Behavioral Footprinting: a New Dimension to Characterize Self-Propagating Worms

Xuxian Jiang, Dongyan Xu

CERIAS and Department of Computer Science, Purdue University, West Lafayette, IN 47907

{jiangx, dxu}@cs.purdue.edu

Abstract

With increasing speed, virulence, and sophistication, self-propagating worms continue to pose a serious threat to the safety of the Internet. To effectively identify and defend against self-propagating worms, a critical task is to characterize a worm along multiple dimensions. Content-based fingerprinting *is a well-established dimension for worm characterization by deriving the most representative content sequence as a worm's signature. However, this dimension alone does not capture all aspects of a worm and may therefore lead to incomplete or inaccurate worm characterization.*

To expand the space of worm characterization, this paper proposes and justifies a new *dimension,* behavioral footprinting. *Orthogonal and complementary to content-based fingerprinting, behavioral footprinting characterizes a worm's unique behavior during each infection session, which covers the probing, exploitation, and replication phases of the infection session. By modeling each infection step as a* behavior phenotype *and the entire infection session as a sequential* behavioral footprint, *we show that behavioral footprinting captures worm-specific behavior which is inherently* different *from a normal access to the vulnerable service. We present advanced sequence analysis techniques to extract a worm's behavioral footprint from its infection traces. Our evaluation with a number of real-world worms clearly demonstrates its feasibility and effectiveness in successfully extracting worm-characterizing behavioral footprints for all experimented worms. Furthermore, by comparing with content-based fingerprinting, our experiments demonstrate the uniqueness and robustness of behavioral footprinting in worm recognition and identification.*

Keywords: *Worm Recognition and Characterization, Behavioral Footprinting, Content-Based Fingerprinting*

1   Introduction

Self-propagating worms continue to pose a serious threat to the safety of the Internet, To effectively identify and defend against self-propagating worms, a critical task is to characterize a worm along *multiple* dimensions. Content-based fingerprinting [26, 28, 33, 43] is a well-established dimension to capture a worm's characteristics by deriving the most representative content sequence as the worm's signature. In practice, various intrusion detection systems (IDSes) [36, 41], together with recent honeypot systems [5, 22, 38, 46], are deployed to collect live worms. Once a worm specimen[1] is collected, anti-worm experts will manually examine the specimen and extract a worm-identifying content fingerprint as the worm's signature. Recent systems [26, 28, 33, 43] take one step further by automatically generating worms' content fingerprints. These systems have demonstrated a degree of success. However, they all

---

[1]The worm specimen might not only contain the worm binary itself, but also include other corresponding traffic associated with a worm infection (e.g., exploitation).

focus on one dimension of worm characterization, namely content, while missing other aspects of a worm. This single-dimension characterization may limit the capability of worm identification and recognition. For example, it has been demonstrated that advanced worms are now capable of exploiting the weakness of content-based fingerprinting by mutating [45] or encrypting [27] their contents or payloads in each infection session, hence escaping recognition and identification by content fingerprints.

We are motivated to explore other dimensions to expand the space of worm characterization and thus enhance worm identification capabilities. Especially, we realize that content-based fingerprinting does not capture a worm's *temporal* infection behavior, which contains valuable self-identifying information that leads to the worm's recognition. In this paper, we present and justify a new dimension, *behavioral footprinting*, to enrich worm characterization. We would like to emphasize that behavioral footprinting is expected to be orthogonal and complementary to other dimensions including content fingerprinting. This new dimension alone also suffers from ineffectiveness towards certain worms. In this paper, we target the type of worms [7, 8, 9, 10, 12, 13, 30, 31, 37] that exploit traditional vulnerable servers (e.g., Apache/IIS, DNS, and Sendmail) to propagate themselves without any human intervention. Other types of worms (e.g., mass-mailing or IM worms [11] involving end user interactions) are subjects of future work. Our contributions are mainly three-fold:

Firstly, we propose behavioral footprinting as a fundamentally new dimension for the characterization of self-propagating worms. Unlike content fingerprinting which extracts one or a few static worm-unique byte sequences as signature, behavioral footprinting essentially captures a worm's unique temporal action sequence during an infection session, which covers the *probing* [2], *exploitation*, and *replication* phases of the infection session. Our evaluation (Section 4) with a number of real-world worms clearly demonstrates the existence of worm-specific behavioral footprints.

Secondly, we develop robust algorithms to extract the behavioral footprint from a worm's infection traces. More specifically, by representing each step within a worm's infection session as a *behavioral phenotype* and the complete infection session as a behavioral phenotype sequence, we observe that the sequence reflects both worm-specific exploitation and propagation strategies. Given traces of only a few infection sessions, our algorithms (Section 3) are able to accurately and robustly extract a worm's behavior footprint, despite possible worm behavior mutation and camouflaging, such as cloaking authentic phenotypes or forging phenotypes.

Thirdly, by comparing with content-based fingerprinting, we demonstrate the uniqueness and robustness of behavior-based footprinting in worm recognition and identification. Because of their orthogonality, behavior-based footprinting is naturally robust against attacks that evade content-based fingerprinting. Our experiments show instances of worms that cannot be identified by content fingerprints but are recognizable using behavior footprints, justifying behavioral footprinting as a complementary dimension for worm characterization.

The rest of this paper is organized as follows: In Section 2, we demonstrate the existence of behavioral footprints

---

[2]Some non-scanning worms may *not* have the probing phase.

in self-propagating worms and make a case for the new dimension of behavior-based footprinting. We then describe in Section 3 our algorithms to extract a worm's behavioral footprint. We present experimental results with a number of real worms in Section 4. Limitations and possible improvements are described in Section 5. We present related work in Section 6 and finally conclude this paper in Section 7.

2 A Case for Behavioral Footprinting

In this section, we first present a staged view of a worm infection session to motivate the characterization of worm behavior. As representative examples, we illustrate the existence of behavioral footprints in two well-known worms: the MSBlaster worm propagating on Windows platform and the Lion worm on Linux platform. Finally, we make a case for behavioral footprinting.

2.1 A Staged View of Worm Infection

In general, the infection of a self-propagating worm from an infected host to a victim host can be broken into three phases:
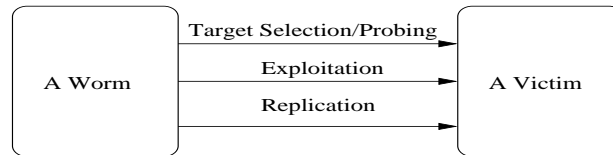

Figure 1. A Staged View of a Worm Infection Session

Phase 1: Target selection and probing   Using a strategy such as random or biased address scanning, a scanning worm during this stage attempts to pick up a victim for infection. For example, an ICMP echo request packet or a TCP *SYN* probe is used to infer the reachability of a chosen target. Additional packets may also be used to obtain the version of a possibly vulnerable service. We note that this phase may *not* exist for non-scanning worms because they may carry a pre-computed target list.

