.18	42.45	504

Table 6.11: Top 10 noisy scanners by # alerts, 5 sites.

Given multiple-site corroboration, we can do some analysis to see if there is any correlation between multiple-site scanners and geographic tendencies, by both the number of scanning sources and the number of alerts generated by IDS sensors. A combination of DNS and WHOIS data was used to determine the geographic distribution of IP addresses. Figures 6.64–6.68 show the results of this analysis. The country codes shown are the ISO codes used by WHOIS. Countries with less than 1% of alerts or IPs are not shown, and are instead lumped into "Other".

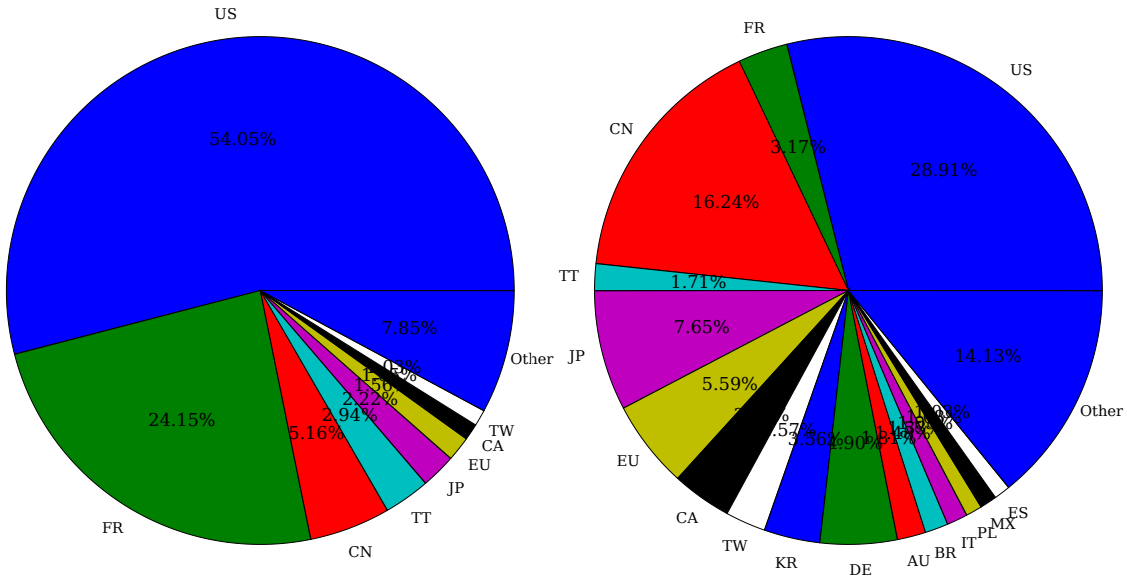

Figure 6.64: Geographic distribution of 1-site scanners, by # of alerts and # of IPs.

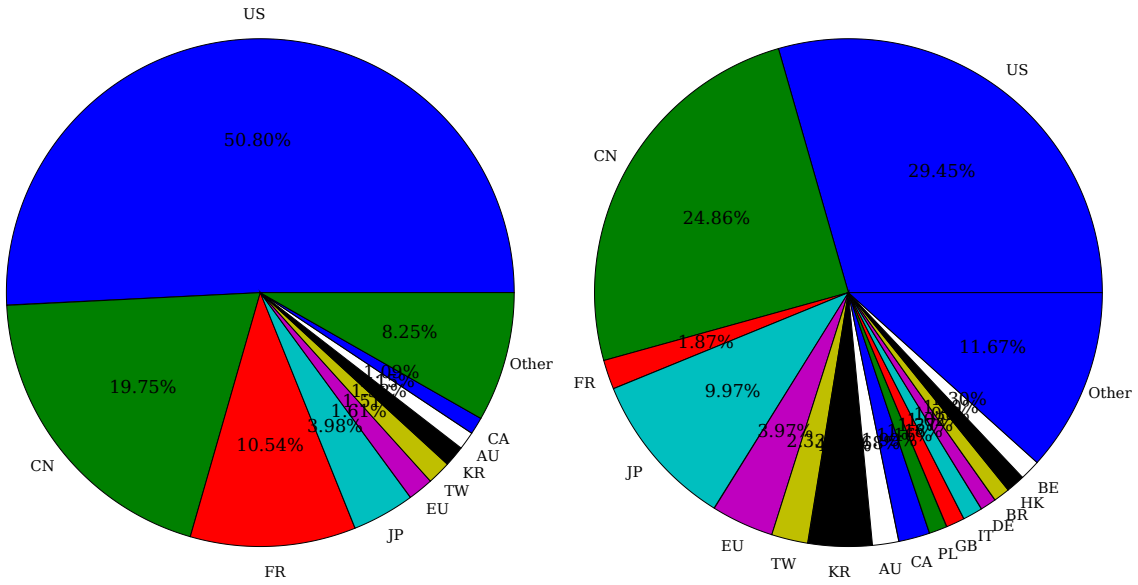

Figure 6.65: Geographic distribution of 2-site scanners, by # of alerts and # of IPs.

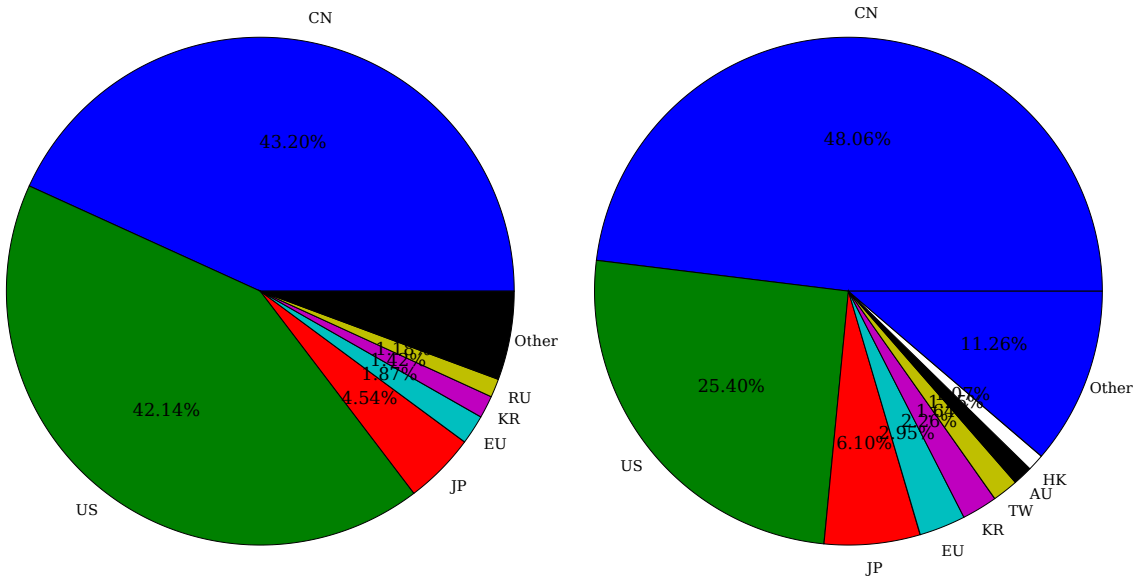

Figure 6.66: Geographic distribution of 3-site scanners, by # of alerts and # of IPs.

The trend from 1-site to 4-site is clear: as more sites' alerts are corroborated, the geographic distribution takes an increasingly international bent; most notable is the shift from the US being the primary source of alerts to China. Part of this is due to the fact that corroboration eliminates most of the false positives observed at local networks; for example, most false positives in CUCS would be attributable

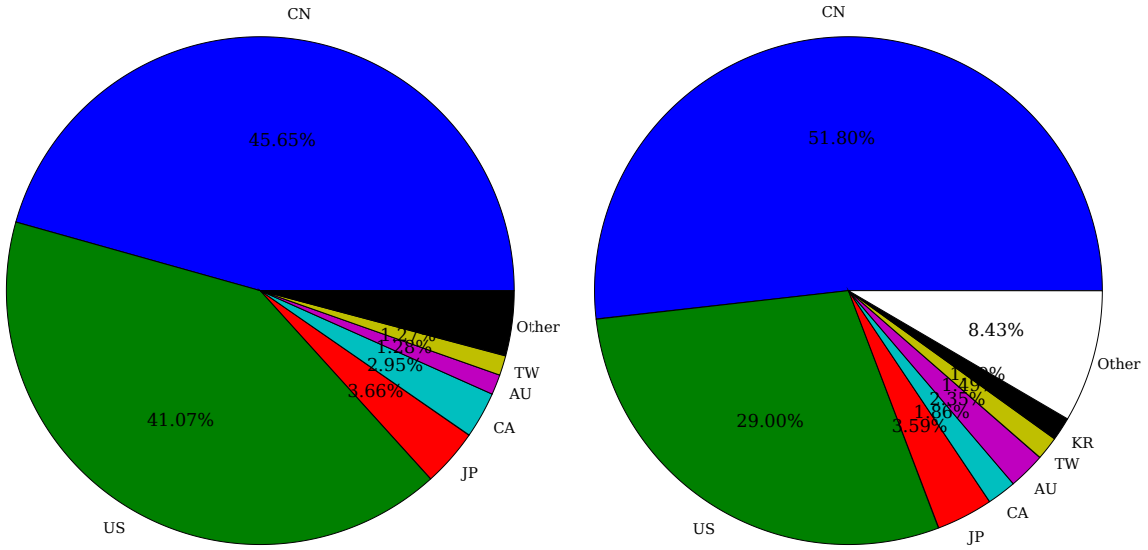

Figure 6.67: Geographic distribution of 4-site scanners, by # of alerts and # of IPs.

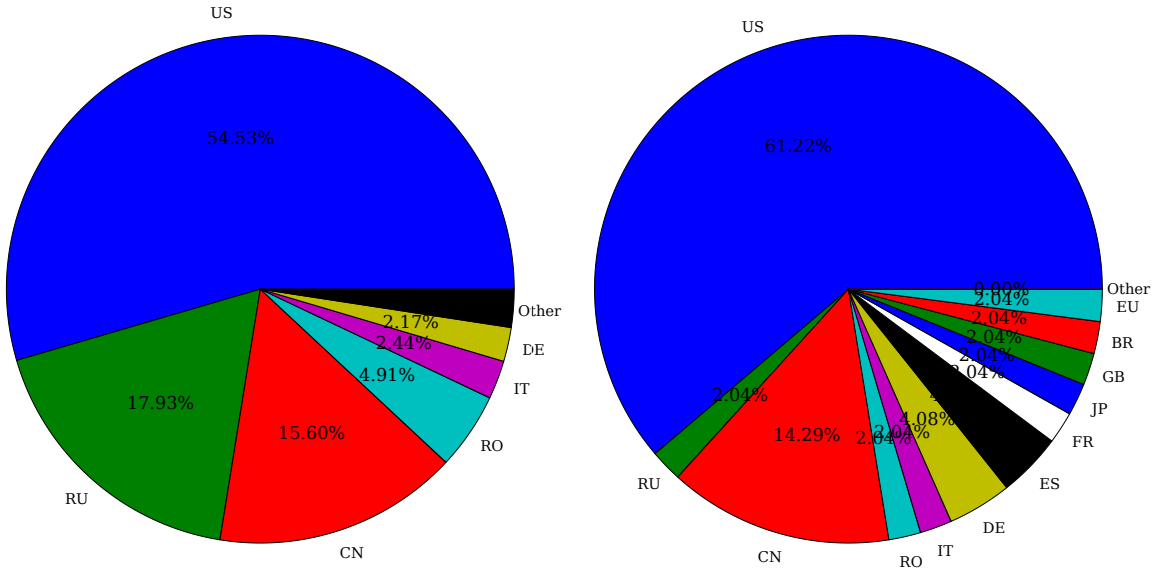

Figure 6.68: Geographic distribution of 5-site scanners, by # of alerts and # of IPs.

to machines on the same LAN. This is already visible in the second chart in figure 6.64, where the US actually has significantly more alerts than actual IP space. By the time 4-site scanners are counted, China has 45% of total alerts and 51% of IP addresses. 5-site scanners are somewhat of an anomaly to this trend. However, the 5-site dataset is small, and it is difficult to draw concrete conclusions from

it. Further study is required to see how this trend continues when more sites are included (although it should be noted that Russia seems to have a significant presence amongst 5-site scanners). What remains clear is that the largest source of scan behavior emanates, by far, from two primary countries. This also suggests that, despite the "great firewall of China", outbound scans and probes are unaffected and continue to propagate to a broad cross-section of the Internet.

Scanning Subnets

As mentioned earlier, the presence of multiple scanners within the same subnet may be indicative of a coordinated scan or attack, which may be of greater interest especially if multiple sites have seen the same behavior. To evaluate this, a scan was made through the IP addresses collected through the IDS to see if any interesting outlier class C subnets (i.e., /24s) were found. To make this more accurate, the "number of sites" criterion was modified to act as a lower bound, since not all addresses may have been detected at all sites. Table 6.12 shows the aggregate statistics for subnet scanners.

# Min Sites	# /24s	Avg	StD	Max
1	208763	1.54	3.45	254
2	10939	1.40	1.91	115
3	2882	1.46	1.50	23
4	703	1.22	0.69	10
5	46	1.11	0.37	3

Table 6.12: Statistics on scanning subnets.

On *average*, very few IPs ($\approx 1-2$) are found to be scanners within any given subnet. This would imply that a subnet with 115 scanners detected on at least *two* sites is a significant anomaly—60 standard deviations above the mean, to be precise! To get a better feel of the outlier distribution of scanning subnets, figure 6.69 shows a logarithmic graph of the largest subnets varying with the minimum number of

sites the scanners are detected upon. While all of the variations show a "long tail", as more sites are involved the "head" of the tail becomes a larger outlier. While the possibility remains that these IPs were independent and coincidentally happened to be many active blocks in the same class C, the statistics make this extremely unlikely.

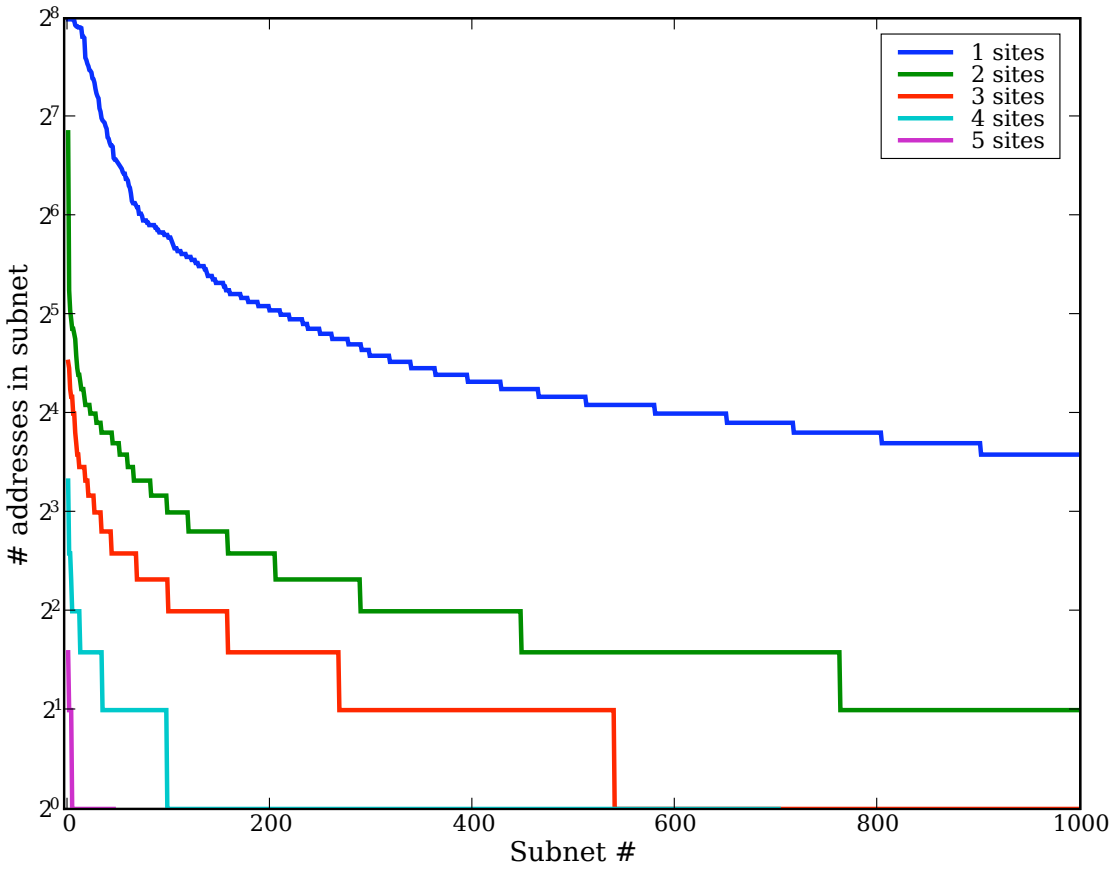

Figure 6.69: Distribution of scanning subnet sizes by varying # min sites. The top line is 1 site, and the bottom/leftmost line is 5 sites.

It is worth mentioning that newer scanning *botnets* are not necessarily restricted to single subnets; indeed, many of the newer scanning approaches use a much broader range of machines, such as compromised computers distributed across the Internet. While this analysis technique cannot detect such botnets, other approaches can, including the exploit-specific corroboration as described in section 6.5.

Target Analysis

One last significant form of analysis is looking for alerts that correspond to certain target *longitudes*, i.e., scanners that may target commercial institutions but not academic institutions, or vice-versa. These may be indicative of scan sources that are more than just purely automated—they may be actively scanning some entities and not others to build more specific hitlists.

Given our collected data, a total of 2,095 sources matched these criteria; 311 sources targeted *all three* academic institutions but neither of the two commercial ones, while 1,784 sources targeted *both* commercial institutions but none of the three academic sites. Tables 6.13–6.16 show the top 10 for each, measured by number of alerts and stealthiness, respectively, along with the top ports for the sources in question. A number of these correspond to well-known services; for a list of ports and their corresponding services, see appendix B.

Source IP	Scan Length	# Alerts	Stealthiness	Top Ports
218.94.124.43	206.47	2913	1.633e-04	8080, 3128, 8000
202.63.188.2	19.87	1062	6.187e-04	25, 3389, 202
61.233.40.205	99.78	832	9.651e-05	1028, 1029, 1032
221.12.161.99	89.81	717	9.240e-05	1029, 1028, 1032
61.138.136.28	74.40	499	7.763e-05	4257, 1029, 1031
193.6.40.135	0.19	482	2.881e-02	22
202.103.86.66	184.99	473	2.959e-05	1026, 1027, 2
62.195.115.67	13.07	331	2.931e-04	1026
219.157.19.157	69.76	319	5.293e-05	1028, 1029, 1030
67.176.227.12	25.38	309	1.409e-04	1026

Table 6.13: Academic-only scanners, top 10 by # alerts.

What these four tables make clear is that there are *both* categories of targeted scanners—those that issue many scan attempts against specific sites, and those that issue as few attempts as possible. The latter make intuitive sense; the former, however, are more unusual; if a scanner is going to generate so many reports against disparate organizations, why would they not broadly scan other networks? One

Source IP	Scan Length	# Alerts	Stealthiness	Top Ports
194.204.0.1	288.15	5	2.008e-07	53, 54463, 54558
193.92.150.3	256.93	5	2.252e-07	53, 1118, 32877
64.15.205.211	278.54	6	2.493e-07	1086, 1118, 32877
200.23.242.197	277.62	6	2.501e-07	46319, 32877, 46796
213.150.135.213	259.05	6	2.681e-07	53, 1118, 32782
68.142.249.189	171.66	4	2.697e-07	80, 53, 443
213.140.2.12	286.52	7	2.828e-07	1118, 56156, 59916
203.116.1.78	294.10	8	3.148e-07	34269, 33020, 32840
194.85.82.254	229.77	7	3.526e-07	80, 443, 42
68.142.251.21	96.37	3	3.603e-07	80, 8080, 8060

Table 6.14: Academic-only scanners, top 10 by stealthiness.

Source IP	Scan Length	# Alerts	Stealthiness	Top Ports
61.241.93.47	2.38	732	3.555e-03	783, 2622, 762
195.7.3.100	168.75	566	3.882e-05	15118
221.202.84.227	368.21	537	1.688e-05	1434
61.175.218.186	171.19	467	3.157e-05	1434
62.210.4.23	9.69	368	4.396e-04	1434
61.139.54.94	171.68	345	2.326e-05	1434, 1433
62.233.215.129	17.04	340	2.309e-04	15118, 445
80.231.169.58	372.60	317	9.847e-06	1434
202.103.207.139	83.90	303	4.180e-05	1434
61.153.15.163	142.26	262	2.132e-05	1434

Table 6.15: Commercial-only scanners, top 10 by # alerts.

Source IP	Scan Length	# Alerts	Stealthiness	Top Ports
69.157.8.5	286.74	3	1.211e-07	80
219.140.177.20	343.79	4	1.347e-07	1434
61.183.37.164	320.83	4	1.443e-07	43868, 139, 19547
80.188.58.18	306.44	4	1.511e-07	445, 135
221.15.233.166	301.49	4	1.536e-07	80
202.145.48.193	296.50	4	1.561e-07	139
200.149.32.170	290.44	4	1.594e-07	139, 135
218.95.64.229	287.86	4	1.608e-07	1434
69.157.174.218	281.53	4	1.644e-07	135
220.229.76.66	348.90	5	1.659e-07	139

Table 6.16: Commercial-only scanners, top 10 by stealthiness.

might chalk it up to coincidence, although the volume of the outliers (on the order of thousands of alerts) and scan length suggests that it is more than a coincidence.

Additionally, we can note that these scanners appear to be targeting different services across domains. For instance, academic sites seem to be more targeted for Messenger (port 1026) spam, while commercial sites seem to be targeted more for SQL vulnerabilities (port 1434); while both are discussed in greater detail in the next subsection, there may be both pragmatic and topological considerations behind these results.

In general, this form of analysis needs longer periods and broader data collection; ideally, such collection would enable analysis per *industry* or segments of industry, instead of the current rough-granular academic vs. commercial. Nevertheless, the results above show promise in this form of analysis.

Targeted Services

The other specific targeting of interest is service—namely, what services are the broad scanners particularly looking at? And does this scan behavior differ significantly between narrow and broad scanners? Tables 6.17–6.26 show the top 15 targeted ports by 1- to 5-way scanning sources by both the number of IP and the frequency of alerts.

A few conclusions can be drawn from these figures. In particular, broad scanners frequently targeted different ports than narrow scanners. Most notable was port 1026; the most common service on that port is the Windows Messenger service, listening for UDP messages. During the time period of this data collection, Windows Messenger spam was still a major open target, and it appears many nodes were scanning for non-firewalled machines to deliver such spam. [95] What is interesting is that these alerts were far more noticeable when corroborated across sites, as opposed to individual site data. This may be due to the fact that unwanted or potentially malicious UDP traffic is far harder to detect via misuse analysis without generating too many false positives, and so many IDS analysis techniques reduce

Port	# IPs	TotFreq
445	51537	999057
113	1087	967649
139	43079	790319
135	41054	753613
53	3320	635161
80	52716	452787
6881	819	309281
1025	60684	248797
1026	32966	244276
6346	1002	197356
1433	24683	185933
1434	9889	149863
25	1929	136517
137	6567	108274
33434	258	100873

Table 6.17: Top ports by frequency, 1+ site scans.

Port	# IPs	TotFreq
1026	2090	143753
33434	83	100067
53	533	98551
113	93	84649
1434	1380	83981
80	2685	65153
139	973	30470
1027	470	23372
137	303	15286
1024	578	14773
445	621	14169
3072	279	14120
1029	223	12583
1028	205	12370
25	282	11786

Table 6.18: Top ports by frequency, 2+ site scans.

Port	# IPs	TotFreq
1026	713	120073
33434	19	99128
1434	255	48687
1027	263	17332
80	431	16285
53	184	12578
1024	355	12334
3072	136	11851
113	38	11363
1029	107	8162
1028	103	8022
25	89	7721
1030	114	7113
1080	94	6288
135	414	5968

Table 6.19: Top ports by frequency, 3+ site scans.

Port	# IPs	TotFreq
1026	207	79425
1434	37	16786
1080	29	4742
1024	115	4453
3072	31	4196
135	98	3422
1029	33	3315
1028	30	3168
1030	30	2638
33437	21	2308
1027	101	2182
80	56	2131
33438	22	2015
33436	25	1792
33439	17	1596

Table 6.20: Top ports by frequency, 4+ site scans.

Port	# IPs	TotFreq
1026	11	6482
1080	4	3843
1029	6	2079
1028	5	1934
1030	5	1688
33436	9	880
135	7	842
33438	9	764
33437	8	744
33439	6	618
33440	4	376
33443	2	368
1434	1	352
80	4	346
22	2	335

Table 6.21: Top ports by frequency, 5+ site scans.

the weight of "suspicious UDP behavior" during their analysis and aggregation.

