Language-Based Generation and Evaluation of NIDS Signatures

Shai Rubin, Somesh Jha, and Barton P. Miller
University of Wisconsin, Madison
Computer Sciences Department
{shai,jha,bart}@cs.wisc.edu

Abstract

We present a methodology to automatically construct robust signatures whose accuracy is based on formal reasoning so it can be systematically evaluated.

Our methodology is based on two formal languages that describe different properties of a given attack. The first language, called a session signature, describes temporal relations between the attack events. The second, called an attack invariant, describes semantic properties that hold in any instance of the attack. For example, an invariant may state that a given FTP attack must include a successful FTP login and can be launched only after the FTP representation mode has been set to ASCII. We iteratively eliminate false positives and negatives from an initial session signature by comparing the signature language to the language of the invariant.

We developed GARD, a tool for session-signature construction, and used it to construct session signatures for multi-step attacks. We show that a session signature is more accurate than existing signatures.

1 Introduction

A misuse Network Intrusion Detection System (NIDS) defines an attack via an *attack signature*: typically, a regular expression that matches a pattern of the attack [21, 26]. Ideally, each time an ongoing activity matches an attack signature, the NIDS raises an alarm. Ultimately, the security that a NIDS provides depends primarily on the accuracy of its signatures.

Conceptually, a signature represents a single attack, a set of events that exploit a given vulnerability. In practice, however, a single attack appears in many forms, or *attack instances*. For example, the *pro-ftpd* attack requires four FTP commands that can be ordered in different ways (CAN-2003-0831 [4, 17]). Thus, a signature is essentially a concise representation of a large set of attack instances.

Construction of an accurate signature is a daunting task. Usually, security analysts inspect a few exemplary attack instances, hypothesize the properties that must hold in any attack instance, and write down an expression that seems to match these properties. The analysts have no systematic way to either identify false positives, cases in which the signature over-approximates the set of attack instances, or false negatives, cases in which the signature under-approximates this set. Furthermore, there is no systematic way to evaluate the impact of changes that the analysts perform to improve the signature's accuracy. While a single change might fix a false positive, it might uncover a false negative. The end result is an ad-hoc, time consuming, and error prone process. It is not surprising that current signatures are inaccurate, producing many false positives [7, 31] and negatives [10, 25, 35].

In this paper we develop a systematic method to construct accurate signatures. The main idea is simple. First, we construct an initial signature for the given attack. Second, we build another, independent representation of the set of attack instances that the signature should match. Third, we compare the signature to this independent representation. This comparison usually reveals false positives and negatives, so we manually refine the signature. We repeat this process until we are satisfied with the signature accuracy.

The signature we develop in this paper is called a *session signature*. Like existing signatures, it is based on a regular language that models the attack events (e.g., FTP messages). However, a session signature models the entire attack, from the attacker's initiation of the connection to the victim's indication of whether the attack succeeded. So, compared to current signatures, it enables precise definition of the temporal relations between all of the events of the attack. As a result, a session signature is potentially more accurate than current signatures (Section 5.1).

To refine the accuracy of a session signature, we develop the notion of an *attack invariant*, an over-approximation of the set of attack instances. The invariant describes semantic properties that are necessary conditions for the attack. For example, the property “successful FTP login is required before launching the attack” is an invariant of the *pro-ftpd* attack mentioned above. Semantic properties enable us to express knowledge about the protocol the attack uses. Usually, these properties are not easily expressed using the signature.

To bridge between a session signature, which represents the attack’s syntactic features, and an attack invariant, which represents semantic features, we develop a *semantic model of a protocol*. It is a finite state machine that specifies how a protocol command changes a set of variables whose values define the *protocol state*. For example, our FTP model contains a `type` variable that tracks the representation mode of an FTP session (either ASCII or BINARY) and describes how the `TYPE` command affects this variable.

This semantic model serves two goals. First, we use it to define the invariant as a set of machine states. For example, since the *pro-ftpd* attack only succeeds in ASCII mode, a *pro-ftpd* invariant is the set of states in which `type` is set to ASCII. Second, we use the model to compute the protocol state for a given sequence of commands, so we can determine whether the invariant holds for that sequence.

Essentially, a session signature and an attack invariant are two independent representations of a set of attack instances. Formally, each of them represents a set of sequences of events. To find false positives and negatives, we compare the set of sequences represented by the signature to the set represented by the invariant. To find a false positive, we search for a sequence that matches the signature but does not satisfy the invariant. Analogously, to find a false negative we search for a sequence that does not match the signature but satisfies the invariant (Figure 1).

Since the invariant over-approximates the set of real attack instances, each sequence that is part of the signature but is not part of the invariant must be false positive. When we find such a false positive we refine the signature. Unfortunately, for the same reason, a sequence that does not match the signature but satisfies the invariant is not necessarily a false negative. It might be a *spurious sequence*: a sequence that does not match the signature, satisfies the invariant, but does not really implement the attack. Hence, we manually distinguish between false negatives and spurious sequences. When we find a false negative, we refine the signature; when we find a spurious sequence, we refine the invariant. We repeat the search for false positives and negatives until we can no longer refine the signature or the invariant, or we are satisfied with the signature’s accuracy.

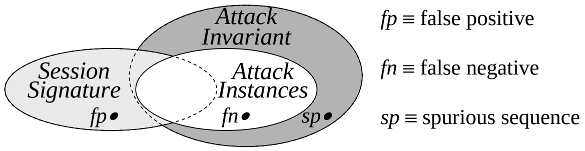


Figure 1. Searching for false positives and negatives.

Obviously, finding false positives and negatives improves our signature accuracy; furthermore, we discovered that spurious sequences also contribute to the accuracy. Since they force us to refine the invariant, we better understand the necessary conditions for identifying the attack. This understanding is the key for building highly accurate signatures.

However, we like to find more false negatives than spurious sequences. To do so, we develop a novel searching strategy. Our search for false negatives starts with sequences of events that do not match the signature but are similar (in terms of string matching) to the sequences that do match it. Research shows that such sequences are likely to be real attacks [15, 27, 35], and thus likely to be false negatives of a signature. Indeed, we show (Section 5.2.3) that this searching strategy is effective for finding false negatives.

To carry out this search, we implement GARD: a tool for <u>G</u>eneration, <u>A</u>nalysis, <u>R</u>efinement, and <u>D</u>eployment of NIDS signatures. GARD uses the semantic model to formalize the notion of an attack invariant and then compares the signature’s regular language to the invariant.

We empirically evaluated GARD’s capabilities. We show that a session signature of a simple attack is more accurate than its Snort [26] and contextual [31] counterparts. We show that, with respect to our attack invariants, our signature produces the least false alarms (none) and that it does not miss any attack instance that the other signatures recognize. We show that GARD is capable of generating a signature for a complex attack [4], an attack that requires multiple steps to succeed. We illustrate an iterative process in which we use GARD to systematically uncover a signature’s inaccuracies.

GARD does *not* guarantee discovery of *all* false positives and negatives. It finds false positives and negative with respect to the attack invariant we use. Defining meaningful invariants is an art based on human expertise. Our experience shows, however, that even with simple invariants, GARD is capable of finding false positives and negatives that we did not anticipate. Furthermore, we believe that the majority of semantic properties required by an invariant can be formally specified using the notion of our semantic model.

Even when we cannot overcome all signature limitations, for example for signatures that require non-regular languages, a priori knowledge of signature limitations is as important as the ability to generate a robust signature. Based on this knowledge, a signature user can make rational decisions about the risks that their system faces. If necessary, the user might address these risks using other security means.

In summary, this paper makes the following contributions.

1. **A session signature** that models the entire attack as a regular language. We show that the accuracy of a session signature can be systematically improved. We show that a session signature, at least for the attacks we analyzed, is more accurate than current signatures.
2. **An attack invariant**, another representation of the attack that is used to evaluate a session signature. It is based on a novel semantic model of the attack protocol.
3. **GARD**, a tool for automatic evaluation and generation of session signatures. We show, based on empirical evaluation, that the signatures GARD produces are superior to current signatures.

2 Related Work

GARD is centered around a combination of capabilities: modeling the entire attack, a signature-specification language based on language operators, a formal representation of the protocol semantics, and a methodology to evaluate a signature accuracy. We review other research and tools with respect to those capabilities.

Signature specification languages. Snort, a widely-used NIDS [3, 28], represents a signature using a set of attributes: packet attributes, like a packet length, and pattern attributes, like a regular expression defined over the attack bytes. A Snort signature corresponds to a single attack event; it does not (and probably cannot) model the entire attack since Snort does not facilitate composition of rules (except for the ability to dynamically invoke rules for logging purposes).

Bro [21] bases its detection on *policy scripts* rather than signatures. A policy script determines the actions (e.g., alerts) that should be taken based on the events Bro identifies (e.g., a sequence of FTP commands). In general, such a script can model the entire attack as a regular language; however, this requires programming in an imperative language similar to C, a fact that impairs the ability to easily define a signature and to analyze a signature's accuracy. Recently, Sommer and Paxson used Bro to implement a *contextual signature* [31] that enables sequencing of events in a declarative way. However, a contextual signature does not model the entire attack and supports only sequencing operators. As our results show (Section 5.1), sequencing alone is not enough for constructing tight signatures.