Phase 2: Exploitation   Once the worm receives a positive response from the victim host, a number of malicious packets[3] may be sent over attempting to exploit the targeted vulnerability. Successful exploitation will result in the execution of a specifically crafted code in the victim node. Different worms usually implement different functionalities in the crafted code.

Phase 3: Replication   If the exploitation is successful, an additional replication phase may follow to transmit a worm replica to the victim node. The replica will be installed in the victim node, completing this infection session.

We will show that the behavior exhibited by the worm during this infection session contains valuable self-identifying information that can be used to characterize and identify the worm. Especially, the temporal order of infection steps

---

[3]There are certain worms such as Slammer[30] which might blindly send exploitation to any probed hosts.

taken by the worm reflects the intrinsic dependencies that must be followed to ensure a successful infection.

2.2 Example I: Windows-Based MSBlaster Worm

We consider the infamous MSBlaster worm [9] as the first motivating example. The MSBlaster worm exploits an RPC-DCOM vulnerability (MS03-026) for its infection. An MSBlaster infection session is illustrated in Figure 2. The infection session consists of the following steps:
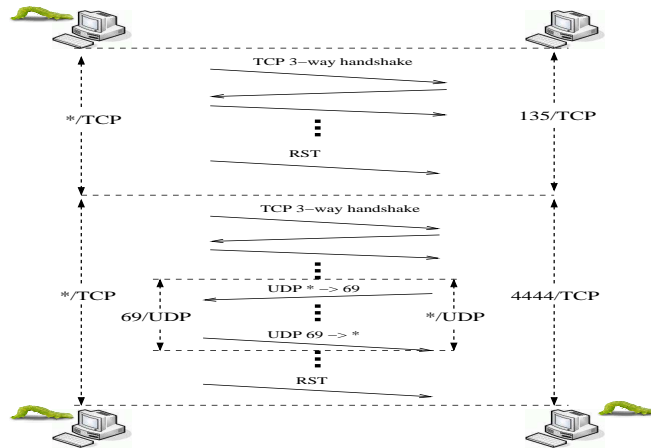

Figure 2. An Infection Session of the MSBlaster/Windows Worm

- A three-way TCP handshake on port 135 [4] is implicitly used by the worm to check the reachability of the selected target (Phase 1).

- Upon the establishment of the TCP connection, the worm sends a number of malicious packets (Phase 2), which exploit the known RPC-DCOM vulnerability[9] and contain a specially crafted shell-code. A successful exploitation will lead to the execution of the shell-code in the victim node. In the case of the MSBlaster worm, a new shell service will be started on TCP port 4444 by the shell-code.

- The new shell service on $4444/TCP$ is immediately contacted by the worm to send instructions on how to download the worm replica, i.e., *msblast.exe* (Phase 3). From Figure 2, the *TFTP* protocol is apparently used for the downloading.

The above sequence of actions significantly deviates from a normal access to the RPC-DCOM service: First, after the "service request", a new shell service would not suddenly appear and listen on $4444/TCP$ in the victim host. Second, a new TCP connection to *this* port would not follow with the service request. Third and most importantly, it should not be observed that the victim *took the initiative* in using the *TFTP* protocol to download a file (with the name *msblast.exe* and size $6,372$ bytes) from the service client.

---

[4]Microsoft's DCOM Service Control Manager (also known as the RPC Endpoint Mapper) uses this port as a well-defined means to provide port-mapping services associating available services with their ports.
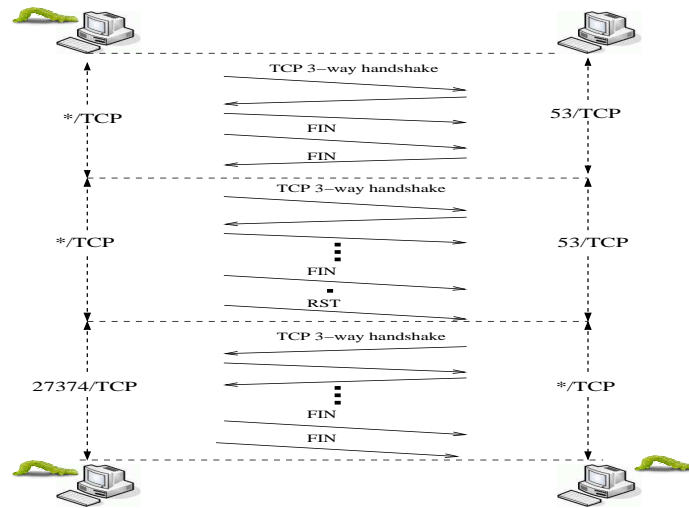
2.3 Example II: Linux-Based Lion Worm

Figure 3. An Infection Session of the Lion/Linux Worm

The second illustrative example is the historical Linux-based Lion worm [4]. The Lion worm exploits a BIND vulnerability (CA-2001-02) for its infection. A Lion worm infection session is shown in Figure 3.

- The Lion worm firstly makes an explicit TCP connection attempt to the destination port 53. A successful connection indicates the reachability and possible vulnerability of the selected target (Phase 1). This connection, if established, is then immediately tore down *without* transmitting any payload.

- Another TCP connection to the same destination port is then established. This time, certain exploitation codes are sent (Phase 2).

- If the exploitation is successful, the shell script, which is transmitted together with the exploitation codes, will be executed to retrieve a worm replica from the infecter to the victim (Phase 3).

Again, deviation from the normal access to DNS lookup service is observed: First, it is unlikely that the access would begin with a plain TCP connection *with no payload*. Second and most importantly, after the DNS lookup request, it is highly unusual that the BIND server on the victim side *initiates* a TCP connection to the *DNS client* on an unusual port $27374/TCP$, followed by an HTTP session on this connection to transfer a file of $71,680$ bytes *from the client to the server*.

2.4 Behavioral Footprinting: a New Dimension

In general, for the same vulnerable service, there exist intrinsic differences between a normal access to the service and a worm infection through the service:

Firstly, during the exploitation phase of a worm infection session, a worm will attempt to misuse a vulnerable service in a way that is different from a normal access. In fact, several recent works [1, 32, 47] have leveraged this difference to derive vulnerability models for worm defense.

Secondly, the replication phase of a worm infection session should *not* happen during a normal access to the vulnerable service. In sharp contrast, it will appear in every successful worm infection. As shown in Figure 2, the $4444/TCP$ connection and its encapsulated TFTP transmission will appear in every MSBlaster worm infection. Similarly, the $27374/TCP$ connection and its encapsulated HTTP session can be observed for every Lion worm infection (Figure 3).

Finally, the entire sequence of infection steps during an infection session characterizes the worm's behavior, and is highly unlikely to appear in normal traffic. In fact, our experiments with real-world network traces result in *zero false positive*. Furthermore, for different worms exploiting the same vulnerable service, their sequences of infection steps are different. The reason is that different worms tend to have different exploitation means, replication idiosyncrasies, and payloads, even though they are exploiting the same vulnerability (Section 4.2).