The same analysis can be applied to many of the high ephemeral ports, such as port 33435, 33436, etc. in tables 6.21 and 6.26; a quick investigation yields [22] that

Port	# IPs	TotFreq
1025	60684	248797
80	52716	452787
445	51537	999057
139	43079	790319
135	41054	753613
1026	32966	244276
1433	24683	185933
6129	13236	62960
443	10339	59947
1434	9889	149863
2745	9068	83470
3127	8098	64363
137	6567	108274
5554	6151	23818
8080	5446	90736

Table 6.22: Top ports by # IPs, 1+ site scans.

Port	# IPs	TotFreq
80	2685	65153
1026	2090	143753
1434	1380	83981
135	1276	11556
1025	1175	11288
139	973	30470
34098	963	2519
22307	907	2034
54296	849	2320
23137	832	1977
1840	832	2467
26159	806	2163
14890	805	1718
1433	793	9753
49188	791	1971

Table 6.23: Top ports by # IPs, 2+ site scans.

Port	# IPs	TotFreq
1026	713	120073
22307	668	1601
23137	629	1613
34098	628	1763
14890	625	1399
26112	617	1460
50739	586	1497
6487	585	1241
54296	575	1658
26159	572	1598
1840	559	1709
14945	558	1406
54316	554	1313
20021	554	1276
11355	537	1292

Table 6.24: Top ports by # IPs, 3+ site scans.

Port	# IPs	TotFreq
1026	207	79425
26112	207	515
22307	206	515
6487	190	403
14890	188	424
34098	187	599
48148	182	399
60766	182	400
11355	180	479
5680	180	405
37948	178	395
4981	178	507
18183	175	383
5411	168	365
15201	167	375

Table 6.25: Top ports by # IPs, 4+ site scans.

Port	# IPs	TotFreq
1026	11	6482
33435	10	222
33436	9	880
33438	9	764
33437	8	744
135	7	842
137	7	258
1029	6	2079
33439	6	618
1028	5	1934
1030	5	1688
80	4	346
1027	4	177
1080	4	3843
33440	4	376

Table 6.26: Top ports by # IPs, 5+ site scans.

this is most likely Van Jacobsen-based traceroute packets, possibly due to newer load-balancers trying to redirect traffic to the closest datacenter. More data, and more sites, may prove to help glean more useful information in this regard. However, it

is already clear that corroboration helps to identify several "chatty" ports that are not ordinarily detected by good misuse detectors, and analysis strategies should likely be reflected to include them.

One interesting change from 4-site to 5-site target results is the incidence reduction in port 1434 (MSSQL UDP). Further analysis suggests that one of the sites appeared to be blocking inbound port 1434 traffic, presumably as a preemptive firewalling strategy against Microsoft SQL-based worms. Ideally, a misuse sensor should be placed *outside* the firewall; barring that, given enough sites and analysis, it should be possible to discount certain data at certain sites due to firewalling or topology-specific considerations.

A more focused analysis of interest is to see which (IP, port) *tuples* have been observed across multiple sites. Tables 6.27 and 6.28 show the results for 4 and 5 sites, respectively, aggregated by scanner.

Source IP	# Ports	Top Ports
211.154.222.56	81	1917, 1803, 1911, 1263, 1352
209.208.0.15	66	1080, 40934, 41457, 1813, 1978
218.30.70.56	49	1393, 1151, 1928, 1093, 1295
61.145.127.92	45	1614, 1674, 1450, 1286, 1087
60.31.184.7	29	1865, 1682, 1660, 1641, 1563
221.174.17.252	28	1639, 1499, 1834, 1642, 1577
69.25.135.154	7	8000, 81, 80, 8080, 8081
82.96.96.3	7	3128, 3802, 3777, 6588, 8080
65.223.84.131	6	2301, 8000, 80, 8080, 3124
199.181.135.4	6	33438, 33440, 33439, 33437, 33436
66.219.100.118	5	1080, 3128, 6588, 8080, 80
161.170.254.232	5	33438, 33436, 33441, 33444, 33435
166.91.254.4	5	2968, 2967, 2970, 2969, 8081
216.183.96.100	5	33439, 33438, 33437, 33436, 33435
61.137.117.208	4	1027, 1026, 1029, 1028

Table 6.27: Most popular (IP, port) tuples by source IP seen at 4 sites.

This form of analysis can, amongst other things, better help sites rank threats. If a site perceives certain services, as delineated by ports, as more vulnerable, they

Source IP	# Ports	Top Ports
209.208.0.15	65	1080, 40934, 41457, 1813, 1978
216.183.96.100	4	33439, 33438, 33437, 33436
69.20.1.77	2	33440, 33438
221.12.161.109	2	1026, 1027
24.164.180.228	1	1026
61.141.32.80	1	80
64.94.45.30	1	33443
66.150.223.54	1	33443
66.151.55.10	1	33440
66.151.55.30	1	33439
66.179.168.100	1	33437
69.25.27.10	1	33440
69.40.165.231	1	1026
69.133.97.207	1	1026
81.74.106.18	1	1026

Table 6.28: Most popular (IP, port) tuples by source IP seen at 5 sites.

may choose to adopt more proactive stances against sources known to be broadly scanning those services across many sites, as opposed to ephemeral ports, possibly sign of a portscan or a less-important protocol. The results in these tables suggest such a distribution; certain sources are very focused, e.g., 69.25.135.154, which is scanning primarily HTTP ports versus 64.94.45.30, which is likely computing traceroutes as discussed earlier.

Comparison with DShield

DShield, as mentioned in the related work, represents one of the largest and most successfully-deployed DIDS around; how does Worminator compare? To further evaluate this question, IPs collected from at least two Worminator sites were checked against DShield; queries were conducted over a span of about two months and covered 12,460 sources. The results for 2-way through 5-way sites is shown in figure 6.70.

The figure makes intuitive sense; the broader the scanner, the higher the

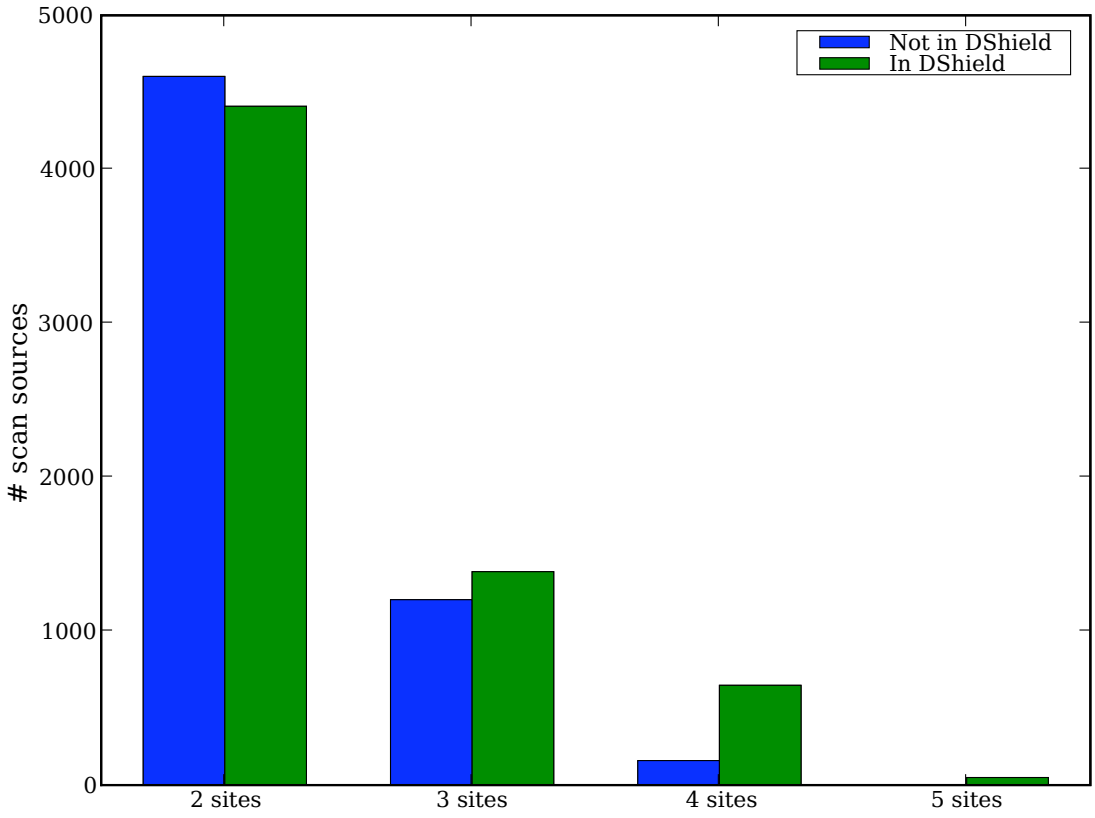

Figure 6.70: Worminator vs. DShield, 2-way through 5-way corroboration.

likelihood that it was going to be detected by DShield. In fact, if we do a stealthiness analysis, i.e., observe the stealthiness metric of the top 10 sources detected by both Worminator and DShield, the results closely match; see table 6.29 and how it compares to table 6.6.[11]

Why would DShield be able to pick up seemingly stealthy sources? Amongst other reasons, it's highly likely that some DShield contributors use honeypots or very verbose IDS sensors, choosing to submit large numbers of alerts and letting DShield find commonality between them. I hypothesize that Worminator has a better ability to pick up stealthy scanners using low volumes of alerts, but this is difficult to prove.

[11]The stealthiness metric itself was computed using solely Worminator data; the only difference here is that the source IPs listed were *also* found in DShield.

Source IP	Scan Length (days)	# Alerts	*St*
207.218.223.92	300.14	12	4.628e-07
207.218.223.103	302.52	17	6.504e-07
69.7.175.21	293.50	41	1.617e-06
69.25.27.10	225.52	33	1.694e-06
161.170.254.232	299.29	51	1.972e-06
219.148.119.199	227.03	45	2.294e-06
66.151.55.10	303.12	62	2.367e-06
62.73.174.150	338.39	90	3.078e-06
64.41.241.171	338.39	90	3.078e-06
64.56.168.66	338.39	96	3.283e-06

Table 6.29: Top 10 stealthy scanners detected at 5 Worminator sites as well as DShield.

We can also compare DShield's performance when looking at targeted sites. Figure 6.71 shows the percentage of Worminator site "targeted" alerts detected by DShield. The results are inconclusive, as the number of sites for each (three and two, respectively) yield somewhat similar results to the non-targeted analysis. Further data is needed to draw a more clear distinction between Worminator and DShield in this regard.

Finally, we can more closely examine the port distributions of IPs not detected by DShield. Tables 6.30 and 6.31 show the most popular ports of remaining scanners, including both those that scan at least 3 sites and those that scan at least 4.

When compared to the results in tables 6.24 and 6.25, there are some clear differences: the non-DShield scanners tend towards less-well-known ports. This is indicative less of an exploit delivered via a well-known execution vector, but rather some form of communication or communication attempts of services installed after-the-fact. Given the large number of IPs involved and the breadth of target port ranges for each, this is suggestive of some form of coordinated network behavior—possibly that of a bot or worm. While the DShield alerts should not be ignored, these also merit closer investigation by site administrators. Ideally, those administrators

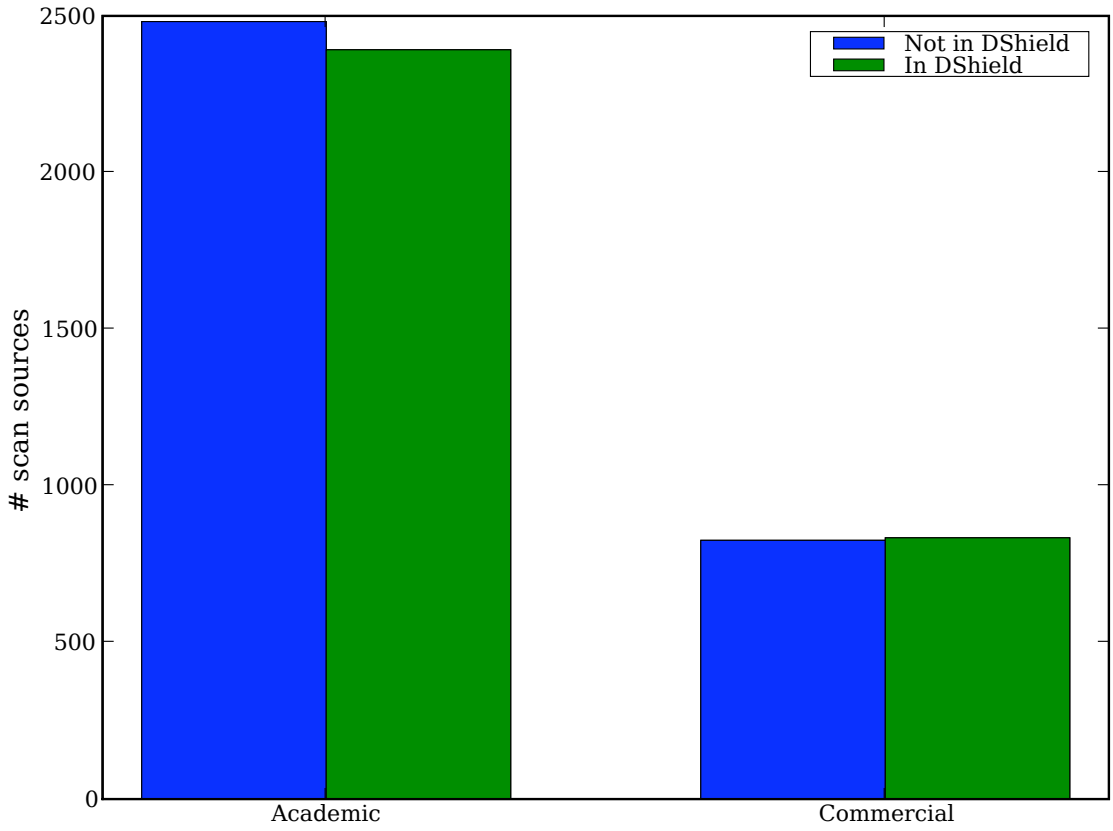

Figure 6.71: Worminator vs. DShield, academic vs. commercially-targeted sites.

would be able to conduct payload analysis on these ports to better determine their purpose.

Other Anecdotal Results

One interesting footnote: while performing analysis on the two-site data, two alerts were generated for the IP `128.9.168.45`. Reverse DNS revealed the URL to be `http://ptr.isi.edu`, which turns out to be an Internet mapping server performing "low-volume" scans. Indeed, this source comprised only .00007% of the total alert exchange, yet we were able to observe activity by the source with a minimum of effort.

Port	# IPs	Total Freq
23137	244	566
50739	231	522
54296	229	608
26159	228	589
34098	227	612
22307	226	524
49188	225	540
20021	222	470
39017	221	562
1840	219	590
54316	217	479
14890	215	470
35840	206	509
41246	205	499
26112	202	454

Table 6.30: Top ports by # IPs, 4+ site scans not in DShield.

Port	# IPs	Total Freq
22307	61	129
6487	57	103
34098	56	159
60766	56	106
4981	55	137
14890	54	103
37948	52	96
11355	52	115
48148	52	99
5411	51	95
26112	51	109
5680	51	100
15201	50	96
49188	49	113
18183	49	91

Table 6.31: Top ports by # IPs, 5+ site scans not in DShield.

6.4.9 Conclusion

Based on this section, two important conclusions can be drawn:

1. IP-based feature sharing, e.g., source address and destination port, can be successfully done in a privacy-preserving manner. The techniques outlined in this section scale effectively while maintaining adequate privacy. The key insight is that brute-force enumeration of any individual site's alerts produces meaningless results, as it is fraught with noise, but corroboration *across sites* effectively eliminates this noise, yielding accurate intersections of *watchlists* and enabling the production of *warnlists*.

2. Sharing even just source addresses and destination ports produces meaningful insight into the nature of suspected intrusions as detected by an Intrusion Detection System (IDS). First, such sharing reduces the volume of alerts that need closer inspection, enabling more detailed inspection using smaller warnlists. Second, insight as to cross-site scanning behavior enables better

classification and ranking of threats.

Based on the results presented and the above conclusions, IP sharing remains a feasible technique, and privacy-preserving corroboration is an effective means of reducing concerns about the possibility of information leakage. Future work includes increasing the number of participating sites, which will greatly increase the depth and breadth of the types of sources of surveillance and we expect it to further validate our approach.

Some significant challenges remain, however. The largest is arguably the proliferation of botnets, which threatens a massive increase in the set of sources that can be used to launch attacks. With such a larger IP space as potential sources, attackers have the flexibility of distributing sources so that cross-site scanning behavior becomes harder to detect. There is arguably no IP header-based detection mechanism that can effectively detect this behavior. Instead, the next generation of IDS sensors will have to focus on the *attack exploit*, which is independent of the attacking source. This approach generates further privacy challenges, and I describe them, as well as approaches to solve the problem, in the next section.

6.5 Payload-Based Collaboration and Signature Generation

As described in the previous section, most information-sharing approaches share header-based alert data, e.g., source IPs, destination ports, and aggregate statistics; the goal of corroborating such features across multiple sites is to detect *common sources of attack*, especially as they are performing initial scans to build hitlists in a future attack. Not only can these lists be used in building fast-propagating worms [140], they can also be used for *targeted* attacks, e.g., an attacker looking to exploit a

critical infrastructure industry that may use common services, such as the financial services industry.

However, the advent of botnets [25, 30, 122] and other forms of indirection have made it far more difficult to discover the *true* attack source, instead of bot machines which play a small role in the actual process. While firewalling can be employed against common IPs [179], there is no guarantee an attacker will not scan using one network and attack using another, thereby defeating proactive attempts. The problem has continued to escalate, and has recently been covered in mass media [96].

Instead, one can approach the problem from the perspective of detecting the actual *exploit* used in the attack attempt: in an increasingly monocultured-software world, specific vulnerabilities are common to a large pool of applications [90]. We can leverage the *commonality* of such attack approaches (and, in particular, the "invariant substrings" as defined in [103] and others) to identify and protect against such attacks even if target machines are unpatched and remain vulnerable. This is particularly advantageous for zero-day worm detection, when common attack vectors may present themselves across many sites in short timeframes, i.e., corroboration of common alerts across space, and for stealthy scanning for long time periods, i.e., corroboration of common alerts across time. In either case, corroborating alerts among collaborating sites requires careful design for accuracy and efficiency.

Of course, exploit-specific vulnerability detection has its challenges: in particular, a reliance on *payload* detection and corroboration is necessary. It is impractical to assume that organizations can exchange raw traffic streams; there is far too much data of a potentially sensitive nature. Even if exchanged material is confined merely to suspicious payloads as classified by an anomaly detector, organizations may fear that some legitimate and/or sensitive traffic may be misclassified and exchanged to

other, possibly competing institutions. Instead, techniques are required to exchange *privacy-preserving* alerts that make it impossible for other entities to determine the actual content of the underlying traffic, yet at the same time exchanging information that can effectively be corroborated. We propose that this is not only possible, but practical and broadly applicable, and propose a collection of techniques to do so.

The rest of this section is organized as follows. Section 6.5.1 introduces the payload-enabled privacy-preserving corroboration techniques at the heart of this section. Section 6.5.4 then shows some early results based on the techniques described in section 6.5.1, including performance, accuracy, and the efficacy of signature generation. Finally, a privacy characterization of the techniques is discussed in 6.5.6.

6.5.1 Corroboration Methodology

Once again, a set of participants $A, B, C, \ldots$ each contain IDS sensors $\mathcal{I}_A, \mathcal{I}_B, \mathcal{I}_C, \ldots$ that produce alerts $\mathcal{E}_A, \mathcal{E}_B, \mathcal{E}_C, \ldots$ at times, i.e., $e_{At_1}, e_{Bt_2}, \ldots$.[12] However, a significant difference is in the *content* of alerts; instead of having discrete features, an event may now contain arbitrary binary *payloads*, e.g., for an event with a payload length m, $\rho_e = \{b_0 b_1 \ldots b_m\}$, b_i referring to a single (typically 8-bit) byte of data.[13] This payload ρ is presumably (but not required to be) generated by an anomaly detector that classifies it to be of interest.

Privacy Transformations

What we would *like* is a transformation that allows us to extract discrete features from ρ_e, which can then be binary-modeled or frequency-modeled and scored appropriately for corroboration. In order to do this, we use the n-gram

[12] And again, a hierarchical characterization of multiple sensors of sites is left to future work.
[13] ρ is not to be confused with p or $\mathcal{P}$, both of which correspond to privacy transforms.

incremental analysis approach described in section 5.1.3. We may also keep source, destination service, and/or timestamp as per the mechanisms described in the previous section. In other words, an alert a_i consists of many n-grams derived from ρ_{e_i}, i.e., $a_i = \{\{b_0 b_1 \dots b_n\}, \{b_1 b_2 \dots b_{n+1}\}, \dots \{b_{m-n} b_{m-n+1} \dots b_{m-1}\}\}$. Each of the alerts, i.e., in the set $\mathcal{A} = a_0, a_1, \dots, a_k$, are then privacy-transformed, i.e., $\mathcal{A}' = \mathcal{P}(A) = \{p(a_1), p(a_2), \dots, p(a_k)\}$, as follows:

- Frequency-modeled 1-gram alerts, i.e., $\mathcal{A}' = \mathcal{M}_F(a_0), \mathcal{M}_F(a_1), \dots, \mathcal{M}_F(a_k)$, where $\mathcal{M}_F$ is the frequency transform, i.e., $\mathcal{M}_F(a_i) = \mathcal{M}_F(b_{i_0}, b_{i_1}, \dots, b_{i_k})$. In other words, for each individual alert, the bytes are treated as individual features (1-grams) and a frequency distribution is composed over them. We *could* also build the frequency distribution of n-grams with $n > 1$, but [173] observes that the sparse frequency matrix is computation-intensive, requires significant amounts of space and does not provide significantly better results than binary-modeled n-grams.

  An alert/packet payload represented by its byte frequency distribution makes it nearly impossible to reconstruct the actual payload except in degenerate cases—the byte distribution contains byte values but no sequential information.

- Optionally, frequency-modeled 1-gram alerts can be reduced yet further into a *Z-String* [174], representing the content rank-ordered by frequency, i.e., $\mathcal{Z}(\mathcal{M}_F(a_i)) = \{b_\alpha b_\beta b_\gamma \dots\}, f(b_\alpha) \geq f(b_\beta) \geq f(b_\gamma)$, etc. Typically, such rank orderings produce Zipf-like distributions (exponentially decreasing frequency values). We rank order the distribution of a suspicious packet from most frequent to least and drop the frequency counts, resulting in a "Zipf String".

- Both frequency models and Z-Strings offer privacy. However, neither can represent a *sequence* of characters. For worms and other malicious binary payloads, we may want to capture such sequences, as they may serve as

invariants across multiple suspect payloads that can be corroborated. We therefore create **binary-modeled n-gram alerts**, i.e.,

$$a' = p(\{\{b_0 b_1 \dots b_n\}, \{b_1 b_2 \dots b_{n+1}\}, \dots, \{b_{m-n} b_{m-n+1} \dots b_{m-1}\}\}) =$$
$$\{p(\{b_0 b_1 \dots b_n\}), p(\{b_1 b_2 \dots b_{n+1}\}), \dots, p(\{b_{m-n} b_{m-n+1} \dots b_{m-1}\})\}.$$

These may then be inserted into a Bloom filter, i.e., $\mathcal{A}' = \mathcal{M}(A)$, by inserting the privacy-transformed n-grams into

$$B = \{a'_0, a'_1, \dots, a'_k\} = \{h(\{b_{0_0} b_{0_1} \dots b_{0_n}\}), h(\{b_{0_1} b_{0_2} \dots b_{0_{n+1}}\}), \dots,$$
$$h(\{b_{0_{m-n}} b_{0_{m-n+1}} \dots b_{0_{m-1}}\}), h(\{b_{1_0} b_{1_1} \dots b_{1_n}\}), h(\{b_{1_1} b_{1_2} \dots b_{1_{n+1}}\}), \dots,$$
$$h(\{b_{1_{m-n}} b_{1_{m-n+1}} \dots b_{1_{m-1}}\}) \dots\}.$$

This implies that the n-grams from multiple alerts may be mixed as a "bag", if desired, for additional space savings over the use of separate Bloom filters per alert.

The resulting Bloom filter(s) and/or frequency distributions essentially serve the purpose of a participant's watchlist as per the previous section, but this time containing a representation of actual payload content, as opposed to IP endpoint information. Publication can be scheduled in a flexible manner, as described earlier. Upon receipt of one or more such watchlists, a warnlist composed of frequency transforms, n-grams, or even n-gram-flattened signatures can be generated; these are discussed in a later subsection.

6.5.2 Evaluating Corroboration

We corroborate content alerts using three main approaches: raw packet alert corroboration, privacy-preserving frequency-based alert corroboration, and privacy-

preserving n-gram alert corroboration. First, however, we develop several metrics as to how we can best compare these techniques.