STATL [8] is a signature specification language for NetSTAT [34]; it represents a signature as a state diagram that describes the sequence of events in the attack. STATL does not support forming regular expressions over events, construction of a state diagram from a signature specification, or evaluation of the signature's accuracy. Since GARD uses a state machine as an intermediate signature representation, it should be easy to translate a GARD signature into STATL; in this work, however, we do not pursue this issue.

Sekar et al. [29, 30] developed a signature specification language based on regular expression over events. However, they did not specify the entire attack, a property that is important for comparing signature accuracies (Section 5.1). While their focus is network-level attacks (e.g., TCP SYN-flood) and ours is application-level, their language and GARD's language can be used to specify both attack types. They do not address the question of signature accuracy and do not suggest any means to evaluate it. Since they also translate their signatures into a finite state machine, we believe that their work can benefit from our evaluation methodology.

Pauzol and Ducassé [23, 24] proposed a language called Sutekh. While their focus was host-based intrusion detection, their language has features similar to ours. They model the attack using sequencing operators and represent a signature using a finite state machine.

LAMBDA [6] and ADeLe [16] also provide abstraction over events. These languages are more expressive than our session-signature language, mainly because they are general enough to express both network and host-based attacks. For example, they provide the ability to define parallel execution of events, something that is usually not required in network-based attacks. These languages, like our language, provide the ability to define the attack preconditions and post-conditions. We believe that translating signatures between our language and these languages should be an easy task. Like the research mentioned above, this work does not provide means to evaluate signature accuracy.

Currently, GARD does not support some features found in other tools: signatures that require detection based on properties of network packets, timers for event scheduling, or explicit event counters (one can count events using regular expression, but this is inefficient). In this work, we focused on the foundational concept behind GARD, using formal language tools for signature specification and evaluation. We believe that the missing features can be integrated into GARD and we plan to do so in the future.

Signature evaluation methodologies. Current techniques for signature evaluation are based on testing (e.g., [19, 25, 35]) and benchmarking (e.g., [14, 33]). To the best of our knowledge, we are the first to apply formal verification techniques for signature evaluation. Fundamentally, verification and testing complement each other. Our verification process strives to uncover signature vulnerabilities or to show their absence with respect to an abstract model of the protocol semantics. Testing, on the other hand, aims to uncover bugs by exercising a signature, usually with real network traffic.

Wagner and Soto [36] applied formal methods to find vulnerabilities in signatures for host-based IDS. They intersect the language that a signature accepts with a language that models the attack and manually construct an instance that evades the signature (false negative). However, they do not provide automatic methods for constructing either false positives or negatives. On the other hand, GARD uses techniques like an attack invariant to automatically perform these tasks.

3 GARD Overview

We illustrate how to use GARD to construct and evaluate a session signature. For these purposes, we chose a simple example attack, called *ftp-cwd*. In Section 5, we show how GARD handles more complex attacks.

The *ftp-cwd* attack (CAN-2002-0126 [17]) exploits a vulnerability in the BlackMoon FTP server for Windows [5]. The attack requires two steps: the attacker logs into the FTP server (e.g., using anonymous login) and then causes a buffer overflow by providing an overly-long argument for the FTP `CWD` (change directory) command. The attacker gains root privileges on the host and communicates with the compromised host through the FTP control port (i.e., port 21).

To define a signature for *ftp-cwd* we perform the following steps:

1. **Signature specification.** We construct a pattern that matches the sequence of events required for the attack detection. This sequence contains the events that occur during the entire attack, from the attacker's initiation of the connection to the victim's indication of whether the attack succeeded.
2. **Signature evaluation.** We search for signature loopholes: a sequence of FTP events that is either a false positive or false negative. To do so, we first construct an *ftp-cwd* invariant and then use GARD to automatically compare the signature's language to the invariant's language. This is an iterative process that continues until we decide that the signature accuracy is satisfactory or until we can no longer refine the signature or the invariant.

To define a session signature, it is important to understand how the signature is matched in practice.

Matching a session signature requires two components. The *lexical scanner* translates the raw network traffic into a stream of events. For example, the scanner identifies `CWD` commands required by the *ftp-cwd* signature. The *matching engine* matches the stream of events it gets from the scanner against the pattern in the signature specification. To keep the signature specification as simple as possible, we assume that the lexical scanner does not pass to the matching engine any event that is not part of the signature specification. For example, the FTP `STORE` command is irrelevant to the detection of *ftp-cwd*, so the scanner does not pass its token to the matching engine. We also assume that network-level protocols (e.g., IP, TCP) are handled by the NIDS and not by the matching process. For example, we assume that when a TCP connection is aborted, the matching engine halts.

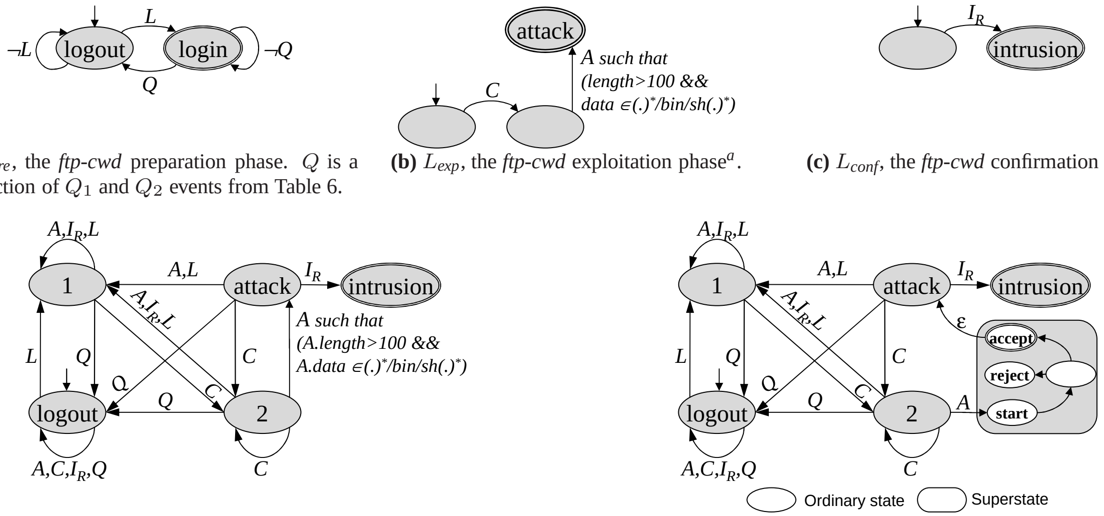
3.1 Signature Specification

A session signature is based on a three-phased abstraction: *preparation*, in which the attacker sets up the attack preconditions; *exploitation*, in which the attacker launches the attack; and *confirmation*, in which the attacker determines whether the attack succeeded or not.

To construct a session signature, we define the events in each phase (i.e., the FTP commands and their arguments) and use them to form a regular expression, the *phase language*. A session signature is a concatenation of the three phase languages.

***ftp-cwd* events.** Intuitively, an event corresponds to a protocol message or a part of it. More formally, an event is an observable sequence of bytes that is part of the attack; it contains bytes sent by either the attacker or the victim (but not both). GARD represents an event using a lexeme, a regular expression that matches the sequence of bytes, and a token that uniquely identifies the event (as done in lexical scanners, e.g., Flex [20]).

We identified five events in *ftp-cwd*. In the preparation phase, we identified the L event, which corresponds to a response of an FTP server confirming a successful login, and Q, which corresponds to an attacker attempt



(a) L_{pre}, the *ftp-cwd* preparation phase. Q is a disjunction of Q_1 and Q_2 events from Table 6.

(b) L_{exp}, the *ftp-cwd* exploitation phase[a].

(c) L_{conf}, the *ftp-cwd* confirmation phase.

(d) M_{main} machine: concatenation of the three phases of *ftp-cwd*.

(e) $SS_{ftp\text{-}cwd}$ machine: an HSM for the *ftp-cwd* attack (superstate not fully detailed). Note that the alphabet of a superstate (bytes) is different than the one of the main machine (events).

[a] We use "`/bin/sh`" as an example. In reality, the attacker may not necessarily target "`/bin/sh`". GARD's libraries contain other expressions that an analyst can use to build the *ftp-cwd* explotation phase.

Figure 2. From an *ftp-cwd* specification to an operational *ftp-cwd* session signature.

to logout from an FTP session. The exploitation phase contains the C event, which corresponds to an FTP `CWD` command, and the A event, which corresponds to an argument of a `CWD` command. Finally, to confirm the intrusion, we identified the I_R event; it corresponds to an *Invalid Response* from an FTP server: a message that cannot be part of a legal FTP response. I_R indicates a compromised server because it cannot be sent by a well-behaved FTP server. I_R matches, for example, a response to a `UNIX id` command, which the attacker uses in *ftp-cwd* to check whether the attack succeeded.

In most cases, there is no need to define new events; GARD contains a library of predefined events for common protocols (e.g., FTP, HTTP). For example, based on the FTP specification [22], our FTP library (Table 6) contains an event definition for every event in the *ftp-cwd* specification.