Based on the above observations, we are motivated to adopt a worm's infection step sequence during an infection session to characterize and thus uniquely identify the worm. We call this new dimension *behavioral footprinting*, in contrast to the well-known dimension of content-based fingerprinting. We emphasize that the two dimensions complement each other and they should be combined to overcome their own weaknesses (Section 5). Especially, since behavioral footprinting does not rely on payload content analysis, it is naturally resistant to content-based mutation and encryption attacks (Sections 4.4.1, 4.4.2).

3 Behavioral Footprint Representation and Extraction

In this section, we first define the behavioral footprint and its representation. A simple *pairwise alignment* algorithm is then presented to extract a behavioral footprint from the traces of two infection sessions. To increase the robustness against more intelligent worms, we develop an advanced footprint extraction algorithm to accurately extract a worm's behavioral footprint from multiple infection sessions.

3.1 Behavioral Phenotype and Footprint

The term "*behavioral phenotype*" was originally coined in 1972 by Nyhan [35] to represent a behavior that was genetically determined in the same way as the physical features of a phenotype. Recall the staged view of worm infection session in Section 2, if we denote a worm's infection steps as the worm's behavioral phenotypes, the sequence of behavioral phenotypes manifested during the infection session will be defined as the worm's intrinsic behavioral footprint. From Section 2, the behavioral footprint uniquely reflects the behavioral characteristics of the worm (e.g., abused vulnerability, working exploitation, adopted propagation, and self-carried payload).

Our proposed algorithms to extract worm behavior footprints are based on the sequence analysis techniques extensively applied in bio-informatics areas. A common and important issue for bio-informatics research is to operate over a large sequences of strings such as DNA, RNA, and protein sequences to find certain pattern(s) among them. Notice that any type of protein is a sequence of amino acid sub-units and there are only 20 different amino acids, which constitute the whole alphabet for protein sequence analysis. Similarly, if we consider all possible behavioral phenotypes during the worm infection as the alphabet, the behavioral footprint of a worm can be represented as a sequence of characters in the alphabet. For example, the behavioral footprint of the MSBlaster worm, based on the infection session in Figure 2, can be represented as $S_1 \overleftarrow{S_1^A} A_1 \cdots R_1 S_2 \overleftarrow{S_2^A} A_2 \cdots \overleftarrow{U_1} U_1 \cdots R_2$, where the characters' definitions are:

$$
\begin{aligned}
S_1 &: \ < TCP, 4581/infecter, 135/victim, SYN > \\
\overleftarrow{S_1^A} &: \ < TCP, 135/victim, 4581/infecter, SYN, ACK > \\
A_1 &: \ < TCP, 4581/infecter, 135/victim, ACK > \\
R_1 &: \ < TCP, 4581/infecter, 135/victim, RST > \\
S_2 &: \ < TCP, 4599/infecter, 4444/victim, SYN > \\
\overleftarrow{S_2^A} &: \ < TCP, 4444/victim, 4599/infecter, SYN, ACK > \\
A_2 &: \ < TCP, 4599/infecter, 4444/victim, ACK > \\
\overleftarrow{U_1} &: \ < UDP, 1552/victim, 69/infecter > \\
U_1 &: \ < UDP, 69/infecter, 1552/victim > \\
R_2 &: \ < TCP, 4599/infecter, 4444/victim, RST >
\end{aligned}
$$

The letters in the above footprint denote either TCP flows with different control bits (SYN, ACK, RST) or UDP/ICMP flows (U/I). The subscripts denote different flows. For example, $\overleftarrow{S_1^A}$ or $\overleftarrow{S_2^A}$[5] represents the second step (SYN and ACK bits set) in a normal three-way TCP handshaking procedure. Without ambiguity, a unique well-known *subsequence* can be further shortened as a single character. For example, a TCP 3-way handshake sequence (e.g., $S_i \overleftarrow{S_i^A} A_i$, $i = 1, 2$, in previous sequence) could be simply defined as C_i (more in Section 4).

In this example, every character is a tuple of several fields: the character representing a specific TCP flow has four fields $< TCP, source\_port, dest\_port, \text{TCP control bits} >$; the character related to a specific UDP flow has three fields $< UDP, source\_port, dest\_port >$. Note that as different infection sequences might have different ports, a special *wildcard* field [6] needs to be introduced. Using the MSBlaster worm as an example, the source ports (e.g., the port 4581, 4599, 1552 in S_1, S_2, $\overleftarrow{U_1}$, respectively) vary with different infection sessions while the destination ports are fixed (e.g., the port 135, 4444, 69 in S_1, S_2, $\overleftarrow{U_1}$, respectively). As such, the special wildcard field (instead of a fixed

---

[5]The arrow sign is used to mark the traffic flow direction and can be omitted when it is implicitly implied.

[6]A finite set containing a limited number of values can also be introduced to more precisely capture the possible contents. For simplicity, this paper only mentions the wildcard field.

port number) is used for the source port field. Also, there are some worms, which might have a constant source port number (e.g., the Witty worm have a constant UDP source port 4000), but a random destination port. In this case, the wildcard is used to represent the destination port field. It is worth mentioning that although a worm infection session usually involves *only* two nodes (infecter and victim), a *coordinated* worm infection might involve more than two nodes (e.g., downloading the worm replica from a third-party). In this case, the wildcard field can be used to represent the infecter field.

In addition, the number of fields in a phenotype may not be fixed. Additional fields can be added to each flow to include other meaningful information such as the packet length, particular content sequence, or even relative timing from the previous one (an example is shown in Section 3.3.1). In fact, the extensible nature of behavioral phenotype representation makes it easier to integrate worm characteristics of *other dimensions*. For example, the content-based fingerprint of a worm can be added to a behavioral phenotype, indicating the occurrence of the content during the corresponding infection step. Protocol compliance analysis and vulnerability-specific information can also be integrated to further improve the accuracy of worm identification.

However, we would like to point out that due to different understanding or emphasis even for the same worm, different researchers might intend to extract different behavioral footprints (e.g., in terms of sequence length or field content). Such situation is similar to the content-based counterpart: Different content fingerprints may be chosen by different researchers for the same worm. For simplicity, this paper chooses a simple representation described in this Section. As shown in Section 4, such representation is capable of accurately characterizing existing worms.

3.2   Pairwise Alignment Algorithm

Based on the behavioral footprint representation, we first present an algorithm to extract a worm's behavioral footprint from two infection sequences of the worm.