One can view payload corroboration techniques as a tradeoff: on one extreme, we can consider the idea of transmitting the raw packets that generated alerts; while this enables any corroboration technique, we consider it infeasible because of the sheer amount of data and the fact it is not privacy-preserving. On the other end of the spectrum, we can consider privately-encrypted packet content: unless the key is shared, it essentially appears as noise to peers—but this requires all or no trust. The payload techniques described here fall somewhere in between, and we characterize their relative merits from two perspectives: our ability to corroborate data given a transformed version of packets, and the amount of privacy that is gained using different privacy-preserving transformations of packet content.

Corroboration ability. The fundamental question, given any technique, is whether it is possible to corroborate alerts with low false positive and low false negative rates. Given *raw*, non-privacy-transformed packets that generate an alert, there are several well-defined algorithms that aim to accomplish this task. We consider the *longest common subsequence*, or LCSeq (discussed below), as an appropriate baseline, as it is able to find any non-semantic commonality in the candidate packets. Other approaches, including semantic matching, are discussed briefly in section 4.5, and are considered outside the scope of this chapter, which focuses on corroboration amongst pure network sensors, i.e., no host-specific information.

Given a technique, and a collection of alerts, a *similarity score distribution* is computed as each pair of alerts is tested (§6.5.4). This score distribution then becomes a useful metric for comparing corroboration ability. If we consider LCSeq as a useful baseline, for instance, we can measure the deviation of other techniques from LCSeq as a comparative measure of how other techniques corroborate alerts.

Ideally, a network sensor would be able to use a privacy-enabled technique and get similar results, signifying an increase in the privacy preservation while maintaining the ability to determine common threats and exploits.

Privacy gain. We characterize the baseline as having *no privacy* as raw packets are exchanged, and having *total privacy* with encrypted content without the corresponding key (noise). To characterize intermediate approaches, we utilize a probabilistic model: given a representation of the encoded payload, what is the likelihood that a curious peer would be able to reconstruct the original, possibly sensitive data? For most of the approaches listed, we can estimate this probability by determining the number of original payloads that could be represented by the encoded alert; the resulting measurements are discussed in section 6.5.6.

Corroboration speed. Finally, one remaining important characteristic is the ability to corroborate *quickly*, especially if many sites are involved with many alerts being generated and exchanged. This "speed" metric is reflected in two aspects: the resulting alert size after a transformation is applied, and the computation overhead necessary to transform the original alert. As with the previous cases, we consider raw packets the baseline: it is the largest unencrypted alert encoding (up to 1500 bytes, i.e., bounded by packet size, per alert) and LCSeq is amongst the slowest corroboration mechanisms (up to polynomial-time with respect to buffer size).

Baseline Corroboration: Raw Payload Techniques

As previously discussed, we choose raw packet alert corroboration as a baseline technique: it contains the most complete original information.

SE: String Equality. This is the simplest and most intuitive corroboration approach. Two alerts are deemed similar to each other only if they have identical content. This metric is very strict and does minimize false positives, but has no tolerance for any variation—fragmentation, polymorphism, obfuscation, etc.

Equality is memory and computationally efficient (linear time).

LCS: Longest Common Substring. LCS is one of the classic string comparison techniques; it is less deterministic than SE, and is not susceptible to fragmentation. The longer the string that LCS computes, the greater the confidence that the compared alerts are similar. While it allows minor payload manipulation, multiple changes often cause a short LCS, reducing confidence in its corroboration ability. LCS is reasonably fast; a suffix-tree implementation is linear-time, but at the cost of having to store a suffix tree per alert (or $O(n^2)$ for a naïve but memory-efficient algorithm).

LCSeq: Longest Common Subsequence. LCSeq can be considered a generalization of LCS; instead of finding a single contiguous matching block, LCSeq allows non-matching characters to be interposed. This enables detection despite a variety of payload manipulation operations, including insertion and reordering, and potentially polymorphism. Like LCS, the length of a LCSeq is an indication of similarity. Its main shortcoming is its computation overhead; at best, sparse dynamic programming can achieve, on average, $O(n \lg n)$ complexity (and can range to $O(n^2 \lg n)$ worst-case).

ED: Edit Distance. Edit distance, also known as Levenshtein distance, is another commonly-used approach to compare string similarity. It computes the smallest number of insertions, deletions, and substitutions required to change one string into another. In general, it has similar properties as LCSeq.

Frequency-Modeled 1-Gram Alert Corroboration

Having discussed different techniques for raw payload comparison and corroboration, we now describe our first alert transformation: frequency modeling. As our work on PAYL demonstrates [174], 1-gram frequency models are a good indicator of the nature of packet content. We can leverage this technique and use frequency

distributions as alerts, either with the corresponding normalized frequency counts or with an approximation of this information.

Frequency Distribution. Given two packets with their respective byte distributions, we can apply standard distance metrics to determine similarity; Manhattan distance is efficient ($O(n)$ in length of the alert) yet produces a good approximation of the actual distance. Frequency-based alerts are comparatively sized to packets; a floating-precision frequency distribution takes 1KB of space.

Z-String. A Z-String relies on the relative notion of frequency just by the ordering of the individual byte values, and since it is a string, we can apply the raw matching techniques described above (SE, LCS, LCSeq, ED) to the Z-Strings themselves. Z-Strings are also often smaller than full packets (e.g., 8-bit byte-based packets would be referenced by a 256-byte Z-String), and as such the string comparison times are generally shorter than on the raw packets themselves. However, Z-Strings still have an $O(n \lg n)$ creation overhead in the size of the alphabet. (See section 6.5.4 for an example of a Z-String.)

Binary-Modeled N-gram Alert Corroboration

As discussed in [173], binary-based modeling produces surprisingly good results and leads to two different possible alert types.

N-gram signature. We can generate a list of n-grams that are found to be suspicious from an originating packet. Such a "signature" is position-independent while capturing specific malicious byte sequences. Given two n-gram signatures, we can simply compute the intersection of the two and threshold the cardinality of the intersected set to determine a similarity score. Such an intersection is linear time in the length of the signatures by using fast set-based data structures; depending on the n-gram size and packet content, this can vary significantly; while most packets are regular and have few n-grams, encrypted traffic, with a very flat byte

distribution, can have as many n-grams as the size of the packet itself. In either case, an n-gram signature is a degenerate form of a raw packet; when distributing large n-grams, this is clearly not privacy-preserving, as even a 5-gram can contain a password. In these cases, we need a transformation on the n-gram itself.

Bloom filter n-gram signature. Instead of publishing an n-gram signature, we can instead insert the n-grams into a Bloom filter and publish it.[14] Since Bloom filters support both insert and verify, set intersections can be done between a (local) "raw" n-gram signature and a published BF n-gram signature, identifying the same n-grams as the previous technique *without* yielding other, potentially sensitive n-grams. This approach is also linear in time but leverages a BF's space efficiency. Optionally, *multiple* alerts can be published via a single Bloom filter, treating the BF as a *bag* of suspicious n-grams. This enables a multiplicative reduction in the amount of data transmitted and work needed to compute intersections.

Incidentally, corroborating *two* BF n-gram signatures from different sites can be done via a bitwise AND "intersection"; this does not yield actual n-gram content, but may help find *commonality* between signatures, increasing confidence that the correct common code has been found when corroborated against local data. BF intersection can also be used for model comparison, e.g., comparing two Anagram models to see if different sites exhibit similar traffic properties. Experiments on these approaches are outside the scope of this paper and are briefly discussed in section 7.3.

6.5.3 Performance and Scalability

In order for privacy-preserving corroboration to be effective, it must impose a small overhead. This subsection discusses the performance overheads for hashing and Bloom filters, both of which are critical for n-gram analysis.

[14] This is *not* to be confused with Anagram's use of a BF model; here, individual alerts are placed into Bloom filters.

Full Payload Analysis

Unlike IP lists, privacy transform performance obviously changes when considering larger items, such a full packet payload (typically between 1 and 1500 bytes). Figure 6.72 shows the computation overhead when a distribution of such packets is considered with a selection of hash functions. More precisely, 100,000 packets were sampled from the CUCS network, covering approximately 10 seconds of traffic, and had an average length of 228 bytes and standard deviation of 410 bytes.[15] The results are shown in figure 6.72. The average packet length is shown as a dotted black line in the figure. The "None" line corresponds to the cost of merely *copying* the packet in memory, i.e., doing a linear read/write without any further transformation.

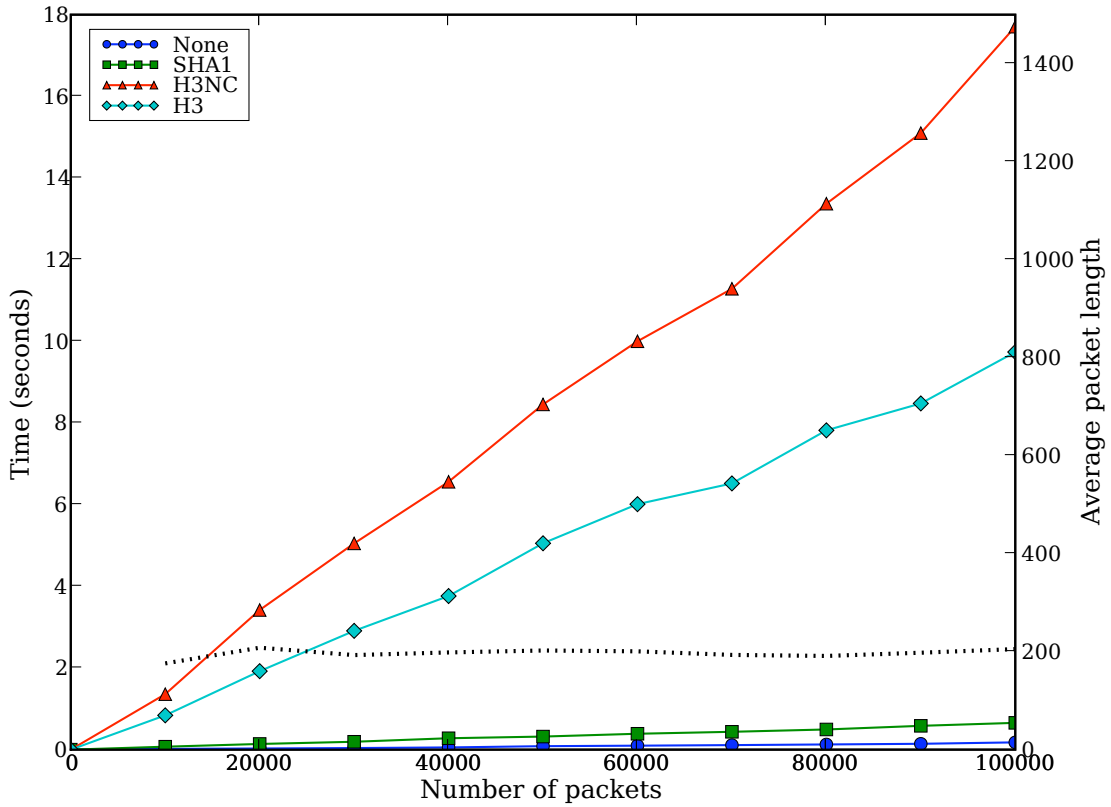

Figure 6.72: Performance Comparison of Hash Functions for Packet Payloads.

[15]This is likely due to the extreme distribution produced by packets that have zero content as opposed to packets carrying large amount of network data. Nevertheless, this does not invalidate the experiment, as the computation trends remain the same even if only "long packets" are considered.

The figure shows that SHA-1 is extremely fast, processing ten seconds of gigabit network traffic in under one second of computation time. SHA-1 is a block cipher, supporting 512-bit blocks, and this likely accounts for the result. H_3, on the other hand, must process bytes individually, significantly increasing the amount of work needed as more packets are processed. (A larger lookup cache for H_3 can be used, if necessary, at the cost of additional memory utilization.)

Per-payload 1-gram frequency analysis is also an extremely fast transform. Figure 6.73 shows the results of both a per-packet frequency transform and Z-String generation, which instead of normalizing the frequency distribution merely sorts it. As such, the two of them still do less data manipulation than the cost of simply copying the packet. A fourth transformation, "Hashcode", is shown; this is the built-in Java Object hash function, which generates fewer bits of output than other hash functions and is extremely efficient, but is less generally useful, as it suffers much higher collision rates. A technique to effectively leverage Hashcode is described in the next subsection.

N-Gram Analysis

Unsurprisingly, partial matching via n-grams significantly raises computation requirements; due to the sliding window over which a set of n-grams is computed, byte values are repeatedly hashed. Unfortunately, individual hash byte values cannot be reused, as bytes are hashed in different positions each time. This adds a significant constant-time overhead to the overall transform. (N-gram values *can* be reused; I discuss this possibility shortly.)

In the experiment detailed in figure 6.74, 10,000 packets were randomly sampled out of the 100,000 used in figure 6.72. N-grams for a range of sizes, from 3 to 10, were computed.[16] Between 2.75 and 2.85 million n-grams were extracted from this

[16]Testing using the Anagram sensor showed that these size n-grams produce the best results for payload anomaly detection [173, 171].

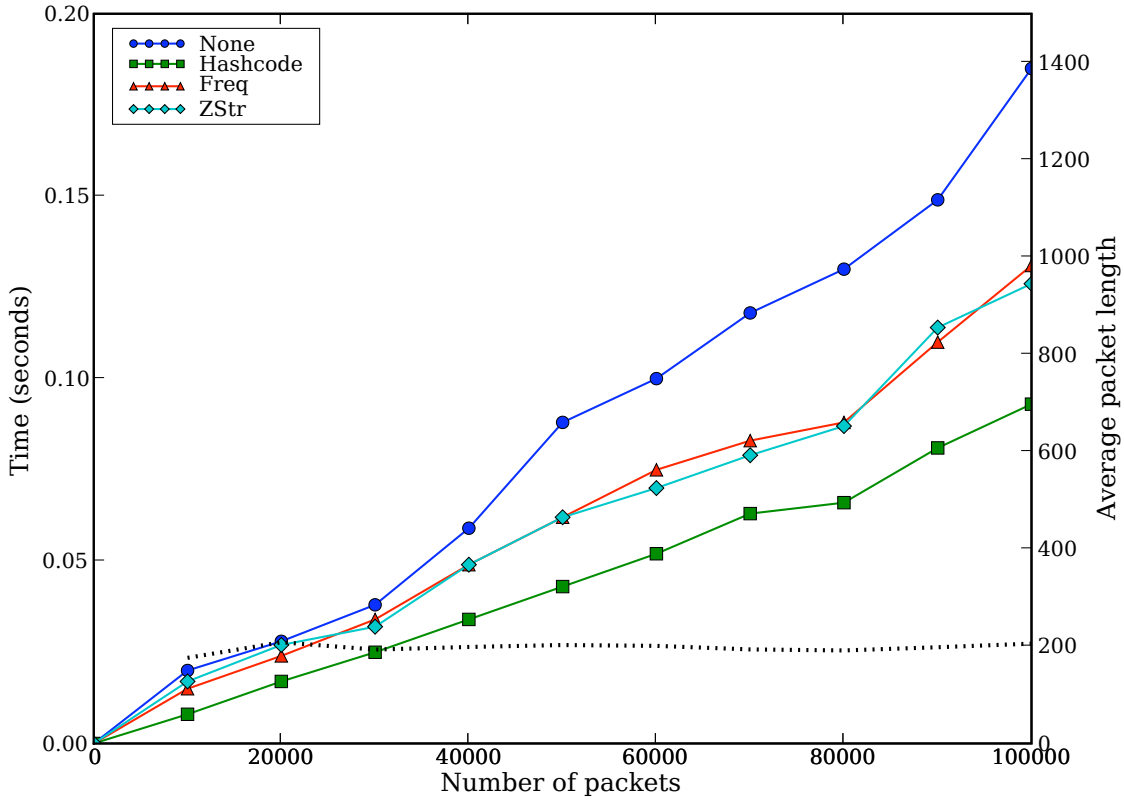

Figure 6.73: Performance Comparison of Frequency Transforms for Packet Payloads.

data (when using sizes ranging from 10-grams through 3-grams, respectively, i.e., there are more 3-grams than 10-grams). As mentioned earlier, the result is a clearly greater computation overhead than for the same number of packets in figure 6.72. One other hash function is present—"H3Inc"—which I discuss below.

The computation overhead is additive if multiple n-gram sizes are desired for any given data. To ameliorate this cost somewhat, two different optimizations can be used: first, the optimization discussed in section 5.2.2 can be used here to reduce the hashing computation overhead to, essentially, the amount of work needed to hash the largest n-gram. This technique is particularly useful when creating models that contain mixtures of n-grams for matching accuracy or flexibility when corroborating against different inputs. The result of using an incremental H_3 implementation is shown as "H3Inc" in the aforementioned figure 6.74. Note that each additional

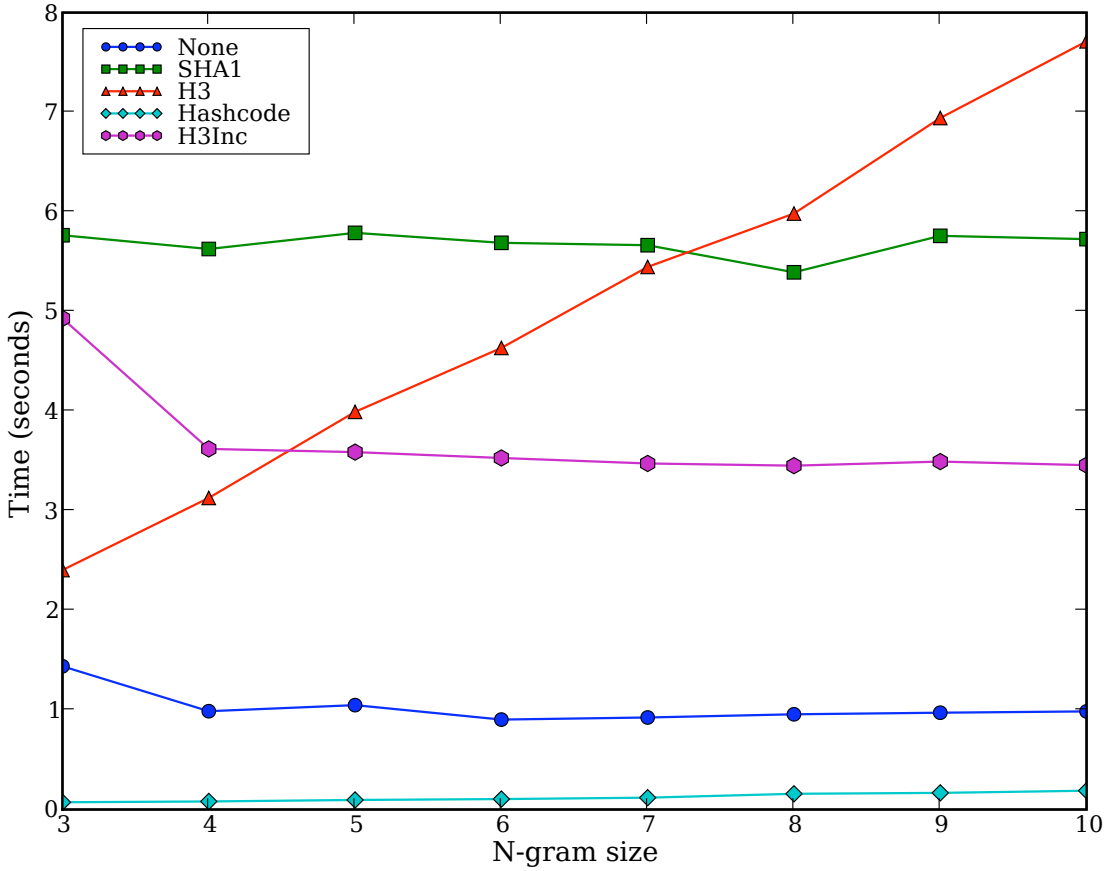

Figure 6.74: Performance Comparison of Hash Functions for N-grams.

n-gram size imposes a constant-time overhead, which for large mixes of n-grams grows far more slowly, additive, than using a non-incremental hash function. (The higher initial cost is more difficult to explain; most likely, the testing mechanism, which tests multiple n-grams for the same information at the same time instead of testing all n-grams of one size separately, leverages the computer's and VM's caching mechanisms less efficiently. Implementation in a lower-level language, like C, may reduce or eliminate this performance difference.)

Second, a *cache* of n-grams can be created to speed up hash value lookup for the most commonly-occurring n-grams. While it may seem unintuitive to build a "hash cache", as the keys in this cache would be looked up via a hash function themselves, the hash function used for these keys can be far simpler than the general

hash function, and would simply index into a small array of the actual n-grams in question, as is illustrated by the "Hashcode" value in figure 6.74. Different caching strategies, as well as different cache sizes, can be adopted. Figures 6.75-6.78 show results with different n-gram sizes and cache policies, using the same dataset as in figure 6.74. Note that each graph has two axes; the left axis, corresponding to the solid lines, refers to the cache hit rate, while the right axis refers to the computation time involved in creating and maintaining the caches. A graph combining the cache maintenance overhead with actual n-gram hashing is shown later, in figure 6.79.

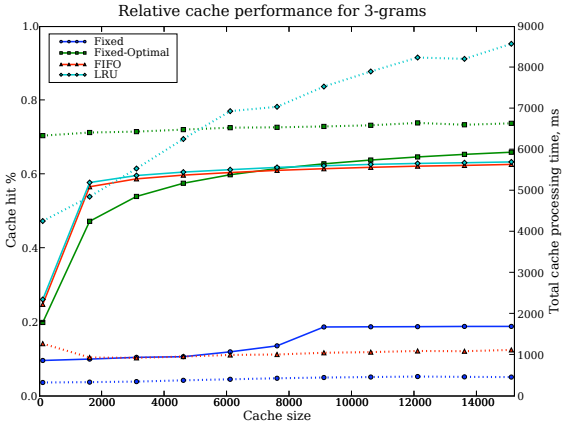

Figure 6.75: Cache performance, 3-grams.

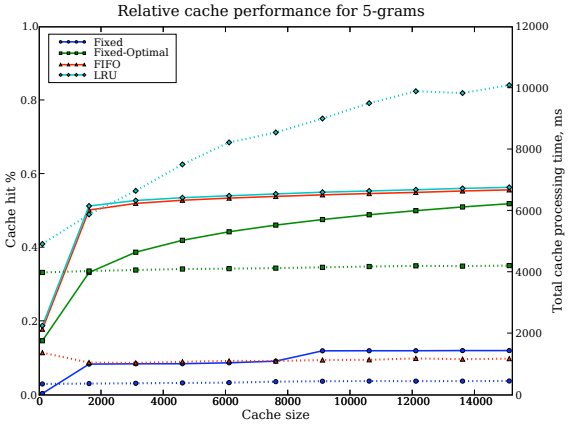

Figure 6.76: Cache performance, 5-grams.

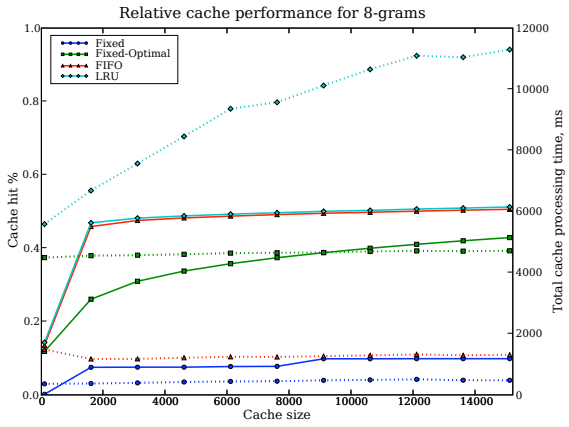

Figure 6.77: Cache performance, 8-grams.

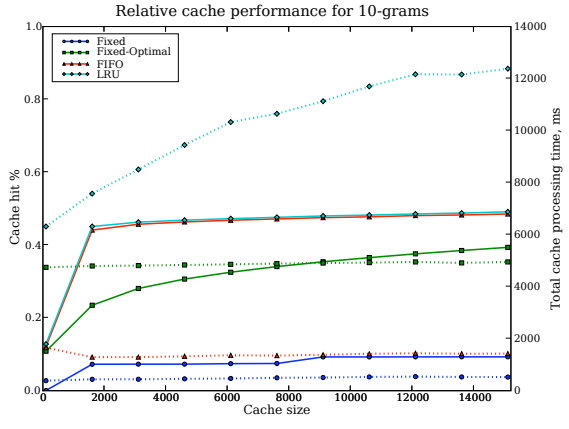

Figure 6.78: Cache performance, 10-grams.

As is shown in the figures, four different hashing functions are used. "Fixed" is the simplest; the cache is populated with misses until it is full, and is only checked thereon. "Fixed-Optimal" employs a two-pass stage: n-gram frequencies are first

computed, and then only the most popular n-grams are statically populated and kept in the cache. "FIFO" and "MRU" refer to first-in-first-out and most-recently-used cache policies.