***ftp-cwd* phase languages.** To explain the process of signature specification and the underlying compilation method of GARD, we present a phase language using a finite state machine. The expression for the languages are given in Section 4.1.

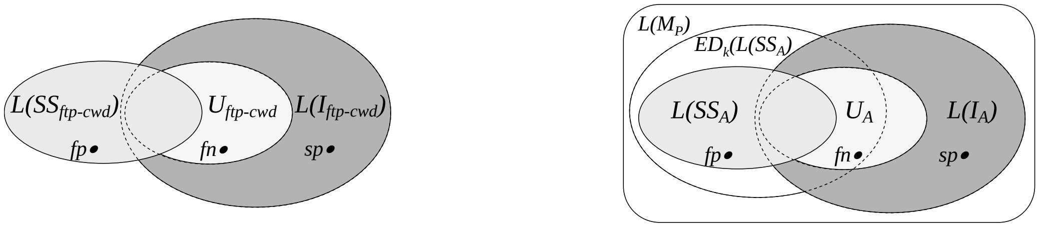
The *ftp-cwd* preparation language, denoted L_{pre}, ensures that an attacker logged into the server before launching the attack. To do so, it imposes two conditions (Figure 2a): (i) the attacker has successfully completed a login procedure, and (ii) the attacker has not yet executed a logout procedure. As we show in Section 5.1, signatures that do not ensure both condition (e.g., Snort *ftp-cwd*'s signature) generate false positives. Since this language (Figure 2a) is common to many FTP attacks, we added it to GARD's library of FTP patterns.

The *ftp-cwd* exploitation language, denoted L_{exp}, concatenates the A event to the C event. However, the concatenation of C and A matches any `CWD` command, even benign ones. Hence, we further restrict the content of a malicious A by specifying that the malicious `CWD` is followed by an argument that is longer than 100 bytes and contains the string "`/bin/sh`" (Figure 2b). GARD supports restrictions on the event *length*, using relational operators (e.g., $>$,$\leq$), and on the event *data*, using regular expressions. So, it is a straightforward matter to translate an event restriction into a regular language.

The *ftp-cwd* confirmation phase, denoted, L_{conf}, contains a single *Invalid Response* event (Figure 2c).

Generating a working signature. Defining a regular language for each phase completes the signature specification. GARD compiles this specification into an intermediate representation. To better explain the signature evaluation process described next, we provide a short summary of this process and the intermediate representation GARD uses. Section 4.2 presents the detailed compilation process.

GARD's compilation process contains two steps.



(a) The main concepts of signature evaluation illustrated through the *ftp-cwd* attack.

(b) Using the semantic model of a protocol to implement and define the search for false positives and negatives.

Language	**Definition**
$L(M_{\mathcal{P}})$	$\Sigma^{\star}_{\mathcal{P}}$: the set of possible sequences of events from protocol $\mathcal{P}$.
$U_{\mathcal{A}}$	Sequences that implement $\mathcal{A}$ (the set we try to unambiguously define).
$L(SS_{\mathcal{A}})$	Sequences that the session signature of $\mathcal{A}$ accepts.
$L(I_{\mathcal{A}})$	Sequences that satisfy $\mathcal{A}$'s invariant.
$ED_k(L(SS_{\mathcal{A}}))$	Sequences that are k-edit distance from $L(SS_{\mathcal{A}})$.
False positive	$fp \in \{L(SS_{\mathcal{A}}) \cap \neg U_{\mathcal{A}}\}$
False negative	$fn \in \{\neg L(SS_{\mathcal{A}}) \cap U_{\mathcal{A}}\}$
Spurious sequence	$sp \in \{\neg L(SS_{\mathcal{A}}) \cap (L(I_{\mathcal{A}}) \setminus U_{\mathcal{A}})\}$

(c) Formalizing signature evaluation as a comparison between regular languages.

Figure 3. The fundamentals of language-based generation of session signatures.

First, GARD concatenates the phase languages into a *main machine* (Figure 2d). Second, GARD embeds the restrictions on events into the main machine as separate finite state machines (restrictions are regular languages, so each restriction can be represented as an FSM). To do so, GARD builds a (non-recursive) Hierarchical State Machine (HSM), a machine whose states are either ordinary states or *superstates* that are FSM themselves [2]. In the *ftp-cwd* case, the superstate imposes the restrictions on A's data and length: in state 2, if the superstate accepts the restrictions, the main machine moves to the *attack* state, else it moves to state 1 (Figure 2e).

It is a straightforward matter to translate an HSM into a working signature. For example, it is possible to translate an HSM into a Snort [26] plugin or Bro's contextual signature [31]. Furthermore, for signature evaluation purposes, it is also easy to translate an into a Spin model (Section 4.4).

3.2 Signature Evaluation

Ultimately, our task is to construct a signature that matches every FTP session that implements the *ftp-cwd* attack. Formally, we denote the set of FTP sessions that implement *ftp-cwd* as $U_{\textit{ftp-cwd}}$, called the *ultimate set* of *ftp-cwd*. We denote the set of sessions that matches our session signature (Figure 2e) as $L(SS_{\textit{ftp-cwd}})$. An ideal signature is a signature such that $L(SS_{\textit{ftp-cwd}}) = U_{\textit{ftp-cwd}}$. A non-ideal signature generates false positives and negatives. A false positive, denoted *fp*, is a sequence in $\{L(SS_{\textit{ftp-cwd}}) \cap \neg U_{\textit{ftp-cwd}}\}$ and a false negative, denoted *fn*, is a sequence in $\{\neg L(SS_{\textit{ftp-cwd}}) \cap U_{\textit{ftp-cwd}}\}$ (Figure 3a).

When the ultimate set is easy to define, for example by a regular language, we can use it as a signature. Unfortunately, in most cases, the ultimate set is difficult to define. For example, we thought that it would be easy to construct a session signature that matches $U_{\textit{ftp-cwd}}$, but GARD found a false positive. The ultimate set for more complex attacks is even more difficult to define (Section 5.2.1).

To systematically find false positives and negatives, we approximate the ultimate set by using an attack invariant, a predicate that must hold in every instance of the attack. For example, we define the *ftp-cwd* invariant as "a login procedure must be completed before the attacker sends the malicious `CWD` command"; such an invariant can be expressed as a regular language, denoted $L(I_{\textit{ftp-cwd}})$. Our evaluation methodology assumes that an invariant over-approximates the ultimate set, that is, $L(I_{\textit{ftp-cwd}}) \supseteq U_{\textit{ftp-cwd}}$. We discuss the reasons for such an over-approximation in Section 4.3.

Given $L(I_{\textit{ftp-cwd}})$, we find a false positive by searching for a sequence in $\{L(SS_{\textit{ftp-cwd}}) \cap \neg L(I_{\textit{ftp-cwd}})\}$. Since $L(I_{\textit{ftp-cwd}}) \supseteq U_{\textit{ftp-cwd}}$, if $fp \in \{L(SS_{\textit{ftp-cwd}}) \cap \neg L(I_{\textit{ftp-cwd}})\}$, then $fp \in \{L(SS_{\textit{ftp-cwd}}) \cap \neg U_{\textit{ftp-cwd}}\}$, that is, *fp* is a false positive (Figure 3a).

To find a false negative we search for a sequence in the set $\{\neg L(SS_{\textit{ftp-cwd}}) \cap L(I_{\textit{ftp-cwd}})\}$. However, since $L(I_{\textit{ftp-cwd}}) \supseteq U_{\textit{ftp-cwd}}$, our search may yield a *spurious* sequence which satisfies the invariant but does not implement *ftp-cwd*; formally, a sequence in $\{\neg L(SS_{\textit{ftp-cwd}}) \cap (L(I_{\textit{ftp-cwd}}) \setminus U_{\textit{ftp-cwd}}\}$ (Figure 3a). Hence, each time we find a sequence in $\{\neg L(SS_{\textit{ftp-cwd}}) \cap L(I_{\textit{ftp-cwd}})\}$, we manually check whether this is a spurious instance.[1] If it is, we refine the invariant and continue searching. We discuss techniques of guiding the search toward false negatives rather than spurious sequences in Section 4.3; we illustrate the invariant refinement process in Section 5.2.3.

A false positive in $SS_{\textit{ftp-cwd}}$. We formally define $L(I_{\textit{ftp-cwd}})$ using our FTP semantic model (Appendix A). GARD compared $L(I_{\textit{ftp-cwd}})$ to $L(SS_{\textit{ftp-cwd}})$ and found a false positive: a sequence of FTP commands that matches the signature but in which the malicious `CWD` appears before a completed login procedure.

Our *ftp-cwd* specification ignores a victim (the FTP server) that voluntarily terminates the connection (event `VQUIT` in Table 6). The false positive includes such an event injected before the malicious `CWD`. An attacker that intentionally ignores this terminating message can continue to send the malicious `CWD`, causing a false alarm. It is an open question whether to include this event in the *ftp-cwd* signature; it is unclear whether attackers can exploit this weakness. In any case, GARD can generate two versions of the signature, one that incorporates the event and one that ignores it. Signature users can choose the version that fits their needs.

The important lesson from the *ftp-cwd* example is not the weakness we revealed in the *ftp-cwd* signature, but the systematic way in which we found the weakness. We illustrated that formal methods and tools can help us systematically construct NIDS signatures, signatures whose quality can be evaluated and understood. We further discuss GARD's advantages and disadvantages in the next section.