Given two infection sequences $\mathscr{F}_1 = x_1 x_2 \cdots x_n$ and $\mathscr{F}_2 = y_1 y_2 \cdots y_m$, a pairwise alignment algorithm is primarily used to align these two sequences so that they could have the same length. Based on a pre-defined scoring matrix (e.g., a match yields 1 while a mismatch yields 0), the alignment algorithm inserts gaps, if necessary, to achieve maximum alignment of the two sequences. The maximum alignment is defined as the sum of terms for each aligned pair of characters $< x_i, y_j >$ within the sequences (representing similarity $s(x_i, y_j)$), plus terms for each gap (representing penalty, p). The similarity and gap penalty are defined as a part of the scoring matrix and might be specific to different applicable scenarios. A global alignment scheme obtains the optimal global alignment between two sequences while a local alignment scheme looks for the best alignment between subsequences of them. There are two corresponding well-known dynamic programming algorithms, i.e., Needleman-Wunsch algorithm[18] and Smith-Waterman algorithm[18].

The idea in Needleman-Wunsch algorithm is to build up an optimal alignment using previous solutions or optimal alignments of smaller subsequences. A matrix $\mathscr{M}$, indexed by i and j with one index for each sequence, is iteratively

constructed. The cell $\mathcal{M}(i,j)$ is the score of the best alignment between the initial segment $x_1 x_2 \cdots x_i$ of x up to x_i and the initial segment $y_1 y_2 \cdots y_j$ of y up to y_j. Initially, $\mathcal{M}(0,0) = 0$, $\mathcal{M}(i,0) = -ip$, $\mathcal{M}(0,j) = -jp$. Then, the matrix is iteratively filled from top-left cells to bottom-right cells based on Eqn.(1).

$$\mathcal{M}(i,j) = \max \begin{cases} \mathcal{M}(i-1,j-1) + s(x_i, y_j), & i \geq 1, j \geq 1 \\ \mathcal{M}(i-1,j) - p, & i \geq 1 \\ \mathcal{M}(i,j-1) - p, & j \geq 1 \end{cases} \tag{1}$$

Each case represents an option how current $\mathcal{M}(i,j)$ cell is derived from one of the other three cells (above-left $[i-1, j-1]$, above $[i-1, j]$, or left $[i, j-1]$). Once all values are calculated, the choices taken at each cell starting from the bottommost rightmost one are *traced* back so that an optimal global alignment is derived. An example alignment applying the Needleman-Wunsch algorithm to the Welchia worm [12] is shown in Figure 4.

Sequence 1:   I $_1$I $_1$C$_1$F $_1$F $_1$C$_2$U$_1$U$_1$$-$ $-$ R$_2$

Sequence 2: $-$ $-$ C$_1$F $_1$F $_1$C$_2$U$_1$U$_1$U$_2$U$_2$R$_2$

Figure 4. Global Alignment with Needleman-Wunsch Algorithm. The choices made during the alignment are shown as "-" and "|". The "-" in the top sequence used as index i for $\mathcal{M}$ corresponds to the choice "above" $[i-1, j]$, the "-" in the bottom sequence used as index j for $\mathcal{M}$ represents "left" choice $[i, j-1]$, while the "|" in the middle shows the option "above-left" $[i-1, j-1]$.

Smith-Waterman algorithm works similarly except that Eqn.(1) is modified for local alignment purpose. Particularly, one more case is added to reflect the possibility of starting a new local alignment. As such, the entry of $\mathcal{M}(i,j)$ is refined with the value $\max(\mathcal{M}(i,j), 0)$ during the iterative calculation of Eqn.(1). The traceback is not performed from the bottommost rightmost cell, but from the cell with the maximum value[7]. Keen readers might find another interesting application with the Smith-Waterman algorithm: if we associate a metric (e.g., number of matches) to the best alignment between subsequences of $\mathcal{F}_1$ and $\mathcal{F}_2$, the metric can also be used to indicate the similarity among the two sequences. In fact, the Smith-Waterman alignment is used in the next algorithm as a similarity-based scoring mechanism to build the relevant phylogenetic tree from a number of worm infection sequences.

It is interesting that most existing self-propagating worms are still primitive with no *behavior-polymorphic* capabilities. Our experiments in Section 4 show that pairwise alignment is highly effective in extracting worm-characterizing behavioral footprints. However, even though the majority of current self-propagating worms are not polymorphic in behavior, it is likely that future worms will be more intelligent, given that certain libraries [16, 25, 42] rendering code polymorphic are readily available. As a result, the pairwise alignment algorithm might not be capable of characterizing future worms.

---

[7]A tie can be broken by arbitrarily choosing any cell with the maximum value.

3.3 Phylogenetic Tree Algorithm

In this section, we propose a robust algorithm to extract behavioral footprints of more advanced worms. The algorithm is based on our observation on the existence of *behavioral invariants*. Before presenting the algorithm, we further justify the existence of behavioral invariants even in advanced worms.

3.3.1 Examining Behavior Invariants

Similar to its counterpart - the content-polymorphic worm, a behavior-polymorphic worm could exhibit varying behavior during different infection sessions. Here we consider single-vector worms which target one vulnerability, because a multi-vector worm can be considered as the combination of several single-vector worm variants, each with only one infection vector. We have so far studied at least twenty self-propagating worms and their variants (including behavior-polymorphic worms we synthesize) targeting a number of different services on top of various operating systems, and have found behavioral invariant in *each* of them. Although we are *not* claiming that all worms will exhibit behavioral invariants, a significant fraction of them do, because behavioral invariants typically result from (1) restrictions imposed for successful exploitations, (2) common components in each infection session (e.g., same payload and replication method of a worm), or (3) in some cases, a worm's idiosyncrasies in its exploitation means, replication mechanisms, and self-carrying payloads. We present two examples to illustrate how restrictions for successful exploitations determine a worm's behavior invariants.

The first example is related to the OpenSSL heap-based buffer overflow exploited by the Slapper worm. As described in [37], the overflow is used *twice* by the worm to achieve a reliable infection. The first OpenSSL exploitation only attempts to locate the over-writable heap address within the vulnerable Apache address space, which is *hardly* predictable across all the servers. After the first exploitation, the acquired heap address is patched in the attack buffer within the second OpenSSL exploitation. It is expected that this two-phase exploitation enables a reliable infection. However, it has one more restriction that the two Apache processes handling these two exploitation connections should have the same heap layout, and thus ensure the validity of the heap address obtained from the first exploitation connection to the second exploitation connection. To satisfy the restriction, the worm must first exhaust the Apache's pool of servers before actual exploitation. The exhaustion is achieved by opening a succession of 20 connections[8] so that two fresh Apache processes can be spawned to handle the two exploitation connections. As such, a reliable Slapper worm infection requires a series of resource-exhausting TCP connections and two additional exploitations. These requirements will be essentially reflected as the *behavioral invariants* or *invariant subsequence* in Slapper worm's behavioral footprint. We will further analyze the Slapper worm in Section 4.4.3.

The second example is related to the Slammer worm exploiting a simple buffer-overrun vulnerability in MS SQL servers. Due to the nature of the exposed vulnerability, only a single UDP packet with the following properties:

---

[8]The number 20 is related to the *StartServers* entry in the Apache configuration file.

destination port 1434, packet type 4, and size larger than 60 bytes, will successfully trigger the buffer overflow. Such requirement leads to the behavioral invariant of the Slammer worm, and is reflected in its behavioral footprint as:

$< UDP, */*, 1434/*, payload : \text{``}|04|\text{''}, size > 60 >$.