Several conclusions can be drawn from these figures. First, cache efficiency will vary by n-gram size; in particular, smaller n-grams work better, which is unsurprising as the total population of n-grams is smaller. Interestingly, Fixed-Optimal's performance declines precipitously as the n-gram size increases; this strongly suggests that global frequencies are not necessarily useful when the population of possible n-grams far outstrips cache sizes. Instead, some locality with respect to the data being processed at any *time quantum* is far more useful. Finally, the most interesting result is that a simple-to-implement and fast FIFO policy works nearly as well as the best, but significantly slower MRU policy. In other words, given network traffic, locality is the most important principle in repetition. In retrospect, this is unsurprising—between whitespace, repeated characters, and consistent data encodings through any arbitrary packet, a cache is most useful at preventing duplicates in a particular network flow or flows at any given time. Best of all, the FIFO cache policy has significantly smaller computation overheads when compared to Fixed-Optimal (whose two-phase pass requires significant precomputation) and MRU (whose requirement of a heap or similar sorting data structure in order to determine the least-recently-used item significantly adds to the cost).

Therefore, FIFO was used as a caching policy when actually processing n-grams; the result with a 15,000-entry cache is shown in figure 6.79. H_3 is used for both the baseline and optimized versions ("H3FIFO"), and the same network traffic as per the previous figures is used.

At first glance, these results seem poor; for a small number of n-grams, there is hardly any difference. This can be largely attributed to the fact that the cache's benefits are neutralized by the additional bookkeeping required to maintain the

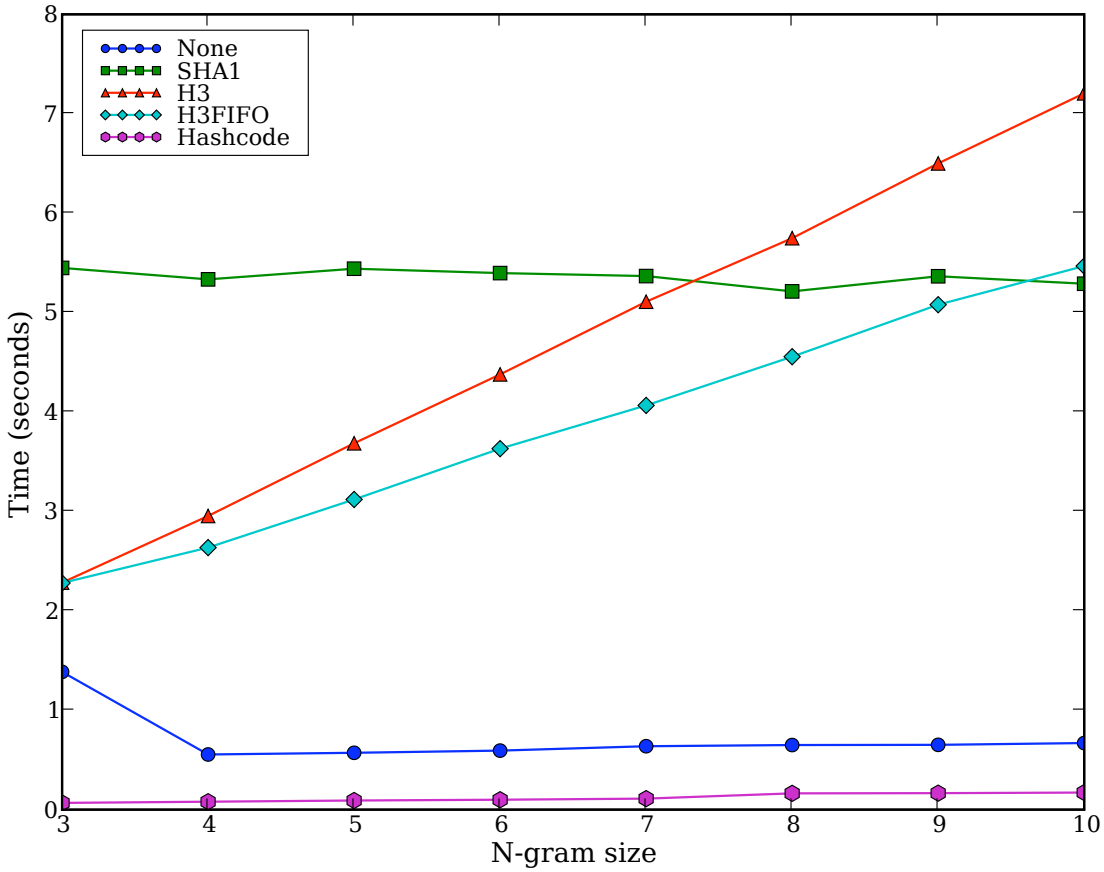

Figure 6.79: Performance Comparison of Hash Functions for N-grams.

cache. The cache's benefits become far more apparent when processing larger n-grams, such as 9- or 10-grams, where the performance improvement is over 20%. The hash cache's performance improves further when *multiple* hash functions are considered (as in a Bloom filter), or when the cache enjoys a higher hit-rate (e.g., when more homogeneous packets—such as those from one protocol—are processed). For instance, figures 6.80-6.82 show results when 7,000 *HTTP* packets are chosen from the same dataset, including both cache hit rates and effective performance. (Given the higher average size of HTTP packets, 7,000 packets yields roughly the same number of n-grams, ranging from 2.95 million to 3 million n-grams.) Given the greater homogeneity of packet content, it's unsurprising to see significantly higher hit rates—reaching into the 90% range for 3-grams through the 70% range for

10-grams. Figure 6.82 shows results when one H_3 instance is used as well as when three H_3 instances, i.e., three distinct hash functions, are used per n-gram. In the latter case, the performance speedup is close to 50% in our prototype system, and presumably can be optimized further given a optimized rewrite using a lower-level language, e.g., C.

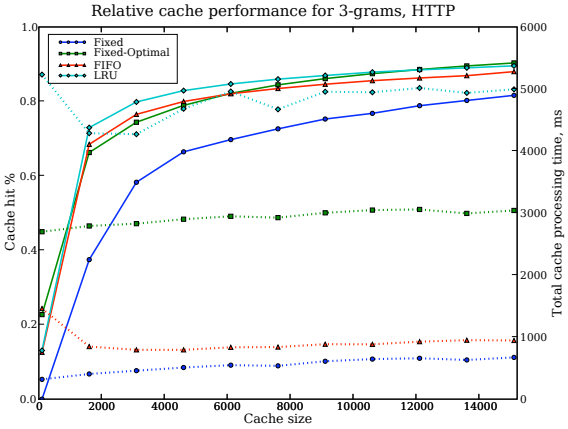

Figure 6.80: HTTP cache performance, 3-grams.

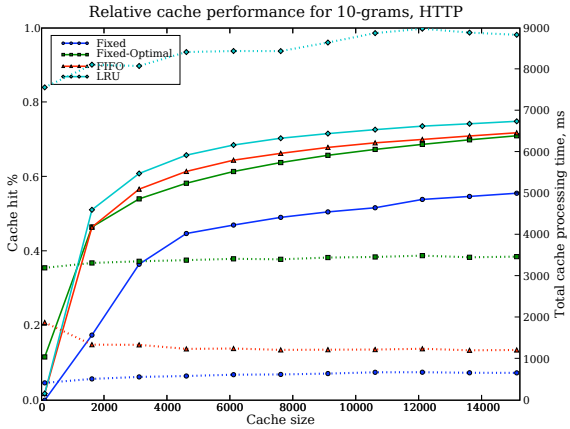

Figure 6.81: HTTP cache performance, 10-grams.

Given these numbers, what conclusions can be drawn? First, hash-based corroboration can indeed be done in real-time. The current prototype easily handles opaque IP-based and payload alerts at alert rates that match the fastest distribution systems. N-grams pose a significantly harder challenge, as many features are being extracted from every single alert (packet). Nevertheless, given the data in figures 6.74-6.82, certain classes of n-gram analysis can be done at wire speed. The analysis in figure 6.82, focusing on 3.2MB of HTTP packets, is capable of supporting approximately 10-20mbits/sec without further optimization.

Bloom Filters

For completeness, I include one more figure, corresponding to figure 6.82, that reflects Bloom filter performance when processing n-grams. 3 H_3 hash functions are used with a 2^{24}-entry BF. Both the results with and without a 15,000-entry FIFO

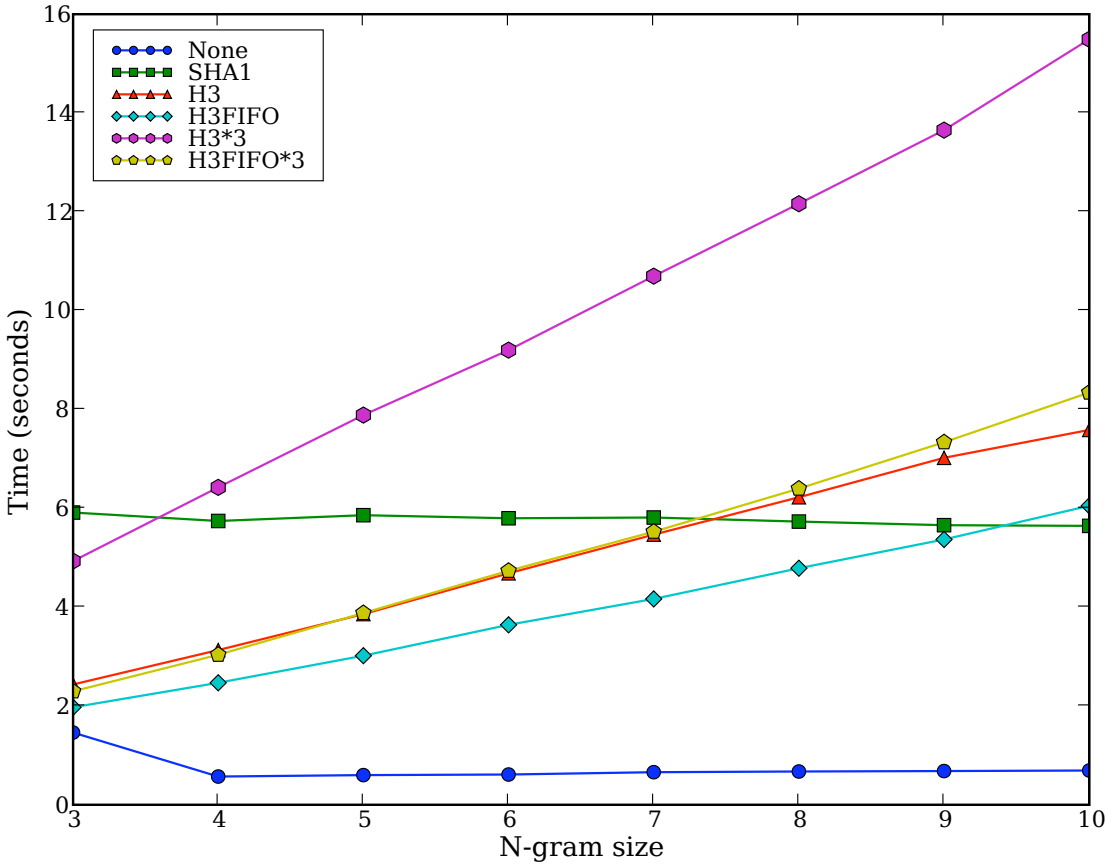

Figure 6.82: Performance Comparison of Hash Functions for N-grams given HTTP traffic.

cache are shown.

The results in figure 6.83 demonstrate that Bloom filters scale as well as, if not significantly better than, hashed n-grams themselves. While the cost of the non-cached Bloom filter hashing is significant, caching offers between a 44% (10-gram) and 52% (3-gram) reduction in computation overhead. Of the n-gram hashing techniques shown, the cached Bloom filter implementation is by far the best. We speculate [173] that an optimized C-based Bloom filter implementation can handle a sustained 100mbps network connection.[17]

Finally, Bloom filters may themselves be exchanged and directly compared

[17]Of course, an out-of-band network would be necessary to support the communication of such alerts.

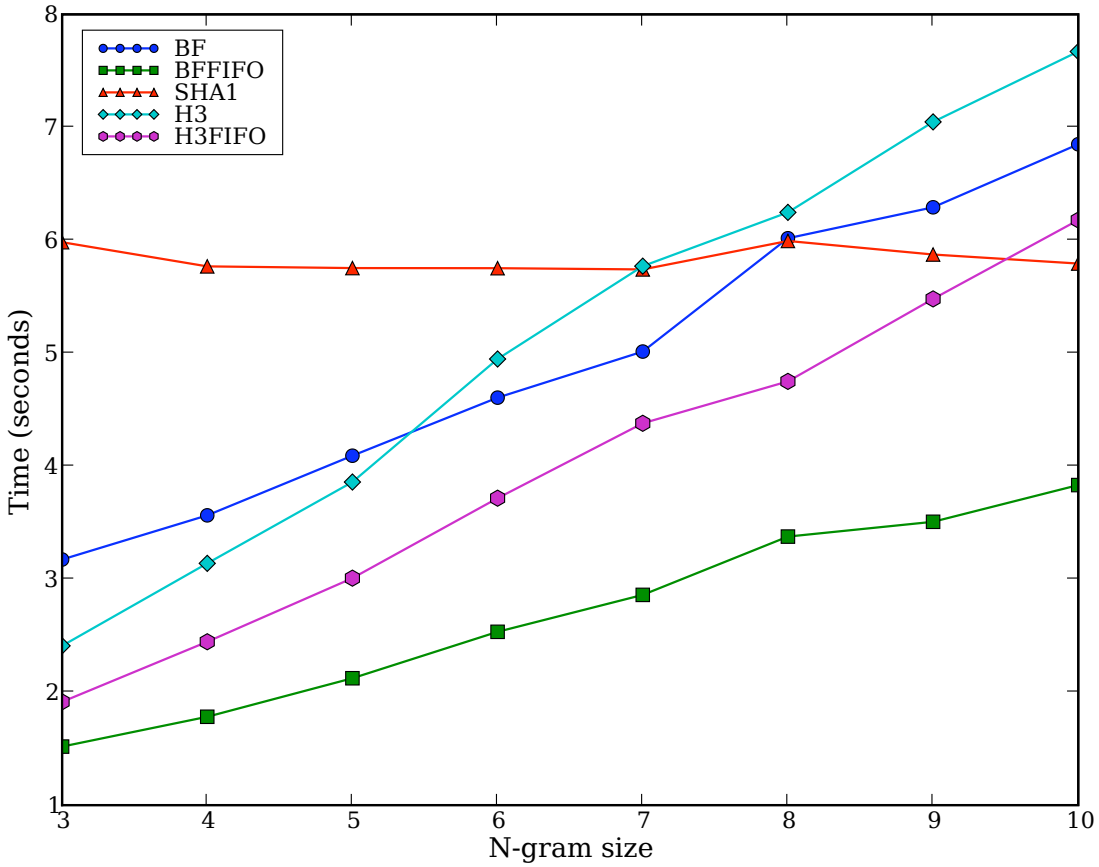

Figure 6.83: Performance Comparison of Bloom Filters and Hash Functions for N-grams given HTTP traffic.

amongst each other as models. While this takes more time, as Bloom filters can be significantly larger than any one single alert, the expectation is that this need not be done at the same rate as network traffic or IDS alerts.

6.5.4 Corroboration Accuracy

As previously discussed, we first measure accuracy by computing a set of *similarity scores* for every corroboration technique, $0 \leq score \leq 1$, with a higher score implying a more similar pair of alerts.

Raw packets and Z-Strings. For both of these alert types, our basket of string comparisons can be used. For SE, the score is binary: 0 or 1, where 1 means equality.

For LCS and LCSeq, we use the percentage of the common LCS or LCSeq length out of the total length of candidate strings: $score = 2 * C/(L_1 + L_2)$, where C is the length of LCS/LCSeq and L_i is the length of string i. For ED, larger values imply dissimilarity; we normalize it as $score = 1 - D/(L_1 + L_2)$, where D is the computed edit distance and L_i the same as LCS/LCSeq.

Frequency distributions. As mentioned before (equation 5.3), frequency distributions are compared using Manhattan distance: $M = \sum_{i=1}^{n} |x_i - y_i|$, $score = M/2$.

Raw and BF n-grams. Since we no longer have full packet content, we instead compute the percentage of common n-grams: $score = 2 * N_c/(N_1 + N_2)$, where N_c is the number of common n-grams and N_i the number of suspicious n-grams in alert i. If a Bloom filter is used, a count may be kept with it *or* approximated by N_b/N_h, i.e., the number of bits set divided by the number of hash functions used.

Testing With Real Traffic

To compare the approaches, we randomly sampled HTTP packets from three sources: clean packets collected from *www* and *www1* (two heavily-trafficked Columbia CS webservers), and malicious packets collected from a sample of attacks (CodeRed, CodeRed II, WebDAV, Mirela, a phpBB forum attack, and an IIS buffer overflow (MS03-022) exploit). These packets were paired off in three sets: 10,000 "good-vs-good" pairs from 100 packets of *www* and *www1* traffic each, 1,540 "bad-vs-bad" pairs formed in the cross-product of the 56 packet malicious dataset, and 5,600 "good-vs-bad" pairs of *www1* and malicious packets. Similarity scores were generated for all of the resulting pairs with all techniques, except SE, which is too brittle to produce meaningful comparisons, and the n-gram analyses, which cannot be compared over an entire packet.

Figure 6.84 visualizes a small random subset (80 pairs) of the scores generated from the "good-vs-good" source. As figure 6.84 shows, the performance plots of the

methods appear similar, although their centers and scale values differ as the scores are not normalized between the corroboration methods. On raw payloads, LCSeq and ED bear very similar results, while comparisons on Z-Strings yield "flatter" results, as less information is compared.

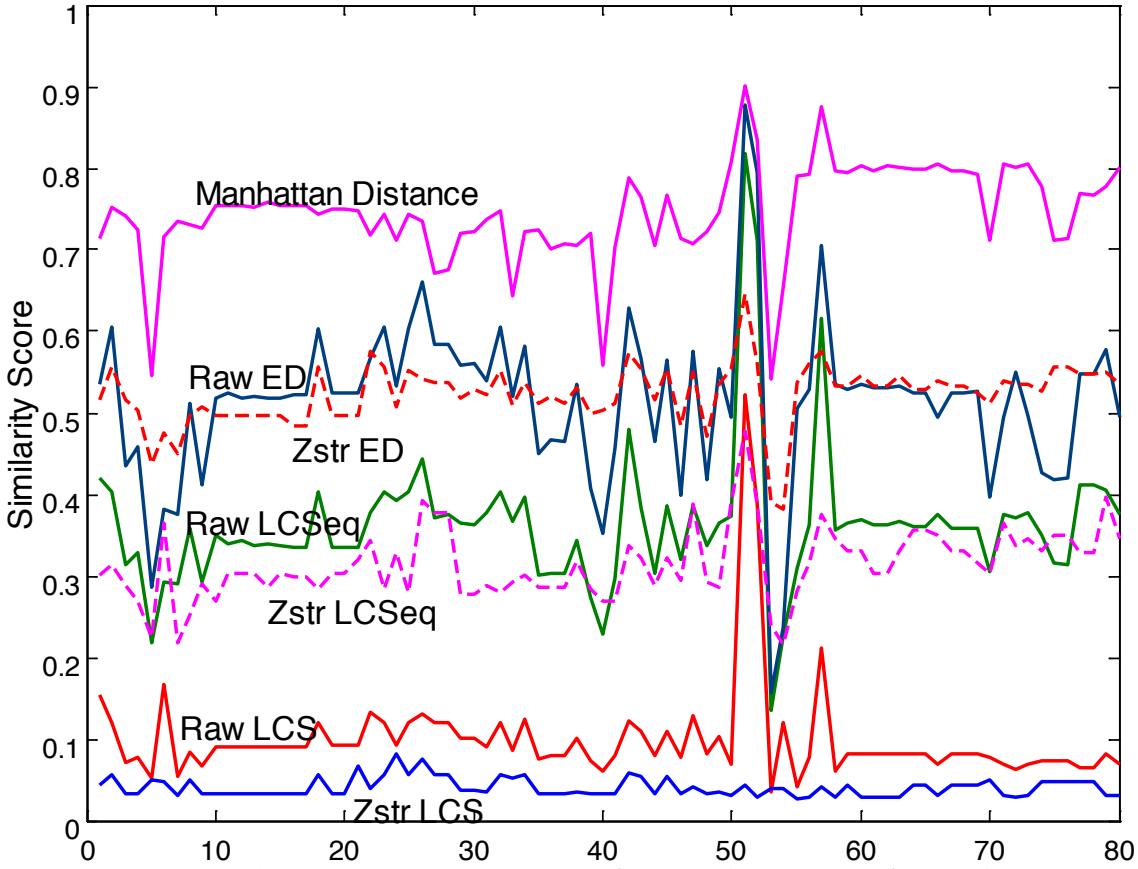

Figure 6.84: Similarity score comparison of 80 random pairs of "good-vs-good" alerts.

As a more complete experiment, normalized scores were generated and compared for all of the pairs formed amongst the three datasets. To normalize the scores for a comparison, we first compute *similarity score vectors* V_A, V_B for the same data over two techniques A and B. The center of the two vectors are then aligned by shifting the median of V_A to match V_B. Finally, V_A's range is scaled proportionally so that its min and max values match V_B's. This normalization allows us to compute the Manhattan Distance of the two vectors, $distance = \sum_{i=1}^{n} |V_{A_i} - V_{B_i}|$; smaller values

imply greater similarity between the two methods. Note that these scores are relative and dependent on the data used; the normalized results are only useful for comparing against a baseline, not as a source of absolute values or across datasets. These pairs were tested with each technique, and the resulting scores were normalized against and compared to the LCSeq score over raw packets. Table 6.32 shows the computed results.

Type	Raw-LCS	Raw-ED	MD	ZStr-LCS	ZStr-LCSeq	ZStr-ED
G-G	.0948	.0336	.0669	.2079	.0794	.0667
B-B	.0508	.0441	.0653	.0399	.0263	.0669
G-B	.0251	.0241	.0110	.0310	.0191	.0233

Table 6.32: Manhattan distance from Raw-LCSeq; lower is better.

Averaged over the three scores, Raw-ED is, unsurprisingly, closest to Raw-LCSeq. When privacy-preserving methods are considered, Manhattan distance performs the best overall, and particularly well for good-vs-bad comparison. All of the privacy-preserving methods are close when corroborating pairs with attack traffic; we conjecture that significantly different byte distributions enable effective comparison even when some information is lost via privacy-preservation.

Cross-Domain Alert Corroboration

Next, we compare the techniques by examining their actual performance in identifying true alerts from false positives. Ideally, all false alerts are eliminated by a small similarity score (i.e., the site that produced the alert was the only site that saw this suspicious packet) while true alerts are identified with high similarity scores (i.e., the attack has been launched against more than one site).

In this experiment, we first randomly mix the aforementioned collection of attacks into two hours' traffic from *www* and *www1*, respectively. Multiple instances of attacks—4 for CodeRed and 3 for CodeRed II—are present to simulate a real-world

worm attack. The attacks are also fragmented differently, as CodeRed does in the wild; for instance, CodeRed may fragment into a sequence of (1448, 1448, 1143) length packets, (4, 375, 1460, 1460, 740) length packets, etc. Multiple instances also enable testing corroboration *between* different attack types (e.g., CodeRed vs. CodeRed II).

Next, the two mixed traffic sets are each run through PAYL and Anagram with previously-built models and with the alerting threshold lowered so that 100% of the attacks are detected, but with higher (and comparable) false positive rates. The resulting alert sets are corroborated against each other using each of the techniques; the results are summarized in figure 6.85. For each method, the stacked bar represents corroboration results for *false positives*. The shaded portion of the bar represents the 99.9% percentile similarity score range, while the white represents the worst-case (highest) score; in other words, while the worst-case FP score can be high, the vast majority of false positives score relatively low.

The asterisk-marked ("*") lines represent the range of similarity scores when instances of the same worm are corroborated, and the open circle-marked ("o") lines represent scores across CodeRed and CodeRed II—a very simple measure of polymorphism. The other worms, which were inserted without fragmentation, all scored at or near 1, and so are not shown.[18]

We can draw several conclusions. First, corroboration of identical (non-polymorphic) attacks works perfectly and accurately for all techniques. Most of the techniques can also corroborate multiple instances of fragmented attacks; of the privacy-preserving techniques, MD, LCSeq and ED on Z-Strings, and n-gram analysis[19] all perform well. (As intuition may suggest, ZStr-LCS is not particularly effective.) Polymorphic worm detection is far harder—even in the case of CR vs. CRII, only Raw-LCSeq and n-grams achieve promising results. N-gram analysis, in

[18]We could have artificially fragmented these worms to simulate the CodeRed experiment, but we expect similar results.

[19]We do not distinguish between published raw n-grams and published BF-based n-grams here, as they produce virtually identical results.