4 GARD's Foundations

We discuss GARD's signature-specification language, the algorithm it uses to translate a signature specification into an hierarchical state machine, and its methodology for finding false positives and negatives.

4.1 Signature Specification Language

The goal of GARD's specification language is to provide a clean separation between event representation and the ability to construct regular languages over events. To achieve this goal, we use one language for event representation and one for constructing regular expressions.

We represent events using regular expressions over raw network bytes. In practice, we use the Flex language [20] to form the expressions. Since an attack might contain events from multiple streams (e.g., one stream for the messages sent by the attacker and one for the victim responses), we annotate each event with its corresponding stream. To identify streams, we use the common convention of IP addresses and port numbers [26, 21]. For brevity, in our FTP model (Appendix A) we specify the stream of an event by using only the sender: a subscript a to denote the attacker and v to denote the victim.

To form a regular expression over events, GARD uses standard operators for language manipulation (Table 7). The only non-standard operator is the `such_that` operator; it is used to restrict the data or length of an event (as in Figure 2b).

Table 1 presents a complete signature specification for *ftp-cwd*. The languages for the three phases, $L_{\textit{pre}}$, $L_{\textit{exp}}$, and $L_{\textit{conf}}$, correspond to the state machines presented in Figure 2a to Figure 2c.

4.2 Compilation Process

Compiling a signature specification (e.g., Table 1) into its corresponding HSM (e.g., Figure 2e) is based on a standard algorithm for translating a regular expression into a finite state machine [12]. The compilation process contains three steps:

1. We use the standard algorithm to translate each `such_that` expression into an FSM, called a *superstate*. For example, we translate the `such_that` expression in the *ftp-cwd* specification into a machine that identifies a string that contains the pattern "`/bin/sh`" and is longer than 100 bytes.

2. We use the standard algorithm to build $M_{\textit{main}}$ (Figure 2d). Formally, we build an $M_{\textit{main}}$ that accepts the language $L_{\textit{pre}} \cdot L_{\textit{exp}} \cdot L_{\textit{conf}}$.

3. We embed the superstates into $M_{\textit{main}}$, obtaining a session signature, an HSM denoted $SS_{\textit{attack-name}}$ (e.g., Figure 2e).

While we build $M_{\textit{main}}$ using the standard algorithm, recall that this algorithm does not handle the non-standard `such_that` operator. Hence, we perform *restricted-event renaming*. We replace each restricted event, a token restricted with the `such_that` operator (Table 7), with a unique identifier. For example, we convert $L_{\textit{exp}}$ in Table 1 from $C \cdot (A \textit{ such\_that data} \in$ `shell` && $\textit{length} > 100)$ into the expression $C \cdot \hat{A}$.

[1] We can automate this process by launching the instance on a vulnerable host [35]. We leave this implementation issue for future work.

Phase	Signature	Description
Preparation (L_{pre})	$login_{ftp}$	A macro denoting a regular expression that matches any FTP session after a successful login. The macro defines a regular expression using tokens L and Q: $((\neg L)^{\star} \cdot L \cdot (\neg Q)^{\star})^{+}$
Exploitation (L_{exp})	$C \cdot (A$ *such_that* *data* $\in (.)^{\star}$`bin/sh`$(.)^{\star}$ && *length* $> 100)$	A malicious `CWD` command whose argument is longer than 100 bytes and contains the pattern `bin/sh`. `bin/sh` is an example; in practice, other patterns from the exploit code can be used.
Confirmation (L_{conf})	I_R	The *Invalid Response* event indicates that the connection is no longer used as an FTP connection.

Table 1. A signature specification for *fpd-cwd*.

Renaming a restricted event is not just syntactic sugar; it actually preserves the signature semantics. Essentially, a restricted event represents a unique sequence of bytes: the restricted event "A *such_that data* $\in$ `shell` && *length* > 100" represents a different sequence of bytes than the event "A" (Table 6). Hence, renaming enables the standard algorithm to distinguish between restricted events and their unrestricted versions (e.g., A and $\hat{A}$) in the same way it distinguishes between any other two events (e.g., A and C).

Renaming also enables us to identify the places in the main machine in which we need to embed the superstates. After step 2 above, each edge in a main machine that is labeled with a renamed event should be replaced with its corresponding superstate.

4.3 Signature Evaluation Algorithms

Let $SS_{\mathcal{A}}$ be the session signature of an attack $\mathcal{A}$, such as $SS_{\textit{ftp-cwd}}$ in Figure 2e. We evaluate a session signature by comparing its language, denoted $L(SS_{\mathcal{A}})$, to the language of $\mathcal{A}$'s invariant, denoted $L(I_{\mathcal{A}})$.

To define an invariant for an attack that uses protocol $\mathcal{P}$, we represent the protocol semantics using the *semantic model of* $\mathcal{P}$: a finite state machine, denoted $M_{\mathcal{P}}$. A state in $M_{\mathcal{P}}$ is a valuation of variables that are called the protocol's *state variables*. A transition describes how an event, usually corresponding to a protocol message, affects the variable values. $M_{\mathcal{P}}$ is defined over $\Sigma_{\mathcal{P}}$, a set of protocol events. Essentially, the language that $M_{\mathcal{P}}$ represents is $\Sigma_{\mathcal{P}}^{\star}$. An example model for FTP is given in Appendix A.

An attack invariant $I_{\mathcal{A}}$ is a logical formula over the state variables of $M_{\mathcal{P}}$. The language $L(I_{\mathcal{A}})$ is the language accepted by an $M_{\mathcal{P}}$ whose accepting states are the states in which $I_{\mathcal{A}}$ holds.

As mentioned in Section 3.2, our goal is to construct a signature that is as close as possible to $U_{\mathcal{A}}$, the ultimate set of $\mathcal{A}$. Unfortunately, in most cases, there is no clear definition of $U_{\mathcal{A}}$. Therefore, we evaluate the accuracy of $L(SS_{\mathcal{A}})$ using $L(I_{\mathcal{A}})$. Even though both $L(SS_{\mathcal{A}})$ and $L(I_{\mathcal{A}})$ are only approximations of $U_{\mathcal{A}}$, our results show that such a comparison is an effective way to improve the accuracy of $L(SS_{\mathcal{A}})$.

We require that $L(I_{\mathcal{A}}) \supseteq U_{\mathcal{A}}$, that is, an attack invariant should be a necessary condition for the attack to occur (Figure 3b). We require this for two reasons. First, necessary conditions are usually easy to define, facilitating fast signature construction. For example, it is easy to see that a successful FTP login is a necessary condition for the *ftp-cwd* attack. Second, since false negatives are considered more harmful than false positives, by over-approximating $U_{\mathcal{A}}$ we ensure that, theoretically at least, we will never miss a false negative.

To find a false positive, we search for a sequence in $\{L(SS_{\mathcal{A}}) \cap \neg L(I_{\mathcal{A}})\}$. As mentioned in Section 3.2, since $L(I_{\mathcal{A}}) \supseteq U_{\mathcal{A}}$, such a sequence must be a false positive. Analogously, to find a false negative we search for a sequence in the set $\{\neg L(SS_{\mathcal{A}}) \cap L(I_{\mathcal{A}})\}$. However, since $L(I_{\mathcal{A}}) \supseteq U_{\mathcal{A}}$, our search may yield a *spurious* sequence, a sequence that satisfies the invariant but does not implement $\mathcal{A}$. Hence, each time we find a sequence in $\{\neg L(SS_{\mathcal{A}}) \cap L(I_{\mathcal{A}})\}$, we check whether this is a spurious instance. If it is, we refine the invariant and continue searching.

Since the search for false negatives involves human intervention, we would like to avoid spurious sequences. The problem becomes even more serious when $L(I_{\mathcal{A}})$ is much larger than $U_{\mathcal{A}}$, yielding many spurious sequences. For example, $L(I_{\textit{ftp-cwd}})$ (Figure 3a) contains all sequences of FTP commands in which FTP login has been completed. This set is much larger than $U_{\textit{ftp-cwd}}$, the set of sequences that implement *ftp-cwd*.

To reduce the probability of hitting a spurious sequence, we search the set $\{\neg L(SS_{\mathcal{A}}) \cap L(I_{\mathcal{A}})\}$ for sequences that are similar, in terms of string matching, to sequences in $L(SS_{\mathcal{A}})$. This strategy is based on the observation that new attack instances can be generated by introducing small changes to already-known instances of the attack and that a signature represents such known instances. This observation forms the basis for many NIDS testing tools [10, 15, 19, 25, 35].

We formalize similarity using the notion of *edit distance* [1]. The edit distance between strings s_1 and s_2, denoted $ed(s_1, s_2)$, is the number of insertions, deletions, or substitutions required to transform s_1 into s_2. For a language L we define its k-*edit-distance* language, denoted $ED_k(L)$, as the set of strings such that their edit distance from a string in L is less than k. Formally, $ED_k(L) = \{x | \exists y \in L \textit{ such that } ed(x, y) < k\}$. It is well known that if L is a regular language, then $ED_k(L)$ is also regular [9, 13, 32]. Hence, $ED_k(L(SS_{\mathcal{A}}))$ forms another regular language, a superset of $L(SS_{\mathcal{A}})$.