3.3.2 Building the Phylogenetic Tree

By operating over a collection of a worm's infection sequences[9], the worm's behavioral invariants can be reliably extracted by advanced sequence analysis techniques. More specifically, pairwise alignment is first performed to derive their relative similarities with each other (a.k.a., the Smith-Waterman alignment). Based on the similarities, a *phylogenetic tree* will be built to guide the final stage of multiple sequence alignment to expose and extract the behavioral invariants.

A phylogenetic tree is originally proposed to depict the evolutionary relationships of a group of life organisms. Here we are building the phylogenetic tree to extract the most fundamental footprint subsequences or invariants that are embedded within a number of related infection sequences $\mathscr{F}_k$, $k = 1..n$. Some of the sequences might be explicitly mutated by inserting irrelevant subsequences or replacing some subsequence with another functionally-equivalent string. An algorithm called UPGMA [18] originally used in gene analysis has been applied to construct such a tree. Initially, each sequence $\mathscr{F}_k$ is considered as a cluster C_k. These clusters are iteratively grouped with the most related one so that, eventually, there is only one cluster left. The relatedness or similarity between two clusters C_i and C_j is defined as d_{ij}:

$$d_{ij} = \frac{1}{\|C_i\| \|C_j\|} \sum_{p \in C_i, q \in C_j} d_{pq} \tag{2}$$

where $\|C_i\|$ and $\|C_j\|$ denote the number of sequences in clusters C_i and C_j. The value of d_{pq} is derived based on the Smith-Waterman scoring algorithm. The clustering algorithm is further described as follows:

PHYLOGENETICTREECONSTRUCTION($\mathscr{F}_k, k = 1 \cdots n$)

1   $C \leftarrow \emptyset; T \leftarrow \emptyset$
2   **for** each sequence $\mathscr{F}_i$ $i \in 1..n$
3   **do**
4   Assign a cluster $C_i \leftarrow \{\mathscr{F}_i\}$
5   and add it into $C \leftarrow C \bigcup C_i$
6   Define a leaf N_i in T for $\mathscr{F}_i$
7   **for** each any other sequence $\mathscr{F}_j, j \in i + 1 \cdots n$
8   **do**
9   Calculate the similarity between $\mathscr{F}_i$ and $\mathscr{F}_j$
10                   $d_{ij} \leftarrow$ SMITH-WATERMAN($\mathscr{F}_i, \mathscr{F}_j$)
11   **while** $\|C\| \neq 1$

---

[9]Such infection sequences can be safely collected by unleashing the worm in our experimental environment (Section 4.1.2).

12	**do** Determine the two clusters C_i and C_j
13	s.t. d_{ij} is maximum
14	Define a new cluster $C_k = C_i \bigcup C_j$
15	and calculate d_{kl} for all l
16	Remove C_i and C_j from C, i.e., $C \leftarrow C - C_i - C_j$
17	Add C_k to C, i.e., $C \leftarrow C \bigcup C_k$
18	Add a parent node N_k to T with children N_i and N_j
19	**return** T

The calculation in d_{kl} in step 15 can be conveniently performed based on following equation:

$$d_{kl} = \frac{d_{il} \, \|C_i\| + d_{jl} \, \|C_j\|}{\|C_i\| + \|C_j\|} \tag{3}$$

The time and space complexity of the algorithm is $O(n^2)$, since there are $n-1$ iterations, with $O(n)$ steps in each one.

3.3.3 Aligning Multiple Sequences

The phylogenetic tree is used to categorize the worm footprint sequences and guide the actual alignment of multiple sequences. Within the generated tree T the leaves contain the raw footprint sequences while intermediate nodes contain the sequences representing their children nodes. A simple post-order tree traversal algorithm (shown below) can be recursively applied to construct the representative sequences until the root of the tree T is reached.

MULTIPLESEQUENCEALIGNMENT($T : PhylogeneticTree$)

1	**if** $T \neq$ **NULL**
2	**then** MULTIPLESEQUENCEALIGNMENT($T.left$);
3	MULTIPLESEQUENCEALIGNMENT($T.right$);
4	**if** $T.left \neq$ **NULL AND** $T.right \neq$ **NULL**
5	**then** $T.sequence \leftarrow$
6	NEEDLEMAN-WUNSCH($T.left, T.right$)

The actual sequence construction is based on the global alignment alignment, i.e., the Needleman-Wunsch algorithm (Section 3.2). An example run of the algorithm against a Welchia worm variant is illustrated in Fig 5. The sequence shown at the root of the tree

$$< variable > C_1 F_1 \overleftarrow{F_1} C_2 \overleftarrow{U_1} U_1 < variable > R_2$$

is extracted as the behavioral footprint for the Welchia worm.

$$I\,I\,C_1F_1F_1C_2U_1U_1 - - R_2$$

$$C_1F_1F_1C_2U_1U_1 - R_2$$

$$I\,I\,C_1F_1F_1C_2U_1U_1 - - R_2 \qquad I\,I\,C_1F_1F_1C_2U_1U_1U_2U_2R_2 \qquad - - C_1F_1F_1C_2U_1U_1U_2U_2R_2$$

Sequence 1   Sequence 2   Sequence 3

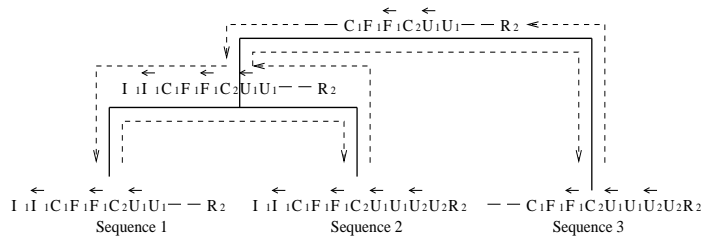**Figure 5. An Example Alignment of Multiple Worm Sequences/Footprints. These three sequences are modified from the original Welchia worm sequence (Section 4.3) for illustration purpose. The first sequence ignores the second *tftp* UDP connection for the $SVCHOST.exe$ file. The second sequence contains the original infection sequence. The last one ignores the first ICMP probing.**

4 Evaluation

In this section, we first describe our experimental environment (Section 4.1), which is used to trap "live" worms and analyze historical worms. We then derive these worms' behavioral footprints (Section 4.2) and demonstrate their validity by showing that they not only differ significantly from normal service access behavior, but also accurately characterize the behavior of corresponding worms. Later, by comparing with content-based fingerprinting, our experiments further demonstrate the uniqueness (Section 4.3) and robustness (Section 4.4) of behavioral footprinting in worm recognition and identification.