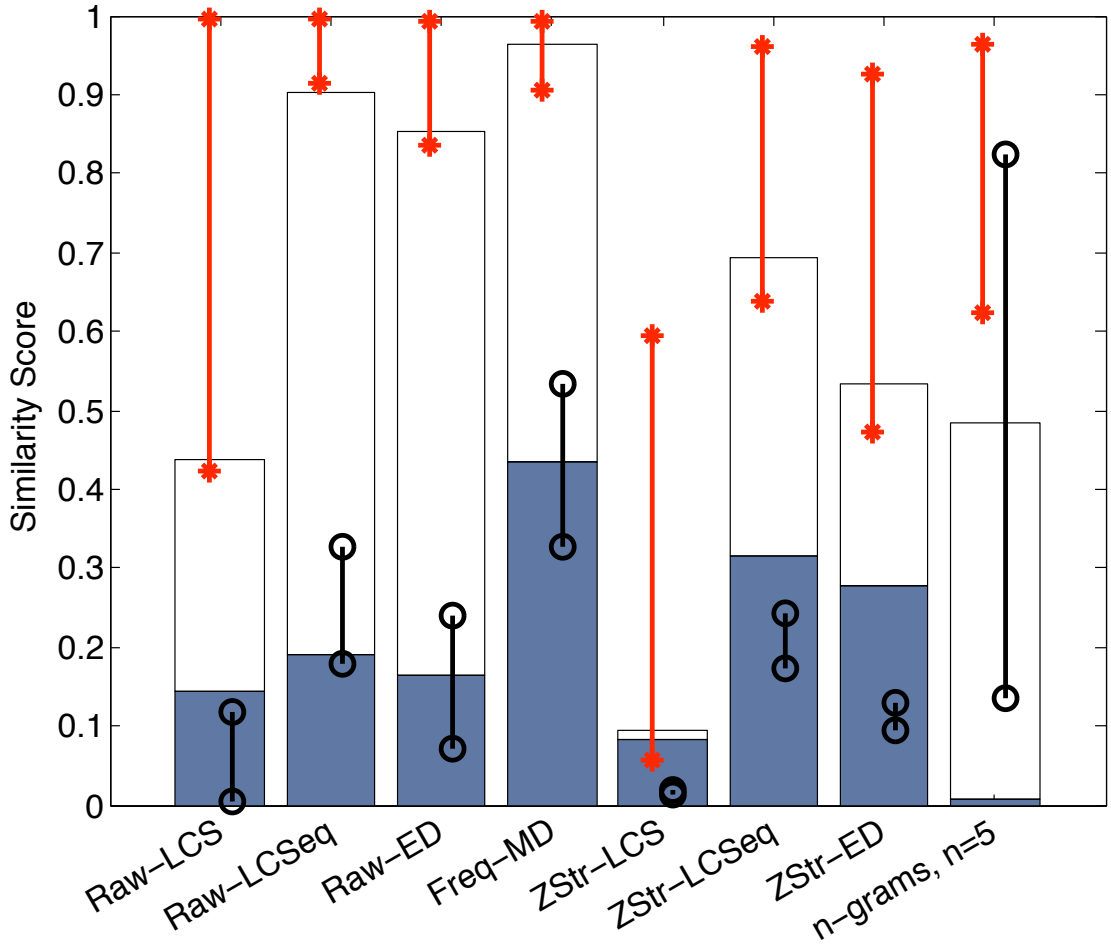

Figure 6.85: Corroboration methods comparison.

particular, stands out; it produces accurate results and is particularly effective at eliminating false positives, and the use of BFs enables privacy-preservation.

Signature Generation

Corroborating alerts across sites also enables the possibility of automatic signature generation and deployment, once true alerts are identified. (We can also potentially use the scores computed during similarity comparison as a "confidence" measure in mitigation strategies to determine whether to deploy a signature.)

Raw packet-based signatures. Given the ability to share raw alerts, we can exchange the LCS or LCSeq of highly similar packets. This has been the subject of

much recent work (§4.5), is not privacy-preserving, and we do not discuss it further here.

Byte frequency/Z-Strings. Given the first packet of a CodeRed II attack in figure 6.86 and its byte distribution displayed in figure 6.87, we can generate a Z-String by ordering the distribution by most frequent to least and dropping frequency information. Figure 6.88 shows the first 20 bytes of the generated Z-String for the distribution in figure 6.87, with nonprintable characters shown by their Unicode values. Both frequency distributions and Z-Strings can be used as signatures.

```
GET./default.ida?XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX%u9090%u6858%ucbd3%u7801%u9090%u6858%u
cbd3%u7801%u9090%u6858%ucbd3%u7801%u9090%u9090%u8190%u00c3%u0003%u8b
00%u531b%u53ff%u0078%u0000%u0
```

Figure 6.86: Raw packet of CRII; only the first 301 bytes are shown for brevity.

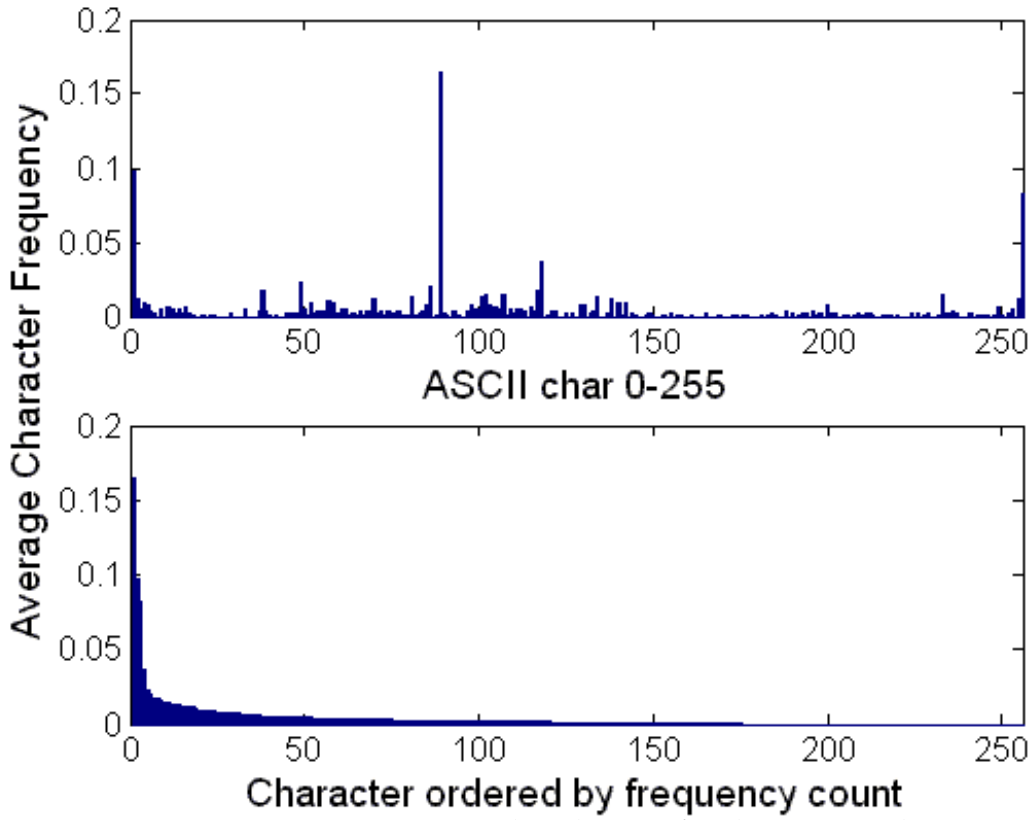

Figure 6.87: Frequency distribution for the CRII packet

```
88 0 255 117 48 85 116 37 232 100 100 106 69 133 137 80 254 1 56 51
```

Figure 6.88: First 20 bytes of the Z-String computed from the CRII packet.

N-Grams. N-grams are an intriguing approach to signature generation; n-grams are position-independent, making them robust to reordering and fragmentation. Additionally, if position information is kept, such a collection *can* be transformed into a flat signature if desired. Figure 6.89 shows the results when a collection of 5-grams based on the CodeRed II example packet are "flattened", with "*" representing a wildcard for signature matching. Compared to the original, figure 6.89 successfully captures the malicious encoding and deemphasizes the padding "noise". Results with different n-gram sizes and another CRII packet are presented in an appendix in [115].

```
* /def*ult.ida?XXXX*XXXX%u9090%u6858%ucbd3%u7801%u9090%u6858%ucbd3%u
7801%u9090%u6858%ucbd3%u7801%u9090%u9090%u8190%u00c3%u0003%u8b00%u53
1b%u53ff%u0078%u0000%u00=a HT*: 3379
```

Figure 6.89: Generated 5-gram signature from the CRII packet; only the first 172 bytes are shown for brevity.

6.5.5 Temporal Corroboration: Z-String Clustering

Section 6.4.6 discussed the performance of MRU BFs and TSBFs, and the same analysis applies here for n-gram Bloom filters. However, temporal corroboration with frequency distributions is far more complex. Unlike Bloom filters, which can trivially be ANDed, frequency distributions cannot be merged without losing data. The alternative, a linear scan, can become very expensive if every exchanged, non-expired frequency distribution must be examined.

Alternatively, we can *partition* the set of frequency transforms into a number of disjoint clusters based on transform similarity. While this will not completely remove the linear-time scanning overhead, given a significant number of partitions,

we can still significantly reduce the work required. In order for this approach to work, however, it must fulfill the following characteristics:

1. The partitioning of transforms into clusters must be fast;
2. The number of clusters should be small enough so that they are not specialized around individual frequency distributions, but not so small that the degenerate case, where a linear scan of a large partition is of the same order-of-magnitude as scanning all of the partitions;
3. The overlap between clusters should be minimal to ensure that incoming models are correctly thrown into the right clusters;
4. The number of clusters generated for a particular type of datum should be minimized, so as to ensure that incoming models thrown into such clusters will be rapidly corroborated.

While many clustering algorithms exist, we propose a simple, fast clustering technique—the use of Z-Strings. A Z-String is an approximate representation of a frequency distribution, and given the first few characters of a Z-String, we can characterize the most prevalent aspects of a frequency distribution with a minimum of computation effort. As shown in section 6.5.3, Z-String computation is fast, and Z-Strings can be used as hash values to reduce the time of partition to the cost of computing the Z-String.

Cluster Key Determination

The key question to this approach's efficacy is effectively determining the number of characters of a Z-String that should be used as a cluster key. If too many characters are used, too many sparse clusters will be generated; if too few are used, a degenerate situation will occur. Instead of empirically setting the string value,

we compute optimal cluster keys by thresholding the frequency distribution. Two simple thresholds are a *minimum threshold*, i.e., eliminate all Z-String characters whose frequency is less than an *a priori* threshold; and a *delta threshold*, i.e., eliminate all Z-String characters after the "first derivative" frequency deltas are smaller than a similar threshold. These two approaches are called the *cluster* and *cluster delta* approach, respectively, in the experiments detailed here.

Clustering Performance

Given the desiderata above, two sets of data were evaluated to see how the techniques perform in partitioning data into clusters. The first was 100,000 randomly-sampled packets of *www*; the second was 100,000 packets from a Code Red II trace. Figures 6.90–6.102 show the results as these two traces are used to evaluate partitioning performance.

First, figures 6.90–6.93 are semilog plots of clustering scores for all three techniques—raw Z-Strings, Clusters, and Clusters Delta—at two different thresholds, 1 and 4 (i.e., restricting string length based on thresholding at 1 and 4, respectively), and against a varying number of packets (ranging from 25,000–100,000).

The difference between the cardinality of raw Z-Strings, Clusters, and Clusters Delta is remarkable. As the Y-axis is semilog, each tick represents an order-of-magnitude difference. By this measure, Clusters Delta produce 1–2 orders of magnitude less clusters than raw Z-Strings or regularly-thresholded Clusters. While increasing the threshold to 4 decreases the difference between Clusters and Clusters Delta, there is still at least one order of magnitude between the two. A homogeneous trace, unsurprisingly, produces fewer clusters; in the case of Cluster Delta and the CRII trace, the algorithm quickly computes around 100 clusters, even in the case of 100 packets. This strongly suggests that the Cluster Delta approach is producing meaningful results.

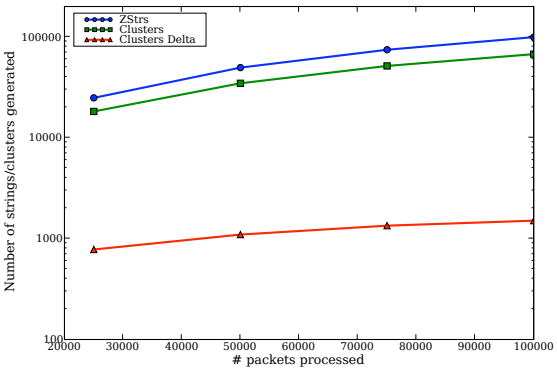

Figure 6.90: Z-String clustering on www, threshold 1.

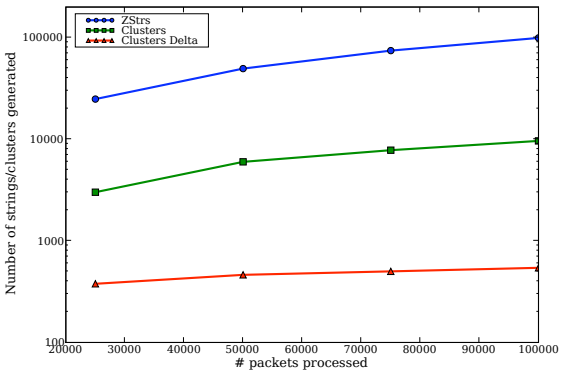

Figure 6.91: Z-String clustering on www, threshold 4.

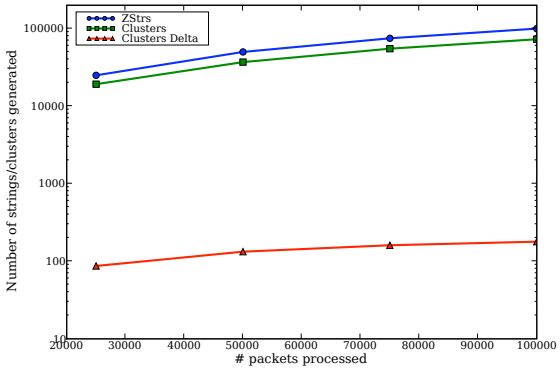

Figure 6.92: Z-String clustering for CRII, threshold 1.

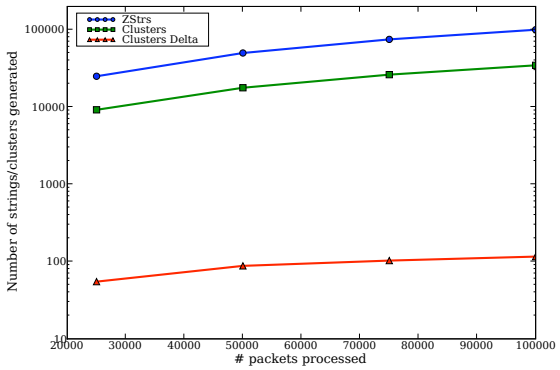

Figure 6.93: Z-String clustering for CRII, threshold 4.

In order to get a better idea of thresholds' effects, figures 6.94–6.97 focus on Cluster and Cluster Delta for varying thresholds and packets.

Once again, Cluster Delta fares far better, especially when CRII is concerned. It's also unsurprising that the CRII clustering algorithm converges faster, and higher thresholds have less of an effect on the number of clusters generated.

One other interesting characteristic worth exploring is the length of the cluster strings generated, as well as the average number of packets per cluster, as shown in figure 6.98 and 6.99; the dashed lines represent the second Y-axis (number of items) as opposed to the first Y-axis (Z-string length).

The results are as expected, although with one twist: in the CRII trace, the average Z-String length actually *decreases* slightly with additional traffic. This may be a sampling coincidence, i.e., the first 25,000 packets tended towards slightly

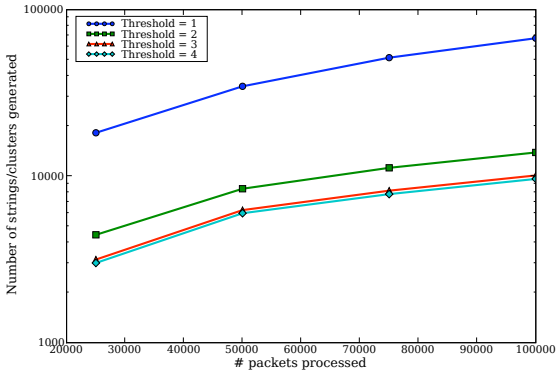

Figure 6.94: Z-String clustering via Cluster on www.

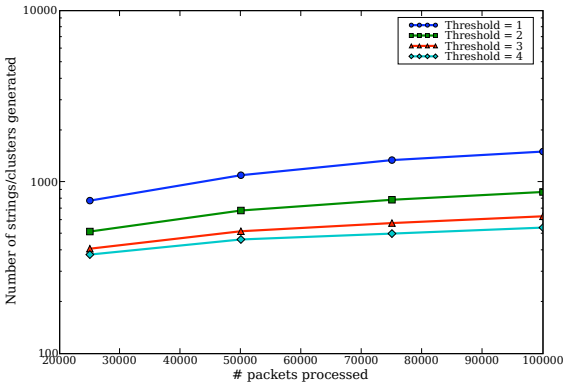

Figure 6.95: Z-String clustering via Cluster Delta on www.

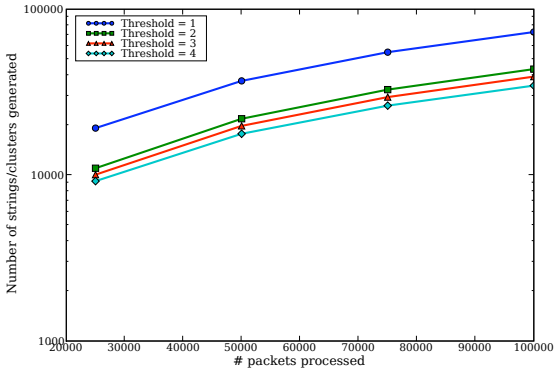

Figure 6.96: Z-String clustering via Cluster for CRII.

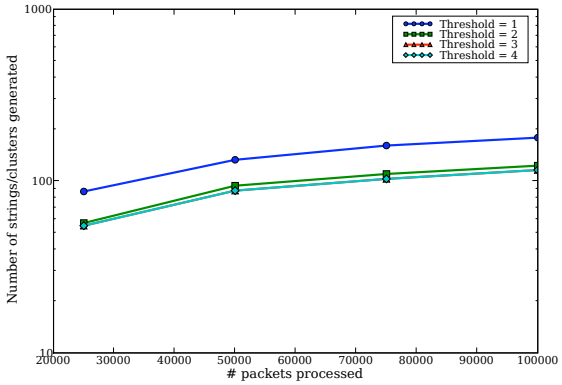

Figure 6.97: Z-String clustering via Cluster Delta for CRII.

longer Z-Strings, which then quickly corrected itself thereafter.

Now that Cluster Delta's ability to generate clusters has been validated, we would like to evaluate the actual distribution and relevance of the generated clusters. We can do this via two means; first is the use of a distance metric to ensure each cluster is cohesive (packets in a cluster are highly similar) and that clusters are only loosely coupled (packets between clusters are not similar). Second, we can actually test for the prevalence of worm bodies in clusters, with the optimal scenario being that a minimum of clusters contain a particular segment of the worm.

Figure 6.100 tests the former with varying thresholds. The distance was computed by randomly sampling up to 100 packets from each cluster, generating their normalized frequency distributions, and computing distances pairwise, i.e., all 100 distributions were tested against each other in the current cluster *and* each of

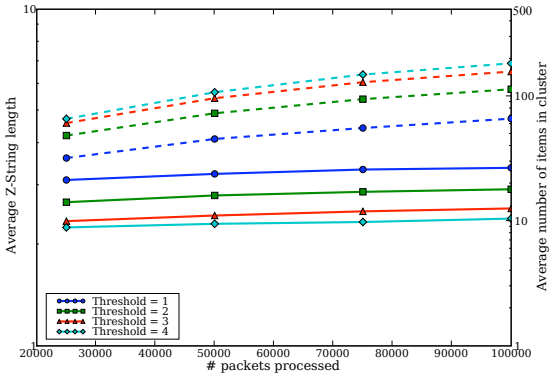

Figure 6.98: Z-String Cluster Delta lengths on www.

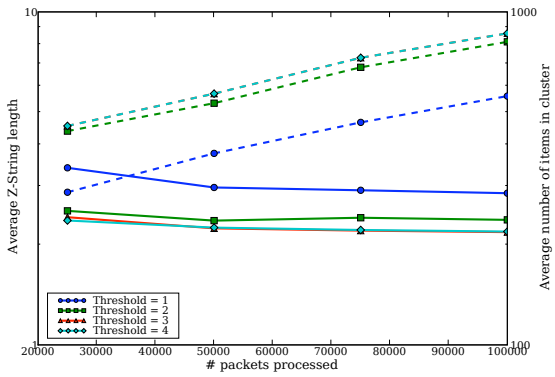

Figure 6.99: Z-String Cluster Delta lengths for CRII.

those packets tested against the other samples in every other cluster. The resulting Manhattan distances were averaged into each category, and are shown below. The dashed line just plots the numbers of clusters generated, as per the second Y-axis.

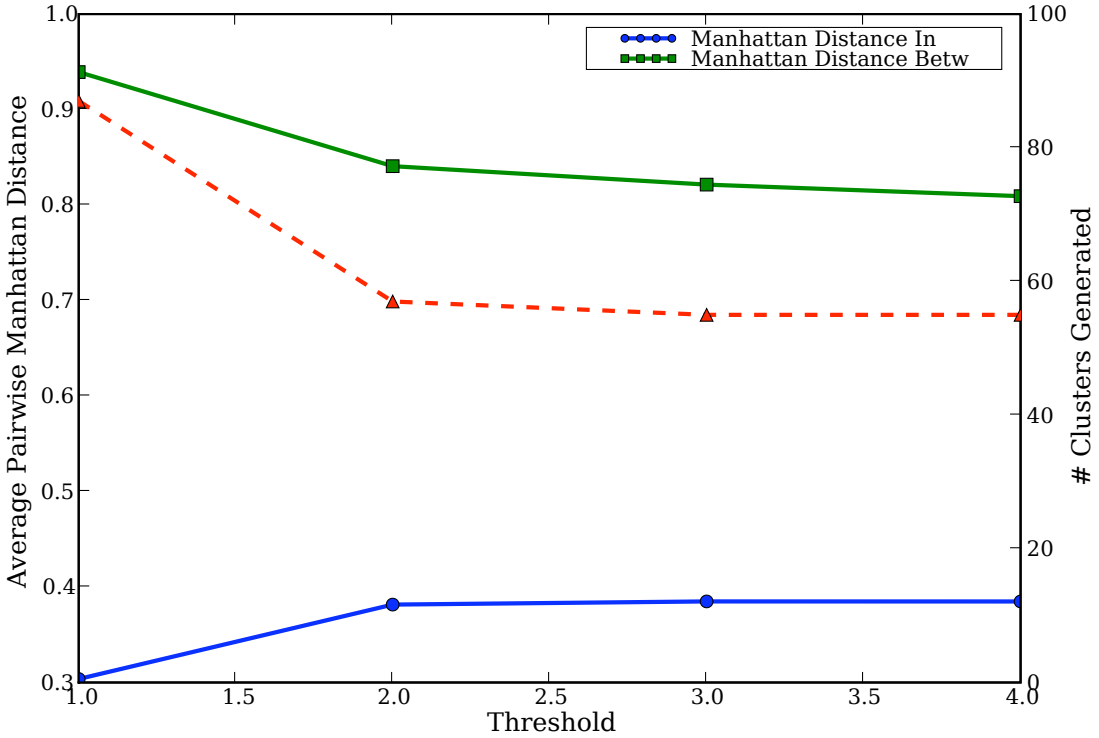

Figure 6.100: Z-String Cluster Delta, Manhattan distance results.

The results are solid; there is a clear delineation between distributions within a cluster and those between clusters. Unsurprisingly, the differences diminish as the threshold increases, as fewer clusters imply that slightly more dissimilar distributions

are shoehorned into other clusters. Nevertheless, the average Manhattan distance within clusters remains below .4, while the average Manhattan distance between clusters stays above .8.

Given the good results for all thresholds distance-wise, is there a preferred choice? While fewer clusters may cause slightly dissimilar distribution clusters to be merged, intuition suggests that a reduced number of clusters increases the likelihood that received frequency distributions are pigeonholed into the correct cluster. We can test this via examining the *prevalence* of a unique packet (or substring thereof). In the CRII case, the simplest example is a portion of the URL that triggers the CRII buffer overflow (e.g., `default.ida?XXXXX`), and indeed, the results in figures 6.101–6.102 produce high prevalence and validates our intuition. Dashed lines represent the average number of distributions in a cluster that match our specified search string.