4.4 GARD: Summary and Pitfalls.

To find false positives, GARD checks whether the set $\{L(SS_{\mathcal{A}}) \cap \neg L(I_{\mathcal{A}})\}$ is empty. To find false negatives, GARD checks whether the set $\{\neg L(SS_{\mathcal{A}}) \cap L(I_{\mathcal{A}}) \cap ED_k(L(SS_{\mathcal{A}}))\}$, for some constant k, is empty.

To check the emptiness of the above sets, GARD uses Spin [11], a publicly available model checker. Since all of these sets are regular, it is straightforward to use Promela, the input language for Spin, to represent the sets as finite state machines. We use Spin because it is capable of not only checking for emptiness, but also of providing a sequence in these sets when they are not empty. This ability greatly simplifies the evaluation process.

When GARD asserts that $\{L(SS_{\mathcal{A}}) \cap \neg L(I_{\mathcal{A}})\} = \emptyset$, it means that, if we interpret events according to our semantic model, then we cannot use the events in the model to construct a false positive that matches the signature but violates the attack invariant.

For example, consider our *ftp-cwd* signature (Figure 2e), our FTP model (Appendix A), and the *ftp-cwd* invariant from Section 3.2. GARD asserted that $\{L(SS_{\textit{ftp-cwd}}) \cap \neg L(I_{\textit{ftp-cwd}})\} = \emptyset$ (after adding Q_3 to the signature). This means that we cannot construct a sequence of events from the events in Table 6 that matches our signature but in which an attacker is not logged in. If one believes, as we believe, that our model accurately describes all possible ways to login and logout from an FTP server, then an attacker cannot cause a false positive for our *ftp-cwd* signature without completing an FTP login procedure. Such a guarantee does not exist in current *ftp-cwd* signatures (Section 5.1).

GARD does *not* guarantee discovery of *all* false positives. GARD will not find a false positive that uses an event that is not part of the model, that is, $fp \notin L(M_{\mathcal{P}})$; in such a case, the semantics model should be refined. GARD will not find a false positive that satisfies the attack invariant, that is, $fp \in \{L(SS_{\mathcal{A}}) \cap (L(I_{\mathcal{A}}) \setminus U_{\mathcal{A}})\}$ (Figure 3b); when such a sequence surfaces, the invariant should be refined.

When GARD asserts that $\{\neg L(SS_{\mathcal{A}}) \cap L(I_{\mathcal{A}}) \cap ED_k(L(SS_{\mathcal{A}}))\} = \emptyset$, it means that, if we interpret events according to our semantic model, then there is no sequence of events, from our model, that implements $\mathcal{A}$ and is k-edit distance from a sequence that matches our signature.

GARD does *not* guarantee discovery of *all* false negatives. GARD will not find a false negative that is more than k-edits away from our signature. Also, GARD will miss a false negative that uses an event that is not part of the model, that is, $fn \notin L(M_{\mathcal{P}})$. In such a case, Figure 3b is inaccurate because $U_{\mathcal{A}} \not\subseteq L(M_{\mathcal{P}})$. There is nothing surprising here, because the vague nature of $U_{\mathcal{A}}$ is the motivation for our work. When such a false negative surfaces, the semantic model should be refined.

Implementation notes. To construct $ED_k(L_{sig}(\mathcal{A}))$ we used a recent methodology proposed by Kari et al. [13]. Their methodology enables us to define the errors that are permitted. In other words, it permits us to restrict the transformations attackers can perform. This feature is useful if we understand that not all transformations really preserve the attack semantics. For example, in the *ftp-cwd* attack one cannot delete or replace the `CWD` command with a different FTP command.

During signature evaluation we use the HSM without its superstates (the M_{main} machine rather than the $SS_{\textit{ftp-cwd}}$ machine from Figure 2). We assume that if a false negative or positive exists in M_{main}, there is also a sequence in M_{sig}. After all, if an attacker finds such a sequence for the M_{main}, they can construct it in a way that satisfies the restrictions imposed by the supernodes. Also recall that the lexical scanner drops any event that does not explicitly appear in the signature specification (Section 3). Since these dropped events appear as self-loops in M_{main}, during signature evaluation, we add these self loops to M_{main}.

5 GARD Evaluation

We evaluate the accuracy of a session signature and GARD's ability to find false positives and negatives in complex attacks. We prove that our *ftp-cwd* signature (Table 1) is more accurate than current signatures: not only it is not vulnerable to false positives (with respect to our invariant), but it does not miss any attack instance that the other signatures recognize. We also show that GARD is able to model complex attacks and to find false positives and negatives that we did not anticipate.

5.1 Session Signature Evaluation

Our initial study used GARD to identify known vulnerabilities in the *ftp-cwd* signatures that current NIDS use: a recent contextual signature developed by Sommer and Paxson [31] and a Snort [26] signature.

To compare the two signatures to ours, we first de-

Sig. name	$L(\mathit{SS}_{\textit{sig-name}})$ For brevity we write $L_{\textit{sig-name}}$	**HSM size**[a]			**false positive**	**Operational false positive**	**Comments**
		Alphabet	states	edges			
Snort	$(\Sigma_{\textit{ftp}})^{\star}(CA)(I_R)$	$\{C, A\}$	2	4	CA	yes	
CS	$((\neg L)^{\star}L(\Sigma_{\textit{ftp}})^{\star})(CA)(I_R)$	$\{C, A, L\}$	4	9	LQ_1CA	yes	
cwd$_1$	$((\neg L)^{\star}L(\neg Q)^{\star})^{+}(CA)(I_R)$	$\{C, A, L, Q_{1,2}\}$	4	12	LQ_3CA	no	Q=$\{Q_1, Q_2\}$
cwd$_2$	$((\neg L)^{\star}L(\neg Q)^{\star})^{+}(CA)(I_R)$	$\{C, A, L, Q_{1,2,3}\}$	4	15	-	-	Q=$\{Q_1, Q_2, Q_3\}$

[a] The size of a deterministic HSM without states and edges of superstates.

Table 2. Comparison of signatures and their weaknesses. $\Sigma_{\textit{ftp}}$ is the set of events from Table 6.

fined the Snort and contextual signatures as session signatures, denoted *Snort* and *CS*, respectively. Since the Snort signature does not include a confirmation phase, we added the I_R event (Section 3.1) as the confirmation phase for all signatures. Then, we used our FTP model (Appendix A) to define the invariant “a login procedure must be completed before the attacker sends the malicious `CWD` command”. Last, we used GARD to find false positives in each of the three signatures. GARD revealed false positives with which we were familiar and one with which we were not. We then refined our initial *ftp-cwd* signature and verified that, with respect to our invariant, it does not have any false positives.

ftp-cwd **Snort signature (signature 1919 revision 19 [26]).** Essentially, this is the exploitation phase of our session signature (Figure 2b). We knew that this signature is not tight: attackers can cause a false positive by sending the malicious `CWD` command before they have logged into the FTP server. Since this signature ignores events that precede the malicious `CWD`, we model its preparation phase as $(\Sigma_{\textit{ftp}})^{\star}$, where $\Sigma_{\textit{ftp}}$ is the set of FTP events in our model (Table 6).

ftp-cwd **contextual signature [31].** This signature ensures that a login event appears before the malicious `CWD`. We knew that this signature is not tight because an attacker can cause a false positive by sending an FTP `QUIT` command immediately before the malicious `CWD` but after they have completed the FTP login procedure. We defined the preparation-phase language as $(\neg L)^{\star}L(\Sigma_{\textit{ftp}})^{\star}$, where L is a successful-login event (Table 6). This means that after observing a single login event the signature moves to the exploitation phase.

ftp-cwd **session signature.** We used the signature we defined in Table 1, denoted *cwd*$_1$.

Evaluation summary. Table 2 presents a summary of the experiment results. Each row in the table presents one signature: Snort, *CS*, *cwd*$_1$, and *cwd*$_2$, which we constructed after GARD had revealed a weakness in *cwd*$_1$. For each signature, we show the language it accepts, the alphabet for the signature's HSM, the number of nodes and edges in the HSM, the false positive GARD found, and whether we were able to create an operational false positive from the sequence GARD provided. A few observations should be noted:

1. We can order the signatures by their accuracy: Snort is the least accurate signature and *cwd*$_2$ is the most accurate one. It is easy to prove that $L_{\textit{Snort}} \supset L_{\textit{CS}} \supset L_{\textit{cwd}_1} \supset L_{\textit{cwd}_2}$. This means that a less accurate signature always generates more false positives than a more accurate signature. It is also possible to prove that there is no sequence that implements *ftp-cwd* in $\{L_{\textit{Snort}} \backslash L_{\textit{CS}}\}$, $\{L_{\textit{CS}} \backslash L_{\textit{cwd}_1}\}$, and $\{L_{\textit{cwd}_1} \backslash L_{\textit{cwd}_2}\}$. This means that a more accurate signature never misses an attack instance that a less accurate signature matches. The formal proofs of these claims is beyond the scope of this paper.

2. The operational overhead of all signatures is low. All signatures require less than 20 edges, indicating insignificant memory footprints of their HSMs.