4.1 Experimental Environments

Behavioral footprints characterize worms by capturing their dynamic infection sequences. The difficulty in validating the proposed scheme lies in the safe collection of infection sequences of real-world worms. To address this challenge, we have implemented and deployed (1) Collapsar [22], a honeyfarm architecture to trap live, real-world worms and (2) vGround [23], a virtual worm playground environment to safely unleash and observe the dynamic infection behavior of historical real-world worms.

4.1.1 Trapping Live Worms

The goal of trapping live worms is to collect their malicious infection sequences. To achieve this goal, there are two important considerations:

- *Honeypot services on dark address space*   There is a high concentration of malicious traffic in a dark (namely, unallocated) IP address space. By further deploying *high-interaction* honeypot services [22] in such dark address spaces, we are able to collect original traces of self-propagating worms. In our experimental environment (Collapsar), honeypots are deployed using virtual machines enabled by both VMware [6] and User-Mode Linux (UML) [17].

- *Off-site and distributed worm capture* A high-interaction honeypot can be infected as a real host by propagating worms. To collect a diverse set of worm infection traces, we have developed a number of honeypot traffic redirectors, which forward dark space traffic from distributed participating sites to a centralized location for easy worm trace collection. By using traffic re-direction techniques such as Proxy-ARP and GRE [20, 21], the traffic redirectors are transparent to remote worm infectors.

We first started the prototype of Collapsar in February 2003 and it was initially deployed in August 2003. Three redirectors are deployed in three Ethernet-based production networks and forward traffic to a centralized facility, which is located in a separate Ethernet LAN. Encouragingly, right after its deployment, it successfully captured one instance of the MSBlaster worm. Later in August 2004, we expanded Collapsar deployment to three more production networks: one local subnet network (20 IP addresses), one wireless LAN, and one DSL network. The DSL network is located in another administrative domain. The honeypots run a variety of commodity operating systems, including RedHat Linux 7.2/8.0, Windows XP Home Edition, FreeBSD 4.2, and Solaris 8.0. All traffic from/to these honeypots are fully logged through the *tcpdump* tool. Using Collapsar, a number of live real-world worms such as MSBlaster [9], Enbiei [8], Welchia [12], and Sasser [13] are captured [10]. Further analysis of the worms are presented in Section 4.2.

4.1.2 Analyzing Historical Worms

Our honeyfarm architecture Collapsar is able to capture currently propagating worms. However, it is unable to analyze other historical worms. To this end, we have created a virtual worm playground environment called vGround, where worms can be safely unleashed and monitored. vGround has the following features:

- *High fidelity with full-system virtualization* Within a vGround, realistic end-hosts and network entities (e.g., routers and firewalls) are emulated using virtual machines [6, 17]. The adoption of virtual machines brings great convenience and flexibility in supporting unmodified vulnerable services and operating systems.

- *Strict confinement with link-layer network virtualization* A vGround is used to experiment with malicious, destructive worms. A confined virtual network is necessary to strictly contain malicious worm traffic and worm damage. To this end, we have developed a link-layer network virtualization technique to safely intercept and completely confine worm traffic within the virtual playground. Our current vGround prototype supports both VMware and UML-based virtual machines.
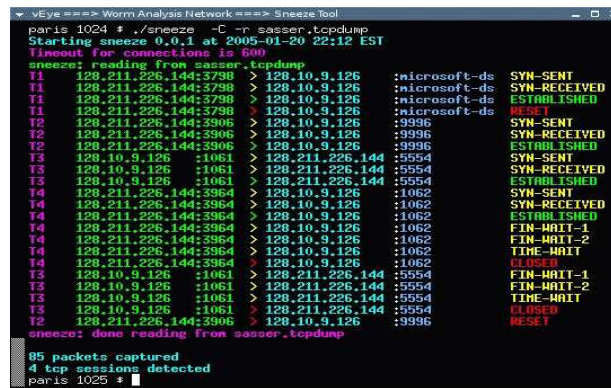
We have successfully experimented with a number of historical worms and their variants, including Lion worm [4], Slapper worm [37], Ramen worm [7], and SARS worm [10]. For each experiment, the dynamic infection traces are captured using the *tcpdump* tool. The analysis is presented in the next section.

---

[10]Note that due to the limited scale of our current deployment, it is less likely to capture all of Internet worms which are active particularly at their early stage. However, Collapsar did capture Sasser worm on the *first* day (May 1, 2004) of its outbreak.

4.2 Extracting Behavioral Footprints

With collected *tcpdump* log files, the next step is to extract flow sequences relevant to worm infections. We develop a tool named *sneeze* for this purpose: all TCP/UDP/ICMP flow sequences contained within the log are extracted and additional packet reassembly or re-ordering, if necessary, is also performed. These TCP/UDP/ICMP sequences are separated with respect to each address pair and are further ordered based on the associated time-stamp. The duration and payload size of each flow is also automatically calculated by sneeze.

An example output from sneeze is shown in Figure 6. The trace input is related to an complete infection session of the Sasser worm, which is captured by Collapsar on May 1, 2004.

Figure 6. An Example Output of the Sneeze Tool

Note that when analyzing related TCP flows, sneeze is able to track relevant TCP states. Specifically, within extracted TCP flows, any TCP control packet with SYN, ACK, FIN, or RST bit set are contained within the resulting infection sequence. The TCP data packets (though ACK bit turned on) are usually ignored. However, as discussed in Section 3.1, additional content sequence, protocol compliance analysis, or even vulnerability-related information can be integrated here to further enrich the accuracy and effectiveness of worm footprints. We are currently extending the prototype for such integration. UDP and ICMP flows are also recorded within the sequence.

By considering each interaction as the behavioral phenotype, the algorithm described in Section 3 is applied on these multiple interaction sequences to extract representative behavioral footprints. The results are shown in Table 1. Within the table, those letters denote either TCP flows with different control bits or UDP/ICMP flows. Also, the letter C_i represents the well-known three-way TCP connection handshake. However, the same letter usually means different field contents (e.g., the destination port number) for different footprints.

It is encouraging to note that we are able to reliably extract behavioral footprints for all worms examined. The footprint of the MSBlaster worm has been pictorially shown in Figure 2. Welchia worm[11] is similar to MSBlaster worm except that an initial ICMP probing packet is generated before actual infection and the second TCP connection

---

[11]The Welchia worm is a multi-vector worm, which takes advantage of two vulnerabilities, i.e., the RPC-DCOM vulnerability (MS03-026) and WebDAV vulnerability (MS03-007). Due to the lack of the vulnerable IIS server in our environment setup, the WebDAV-based infection is not able to be reproduced.