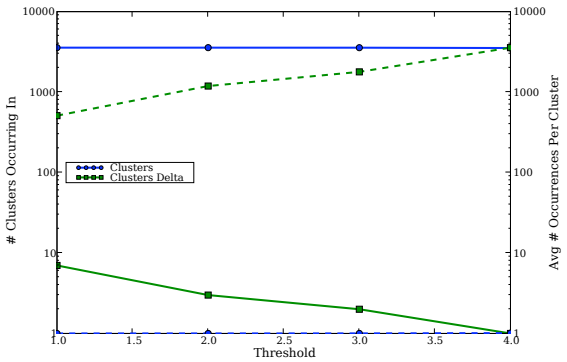

Figure 6.101: Z-String CRII prevalence, 25,000 packets.

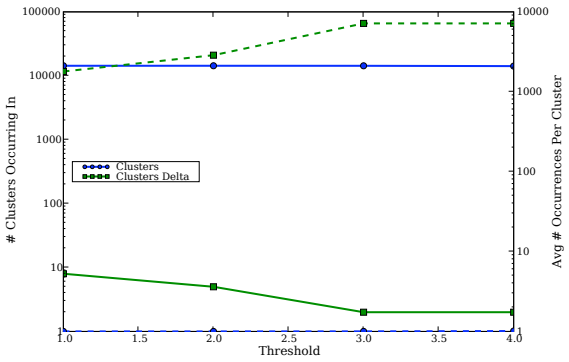

Figure 6.102: Z-String CRII prevalence, 100,000 packets.

These results, in fact, serve as the strongest indicator that the clustering technique works. When 25,000 packets are processed, Cluster Delta successfully converges all of the CRII URL overflow distributions into a single cluster. The algorithm is not fully successful with more packets, but still sorts the distributions down into 4 clusters. Additional heuristics, such as aggressively merging 1-clusters, may help in reducing the number of clusters for a distribution of interest. Still, 4 clusters is

sufficient to ensure enough alerts are pigeonholed into each such that corroboration can be done quickly as new distributions are received.

6.5.6 Privacy Gain

As is shown here, the analysis done in section 6.4.7 is not practical for payloads; the cost of doing a brute-force analysis is clearly intractable with standard computation hardware and techniques. Instead, as mentioned earlier, we can use a probabilistic model as a first-order approximation to measure the relative privacy and effectiveness of each privacy-preserving technique.

Frequency-Based Approaches

Recovery of the original text from its privacy-preserving encoding can be modeled as follows: given a frequency distribution $f = \{(b_0, \pi_0), (b_1, \pi_1), ..., (b_{n-1}, \pi_{n-1})\}$, where n is the size of the alphabet, b_i is the byte value with probability π_i, and a target length l, we construct the content of a packet $p = \{b_0 b_0 ... b_0 b_1 b_1 ... b_1 \cdots b_{n-1} b_{n-1} ... b_{n-1}\}$, with $\pi_0 l$ copies of b_0, $\pi_1 l$ copies of b_1, and so on.

We can now characterize the *recovery likelihood* $R = 1/(l!/((\pi 0 l)!(\pi 1 l)!...(\pi n - 1 l)!))$, where the denominator is simply the count of all permutations of p. This is an effective estimate of the privacy of the frequency distribution, as it represents the likelihood an attacker will be able to correctly guess the true content of the original packets. This number is, additionally, vanishingly small. For the frequency distribution of the CRII packet shown in figure 6.87, $R \approx 1/2^{8208}$, well beyond the scope of feasibility, despite the packet's regularity thanks to the large padded section.

The value of R for a Z-String is orders-of-magnitude smaller; not only do permutations of a packet p_i have to be computed, there are many such packets; since no frequency information is stored, one must guess the frequencies for each of the

bytes b_i in the packet. In short, effectively guessing the correct base packet p_i *and* its correct permutation is intractable.

N-Grams

We can consider the privacy of both a raw collection of n-grams and a corresponding Bloom filter encoding. The raw collection is *not* very privacy-preserving; not only can a byte sequence contain valuable information, significant n-gram collections enable reassembly of much of the original packet, even without position information. Given a 5-gram $\{b_1b_2b_3b_4b_5\}$ from packet p, one can search for the 5-gram $\{b_2b_3b_4b_5b_x\}$ in the collection, where b_x is any byte; if found, one can reasonably assume the presence of the 6-gram $\{b_1...b_5b_x\}$ in packet p; since 256^4, the number of "common" 4-grams contained in both two 5-grams, is much greater than a packet's size, it is highly unlikely that the common 4-gram happens to be a coincidence. Combined with the fact that high-scoring alerts can contain nearly as many n-grams as the original packet's size, this is impractical from a privacy perspective.

Instead, as previously proposed, we insert the collection of n-grams into a Bloom filter before publishing it. The size of the Bloom filter need not be much more than the number of n-grams; we can pick a size, say 2^{12} bits, which is more than twice the size of any individual packet and not prone to significant false positives. (This Bloom filter still takes substantially less memory than the n-gram collection itself.) Given such an encoding, the only practical way of recovering the data is to brute-force verify every possible n-gram against the Bloom filter. For example, if we know that only 5-grams are contained in the Bloom filter, there are 256^5 possible n-grams. Not only is testing all such n-grams computationally infeasible, a brute-force attempt is likely to generate many, many false positives, since there are $256^5/2^{12}$ possible n-grams for each set bit in the Bloom filter (assuming one

hash function[20]); the recovery likelihood $R = (2^{12}/256^5)^m$, where m is the number of n-grams recovered, is again vanishingly small. This number grows even smaller if multiple n-gram sizes are embedded in the same Bloom filter.

Interestingly, despite the number of possible n-grams for each bit of the Bloom filter, corroborating such filters is *not* prone to significant misclassification. We can characterize the "unlucky coincidence" rate = $(\frac{256^5/2^{12}}{256^5})^m$, that is, the likelihood that we happen to incorrectly verify m possible n-grams, each represented by a particular bit b_i, out of all possible n-grams. This simplifies to $(\frac{1}{2^{12}})^m$, which rapidly grows smaller with increasing m. In our experiments, we found that a similarity score threshold of 0.1 produced good results; combined with the fact that the average number of n-grams in a false positive alert is approximately 55, the probability of miscorroborating a Bloom filter alert due to *5* unlucky coincidences is $(\frac{1}{2^{12}})^5$—not a major concern. In short, testing multiple n-grams eliminates coincidences very rapidly. Sizing the Bloom filter appropriately to avoid saturation is a far more important issue.

Given the effectiveness of n-gram analysis, combined with its strong privacy guarantees and compact size, we believe there is great promise for this form of payload-based corroboration.

6.5.7 Conclusion

Content-based alert sharing holds great promise in being able to detect and track attack behaviors that are difficult to monitor via pure IP-based intrusion detection. In particular, content-based alerts generated by locally-trained payload anomaly detectors reveals an opportunity to detect the early onset of zero-day worm or targeted attacks.

We presented a comparative evaluation of alternative corroboration strategies

[20] Additional hash functions do not affect our analysis.

and accuracy measures using test data sets with known worm exploits, and included a proposed estimate of the "privacy gain" each method affords. This is important in approaching the problem analytically in order to help break down barriers to collaboration. We find that cross-site and cross-domain privacy-preserving "suspect payload" alert sharing is feasible and useful, particularly shown in the analysis of Bloom filter-exchanged alerts encoding suspect anomalous n-grams.

6.6 Model-Driven Collaboration

The previous sections in this chapter have focused on the exchange of individual *alerts*, or combinations thereof. While this is useful for purely alert-driven intrusion detection, there are a number of specialized applications where this is an insufficient approach—for example, environments where new communication partners are constantly encountered, for which waiting for intrusion alerts or making meaningful sense of them in a short time period is infeasible.

One such application is that of mobile ad-hoc networks (MANETs), which have recently gained adoption in a broad variety of environments thanks to improvements in wireless networking technology and the need for rapid mobile deployment. However, MANETs pose unique security requirements and challenges. Since they enable devices to enter and leave a network without previous authentication or certification, a MANET node cannot be assumed to be trusted. Traditional security approaches, like firewalls, do not extend well to MANETs, where both benign and malicious parties have full access to communicate with peers. Additionally, the limited processing power and battery life of these devices also prevent the adoption of heavy-duty cryptographic techniques.

While traditional misuse-based IDSes may work in a MANET, the efficacy of these techniques is in question. A traditional IDS might watch for packet dropouts

or unknown outsiders as a sign that an inbound communication may be malicious. In a MANET however, both of these occurrences are commonplace amongst benign nodes as well. Additionally, most MANET-based misuse detectors have focused on routing-specific attacks, e.g. [148], at the cost of ignoring actual application vulnerabilities.

Anomaly detection approaches hold out more promise, as they utilize learning techniques to enable adaptation to the wireless environment and to the tasks and communications being utilized in that environment. Anomaly detectors generate a model of the observed data (traffic, behavior, etc.), and compare new data against this model to check for anomalies. It is relatively simple to determine if peer communications fit that model, and to establish policies ignoring data that is flagged as malicious (e.g., [174, 172]). The model also acts as a profile of device behavior, which can be utilized by peers to help determine its trustworthiness by comparing their mutual models exchanged between the devices.

This concept extends the notion of mutual authentication; rather than proving one's trustworthiness via a certificate or a credential; here, MANET nodes are authenticated by their *behavior*—a profile of how they typically interact. Other nodes may validate the node by conformance to their own profiles, and to ensure the new node subsequently behaves in conformance with their announced profile.

Early work in building anomaly detectors for MANETs was primarily focused on header-level and routing-level features [64, 117, 142, 164]. Computing the anomaly model for traffic payload or other rich feature sets is a time-consuming and processor-heavy task, one that needs to be avoided in a battery-conscious, reduced-communication environment. To solve this, we proposed the use of *model exchange* in a MANET to provide a balance between the need for adaptation as a device moves between different networks and the need to minimize computation and traffic utilization. Any node should be able to obtain peers' model(s) and

evaluate it against its own model of "normal" behavior. The node should be able to either integrate the peer's model with its own to get a better idea of legitimate traffic being conducted on the network, or to flag the peer as suspicious if the profile is significantly different than its own. Moreover, it should be able to do this form of exchange without revealing any data deemed private—that is, we wish to separate the notion of *behavior* from *data*. Finally, in order to ensure the peer is not lying when it presents its model, the architecture allows for *continuous validation* of the model—to ensure that subsequent received traffic matches the presented model.

We present this model, discuss scenarios in which it may be used, provide early results about model integration and comparison, and provide a framework for future implementation. While worms and similar malicious payloads have not yet become prevalent on MANETs, it's only a matter of time before such intrusion detection techniques are necessary [24]. Note that this section of this chapter represents early feasibility work, and does *not* aim to be a complete dissertation on the subject. There are significant opportunities for future work, including an actual model exchange implementation and deployment based on the Worminator framework. These are discussed in section 7.3 and an implementation is underway.

6.6.1 Corroboration Model

Once again, we have a set of participants $A, B, C, \ldots$; each may correspond to an entity of varying size, ranging from a standard intrusion detection node to a small MANET entity, and the same multisensor-per-participant analysis discussed could also be applied here. Each contain anomaly sensors $\mathcal{I}_A, \mathcal{I}_B, \mathcal{I}_C, \ldots$ that produce *models* $\mathcal{M}_A, \mathcal{M}_B, \mathcal{M}_C, \ldots$. These models may evolve over time, so we may choose to express instances of these models as a function of time, i.e., $\mathcal{M}_{At_\alpha}, \mathcal{M}_{Bt_\beta}, \mathcal{M}_{Ct_\gamma}, \ldots$. The sensors may also generate events $\mathcal{E}_A, \ldots$, but we ignore them here, choosing to focus exclusively on the anomaly models themselves.

As discussed earlier in this thesis, models can be generated in one of several ways:

- 1-gram frequency distributions compiled over aggregate traffic, i.e., $\mathcal{A}' = \mathcal{M}_F(\rho(A)_{t_0-t_1})$, where $\mathcal{M}_F$ is the frequency transform over payloads ρ between time t_0 and t_1, i.e., $\mathcal{M}_F(\rho(A)_{t_0-t_1}) = \mathcal{M}_F(b_{A_0}, b_{A_1}, \ldots, b_{A_k})$, $b_{A_i} \in \rho(A)$ and $t(b_{A_0}) \geq t_0, t(b_{A_k}) \leq t_1$. The anomaly sensor PAYL is particularly designed for this task; PAYL can compute *multi-centroid* models conditioned on length, i.e., $\mathcal{M}_A = \{c_0, c_1, \ldots, c_n\}, c_i = \mathcal{M}_F(\rho(A)_0, \rho(A)_1, \ldots, \rho(A)_k)$ and $|\rho(A)_j| = |\rho(A)_k|$. A more complete discussion of PAYL can be found in section 6.3.2.

- Z-Strings, i.e., reduced frequency-modeled 1-gram alerts. These are not explored, because they are far too coarse to effectively model large amounts of content (as opposed to the previous section, where Z-Strings were used for more limited individual payloads).

- Binary n-gram models, possibly inserted in Bloom filters. Much like how frequency models can be generated by PAYL, binary n-gram models can be generated by Anagram and compared. This approach is left for future work.

Once the models are exchanged, we can choose to either *aggregate* or *differentiate* them, depending on the similarity of individual centroids. Both techniques are discussed in detail later in this section.

6.6.2 Practical Model Distribution

In a MANET, we make the fundamental assumption that most nodes cannot (or prefer not to) compute an anomaly model for payloads, due to the lack of traffic, battery power, or computation ability. This requires the existence of a node that is sufficiently powerful to perform anomaly model learning and can bootstrap the

MANET's model set. Depending on the location of this node, several different distribution models can be adopted:

- Use a server/desktop entity to generate the anomaly model. This is ideal for situations where the MANET is running a replica or a lightweight version of the desktop application (e.g., SMTP messaging or HTTP data transfer). In these cases, training can be done on the desktop and the model distributed to the MANET nodes when possible:
  - If the MANET nodes have WAN connectivity, they can initiate download requests to obtain the latest model from the server. (Some WAN topologies now allow for "push" models, which could be leveraged to let the desktop administer the download interval.) A hierarchical distribution can also be accomplished, whereby a single MANET node downloads the data over a potentially expensive WAN link and then utilizes the WLAN links to distribute the updated model to neighboring nodes.
  - Without WAN connectivity, MANET nodes can be initialized before deployment. This is a natural arrangement for "syncable" handheld devices (e.g., Palm/WinCE PDAs), which often have a cradle at the office/base and allow one-touch synchronization. We call this mechanism pre-charging. Ideally, the handheld device would contact the desktop at a regular basis, but high-quality models can reduce this dependency. Synchronization can also be accomplished with intermittent network links.

- If a desktop cannot be deployed, a more powerful MANET node can be deployed, with sufficient processing and/or battery power to perform anomaly training. This "supernode" would listen promiscuously to all visible traffic on

the MANET, generate models, and distribute them to the (potentially weaker) peers. This model is decentralized and does not require WAN connectivity. However, the supernode does not see traffic that is not routed within its vicinity. A workaround to enable broader model coverage would entail periodic traffic reports from all nodes; these traffic samples should be sufficient to construct a representative model.

- Use a precomputed anomaly model. This scenario is worst-case, but can be practical in situations where the MANET's behavior is well-defined and follows a standard protocol definition. This is a variation on the first scenario, but one where the regular synchronization requirement is dropped.

- Introduce node(s) from a different MANET who has been able to compute an anomaly model. Much like the previous scenario, this works best when MANET functionality is well-defined and compatible with the other MANET.

Degraded modes can also be adopted, i.e., in scenarios where anomaly models are unavailable, mobile nodes can adopt a "defensive" posture and reject otherwise accepted traffic. While model exchange imposes an additional restriction as opposed to standalone misuse or anomaly detectors, we believe that the savings in computation time and the benefits justify these requirements.

6.6.3 Case study: PAYL models

To verify our hypothesis, we examine the use of model exchange with PAYL. PAYL has favorable characteristics for MANET model exchange; in particular, it uses small-size models that can be easily exchanged, profiled and aggregated between nodes ($\tilde{5}$0KB after compression, making it useable for low-bandwidth links). In this section, we show the methods that can be used to accomplish the above mentioned tasks. We will also show early experimental results that validate these methods.

Model aggregation

As mentioned earlier, PAYL models are composed of centroids that capture payload byte distribution. With PAYL's incremental learning technique [174], merging models is as simple as averaging one model onto the other. If, for a specific payload length, only one of the models contains one or more computed centroids, the aggregate one will simply inherit these centroids.

This aggregation algorithm requires linear execution time (relative to the size of the model), thus satisfying the computational limitations typical to MANETs. Also, since PAYL models only contain statistical distributions, they can be distributed without encryption, as sensitive content will not be revealed.

Model differentiation

We can differentiate models at multiple levels of detail; at the first level, we extract the payload length distribution for the two models that we compare, and compute the Manhattan distance [174] between them. If this distance is greater than a significant threshold, we might conclude that the models display a significant difference. If not, we perform the analysis at a higher level of detail and compare based on the distance between each model's centroids for any port and packet length. This can be done either by considering only the predominant centroid for each packet length, or all centroids contained in the model.

The advantage of performing a multi-level analysis is that negative answers for divergent models can be returned very quickly, while in-depth comparisons will be performed only for very similar models. As a result, this method can satisfy low computational constraints.

6.6.4 Experimental results

In order to confirm the characteristics of the PAYL model differentiation and aggregation, we have conducted a series of experiments on a set of four models, which we name *model1* through *model4*.

- *model1* and *model2* were generated on machines accepting similar, primarily hypertext (i.e., ASCII/HTML) traffic (as both machines service the same population).
- *model3* was generated on a machine that sees a more complex traffic mix, including more streaming video and audio.
- *model4* was built out of mostly abnormal traffic populated with Code Red II, an IIS WebDAV exploit, etc.

As an example, figure 6.103 displays two centroids built for the same payload length and the same port in *model1* and in *model4*.

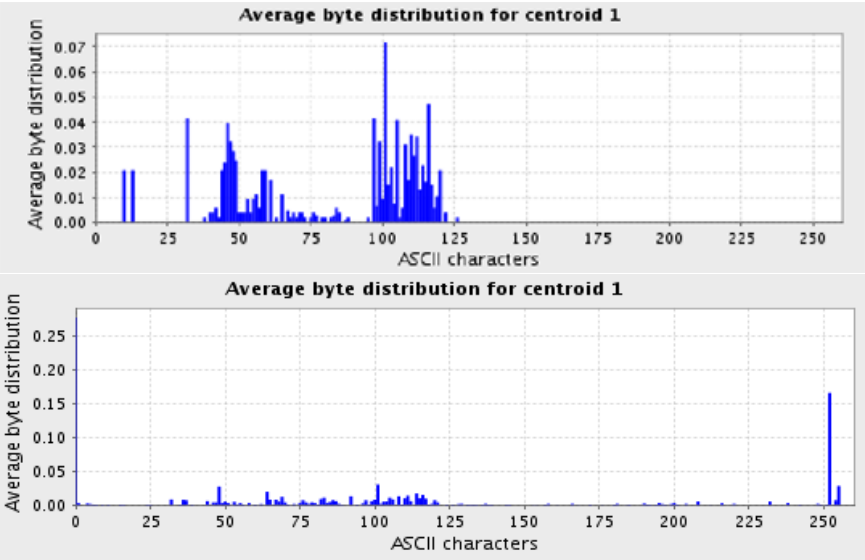

Figure 6.103: First centroid for port 80, length 1058 for model1 (top) and model4 (bottom).

The differentiation methodology discussed above was used to perform comparisons between these models. It correctly decided that *model1* and *model2* are similar,

while *model3* and *model4* present significant differences relative to other models. Numerical results reported by this method at various level of detail can be found in table 6.33.

	model1 *model2*	*model1* *model3*	*model1* *model4*	*model3* *model4*
Dist btw payload lengths	0.4210	1.5201	1.8981	0.7898
Avg dist over first centroids	0.5946	0.7400	1.6368	1.6330
Avg dist over all centroids	0.4276	0.6112	1.5220	1.5096

Table 6.33: Manhattan distances between models by three metrics.

This table shows two different metrics for comparison (payload length distributions and payload content distributions). We observe that *model1* is quite similar to *model2* with respect to their average payload length distributions and content distributions. *model4* clearly appears different than the other models for both length and content distributions. However, while *model1* and *model3* are similar with respect to their content distributions, there is a significant discrepancy between their length distributions. This fact leads us to the conclusion that we need to explore more ways of calculating similarity between the models. Our ongoing research is focused on correlating these and other metrics for better performance.

After differentiating neighbor models, a MANET node could be ready to aggregate its own model with ones that are similar to it. Aggregation was tested to demonstrate that important information is not lost from the original, individual models. We tested both the simple (unaggregated) models and the aggregated models against the same data. Based on Table 6.34, we can observe that even if the number of alerts is different for each simple model, the aggregated models do not significantly shorten the spectrum of alerts that each simple model can generate. The aggregated model shows similar behavior to the ones used to create it, which implies that the aggregation method is reliable.

total # packets / # content packets	*model1 # alerts*	*model2 # alerts*	*model1+2 #alerts*
127023	149	184	149
10414			
304182	2705	2829	2672
21812			
276332	9684	11128	9669
26294			
353897	11201	3394	2187
36780			

Table 6.34: Testing PAYL using model1 and model2, and their aggregate.

In table 6.35, we observe that there is a significant difference between the number of alerts generated by *model1* and *model3*, leading us to conclude that the two are not similar models. We plan to capture more data for similarity measurements in future experiments.

total # packets / # content packets	*model1 # alerts*	*model3 # alerts*	*model1+3 #alerts*
127023	149	81	148
10414			
304182	2705	1789	2613
21812			
276332	9684	1138	9530
26294			
353897	11201	2919	11040
36780			

Table 6.35: Testing PAYL using model1 and model3, and their aggregate.

6.6.5 Privacy Gain

An alert/packet payload represented by its byte frequency distribution makes it nearly impossible to reconstruct the actual payload except in degenerate cases—the byte distribution contains byte values but no sequential information. As previously discussed in section 6.5.6, $R \approx 1/2^{8208}$ for an example worm packet.

Now, given that a model represents *many* such packets, the problem becomes far more difficult. Despite PAYL's use of payload length clustering, a single centroid still may represent the frequency distribution over many thousands of packets. We can safely assert, therefore, that unless every packet is *identical*, a curious entity could only hope to glean general trends about packet content, as opposed to packets themselves. The only such trend that is recoverable given a 1-gram distribution is the relative frequency of individual ASCII values, essentially mirroring the analysis of a Z-String in section 6.5.6, and is similarly intractable.

6.6.6 Conclusion

This section describes early work in the notion of model exchange as an effective privacy-preserving anomaly-detection technique in certain applications, most particularly that of mobile ad-hoc networks (MANETs), where a model can be used to determine appropriate communication policies between mutually newly-introduced nodes. An initial feasibility study of model exchange was conducted, using the PAYL anomaly detector. Early results are promising, but there is further research to be conducted, as discussed in the future work section (§7.3).

6.7 Summary

This chapter has presented a comprehensive application and analysis for corroboration—collaborative intrusion detection, or CIDS. After introducing the corroboration model, three applications were detailed with the potential of significantly improving the way intrusion detection alerts are generated—at the IP header-level, at the payload level, and even at the traffic model level. Not only are all three techniques practical, but also have fewer barriers to entry compared to existing DIDS approaches, thanks to their use of privacy-preserving transforms. This enabled the

validation of the hypotheses introduced in section 6.1.1.

As adversaries grow more sophisticated in their scan and attack strategies, a more comprehensive defense is needed. Collaborative intrusion detection is a powerful tool against such adversaries, including those that do not generate high-visibility events that can be easily caught by traditional means. Worminator's decentralized, privacy-preserving approach provides a natural vehicle for such distributed intrusion detection applications, and its modular approach enables the use of a broad variety of sensors and techniques. While many of these techniques were described here, there are many venues for future research; if anything, Worminator has laid the foundation for significant new possibilities in attacker and defender profiling and data collection; several of these are discussed in section 7.3.

Chapter 7

Contributions, Future Work and Conclusion

7.1 Thesis Contributions

The contributions of this thesis include the following:

1. A **generalized approach** to supporting privacy-preserving event corroboration. As discussed in section 4, most work in this field has been extremely specialized. By building a more general framework, we expect to enable the application of privacy-preserving mechanisms and collaboration to a broader variety of fields. The techniques developed in this work are geared towards such a general application, including the notion of "retrofitting" such privacy-preserving techniques to legacy event correlators.

2. Additionally, this approach has been applied towards the development of a practical **collaborative intrusion detection** system. The methods in this thesis have been specifically geared for real-world solutions that establish privacy requirements, and Worminator's current and ongoing deployment is a key example of the feasibility of the proposed approach.