3. Using the ProFTPD server [18] and Bro, we could not generate an operational false positive with the sequence GARD provided for *cwd*$_1$. In this sequence, the server voluntarily terminates the FTP and TCP connections, using the Q_3 event. Since Bro stops monitoring an FTP session immediately after observing the terminating TCP sequence, it ignores the C and A events and does not generate a false alarm. Since the additional overhead of *cwd*$_2$, in terms of memory consumption, is insignificant, we believe that *cwd*$_2$ is preferable because it prevents this potential false positive in *cwd*$_1$.

5.2 Constructing Complex Signatures with GARD

In our pilot study we examined a relatively simple attack. Given its success there, we wanted to challenge GARD's capabilities with a much more complex attack. To do so, we chose the multi-step *pro-ftpd* attack.

The *pro-ftpd* attack (CAN-2003-0831 [4, 17]). The *pro-ftpd* attack exploits a buffer overflow in the ProFTPD [18] server. This vulnerability occurs when the attacker transfers a file in ASCII mode. During such

Phase	Signature	Description
L_{pre}	$(\mathit{login}_{\mathit{ftp}} \cap \mathit{type}_A) \cdot S$	$\mathit{login}_{\mathit{ftp}} \equiv ((\neg L)^\star L(\neg Q)^\star)^+$ where L is a SLOGIN event and Q is a QUIT event ($Q \equiv \{Q_1 \cup Q_2 \cup Q_3\}$,, see Table 6). We also used this pattern in the *ftp-cwd* signature (Figure 2a).
L_{exp}	$R \cdot R$	$\mathit{type}_A \equiv ((\neg T_a)^\star T_a(\neg T_b)^\star)^\star$ where T_a is a `TYPE A` event and T_b is a `TYPE B` event (Table 6). The intuition is that an ASCII mode requires a T_a event that is not followed by a T_b event.
L_{conf}	W	$W \equiv ((.)^\star)_{x::y \rightarrow \mathit{FTP}::4660}$. This is an event that identifies an opening of a `UNIX` shell on port 4660 of the FTP server. The pattern $(.)^\star$ matches any traffic that traverses this TCP session. x is the IP address of all events sent by the attacker. y is an arbitrary port number.

Table 3. Our initial signature of the *pro-ftpd* attack. This signature is susceptible to false positives and negatives which we fix in Section 5.2.2 and Section 5.2.3, respectively. Event definitions are given in Table 6.

a transfer, the ProFTPD server stores data in 1024 byte chunks to check for newline characters. Due to incorrect handling of these characters, a buffer overflow occurs when ProFTPD parses a specially crafted file. This attack requires four steps: The attacker (i) logs into the FTP server, (ii) changes the representation mode to ASCII, (iii) uploads, to the FTP server, a file that contains their shell code, and (iv) downloads the same file; during this download a buffer overflow occurs. In this attack, the attacker gains root privileges on the host, and then communicates with the compromised host through a new TCP connection opened by the shell code.

5.2.1 Initial *pro-ftpd* specification

The attack requires three preconditions: an FTP session both in the login and ASCII states, and a `STORE` command that uploads the attacker's file. To simultaneously impose login and ASCII states, we intersected the languages for $\mathit{login}_{\mathit{ftp}}$ and type_A (Table 3). To the resulting pattern, we concatenated the `STORE` event.

We modeled the two retrieval, or download, operations as the *pro-ftpd* exploitation phase (Table 3). The boundary between the preparation and exploitation phases can be set arbitrarily because it does not have any operational meaning. However, the boundary between the exploitation and confirmation phases signals the NIDS to raise the *attack* alert, so it is important to set it according to our interpretation of the attack. In the case of *pro-ftpd*, it seems clear that the second retrieval operation marks the end of the attack.

The confirmation phase of *pro-ftpd* consists of a single event, opening a `UNIX` shell. Unlike the case of *ftp-cwd*, the shell is opened on a new TCP connection using the server's port 4660. To identify this activity, we defined an event that matches any communication that occurs on this port.

5.2.2 *pro-ftpd* False Positives

Our *pro-ftpd* invariant states that any successful *pro-ftpd* attack must end in the login and ASCII states. We formally defined it using the $\mathit{FTP}_{\mathit{type}}$ and $\mathit{FTP}_{\mathit{login}}$ variables in our FTP model (Table 5).

To search for false positives and negatives, we must initialize the variable $\mathit{FTP}_{\mathit{type}}$ to the default value used by the ProFTPD server. Since the ProFTPD enables an administrator to determine the default mode, we performed the search twice: once with ASCII as the default and once with BINARY.

When the default representation mode is ASCII, GARD verified that our *pro-ftpd* specification has no false positives with respect to the given invariant. However, when the default is BINARY, GARD found a false positive: the sequence $\langle$`TYPE A`, `LOGIN`, `STORE`, `RETV`, `RETV`$\rangle$. This sequence matches the signature but ends in BINARY mode, so it does not implement the attack; the `TYPE A` has no effect because it appears before a login procedure. The sequence matches the signature because the intersection between the languages $\mathit{login}_{\mathit{ftp}}$ and type_A does not enforce the required order between the events `LOGIN` and `TYPE A` (see Figure 4a).

The problem is the type_A pattern. This pattern intends to ensure that a `TYPE A` event is not followed be a `TYPE B` one, that is $T_a(\neg T_b)^\star$. However, GARD instantiates $\neg T_b$ as the set $\{L, Q, S, R, T_a\}$. When BINARY is the default mode, performing a logout and then login (using the `QUIT` and `LOGIN` events) behaves like T_b: it switches the mode back to BINARY. Hence, we modified type_A into $((\neg T_a)^\star T_a((\neg T_b) \cup (\neg Q) \cup (\neg L))^\star)^\star$. After this change, GARD verified that no false positive exists when the default mode is BINARY.

5.2.3 *pro-ftpd* False Negatives

We illustrate an iterative process in which we found false negatives in our initial *pro-ftpd* signature. In each iteration, GARD provided a sequence that does not match the signature but satisfies our invariant. We manually

Round	Sequence Type (Figure 3b)	Attack Invariants	Invariant/Model Refinement	Signature Fix
1	original: $\langle T_a, L, S, R, R\rangle$ false neg: $\langle R, L, S, R, R\rangle$	$\mathit{FTP}_{\mathit{login}}$=*true* $\mathit{FTP}_{\mathit{type}} = A$	—	Fix $(\mathit{login}_{\mathit{ftp}} \cap \mathit{type}_A)$. Compare old (Figure 4a) to new pattern (Figure 4b).
2	original: $\langle L, S, R, R\rangle$ spurious: $\langle L, C, R, R\rangle$	$\mathit{FTP}_{\mathit{login}}$=*true* $\mathit{FTP}_{\mathit{type}}$=*A*	One `STORE` command must appear in any instance of the attack.	—
3	original: $\langle L, S, R, R\rangle$ spurious: $\langle L, S, C, R\rangle$	$\mathit{FTP}_{\mathit{login}}$=*true*, *st*=*1* $\mathit{FTP}_{\mathit{type}}$=*A*	Two `RETV` command must appear in any instance of the attack.	—
4	original: $\langle L, AP, S, R, R\rangle$ false neg: $\langle L, AP, C, R, R\rangle$	$\mathit{FTP}_{\mathit{login}}$=*true*, *st*=*1* $\mathit{FTP}_{\mathit{type}}$=*A*, *rt*=*2*	—	Replace $(\mathit{login}_{\mathit{ftp}} \cap \mathit{type}_A)S$ with $(\mathit{login}_{\mathit{ftp}} \cap \mathit{type}_A)(S \cup A)$.
5	original: $\langle L, R, AP, R, R\rangle$ spurious: $\langle L, R, AP, R, C\rangle$	$\mathit{FTP}_{\mathit{login}}$=*true*, *st*=*1* $\mathit{FTP}_{\mathit{type}}$=*A*, *rt*=*2*	Limit length of attacks. Limit attacker transformations.	—
6+	—	$\mathit{FTP}_{\mathit{login}}$=*true*, *st*=*1* $\mathit{FTP}_{\mathit{type}}$=*A*, *rt*=*2*, *length*<*k*, . . .	—	—

Table 4. A search for false negatives with 1-edit distance. An original sequence matches the *pro-ftpd* signature in Table 3. Event definitions are given in Table 6. In Round i we applied the Invariant Refinements or Signature Fixes we performed after Round $i-1$.

determined whether this sequence was a false negative or a spurious sequence. In the first case, we refined the signature; in the second, we refined the invariant. We repeated these steps until we were satisfied with the signature accuracy.

We searched for false negatives that are 1-edit distance from our session signature. We assumed that the default representation mode is ASCII. During the process, we found two false negatives. The whole process took less than 3 hours for an experienced GARD user.

In the first false negative we found that GARD replaced a `TYPE A` with a `RETV` command (Round 1 in Table 4). Essentially, GARD showed us that when the default representation mode is ASCII, a `TYPE A` command is not required for a successful *pro-ftpd* attack; a successful login puts the FTP session into ASCII mode (Figure 4a). We changed the *pro-ftpd* preparation phase into the expression represented by Figure 4b and removed this type of false negatives.