Name	Infection Vector	Behavioral Footprints Derived	Captured Date	Platforms
MSBlaster	RPC-DCOM vulnerability (MS03-026)	$C_1 R_1 C_2 \overleftarrow{U_1} U_1 R_2$	Aug. 28, 2003	Windows
Welchia	RPC-DCOM vulnerability (MS03-026) WebDAV vulnerability (MS03-007)	$I_1 \overleftarrow{I_1} C_1 F_1 \overleftarrow{F_1} \overleftarrow{C_2} \overleftarrow{U_1} U_1 \overleftarrow{U_2} U_2 R_2$	Sep. 17, 2003	Windows
Enbiei	RPC-DCOM vulnerability (MS03-026)	$C_1 R_1 C_2 \overleftarrow{U_1} U_1 R_2$	Oct. 12, 2003	Windows
Sasser	LSASS vulnerability (MS04-011)	$C_1 R_1 C_2 \overleftarrow{C_3} C4 F_4 \overleftarrow{F_4} F_3 \overleftarrow{F_3} R_2$	May 1, 2004	Windows
Ramen	LPRng vulnerability (CVE-2000-0917)	$S_1^F \overleftarrow{S_1} R_1 C_2 F_2 \overleftarrow{F_2} C_3 \overleftarrow{C_4} F_4 \overleftarrow{F_4}$	-	Linux
	WU-FTPD vulnerability (CVE-2000-0573)	$S_1^F \overleftarrow{S_1} R_1 C_2 R_2 C_3 R_3$ (flawed)	-	
	NFS-UTILS vulnerability (CVE-2000-0666)	$S_1^F \overleftarrow{S_1} R_1 U_1 \overleftarrow{U_1} U_2 C_2 \overleftarrow{C_3} F_3 \overleftarrow{F_3} R_2$	-	
Lion	BIND vulnerability (CA-2001-02)	$C_1 F_1 \overleftarrow{F_1} C_2 \overleftarrow{C_3} F_3 \overleftarrow{F_3} R_2$	-	Linux
Slapper	OpenSSL vulnerability (CA-2002-23)	$C_1 F_1 \overleftarrow{F_1} C_2 \overleftarrow{F_2} \prod_{i=3}^{22} C_i C_{23} C_{24}$	-	Linux
SARS	Samba vulnerability (CAN-2003-0201)	$U_1 \overleftarrow{U_1} U_2 \overleftarrow{U_2} C_1 F_1 C_2 F_2 \overleftarrow{F_2} C_3 \overleftarrow{C_4} F_4 F_4 R_3$	-	Linux/BSD

Table 1. Characterizing Self-Propagating Worms with Behavioral Footprints

(a) The Behavioral Footprint of the Ramen Worm Exploiting the LPRng IV

(b) The Behavioral Footprint of the Ramen Worm Exploiting the WU-FTPD IV (w/ Flawed Implementation)

(c) The Behavioral Footprint of the Ramen Worm Exploiting the RPC.STATD IV
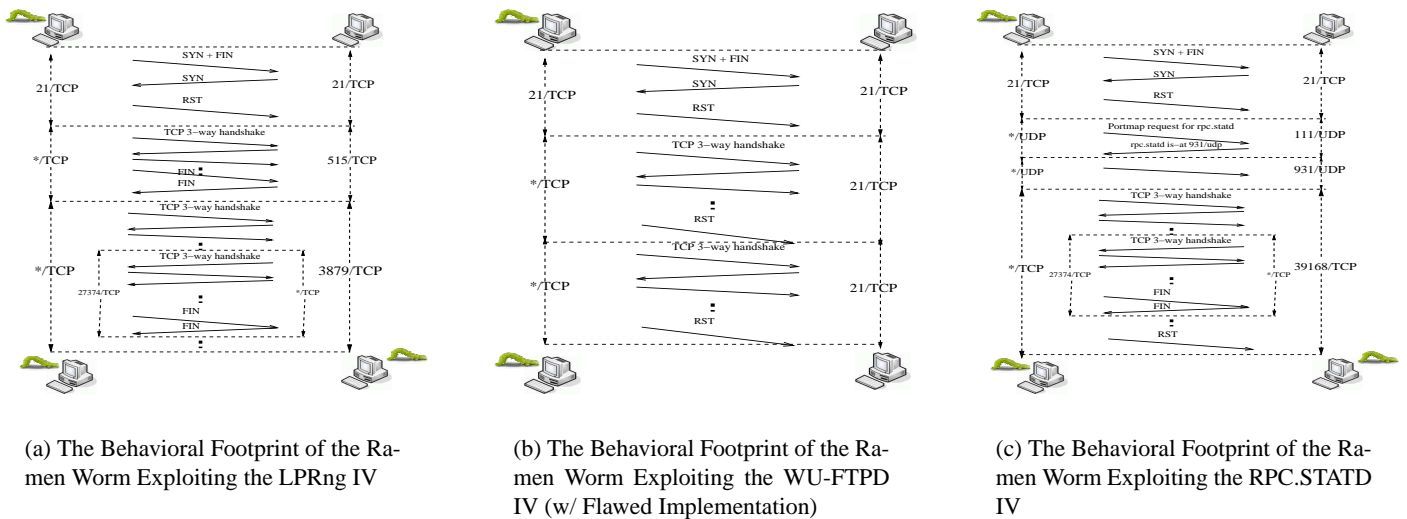
Figure 7. Behavioral Footprints of the Ramen Worm, a Multi-Vector Worm

($\overleftarrow{C_2}$) is initiated from the victim with a connect-back shell-code. Note that though MSBlaster and Welchia exploit the *same* vulnerability, their behavioral footprints are *different*. Enbiei worm exhibits a footprint similar to that of MSBlaster worm but has a different worm binary and payload. Sasser worm uses the *ftp* protocol ($\overleftarrow{C_3}$) to download the worm replica. Within the ftp session, a *PORT* primitive is initiated to start another reverse connect-back activity (C_4).

Table 1 also shows the footprints of several historical worms, which are derived from our worm playground environment (vGround). Ramen worm is a multi-vector worm, which has three infection vectors (IVs): LPRng (CVE-2000-0917), wu-ftpd (CVE-2000-0573), and nfs-utils (CVE-2000-0666). Interestingly, the exploitation on the wu-ftp IV is flawed, which could not result in a success infection. The footprints for Ramen worms on different infection vectors are also visualized in Figure 7. Note that an initial TCP control packet with SYN and FIN bits (S_1^F) set, source port being 21, and destination port being 21, is used to probe the victim among *all* three IV-specific footprints. Another three examined worms, i.e., Lion, Slapper, and SARS worms, are single-vector worms. Lion worm has been

described in Section 2.3. We defer the discussion of Slapper worm in Section 4.4. SARS worm is a multi-platform worm, which is able to spread across various platforms (e.g, Debian 3.0, Gentoo 1.4.x, Mandrake 8.x/9.0, Redhat 6.x/7.x/8.0/9.0, Slackware 8.x/9.0, SuSE 7.x/8.x, FreeBSD 4.x/5.0, NetBSD 1.5/1.6, and OpenBSD 3.2). Its visual presentation is omitted due to space constraint.

We would like to highlight that all of these behavior sequences are uniquely exhibited by the corresponding worms and to the best of our knowledge, are *not* exhibited within any other normal accesses to corresponding services.