3. Characterization of the **effectiveness of privacy-preserving mechanisms**: to date, people have focused on proving the feasibility of privacy-preserving computation, but less so the advantages/disadvantages of picking a particular mechanism. The evaluation of the techniques in this thesis is a first significant step towards developing and using metrics to determine appropriate solutions based on the required application.

4. Development of **fast corroboration data structures**, including noisy Bloom filters, MRU and Timestamp Bloom filters, frequency transforms, and Z-Strings. Of these, the MRU and Timestamp BFs are completely new, and the application of noisy BFs, frequency transforms, and Z-Strings to privacy preservation and intrusion detection is novel. These data structures support a broad variety of applications, including IP-based alert dissemination, anomalous payload corroboration and signature generation, model corroboration, etc.

5. The Worminator IP results form a **longitudinal study** that will hopefully serve as a useful baseline for evaluating collaborative intrusion detection (and, perhaps, a helpful step forward for intrusion detection in general).

7.2 Research Accomplishments

In addition to the contributions listed above, the following practical accomplishments have already been completed to date:

- Published papers, including [116], [173], [114], [88], [59], [71], [69], [68], [58], and a poster presentation [113].

- Grant support by several agencies, including DARPA, NSA, DHS, and ARO. The author was involved in the writing of two phases of DHS grants, as well

as the ARO grant. Finally, successful demonstrations have been conducted for DARPA, DHS and ARO as partial fulfillment of grants' requirements.

- Successful application of the XUES platform to a DARPA challenge problem, demonstrated during DARPA DASADA [32] Demo Days in June, 2002.
- Successful deployment of Worminator to 5+ sites [112], demonstration of Worminator to program managers at NSF, DHS, ARO and others.
- Patent applications filed on aspects of Worminator work.

7.3 Future Work

As the focus of this thesis has been a generalized framework for privacy-preserving event corroboration, there are numerous interesting future work possibilities. I briefly describe two categories of future work: immediate applications of the model and Worminator platform, as well as possible new directions for this area of research.

7.3.1 Immediate Future Applications

First, the most immediate extension is to **deploy Worminator on a wider scale**. The techniques described in this thesis, and specifically the payload-based techniques, are specifically designed to be deployed at many enclaves to increase the correlation power and confidence provided by sensors at different sites with different content flows. However, encouraging peers to participate in anything but simple IP-based misuse alerts is challenging. We are integrating the work reported in this paper with a new and substantially different content exchange and sharing network known as DNAD-2 (Distributed Network Anomaly Detection) and seek collaborators to share their respective suspicious content detected by whatever local sensors may be available to them.

Additionally, the payload work in section 6.5 can be extended to support **polymorphic/obfuscated worm detection and mimicry attack.** While n-gram analysis has the potential of detecting polymorphic worms, e.g., [37], it becomes significantly harder as polymorphic worm engines launch *mimicry attacks* [168, 77] to mask themselves, such attacks are generally site-specific. Intersecting n-gram BFs across sites may provide the opportunity to identify even the few bytes of invariant common "code" that appear anywhere in such attacks. Having more BFs that can correlate these short regions increases the confidence in having found the correct snippets. *High-entropy* regions, such as those containing polymorphic or obfuscated code, would likely not be correlated. Another technique would be to apply n-gram analysis to host intrusion detection (HIDS), such as system call sequences.

Given the techniques in this thesis, another important area of research is to develop **aggregation/exchange policies**. Currently, Bloom filters are exchanged when they reach a particular "fill" threshold or when a certain amount of time has elapsed since the previous filter was sent. To minimize communication overhead *and* improve latency, one of several optimization techniques could be adopted, e.g., a *posture*-like approach [76] to communication: if nodes are deemed as being under threat, they may choose to communicate more frequently (e.g., 1 second latency) than when nodes are idle (once or twice a minute).

While many of the techniques discussed here have a broad variety of applications, mapping a technique to an application requires additional semantics. Therefore, one may choose to develop a **privacy-preserving policy language** for Worminator. The language would most likely be designed in XML, as an extension of the W3C P3P standard [167]. Each privacy-preserving data structure would then have associated metadata, enabling automatic matching between policies and implementations.

As mentioned in chapter 2, one application of privacy-preserving software monitoring is the building of an **"Application Communities"** [33] peer-to-peer

framework. The fundamental idea behind Application Communities is to accept software monoculture and to *leverage* it. Early work [89] has demonstrated the possibility of distributing the task of detecting faults in instrumented code, which typically runs orders-of-magnitude slower than native code, thereby reducing the effect on any one individual instance. Worminator could be adapted to serve as a privacy-preserving infrastructure; in fact, several of this thesis' committee members submitted a grant proposal to DARPA to explore this possibility.

While this thesis introduces a first-order privacy analysis, further discussion may optionally include an **information-theoretic analysis.** A more general approach to comparing correlation techniques is to measure the comparative *information gain* each correlation technique provides. However, this requires an accurate characterization of the distribution of packet content, which is both protocol and site content flow-specific. Additionally, information gain does not necessarily translate to correlation ability: as the results show, it is possible to correlate alerts reasonably well with significantly less information.

Finally, while the work in section 6.6 introduced the basic concept, there are multiple directions in which **privacy-preserving model correlation** can be extended. Site anomaly models need not just be restricted to frequency models, e.g. given two Bloom filters that represent anomaly models for Anagram, we can do a bitwise AND of the two Bloom filters to estimate the number of common "good" n-grams, or a bitwise OR of the two Bloom filters to aggregate and update the respective models. Other anomaly models may also be useable, such as those computed by the PAD algorithm [146]. Further discussion of this concept, as well as an extensive protocol to exchange and authorize peers based on their content models, is beyond the scope of this thesis; see [47] for an application of this approach to enhance access control.

7.3.2 Future Directions

Unlike the aforementioned items, the following focus on longer-term research directions based off of the work done in this thesis.

First, section 5.5 detailed several potential **subversion approaches** to the protocols and techniques described in this thesis. Ideally, a framework should incorporate as many mitigation strategies as possible to minimize the possibility of subversion. It is important to note that "complete security" does not exist; rather, the goal is to stay at least one step ahead of the attacker in the security race and to ensure that practical privacy is achieved.

Another major area of research beyond the scope of this thesis, but still of interest, is the evaluation of current **event distribution strategies** and integration of other distribution strategies underlying publish/subscribe infrastructures. We have continued to study methods for low cost alert distribution and propagation. We have done some exploratory work in this field [88], and work is ongoing on developing a complete decentralized-yet-secure exchange model.

Ideally, the techniques described here can enable the development of a solution for **automatically profiling** scanners and attackers. Preliminary work done in our longitudinal study demonstrates there are unique scanner profiles—the worm-infested home computer connected to a cable modem, the targeted attacker that has gained access to full class C (/24) subnets to distribute his behavior, the very long-term stealthy scanner, etc. A logical extension of this platform, to improve autonomic response mechanisms, would be to cluster this data and assign threat levels based on the resulting profile.

One can derive a more **general event typing and versioning** framework from the models presented with XUES and Worminator. Our work in [114] discusses mechanisms to support more flexible type discovery and versioning via the use of semantically-discovering XML processing toolkits. More specifically, if a recipient

receives XML messages that it does not understand, a protocol can be established to: *discover* the appropriate syntactic (schema) rules, and *process* the data into an intermediate form that can be processed. Our toolkits, known as *FleXML* and *tag processors*, respectively add these functionalities to XML parsers. This work could be extended to privacy-preserving types and corresponding transforms; the retrofitting model discussed in section 5.4 may also be of use to integrate with existing event type systems.

One extension of much intrusion detection work, including the Worminator work here, is the notion of **proactive response.** We envision developing metrics for ranking the threat level for particular "verticals", or groups of organizations. The deployment of collaborative distributed intrusion detection allows an organization to take mitigation and preemptive threat responses without having been directly attacked.

Finally, it may be desirable to have **automated development of rules or gauges** as to desired versus undesired behavior. Right now, pre-defined collections of events, such as software failures or a preponderance of suspicious scans, conveys significance. However, the ability to automatically parse system and behavioral models and generate important patterns or behaviors for detection purposes would be desirable. We are also considering a wider range of application domains. One particularly intriguing application area is autonomic service survivability in the face of insider and outsider security attacks, to coordinate what would otherwise be isolated, independently operating security mechanisms, as we proposed in [71].

7.4 Conclusion

In this thesis, I introduced the notion of *privacy-preserving decentralized event corroboration*, with the primary goal of enabling collaboration between peers whose

privacy policies typically prevent data disclosure. In doing so, I introduced several new ideas and methodologies:

- A generalized, type- and temporally-driven event correlation/corroboration model;
- A robust model for event corroboration with a heterogeneous set of privacy- and anonymity-preserving transforms;
- Extensive evaluation of the privacy-preserving transforms and practical implementation techniques to make them feasible for a broad variety of applications;
- Introduction of three significant new applications in the field of Collaborative Intrusion Detection (CIDS).

As the future work above demonstrates, this thesis enables the beginning of new research fields, especially in intrusion detection and network security, and it lays down the framework for enabling a new set of collaborative approaches to distributed security and application resiliency. Ultimately, I hope the work done here improves and extends the domain of collaborating applications and peers, spurs the development of new languages for expressing such policies, and ultimately encourages effective Internet-scale collaboration and participation amongst a broad variety of organizations.

Chapter 8

Bibliography

[1] Rakesh Agrawal, Alexandre Evfimievski, and Ramakrishnan Srikant. Information Sharing Across Private Databases. In *SIGMOD*, 2003.

[2] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Hippocratic Databases. In *VLDB*, 2002.

[3] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Order Preserving Encryption for Numeric Data. In *ACM SIGMOD*, Paris, France, 2004.

[4] Rakesh Agrawal and Ramakrishnan Srikant. Privacy-Preserving Data Mining. In *ACM SIGMOD*, 2000.

[5] Kostas G. Anagnostakis, Michael B. Greenwald, Sotiris Ioannidis, Angelos D. Keromytis, and Dekai Li. A Cooperative Immunization System for an Untrusting Internet. In *IEEE International Conference on Networks*, 2003.

[6] Stanford Program Analysis and Verification Group. Rapide website, 1999. `http://pavg.stanford.edu`.

[7] Apache Project. SpamAssassin. `http://spamassassin.apache.org/`.

[8] Robert Balzer and Neil M. Goldman. Mediating Connectors: A Non-ByPassable Process Wrapping Technology. In *DARPA Information Survivability Conference and Exposition*, volume 2, pages 361–368, Hilton Head, SC, 2000.

[9] Guruduth Banavar, Marc Kaplan, Kelly Shaw, Robert E. Strom, Daniel C. Sturman, and Wei Tao. Information Flow Based Event Distribution Middleware. In *Middleware Workshop at the International Conference on Distributed Computing Systems*, 1999.

[10] Mayank Bawa, Roberto J. Bayardo Jr., and Rakesh Agrawal. Privacy-Preserving Indexing of Documents on the Network. In *VLDB*, 2003.

[11] Scott Bekker. Microsoft Error Reporting Drives Bug Fixing Efforts. *ENT News*, 2002. `http://entmag.com/news/article.asp?EditorialsID=5532`.

[12] Steven M. Bellovin and William R. Cheswick. Privacy-Enhanced Searches Using Encrypted Bloom Filters, 2004.

[13] Bernhardt, Thomas. Esper. `http://esper.codehaus.org/`.

[14] Burton H. Bloom. Space/time trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970.

[15] George Candea, Emre Kiciman, Steve Zhang, Pedram Keyani, and Armando Fox. JAGR: An Autonomous Self-Recovering Application Server. In *Autonomic Computing Workshop*, 2003.

[16] Carnegie Mellon University ABLE Group. Acme Architectural Description Language. http://www-2.cs.cmu.edu/~acme/.

[17] Carnegie Mellon University ABLE Group. AcmeStudio Development Environment. http://www-2.cs.cmu.edu/~acme/AcmeStudio/AcmeStudio.html.

[18] Carnegie Mellon University ABLE Group. DASADA Gauge Infrastructure. http://www.cs.cmu.edu/~able/rainbow/gaugeinf.html.

[19] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Achieving Scalability and Expressiveness in an Internet-Scale Event Notification Service. In *Principles of Distributed Computing*, 2000.

[20] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.

[21] Francis Chang, Wu-chang Feng, and Kang Li. Approximate Caches for Packet Classification. In *IEEE INFOCOM*, 2004.

[22] Chase, Timothy. ISC SANS Intrusion Mailing List: UDP Traffic on Ports 33435-8, TCP on 2082 and 2745. http://lists.sans.org/pipermail/intrusions/2004-August/008268.html.

[23] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *International Workshop on Design Issues in Anonymity and Unobservability*, 2001.

[24] Robert G. Cole, Nam Phamdo, Moheeb A. Rajab, and Andreas Terzis. Requirements on Worm Mitigation Technologies in MANETS. In *Workshop on Principles of Advanced and Distributed Simulation*, 2005.

[25] Evan Cooke, Farnam Jahanian, and Danny McPherson. The Zombie Roundup: Understanding, Detecting and Disrupting Botnets. In *USENIX SRUTI Workshop*, Cambridge, MA, 2005.

[26] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-End Containment of Internet Worms. In *ACM SOSP*, 2005.

[27] Murilo Coutinho, Robert Neches, Alejandro Bugacov, Ke-Thia Yao, Vished Kumar, In-Young Ko, and Ragy Eleish. GeoWorlds: A Geographically Based Information System for Situation Understanding and Management. In *International Workshop on TeleGeoProcessing (TeleGeo 99)*, Lyon, France, 1999.

[28] Frederic Cuppens and Alexandre Miege. Alert Correlation in a Cooperative Intrusion Detection Framework. In *IEEE Security and Privacy*, 2002.

[29] Frederic Cuppens and Rodolphe Ortalo. LAMBDA: A Language to Model a Database for Detection of Attacks. In *RAID*, 2000.

[30] David Dagon, Cliff Zou, and Wenke Lee. Modeling Botnet Propagation Using Time Zones. In *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2006.

[31] R. Danyliw, J. Meijer, and Y. Demchenko. The Incident Object Description Exchange Format Data Model and XML Implementation, 2005. http://tools.ietf.org/wg/inch/draft-ietf-inch-iodef/draft-ietf-inch-iodef-04.txt.

[32] DARPA. DASADA Program. http://www.rl.af.mil/tech/programs/dasada/program-overview.html.

[33] DARPA. Application Communities Proposer Information Pamphlet/BAA, 2005. http://www.darpa.mil/ipto/Solicitations/open/05-51_PIP.htm.

[34] Julie Smith David, David Schuff, and Robert St. Louis. Managing Your Total IT Cost of Ownership. *Communications of the ACM*, 45(1):101–106, 2002.

[35] H. Debar, D. Curry, and B. Feinstein. The Intrusion Detection Message Exchange Format, 2005. http://tools.ietf.org/wg/idwg/draft-ietf-idwg-idmef-xml/draft-ietf-idwg-idmef-xml-14.txt.

[36] I. DeBare. Programmers in the Driver's Seat: Companies Clamor for Year 2000 Programmers. *Dr. Dobb's Journal*, Spring 1998 1998.

[37] Theo Detristan, Tyll Ulenspiegel, Yann Malcom, and Mynheer von Underduk. Polymorphic Shellcode Engine Using Spectrum Analysis, 2003.

[38] Sarang Dharmapurikar, Praveen Krishnamurthy, Todd Sproull, and John Lockwood. Deep Packet Inspection using Parallel Bloom Filters. In *IEEE Symposium on High Performance Interconnects (HOTI)*, 2003.

[39] John R. Douceur. The Sybil Attack. In *International Workshop on Peer-to-Peer Systems*, 2002.

[40] Wenliang Du and Mikhail Atallah. Secure Multi-Party Computation Problems and Their Applications: A Review and Open Problems. In *New Security Paradigms Workshop*, 2001.

[41] Alexander Dupuy, Soumitra Sengupta, Ouri Wolfson, and Yechiam Yemini. NETMATE: A Network Management Environment. *IEEE Network*, 5(2):35–43, 1991.

[42] Cynthia Dwork, Moni Naor, and Amit Sahai. Concurrent Zero-Knowledge. *Journal of the ACM*, 51(6):851–898, 2004.

[43] EPIC. Privacy, 2005. http://www.epic.org/privacy/.

[44] Justin R. Erenkrantz. Handling Hierarchical Events In An Internet-Scale Service. Technical report, UCI, 2001. http://www.ucf.ics.uci.edu/~jerenk/siena-xml/SienaPaper.html.

[45] Li Fan, Pei Cao, Jussara Almeida, and Andrei Broder. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. In *ACM SIGCOMM*, 1998.

[46] Michael J. Freedman, Emil Sit, Josh Cates, and Robert Morris. Introducing Tarzan, a Peer-to-Peer Anonymizing Network Layer. In *International Workshop on Peer-to-Peer Systems*, 2002.

[47] Vanessa Frias-Martinez and Salvatore J. Stolfo. BARTER: Profile Model Exchange for Behavior-based Access Control. Technical report, Columbia University, 2006. Submitted to conference.

[48] A.G. Ganek and T.A. Corbi. The dawning of the autonomic computing era. *IBM Systems Journal*, 2003.

[49] David Garlan, Shang-Wen Cheng, and Bradley Schmerl. Increasing System Dependability through Architecture-Based Self-Repair. In *Workshop on Architecting Dependable Systems*, Ottawa, Canada, 2003.

[50] David Garlan, Robert T. Monroe, and David Wile. Acme: Architectural Description of Component-Based Systems. In *Foundations of Component-Based Systems*, pages 47–67. Cambridge University Press, 2000.

[51] Peter W. Gill. Probing for a Continual Validation Prototype. Technical report, Worcester Polytechnic Institute, 2001. MS Thesis. http://www.wpi.edu/Pubs/ETD/Available/etd-0826101-235008/.

[52] Eu-Jin Goh. Secure Indexes. Technical report, Stanford University, 2004.

[53] Oded Goldreich and Yair Oren. Definitions and Properties of Zero-Knowledge Proof Systems. *Journal of Cryptology*, 7(1):1–32, 1994.

[54] David Goldschlag, Michael Reed, and Paul Syverson. Onion Routing for Anonymous and Private Internet Connections. *Communications of the ACM*, 43(2):39–41, 1999.

[55] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The Knowledge Complexity of Interactive-Proof Systems. In *ACM Symposium on Theory of Computing*, Providence, RI, 1989.

[56] Li Gong. JXTA: A Network Programming Environment. *IEEE Internet Computing*, 5(3):88–95, 2001.

[57] Antone Gonsalves. IBM Releases Blueprint For Automated Computing. *TechWeb*, 2003. `http://www.techweb.com/wire/26801207`.

[58] Philip Gross, Suhit Gupta, Gail Kaiser, Gaurav S. Kc, and Janak J. Parekh. An Active Events Model for Systems Monitoring. In *Working Conference on Complex and Dynamic Systems Architectures*, Brisbane, Australia, 2001.

[59] Philip Gross, Janak J. Parekh, and Gail Kaiser. Secure "Selecticast" for Collaborative Intrusion Detection Systems. In *International Workshop on Distributed Event-Based Systems*, Edinburgh, UK, 2004.

[60] The Open Group. Universal Unique Identifier, 1997. `http://www.opengroup.org/onlinepubs/9629399/apdxa.htm`.

[61] Heineman, George T. and Calnan, Paul and Kurtz, Ben. Active Interface Development Environment (AIDE). `http://www.cs.wpi.edu/~heineman/dasada/`.

[62] Joseph L. Hellerstein, Dawn M. Tilbury, Sujay Parekh, and Yixin Diao. *Feedback Control of Computing Systems*. Wiley, 2004.

[63] Qiang Huang, Helen J. Wang, and Nikita Borisov. Privacy-Preserving Friends Troubleshooting Network. In *NDSS*, San Diego, CA, 2005.

[64] Yi-an Huang and Wenke Lee. A Cooperative Intrusion Detection System for Ad Hoc Networks. In *ACM Workshop on Security in Ad Hoc and Sensor Networks*, Fairfax, VA, 2003.

[65] IANA. Internet Port Numbers. `http://www.iana.org/assignments/port-numbers`.

[66] IEEE. Autonomic Computing Workshop: Fifth Annual International Workshop on Active Middleware Services. `http://www.caip.rutgers.edu/ams2003/`.

[67] Ramaprabhu Janakiraman, Marcel Waldvogel, and Qi Zhang. Indra: A peer-to-peer approach to network intrusion detection and prevention. In *WETICE*, 2003.

[68] Gail Kaiser, Philip Gross, Gaurav S. Kc, Janak J. Parekh, and Giuseppe Valetto. An Approach to Autonomizing Legacy Systems. In *Workshop on Self-Healing, Adaptive and Self-MANaged Systems*, New York, NY, 2002.

[69] Gail Kaiser, Janak J. Parekh, Philip Gross, and Giuseppe Valetto. Kinesthetics eXtreme: An External Infrastructure for Monitoring Distributed Legacy Systems. In *Autonomic Computing Workshop*, Seattle, WA, 2003.

[70] Gail Kaiser, Adam Stone, and Steve Dossick. A Mobile Agent Approach to Lightweight Process Workflow. In *International Process Technology Workshop*, 1999.

[71] Angelos D. Keromytis, Janak J. Parekh, Philip Gross, Gail Kaiser, Vishal Misra, Jason Nieh, Dan Rubenstein, and Salvatore J. Stolfo. A Holistic Approach to Service Survivability. In *ACM Workshop on Survivable and Self-Regenerative Systems*, Fairfax, VA, 2003.

[72] Hyang-Ah Kim and Brad Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. In *USENIX Security Symposium*, San Diego, CA, 2004.

[73] Lea Kissner. *Thesis Proposal: Privacy Preserving Distributed Information Sharing*. PhD thesis, CMU, 2005.

[74] Lea Kissner. *Privacy-Preserving Distributed Information Sharing*. PhD thesis, Carnegie Mellon University, 2006.

[75] Lea Kissner and Dawn Song. Privacy-Preserving Set Operations. In *CRYPTO*, 2005.

[76] John Knight, Dennis Hembigner, Alexander L. Wolf, Antonio Carzaniga, Jonathan Hill, Premkumar Devanbu, and Michael Gertz. The Willow Architecture: Comprehensive Survivability for Large-Scale Distributed Applications. In *Dependable Systems and Networks (DSN)*, 2002.

[77] Oleg Kolesnikov, David Dagon, and Wenke Lee. Advanced Polymorphic Worms: Evading IDS by Blending in with Normal Traffic, 2006.

[78] Alexander V. Konstantinou, Danilo Florissi, and Yechiam Yemini. Towards Self-Configuring Networks. In *DARPA Active NEtworks Conference and Exposition*, pages 143–156, San Francisco, CA, 2002.

[79] Alexander V. Konstantinou and Yechiam Yemini. Programming Systems for Autonomy. In *Autonomic Computing Workshop*, pages 186–195, 2003.

[80] Christian Kreibich and Jon Crowcroft. Honeycomb - Creating Intrusion Detection Signatures Using Honeypots. In *ACM Workshop on Hot Topics in Networks*, Boston, MA, 2003.

[81] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Polymorphic Worm Detection Using Structural Information of Executables. In *Symposium on Recent Advances in Intrusion Detection*, Seattle, WA, 2005.

[82] Christopher Kruegel, Thomas Toth, and Clemens Kerer. Decentralized Event Correlation for Intrusion Detection. In *International Conference on Information Security and Cryptology*, 2002.

[83] Leslie Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[84] LANDesk Software. LANDesk Management Software. `http://www.landesksoftware.com/`.

[85] Zhenkai Liang and R. Sekar. Fast and Automated Generation of Attack Signatures: A Basis for Building Self-Protecing Servers. In *ACM Conference on Computer and Communications Security*, Alexandria, VA, 2005.

[86] Patrick Lincoln, Phillip Porras, and Vitaly Shmatikov. Privacy-Preserving Sharing and Correlation of Security Alerts. In *USENIX Security*, 2004.

[87] Yehuda Lindell and Benny Pinkas. Privacy Preserving Data Mining. *Cryptology*, 15(3):177–206, 2002.

[88] Michael E. Locasto, Janak J. Parekh, Angelos D. Keromytis, and Salvatore J. Stolfo. Towards Collaborative Security and P2P Intrusion Detection. In *IEEE Information Assurance Workshop*, West Point, NY, 2005.

[89] Michael E. Locasto, Stelios Sidiroglou, and Angelos D. Keromytis. Application Communities: Using Monoculture for Dependability. In *HotDep*, 2005.

[90] Michael E. Locasto, Stelios Sidiroglou, and Angelos D. Keromytis. Software Self-Healing Using Collaborative Application Communities. In *Internet Society (ISOC) Symposium on Network and Distributed Systems Security*, pages 95–106, San Diego, CA, 2006.

[91] Michael E. Locasto, Ke Wang, Angelos D. Keromytis, and Salvatore J. Stolfo. FLIPS: Hybrid Adaptive Intrusion Prevention. In *Symposium on Recent Advances in Intrusion Detection*, Seattle, WA, 2005.