In the next sequence, GARD replaced a `STORE` command with a `CWD` (Round 2 in Table 4). Since the `STORE` is a necessary condition for *pro-ftpd*, this sequence is not a false negative but a spurious sequence. To avoid this type of sequences, we added to our semantic model a variable, denoted *st*, that counts the number of `STORE`s in a sequence. We used *st* to ensure that one `STORE` command appears in any sequence GARD provides. Not surprisingly, in the next sequence, GARD replaced a `RETV` with a `CWD` (Round 3 in Table 4); this sequence also does not implement the attack. Hence, we added the *rt* variable that counts `RETV` commands and used it to ensure that a sequence contains two `RETV` commands.

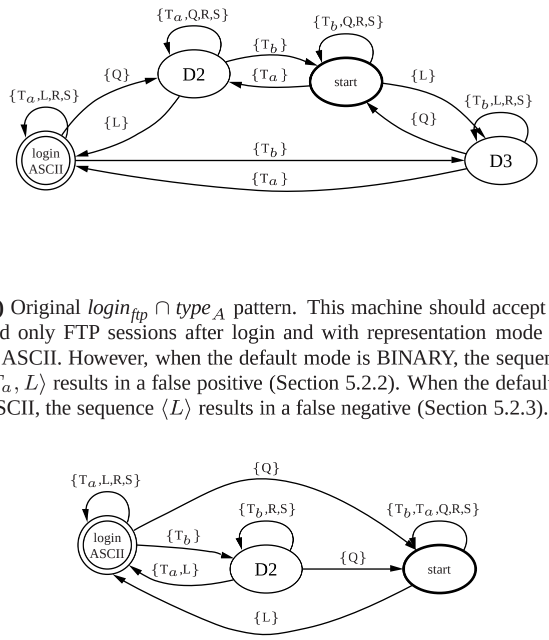
Next, GARD added an `APPEND` command and replaced the `STORE` with a `CWD` (Round 4 in Table 4). This is a false negative, as the *pro-ftpd* attack can be implemented using an `APPEND` instead of a `STORE`. Since we modeled an `APPEND` in the same way we modeled a `STORE`, both increment *st* (Table 6), we anticipated this false negative; to prevent it, we added an `APPEND` command to the *pro-ftpd* preparation phase.

The question of whether `STORE` and `APPEND` are equivalent in general is beyond the scope of this paper. We believe that the answer is attack-dependent; a security analyst should address it in every FTP attack they define. GARD affords an analyst the ability to investigate the implications of their decision. We defined the two commands in an equivalent way, so GARD can warn an analyst of an unforeseen false negative. If this behavior is undesired, an analyst can disable it easily.

In the next sequence, GARD replaced a `RETV` command with a `CWD` and increased the length of the sequence to include two `RETV` commands as required by the attack invariants (Round 5 in Table 4). Again, this is a spurious sequence. At this point we used our edit-distance mechanism to limit the transformations attackers can perform (Section 4.4). For example, in the *pro-ftpd* attack the last `RETV` cannot be replaced with a `CWD`, so we forbade this replacement. Based on the restrictions we imposed, we did not find other false negatives.

5.3 Summary of GARD Evaluation

We analytically showed that our *ftp-cwd* session signature is more accurate than its Snort and contextual counterparts. To the best of our knowledge, we are the



(a) Original $login_{ftp} \cap type_A$ pattern. This machine should accept all and only FTP sessions after login and with representation mode set to ASCII. However, when the default mode is BINARY, the sequence $\langle T_a, L \rangle$ results in a false positive (Section 5.2.2). When the default is ASCII, the sequence $\langle L \rangle$ results in a false negative (Section 5.2.3).

(b) The $login_{ftp} \cap type_A$ pattern after false negative fix (Section 5.2.3). This pattern ensures that after login the session is in ASCII mode, given a default ASCII mode.

Figure 4. Two versions of the $login_{ftp} \cap type_A$ pattern.

first to provide such an evidence of signature accuracy.

GARD taught us that we need to tune the *pro-ftpd* signature according to the ProFTPD server configuration. We did not anticipate this outcome when we began our experiment. Our experience shows that constructing an accurate regular expression is a delicate issue. Even in simple cases, using inaccurate expressions might lead to unforeseen false positives and negatives. We also learned the effectiveness of formal methods to reveal these inaccuracies.

We must remember that GARD does not guarantee that our signature lacks all false positives and negatives. GARD only guarantees that there are no false positives and negatives with respect to our invariant and its underlying semantic model. For example, it is possible to split the *pro-ftpd* attack into two FTP sessions so that it is no longer detected by our signature. The refinement of the signature to handle such cases is left for future work.

6 Conclusion and Future Work

We believe signatures that are based on formal reasoning and verifiable accuracy is a worthy cause. In this paper, we took the first step toward this goal. We present a methodology to construct, evaluate, and improve signatures.

We intend to continue this work as follows. First, although our initial results indicate that the operational cost of session signature is comparable to current signatures (Table 2), we intend to perform a thorough investigation of this issue. Second, since session signatures use HSMs, it seems possible to share machines between signatures and improve the ability of a NIDS to handle many signatures simultaneously. Last, it is necessary to develop semantic models for other protocols.

Acknowledgments. We deeply thank the anonymous referees for their useful comments that have helped us refine the concepts presented in this paper.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. ***Compilers: Principles, Techniques and Tools***. Addison-Wesley, 1986.

[2] R. Alur and M. Yannakakis. Model checking of hierarchical state machines. In *Foundations of Software Engineering*, Lake Buena Vista, FL, Nov. 1998.

[3] A. R. Baker, Barnyard, B. Caswell, M. Poor, R. Alder, J. Babbin, J. Beale, A. Doxtater, J. C. Foster, T. Kohlenberg, and M. Rash. ***Snort 2.1 Intrusion Detection***. Syngress, 2 edition, May 2004.

[4] Beyond Security Inc. ProFTPD ASCII file remote root exploit. Available at `www.securiteam.com/exploits`.

[5] BlackMoon Inc. BlackMoon FTP Server . Available at `www.blackmoonftpserver.com`.

[6] F. Cuppens and R. Ortalo. LAMBDA: A language to model a database for detection of attacks. In *International Symposium on Recent Advances in Intrusion Detection*, Toulouse, France, Oct. 2001.

[7] M. Dacier, editor. *Design of an Intrusion-Tolerant Intrusion Detection System*. IBM Zurich Research Laboratory, Aug. 2002. Deliverable D10, Project MAFTIA IST-1999-11583, Available at `www.maftia.org`.

[8] S. T. Eckmann, G. Vigna, and R. A. Kemmerer. STATL: An attack language for state-based intrusion detection. *J. Computer Security*, **10**(1/2), 2002.

[9] S. Ginsburg. ***The Mathematical Theory of Context Free Languages***. McGraw Hill, 1966.

[10] M. Handley and V. Paxson. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *USENIX Security Symposium*, Washington, DC, Aug. 2001.

[11] G. J. Holzmann. ***The Spin Model Checker: Primer and Reference Manual***. Addison-Wesley, 2003.

[12] J. Hopcroft, R. Motwani, and J. Ullman. ***Introduction to Automata Theory, Languages, and Computation***. Addison-Wesley, 2 edition, 2001.

[13] L. Kari, S. Konstantinidis, S. Perron, G. Wozniak, and J. Xu. Finite-state error/edit-systems and difference-measures for languages and words. Technical Report

2003_001, Saint Mary's University Department of Mathematics and Computing Science, 2003.

[14] R. Lippmann, J. W. Haines, D. J. Fried, J. Korba, and K. Das. Analysis and results of the 1999 DARPA offline intrusion detection evaluation. In *International Symposium on Recent Advances in Intrusion Detection*, Toulouse, France, Oct. 2000.

[15] R. Marti. THOR: A tool to test intrusion detection systems by variations of attacks. Master's thesis, Swiss Federal Institute of Technology, Mar. 2002.

[16] C. Michel and L. Mé. ADeLe: An attack description language for knowledge-based intrusion detection. In *International Conference on Information Security*, Paris, France, June 2001.

[17] MITRE Corporation. CVE: Common Vulnerabilities and Exposures. Available at `www.cve.mitre.org`.

[18] J. Morrissey, T. Saunders, M. Lowes, D. Roesen, and M. Renner. ProFTPD: Highly configurable GPL-licensed FTP server software . Available at `www.proftpd.org`.

[19] D. Mutz, G. Vigna, and R. A. Kemmerer. An experience developing an IDS stimulator for the black-box testing of network intrusion detection systems. In *Annual Computer Security Applications Conference*, Las Vegas, NV, Dec. 2003.

[20] V. Paxson. *Flex, version 2.5, A fast scanner generator*. Free Software Foundation, Mar. 1995.

[21] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, **31**(23/24), Dec. 1999.

[22] J. Postel and J. Reynolds. *RFC 959 - File Transfer Protocol*. The Internet Engineering Task Force, 1985.

[23] J.-P. Pouzol and M. Ducassé. From declarative signatures to misuse IDS. In *International Symposium on Recent Advances in Intrusion Detection*, Davis, CA, Oct. 2001.