[92] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley/Pearson Education, Indianapolis, 1st edition, 2002.

[93] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, 1995.

[94] David C. Luckham and James Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21(9):717–734, 1995.

[95] LURHQ Threat Intelligence Group. Windows Messenger Popup Spam on UDP Port 1026. http://www.lurhq.com/popup_spam.html.

[96] John Markoff. Attack of the Zombie Computers Is Growing Threat. *New York Times*, January 7 2007. http://www.nytimes.com/2007/01/07/technology/07net.html.

[97] Bill McCarty. Botnets: Big and Bigger. *IEEE Security and Privacy*, 1(4):87–90, 2003.

[98] David L. Mills. RFC 958: Network Time Protocol (NTP), 1985. http://www.faqs.org/rfcs/rfc958.html.

[99] Naftaly H. Minsky. On Conditions for Self-Healing in Distributed Software Systems. In *Autonomic Computing Workshop*, 2003.

[100] Michael Mitzenmacher. Compressed Bloom Filters. *IEEE Transactions on Networking*, 10(5):604–612, 2002.

[101] David Moore, Colleen Shannon, Geoffrey M. Voelker, and Stefan Savage. Internet Quarantine: Requirements for Containing Self-Propagating Code. In *INFOCOM*, 2003.

[102] James Newsome, David Brumley, and Dawn Song. Vulnerability-Specific Execution Filtering for Exploit Prevention on Commodity Software. In *Network and Distributed Security Symposium (NDSS)*, San Diego, CA, 2006.

[103] James Newsome, Brad Karp, and Dawn Song. Polygraph: Automatically Generating Signatures for Polymorphic Worms. In *IEEE Security and Privacy*, Oakland, CA, 2005.

[104] Peng Ning, Yun Cui, and Douglas Reeves. Constructing Attack Scenarios Through Correlation of Intrusion Alerts. In *ACM Conference on Computer and Communications Security*, Washington, DC, 2002.

[105] NIST. FIPS 180-1: Secure Hash Standard (SHA-1), 1995. http://www.itl.nist.gov/fipspubs/fip180-1.htm.

[106] OC Systems. AProbe: A New Approach for Testing Web Applications. http://www.ocsystems.com/aprobe_web_testing.html.

[107] OC Systems. Improving Availability of Enterprise Applications with RootCause. http://www.ocsystems.com/rootcause_white_paper.html.

[108] US Department of Health and Human Services. HIPAA, 2005. http://www.hhs.gov/ocr/hipaa/.

[109] US Department of Labor/OSHA. SIC Division Structure, 2005. http://www.osha.gov/pls/imis/sic_manual.html.

[110] Ruoming Pang and Vern Paxson. A High-Level Programming Environment for Packet Trace Anonymization and Transformation. In *ACM SIGCOMM*, 2003.

[111] Janak J. Parekh. Candidacy Exam, 2003.

[112] Janak J. Parekh. Worminator homepage, 2005. http://worminator.cs.columbia.edu.

[113] Janak J. Parekh. Worminator (poster). In *Symposium on Recent Advances in Intrusion Detection*, Seattle, WA, 2005.

[114] Janak J. Parekh, Gail Kaiser, Philip Gross, and Giuseppe Valetto. Retrofitting Autonomic Capabilities onto Legacy Systems. *Journal on Cluster Computing*, 9(2):141–159, 2006.

[115] Janak J. Parekh, Ke Wang, and Salvatore J. Stolfo. Privacy-Preserving Payload-Based Correlation for Accurate Malicious Traffic Detection. Technical report, Columbia University Dept. of CS, 2006. http://mice.cs.columbia.edu/getTechreport.php?techreportID=409.

[116] Janak J. Parekh, Ke Wang, and Salvatore J. Stolfo. Privacy-Preserving Payload-Based Correlation for Accurate Malicious Traffic Detection. In *Large-Scale Attack Detection, Workshop at SIGCOMM*, Pisa, Italy, 2006.

[117] Anand Patwardhan, Jim Parker, Anupam Joshi, Michaela Iorga, and Tom Karygiannis. Secure Routing and Intrusion Detection in Ad-Hoc Networks. In *Third IEEE International Conference on Pervasive Computing and Communications*, 2005.

[118] Vern Paxson. BRO: A System for Detecting Network Intruders in Real Time. 1998.

[119] Paul Pazandak and David Wells. ProbeMeister: Distributed Runtime Software Instrumentation. In *First International Workshop on Unanticipated Software Evolution*, 2002.

[120] Dan Phung, Giuseppe Valetto, and Gail Kaiser. Adaptive Internet Interactive Team Video. In *International Conference on Advances in Web-Based Learning (ICWL)*, Hong Kong, China, 2005.

[121] Phillip Porras and Peter G. Neumann. EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances. In *National Information Systems Security Conference*, 1997.

[122] Honeynet Project and Research Alliance. Know your Enemy: Tracking Botnets, 3/13/05 2005. `http://www.honeynet.org/papers/bots/`.

[123] Vaclav Rajlich, Norman Wilde, Michelle Buckellew, and Henry Page. Software Cultures and Evolution. *IEEE Computer*, 34(9):24–28, September 2001 2001.

[124] M.V. Ramakrishna, E. Fu, and E. Bahcekapili. A Performance Study of Hashing Functions. 1994.

[125] IBM Research. Autonomic Computing. `http://www.research.ibm.com/autonomic`.

[126] Research and Education Networking Information Sharing and Analysis Center. `http://www.ren-isac.net/`.

[127] Retrologic Systems. Retroguard. `http://www.retrologic.com/`.

[128] Seth Robertson, Eric V. Siegel, Matt Miller, and Salvatore J. Stolfo. Surveillance Detection in High Bandwidth Environments. In *DARPA Information Survivability Conference and Exposition (DISCEX)*, 2003.

[129] Martin Roesch. Snort - Lightweight Intrusion Detection for Networks. Technical report.

[130] Antony Rowstron and Peter Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems. In *IFIP/ACM International Conference on Distributed Systems*, Heidelberg, Germany, 2001.

[131] Antony Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *COST264 Workshop on Networked Group Communication*, 2001.

[132] Elizabeth M. Royer and Chai-Keong Toh. A Review of Current Routing Protocols for Ad Hoc Mobile Wireless Networks. *IEEE Personal Communications*, 6(2):46–55, 1999.

[133] SANS Institute. Intrusion Detection FAQ: What is host-based intrusion detection? `http://www.sans.org/resources/idfaq/host_based.php`.

[134] Lambert Schaelicke, Matthew R. Geiger, and Curt J. Freeland. Improving the Database Logging Performance of the Snort Network Intrusion Detection Sensor. Technical report, University of Notre Dame, 2002.

[135] Bill Segall, David Arnold, Julian Boot, Michael Henderson, and Ted Phelps. Content-Based Routing with Elvin4. In *Australian UNIX and Open Systems User Group Winter Conference (AUUG2K)*, 2000.

[136] Sendmail, Inc. Sendmail Mail Filter API. `http://www.sendmail.com/partner/resources/development/milter_api/`.

[137] Sendmail, Inc. Sendmail Mail Server. http://www.sendmail.org/.

[138] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated Worm Fingerprinting. In *6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, CA, 2004.

[139] Stuart Staniford-Chen, Steven Cheung, R. Crawford, and M. Dilger. GrIDS - A Graph Based Intrusion Detection System for Large Networks. In *National Information Computer Security Conference*, Baltimore, MD, 1996.

[140] Stuart Staniford-Chen, Vern Paxson, and Nicholas Weaver. How to 0wn the Internet in Your Spare Time. In *USENIX Security*, 2002.

[141] Malgorzata Steinder and Adarshpal Sethi. Probabilistic Event-Driven Fault Diagnosis Through Incremental Hypothesis Updating. In *International Symposium on Integrated Network Management*, 2003.

[142] D. Sterne, P. Balasubramanyam, D. Carman, B. Wilson, R. Talpade, C. Ko, R. Balupari, C-Y. Tseng, T. Bowen, Karl N. Levitt, and Jeff Rowe. A General Cooperative Intrusion Detection Architecture for MANETs. In *Workshop on Information Assurance*. IEEE, 2005.

[143] Roy Sterritt, Mary Shapcott, Kenny Adamson, and Edwin Curran. High Speed Network First-Stage Alarm Correlator. In *International Conference on Intelligent Systems and Control*, 2000.

[144] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *SIGCOMM*, San Diego, 2001. ACM.

[145] Salvatore J. Stolfo. Worm and Attack Early Warning: Piercing Stealthy Reconnaissance. *IEEE Security and Privacy*, 2004.

[146] Salvatore J. Stolfo, Frank Apap, Eleazar Eskin, Katherine Heller, Shlomo Hershkop, Andrew Honig, and Krista Svore. A Comparative Evaluation of Two Algorithms For Windows Registry Anomaly Detection. *Journal of Computer Security*, 13(4):659–693, 2005.

[147] Salvatore J. Stolfo, Andreas L. Prodromidis, Shelley Tselepis, Wenke Lee, Dave W. Fan, and Philip Chan. JAM: Java Agents for Meta-Learning over Distributed Databases. In *International Conference on Knowledge Discovery and Data Mining*, Newport Beach, CA, 1997.

[148] Dhanant Subhadrabandhu, Saswati Sarkar, and Farooq Anjum. Efficacy of Misuse Detection in Adhoc Networks. In *IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks (SECON)*, 2004.

[149] Sun. Java Management Extensions (JMX). http://java.sun.com/products/JavaManagement/.

[150] Sun. Jini Technology. http://www.sun.com/software/jini/.

[151] Sun. Java Message Specification, 2002. http://java.sun.com/products/jms/.

[152] Latanya Sweeney. k-Anonymity: A Model for Protecting Privacy. *International Journal on Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(5):557–570, 2002.

[153] Inc. Symantec. Symantec Internet Security Threat Report, Trends for January 06-June 06. Technical report, 2006.

[154] System Management ARTS. http://www.smarts.com.

[155] Yong Tang and Shigang Chen. Defending Against Internet Worms: A Signature-Based Approach. In *IEEE Infocom*, Miami, FL, 2005.

[156] The Workflow Management Coalition. http://www.wfmc.org.

[157] Johannes Ullrich. DShield home page, 2005. http://www.dshield.org.

[158] Risto Vaarandi. SEC - Simple Event Correlator, 2005. `http://simple-evcorr.sourceforge.net/`.

[159] Giuseppe Valetto. *Orchestrating the Dynamic Adaptation of Distributed Software with Process Technology*. PhD thesis, Columbia University, 2004.

[160] Giuseppe Valetto and Gail Kaiser. Using Process Technology to Control and Coordinate Software Adaptation. In *International Conference on Software Engineering*, 2003.

[161] Giuseppe Valetto, Gail Kaiser, and Gaurav S. Kc. A Mobile Agent Approach to Process-based Dynamic Adaptation of Complex Software Systems. In *Eighth European Workshop on Software Process Technology*, 2001.

[162] Robbert Van Renesse, Kenneth P. Birman, and Werner Vogels. Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining. *ACM Transactions on Computer Systems*, 2003.

[163] Ved Prakash, Vipul. Vipul's Razor. `http://razor.sourceforge.net/`.

[164] Giovanni Vigna, Sumit Gwalani, Kavitha Srinivasan, Elizabeth M. Belding-Royer, and Richard A. Kemmerer. An Intrusion Detection Tool for AODV-based Ad hoc Wireless Networks. In *Computer Security Applications Conference*, 2004.

[165] W3C. Extensible Markup Language (XML). `http://www.w3.org/XML`.

[166] W3C. XML Path Language (XPath). `http://www.w3.org/TR/xpath`.

[167] W3C. Platform for Privacy Preferences (P3P) Project, 2005. `http://www.w3.org/P3P/`.

[168] David Wagner and Paolo Soto. Mimicry Attacks on Host-Based Intrusion Detection Systems. In *ACM CCS*, 2002.

[169] Helen J. Wang, Chuanxiong Guo, Daniel R. Simon, and Alf Zugenmaier. Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits. In *ACM SIGCOMM*, 2004.

[170] Helen J. Wang, John C. Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. Automatic Misconfiguration Troubleshooting with PeerPressure. In *OSDI*, San Francisco, 2004.

[171] Ke Wang. *Network Payload-based Anomaly Detection and Content-based Alert Correlation*. PhD thesis, Columbia University, 2006.

[172] Ke Wang, Gabriela Cretu, and Salvatore J. Stolfo. Anomalous Payload-based Worm Detection and Signature Generation. In *Symposium on Recent Advances in Intrusion Detection*, Seattle, WA, 2005.

[173] Ke Wang, Janak J. Parekh, and Salvatore J. Stolfo. Anagram: A Content Anomaly Detector Resistant to Mimicry Attack. In *Symposium on Recent Advances in Intrusion Detection*, Hamburg, Germany, 2006.

[174] Ke Wang and Salvatore J. Stolfo. Anomalous Payload-based Network Intrusion Detection. In *Symposium on Recent Advances in Intrusion Detection*, Sophia Antipolis, France, 2004.

[175] Alexander Wise, Aaron G. Cass, Barbara S. Lerner, Eric K. McCall, Leon J. Osterweil, and Stanley M. Sutton Jr. Using Little-JIL to Coordinate Agents in Software Engineering. In *15th IEEE International Conference on Automated Software Engineering*, Grenoble, France, 2000.

[176] Dingbang Xu and Peng Ning. Privacy-Preserving Alert Correlation: A Concept Hierarchy Based Approach. In *21st Annual Computer Security Applications Conference*, Tucson, AZ, 2005.

[177] Andrew C. Yao. Protocols for Secure Computations. In *IEEE Symposium on Foundations of Computer Science*, 1982.

[178] Andrew C. Yao. Theory and Application of Trapdoor Functions. In *Foundations of Computer Science (FOCS)*, Chicago, IL, 1982.

[179] Vinod Yegneswaran, Paul Barford, and Somesh Jha. Global Intrusion Detection in the DOMINO Overlay System. In *NDSS*, 2004.

[180] Vinod Yegneswaran, Jonathon T. Giffin, Paul Barford, and Somesh Jha. An Architecture for Generating Semantics-Aware Signatures. In *USENIX Security Symposium*, 2005.

[181] Shaula Alexander Yemini, Shmuel Kliger, Eyal Mozes, Yechiam Yemini, and David Ohsie. High Speed and Robust Event Correlation. *IEEE Communications*, 1996.

[182] Yuanyuan Zhao and Robert E. Strom. Exploiting Event Stream Interpretation in Publish-Subscribe Systems. In *Principles of Distributed Computing*, 2001.

Appendix A

Event Packager and Event Distiller Rulesets

A.1 Event Packager

A.1.1 Event Packager Rule Language

The full Event Packager documentation may be found at `http://www.psl.cs.columbia.edu/xues/EventPackager.html`.

A.1.2 Event Packager Example

Figure A.1 shows a configuration that subscribes to one event bus (Siena) for events which have the attribute-value pair ("TestAttribute", "TestValue"), stores the results in a SQL database, and finally copies the results to another event bus. The references to the Console are needed if console-level control is desired of the Event Packager; it perceives the user console to be yet another source (and, potentially, a sink) for events.

A.2 Event Distiller

The full Event Distiller documentation may be found at `http://www.psl.cs.columbia.edu/xues/EventDistiller.html`; I present an abbreviated description here.

A.2.1 Event Distiller Rule Language

- `<rulebase xmlns="http://www.psl.cs.columbia.edu/2001/01/DistillerRule.xsd">`

  This line is the top-level XML rulebase declaration, and is required.

- `<rule name="rule name">`

  This is the top-level rule declarator, is declared within ruleBase, and is repeated at the beginning of each rule. The following parameters may be supplied:

  - name (required): Rule names are used primarily for disambiguation at this time; they are not referred elsewhere.
  - position (optional): the position where this rule is inserted, starting with 0. The position specifies the priority of this rule in receiving events: higher priority rules (smaller numbers) receive events first. This is useful in the case of event absorption on the part of a state (see the absorb attribute in state).
  - instantiation (optional): the criterion for instantiating this rule. Legal values are: 0, 1, 2. '0' means the rule will only be instantiated once. '1' means there will always be only one instance at any given time, so a new instance is created as the previous succeeds or one times out. '2' means a new instance is created whenever a previous instance starts (i.e., receives its first event), so that there will always be one instance listening for the starting event(s).

- `<states>`

  This declarator, contained in a rule, signifies the beginning of state declarations (of which there may be many). Note: There can be only one "states" declaration in a rule.

- `<state name="state name" timebound="milliseconds" children="CSV-list of children names" actions="CSV-list of actions" fail_actions="CSV-list of actions" absorb="boolean value" count="integer value">`

  This is the beginning of declarator for a single state in the state-list (pattern) to be matched for this rule. A number of parameters are supplied alongside the rule declarator:

  - name (required): specifies the state name. This name should not conflict with other state names within this rule.
  - timebound (required): specifies the timebound in which this event can happen relative to the previous event. For the first event, the timebound is generally set to -1 (e.g. no time limit); for subsequent timebounds it is recommended, although not required, that the timebound be positive. If the first state of a rule has positive timebound, it is measured against the time at which the rule is created.(Note that timebound is not precise - a fudge factor is present in the Event Distiller to take care of race conditions.)
  - children (optional): specifies (in a comma-delimited list, no spaces) states that can follow this particular state.
  - actions (optional): specifies the notification(s) to be sent out when this state is matched. Usually, this is intended for the last state in a rule, which would signify as the "rule matching", but intermediate notifications can be sent out.

  - fail_actions (optional): specifies the notification(s) to be sent out if this state times out while waiting for input. Unlike actions, this is intended for each state, so a different notification may be sent out for a failure at any given point in the state machine. Note that if a rule allows multiple paths between states (so that at one given time multiple states are subscribed), failure notifications for one of the states that timed out will be sent, only if all currently subscribed states time out.
  - absorb (optional): Whether this state will absorb the events that match it. If this is set to true, when an event matches this state, the event will not be passes on to any other state currently subscribed. If this field is not specified, a default value of 'false' will be used.
  - count (optional): the number of times this event will need to be matched before it passes. A default value of '1' is used if this field is not specified. If the value specified is greater than '1' (say, n, children and actions will only apply to the nth time that the state is matched; while fail_actions, if specified, will apply at all times. The special value '-1' indicates that the event may occur any number of times (within the specified timebound), until one of the children is matched, thus terminating the loop.

- `<attribute name="attribute name" value="attribute value" op="comparison operator" type="value type" />`

  Declarator for an attribute-value pair required in this particular state. Note that there may be many attributes per state, and they are ANDed together in the filter that matches incoming notifications to this state. Since there are no embedded tags within this tag, you can use the compact end-tag notation (e.g. "/>"). The comparison operator is a string representation of the operator to be used for matching the value. For instance, you would specify op=">" to match all values greater than the one that the specified value. The default operator is

the equality operator. Type needs to be specified since some operators only make sense for certain types; the default type is the string type. Note that the value field here is matched using a string-equals comparison, unless a special wildcard-binding notation is used (see below).

- `</state>`
  `</states>`

- `<actions>`

  This declarator, contained in a rule, signifies the beginning of action declarations (of which there may be many). Note: There can be only one "actions" declaration in a rule.

- `<notification name="notification name">`

  This is the beginning of the notification (action) declarator. There is one parameter, name, which corresponds to the actions and fail_actions references above. It is strongly recommended that this name be one contiguous phrase without whitespace or punctuation to avoid conflicts in the CSV-lists referenced above.

- `<attribute name="attribute name" value="attribute value" />`

  Declarator for an attribute-value pair to be included in this particular notification. Note that there may be many attributes per notification. Since there are no embedded tags within this tag, you can use the compact end-tag notation. If you use wildcard-binding above, you may use the same wildcard-binding tag here - it will be substituted with the actual bound value (again, see below).

- `</notification>`
  `</actions>`
  `</rule>`
  `</rulebase>`

A.2.2 Event Distiller Example

Figure A.2 presents an extended view of the aforementioned rules, including the corresponding notifications if the pattern is matched.

In this case, we allocated 15 seconds for the method to complete, and in the case of a crash, both static and dynamic (i.e., wildcard-bound) data were reported (note that we abbreviated a few URLs for readability).

```
<EventPackagerConfiguration>
 <Inputters>
  <Inputter Name="SienaInput1" Type="psl.xues.ep.input.SienaInput"
            SienaReceivePort="7890">
   <SienaFilter Name="TestFilter1">
    <SienaConstraint AttributeName="TestAttribute" Op="="
                     ValueType="String" Value="TestValue" />
   </SienaFilter>
  </Inputter>
  <Inputter Name="ConsoleInput1" Type="psl.xues.ep.input.ConsoleInput"/>
 </Inputters>
 <Outputters>
  <Outputter Name="SienaOutput1" Type="psl.xues.ep.output.SienaOutput"
             SienaReceivePort="7891" />
  <Outputter Name="NullOutput1" Type="psl.xues.ep.output.NullOutput" />
 </Outputters>
 <Transforms>
  <Transform Name="Store1" Type="psl.xues.ep.transform.StoreTransform"
             StoreName="HSQLDB1" />
 </Transforms>
 <Stores>
  <Store Name="HSQLDB1" Type="psl.xues.ep.store.JDBCStore"
         DBType="hsqldb" DBDriver="org.hsqldb.jdbcDriver"
         DBName="xues" DBTable="xues" Username="sa" Password=""/>
 </Stores>
 <Rules>
  <Rule Name="TestRule1">
   <Inputs><Input Name="SienaInput1" /></Inputs>
   <Transforms><Transform Name="Store1" /></Transforms>
   <Outputs><Output Name="SienaOutput1" /></Outputs>
  </Rule>
  <Rule Name="ConsoleRule">
   <Inputs><Input Name="ConsoleInput1" /></Inputs>
   <Outputs><Output Name="NullOutput1" /></Outputs>
  </Rule>
 </Rules>
</EventPackagerConfiguration>
```

Figure A.1: Event Packager example.

```
<rulebase xmlns="http://www.psl.cs.columbia.edu/2001/01/
                 DistillerRule.xsd">

<rule name="ActiveEvent">
 <states>
  <state name="Start" timebound="-1" children="End" actions=""
         fail_actions="">
   <attribute name="Service" value="*service"/>
   <attribute name="Status" value="Started"/>
   <attribute name="ipAddr" value="*ipaddr"/>
   <attribute name="ipPort" value="*ipport"/>
   <attribute name="time" value="*time"/>
  </state>
  <state name="End" timebound="15000" children="" actions="Debug"
         fail_actions="Crash">
   <attribute name="Service" value="*service"/>
   <attribute name="State" value="FINISHED_STATE"/>
   <attribute name="ipAddr" value="*ipaddr"/>
   <attribute name="ipPort" value="*ipport"/>
   <attribute name="time" value="*time2"/>
  </state>
 </states>
 <actions>
  <notification name="Crash">
   <attribute name="Notification_Type" value="GW_Alarm"/>
   <attribute name="Message" value="Dead_Service"/>
   <attribute name="KX_Reaction_Type" value="Workflow"/>
   <attribute name="KX_Reaction_Spec" value="Disable_Service"/>
   <attribute name="Timestamp" value="*time"/>
   <attribute name="Service" value="*service"/>
   <attribute name="Name" value="gwHostAdapter"/>
   <attribute name="IPaddress" value="*ipaddr"/>
   <attribute name="port" value="*ipport"/>
   <attribute name="serviceURI" value="http://www.isi.edu/..."/>
   <attribute name="schemaURI" value="http://www.isi.edu/..."/>
  </notification>
  <notification name="Debug">
   <attribute name="GWFinish" value="Yes"/>
   <attribute name="Timestamp" value="*time2"/>
  </notification>
 </actions>
</rule>

</rulebase>
```

Figure A.2: Event Distiller example.

Appendix B

Well-Known Ports

I briefly list here port values and their corresponding services. This information is used in the analysis in section 6.4.8. A more complete list can be obtained via `/etc/services` on a UNIX machine, via `C:\WINDOWS\SYSTEM32\DRIVERS\ETC\PORTS` on a Windows machine, or via IANA [65].

Port	Transport	Service
22	TCP	ssh
25	TCP	SMTP
53	UDP	DNS
80	TCP	HTTP
113	TCP & UDP	ident protocol
135	TCP	Windows RPC
137	UDP	NetBIOS Name Service (Windows name lookup)
139	TCP	NetBIOS Session Layer (Windows file sharing)
443	TCP	HTTPS (SSL)
445	TCP	Microsoft (Windows) Domain Services
1026	UDP	Windows Messenger Service [95]
1027	UDP	- (additional Messenger port)
1028	UDP	- (additional Messenger port)
1080	TCP & UDP	SOCKS proxy protocol
1433	TCP & UDP	Microsoft SQL
1434	TCP & UDP	Microsoft SQL (monitor)
3128	TCP	Squid HTTP proxy
8080	TCP	HTTP (popular alternate port)
33435+	UDP	Van Jacobsen traceroute