[24] J.-P. Pouzol and M. Ducassé. Formal specification of intrusion signatures and detection rules. In *IEEE Computer Security Foundations Workshop*, Cape Breton, NS, Canada, June 2002.

[25] T. H. Ptacek and T. N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical Report T2R-0Y6, Secure Networks, Inc., Calgary, AB, Canada, 1998.

[26] M. Roesch. Snort: the Open Source Network Intrusion Detection System. Available at `www.snort.org`.

[27] S. Rubin, S. Jha, and B. P. Miller. Automatic generation and analysis of NIDS attacks. In *Annual Computer Security Applications Conference*, Tucson, AZ, Dec. 2004.

[28] Security Administrator Newsletter. Instant poll: Do you use Snort to implement an intrusion detection system (IDS) on your network?, Oct. 2002. Available at `www.winnetmag.com/Poll`.

[29] R. Sekar, Y. Guang, S. Verma, and T. Shanbhag. A high-performance network intrusion detection system. In *ACM Conference on Computer and Communications Security*, Singapore, Nov. 1999.

[30] R. Sekar and P. Uppuluri. Synthesizing fast intrusion prevention/detection systems from high-level specifications. In *USENIX Security Symposium*, Washington, DC, Aug. 1999.

[31] R. Sommer and V. Paxson. Enhancing byte-level network intrusion detection signatures with context. In *ACM Conference on Computer and Communications Security*, Washington, DC, Oct. 2003.

[32] R. Teitelbaum. *Minimal Distance Analysis of Syntax Errors in Computer Programs*. PhD thesis, Computer Science Department, Carnegie-Mellon University, Sep. 1975.

[33] The NSS Group. Intrusion detection systems (IDS) group test (Edition 4), 2003. Available at `www.nss.co.uk/ids/edition4/index.htm`.

[34] G. Vigna and R. A. Kemmerer. NetSTAT: A Network-based Intrusion Detection System. *J. Computer Security*, **7**(1), 1999.

[35] G. Vigna, W. Robertson, and D. Balzarotti. Testing network-based intrusion detection signatures using mutant exploits. In *ACM Conference on Computer and Communications Security*, Washington, DC, Oct. 2004.

[36] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *ACM Conference on Computer and Communications Security*, Washington, DC, Nov. 2002.

A FTP Semantic Model

The state of FTP is defined using 8 variables (Table 5). The alphabet of the model (Section 4.3), denoted Σ_{ftp}, is the set of tokens defined in Table 6. We modeled only the most common FTP commands; others can be modeled in a similar way. For brevity, we define lexemes using upper case letters. However, the FTP specification allows any combination of upper and lower case letters [22]. The events from Table 6 form the event library for FTP. The model is complete: for every event that is not defined in Table 6, the state does not change.

Var.	Values	Semantic	Comments
x_1	$\{0, 1\}$	A USER command was issued.	
x_2	$\{0, 1\}$	A PASS command was issued.	
x_3	$\{0, 1\}$	Victim has indicated a successful login.	Alias to *FTP*$_{login}$ (Section 3.2)
x_4	$\{U = 0, A = 0, B = 1, E = 2\}$	Holds session representation type (i.e., TYPE)	Alias to *FTP*$_{type}$ (Section 5.2.2). A=ASCII, B=BINARY, E=EBCDIC, U=undefined.
x_5	$\{U = 0, S = 0, B = 1, C = 2\}$	Holds session transmission mode (i.e., MODE)	S=STREAM,B=BLOCK, C=COMPRESSED, U=undefined.
x_6	$\{0, 1\}$	A session is in passive mode.	
x_7	$\{0, \ldots, MAX\}$	Counts number of files uploaded in this session.	
x_8	$\{0, \ldots, MAX\}$	Counts number of files downloaded in this session.	

Table 5. State variables for the FTP semantic model.

Event	Token	Lexeme[a]	Flow	Description	Precondition	Postcondition
USER	U	(ˆ“*USER*”)$_a$	$A \rightarrow V$	Specifying a user trying to login.	$x_1 = 0$	x_1 =1
PASS	P	(ˆ“*PASS*”)$_a$	$A \rightarrow V$	Specifying a user’s password.	$x_1 = 1$	x_2 = 1
CWD	C	(ˆ“*CWD*”)$_a$	$A \rightarrow V$	Change directory.	-	-
CQUIT	Q_1	(ˆ“*QUIT*”$\backslash n$)$_a$	$A \rightarrow V$	Client terminates the session.	-	$\forall x_i = 0$
REIN	Q_2	(ˆ“*REIN*”$\backslash n$)$_a$	$A \rightarrow V$	User logged out, session can be restarted.	-	$\forall x_i = 0$
PASV	V	(ˆ“*PASV*”$\backslash n$)$_a$	$A \rightarrow V$	Enter passive mode.	$x_3 = 1$	$x_6 = 1$
(TYPE)$_a$	T_a	(ˆ“*TYPE*”[*SP*]“*A*”$\backslash n$)$_a$	$A \rightarrow V$	Change representation type to ASCII.	$x_3 = 1$	$x_4 = A$
(TYPE)$_i$	T_b	(ˆ“*TYPE*”[*SP*]“*B*”$\backslash n$)$_a$	$A \rightarrow V$	Change representation type to BINARY.	$x_3 = 1$	$x_4 = B$
(TYPE)$_e$	T_e	(ˆ“*TYPE*”[*SP*]“*E*”$\backslash n$)$_a$	$A \rightarrow V$	Change representation type to EBCDIC.	$x_3 = 1$	$x_4 = E$
(MODE)$_s$	M_s	(ˆ“*MODE*”[*SP*]“*S*”$\backslash n$)$_a$	$A \rightarrow V$	Change transmission mode to STREAM.	$x_3 = 1$	$x_5 = S$
(MODE)$_b$	M_b	(ˆ“*MODE*”[*SP*]“*B*”$\backslash n$)$_a$	$A \rightarrow V$	Change transmission mode to BLOCK.	$x_3 = 1$	$x_5 = B$
(MODE)$_c$	M_c	(ˆ“*MODE*”[*SP*]“*C*”$\backslash n$)$_a$	$A \rightarrow V$	Change transmission mode to COMPRESSED.	$x_3 = 1$	$x_5 = C$
RETR	R	(ˆ“*RETR*”)$_a$	$A \rightarrow V$	Retrieve a file from the server.	$x_3 = 1$	$x_8 = x_8 + 1$
STOR	S	(ˆ“*STOR*”)$_a$	$A \rightarrow V$	Store a file on the server.	$x_3 = 1$	$x_7 = x_7 + 1$
APPE	A	(ˆ“*APPE*”)$_a$	*AP* $\rightarrow V$	Append a file on the server.	$x_3 = 1$	$x_7 = x_7 + 1$
DELE	D	(ˆ“*DELE*”)$_a$	$A \rightarrow V$	Delete a file from the server.	$x_3 = 1$	$x_7 = x_7 - 1$
LIST	*LS*	(ˆ“*LIST*”)$_a$	$A \rightarrow V$	List files on the server.	-	-
SLOGIN	L	(ˆ“230”$(\backslash w)^\star\backslash n$)$_v$	$V \rightarrow A$	User has successfully logged in.	-	$x_3 = 1$
VQUIT	Q_3	(ˆ[$2 \mid 4$]“21”$(\backslash w)^\star\backslash n$)$_v$	$V \rightarrow A$	Victim voluntarily terminates session	-	$\forall x_i = 0$
ARG	A	([*SP*] $<$ *str* $>$ $\backslash n$)$_a$	$A \rightarrow V$	An argument of an FTP command.	-[b]	-
INVALID RESPONSE	IR	(ˆ*[*ˆ1-5*]*)$_a$	$V \rightarrow A$	A non-FTP response (any valid response must start with a digit between 1 to 5).	-	-

[a] *str* denotes a string according to the FTP specification [22], ˆ denotes match only at the beginning of a line. $\backslash w$ denotes alphanumeric plus “_”.

[b] Must be precede with a command that requires an argument. For brevity, we do not add the state variables required to track the type of the last command.

Table 6. Events and their transitions for the FTP semantic model.

	Rule	Description
1	$E \rightarrow$ *token*	A single token is a valid expression.
2	$E \rightarrow (E)^\star \mid (E)^+$	Closure of a valid expression is a valid expression.
3	$E \rightarrow \neg(E)$	Negation of a valid expression is a valid expression.
4	$E \rightarrow (E\ op_1\ E)$	Concatenation, intersection, and union of two valid expressions is a valid expression.
5	$op_1 \rightarrow \cdot \mid \cap \mid \cup$	
6	$E \rightarrow$ (*token such_that* R)	A restricted event: a valid expression restricted with R is a valid expression.
7	$R \rightarrow$ (*data* $\in$ *raw_expr*)	A regular restriction imposed on the data attribute of a of an token.
8	$R \rightarrow$ (*length* op_2 *INT*)	A regular restriction imposed on the length attribute of a of an token.
9	$op_2 \rightarrow < \mid > \mid = \mid \neq$	
10	$R \rightarrow (R\ op_3\ R)$	A logical combination of two restrictions is a valid restriction.
11	$op_3 \rightarrow \vee \mid \wedge$	

Table 7. Grammar for construction regular expressions over events.