4.3 Uniqueness of Behavioral Footprinting

Behavioral footprinting captures worms' characteristics based on their infection cycles. In this section, we demonstrate the benefit obtained from this new dimension for worm identification. To this end, we perform trace-driven worm recognition experiments. More specifically, the sneeze utility (Section 4.2) is modified to serve as a worm recognition tool using worms' behavioral footprints. We use a 7-hour trace ($80M$ containing 3 live worm infections) collected by Collapsar [22] to demonstrate the benefit of worm behavioral footprinting. For comparison, we first apply a popular open-source content-based IDS, i.e., *snort*, to detect possible intrusions[12]. Our own tool, *sneeze*, is then applied to the same trace. Sneeze is able to identify all three worm infections in the trace with 0% false positive. The results from snort and sneeze are shown in Table 2 and Figure 8, respectively.

	Snort Signature	# Alerts	# Sources	# Dests
1	NETBIOS DCERPC ISystemActivator path overflow attempt little endian	539	12	201
2	NETBIOS SMB-DS Session Setup And X request unicode username overflow attempt	15	1	1
3	NETBIOS SMB-DS DCERPC NTLMSSP asn1 overflow attempt	14	2	1
4	ICMP Source Quench	28	28	1
5	ICMP redirect host	27	1	1
6	TFTP Get	24	1	4
7	ICMP Large ICMP Packet	3	2	2
8	ICMP PING CyberKit 2.2 Windows	307551	33	153549
9	ICMP Destination Unreachable Communication Administratively Prohibited	156	2	1
10	SCAN UPnP service discover attempt	30	1	1
11	NETBIOS SMB-DS IPC$ share unicode access	6	3	1

Table 2. Worm Detection with Content Fingerprints

As Table 2 shows, snort performs reasonably well in recognizing various RPC DCOM buffer overflow attempts, and in reporting numerous alerts for "ICMP PING CyberKit 2.2 Windows", which correspond to the probing traffic from Welchia worms. However, these alerts are distinct alerts even though they might be caused by the same worm infection session. Figure 8 shows the result from sneeze. Sneeze naturally identifies 3 *successful* worm infections and also reports 2 *unsuccessful* worm infections (which were not discovered in [22]). Further manual analysis shows that one unsuccessful worm infection has erroneously generated a *wrong* address (*192.168.1.59*) to download the worm replica while another unsuccessful infection has a flawed exploitation in binding the command shell service. Since *tftp* protocol is used for all these worms, we would like to compare both outputs in this aspect. Table 2 reports 4

---

[12]The signature database used in the snort has been updated to contain *latest* content fingerprints for known intrusions.

TIME	DUR.	SRC:PORT <-> DST:PORT	SERVICE	BYTES	SIGNATURE
Thu Nov 27 02:48:57 2003	8 s	81.168.168.127:4030 -> 128.10.9.127:135	135/TCP	1896 bytes	
Thu Nov 27 02:49:05 2003	25 s	81.168.168.127:4043 -> 128.10.9.127:4444	krb524/TCP	671 bytes	Enbiei Worm [sid]
Thu Nov 27 02:49:05 2003	21 s	128.10.9.127:1036 <-> 81.168.168.127:69	tftp/UDP	12134 bytes	
Thu Nov 27 04:58:24 2003	10 s	204.188.17.242:42948 -> 128.10.9.127:135	135/TCP	1908 bytes	MSBlaster/Enbiei Worm? [sid]
Thu Nov 27 04:58:35 2003	25 s	204.188.17.242:43362 -> 128.10.9.127:4444	krb524/TCP	349 bytes	
Thu Nov 27 06:08:24 2003	7 s	208.29.230.14:56429 -> 128.10.9.127:135	135/TCP	1572 bytes	MSBlaster/Enbiei Worm? [sid]
Thu Nov 27 06:08:30 2003	0 s	208.29.230.14:57021 -> 128.10.9.127:4444	krb524/TCP	78 bytes	
Thu Nov 27 07:23:46 2003	2 s	66.66.221.210:4581 -> 128.10.9.127:135	135/TCP	1896 bytes	MSBlaster Worm [sid]
Thu Nov 27 07:23:49 2003	17 s	66.66.221.210:4599 -> 128.10.9.127:4444	krb524/TCP	666 bytes	
Thu Nov 27 07:23:49 2003	11 s	128.10.9.127:1552 <-> 66.66.221.210:69	tftp/UDP	6372 bytes	
Thu Nov 27 08:03:55 2003	0 s	128.10.16.220 -> 128.10.9.127	ICMP	64 bytes	Welchia Worm [sid]
Thu Nov 27 08:03:55 2003	0 s	128.10.9.127 -> 128.10.16.220	ICMP	64 bytes	
Thu Nov 27 08:03:55 2003	0 s	128.10.16.220:1567 -> 128.10.9.127:135	135/TCP	436 bytes	
Thu Nov 27 08:03:55 2003	4 s	128.10.9.127:3912 -> 128.10.16.220:707	707/TCP	1686 bytes	
Thu Nov 27 08:03:56 2003	0 s	128.10.9.127:3953 <-> 128.10.16.220:69	tftp/UDP	40 bytes	
Thu Nov 27 08:03:56 2003	0 s	128.10.9.127:3953 <-> 128.10.16.220:1605	1605/UDP	20196 bytes	
Thu Nov 27 08:03:56 2003	0 s	128.10.9.127:3954 <-> 128.10.16.220:69	tftp/UDP	40 bytes	
Thu Nov 27 08:03:56 2003	0 s	128.10.9.127:3954 <-> 128.10.16.220:1606	1606/UDP	10488 bytes	

Figure 8. Worm Detection and Identification with Behavioral Footprints

alerts with messages "TFTP GET" while Figure 8 further shows that one tftp is related to the Enbiei worm, one tftp is related to the MSBlaster worm, and the other two tftp are related to the Welchia worm, which uses one *tftp* session to download the file $DLLHOST.exe$ (the worm payload) and the other *tftp* session for $SVCHOST.exe$ (a tftpd daemon).

The comparison clearly demonstrates the uniqueness of the behavioral footprinting dimension. From the content dimension, snort inspects every incoming/outgoing packet and raises a general alert if a malicious content sequence is detected. From the behavior dimension, sneeze is able to recognize individual worms once the behavioral footprints are matched.

4.4 Robustness of Behavioral Footprinting

Previous subsections demonstrate the feasibility and effectiveness in extracting behavioral footprints for worm characterization and recognition. In the following, we further compare the robustness of behavioral footprinting with the popular content fingerprinting dimension under three different types of mutation attacks.

4.4.1 A Content-Mutation Attack

In this experiment, we examine the robustness under a simple content-mutation attack. The Slapper worm is chosen for the comparison.

Within the snort system, there are two Slapper-related signatures shown in Table 3. To compare, a vGround with 100 virtual nodes is firstly instantiated and an instance of the original Slapper worm is introduced into the environment. A *tcpdump* trace file containing the infection of slapper worms is randomly selected. Snort reports two alerts on the log file with the message "MISC OpenSSL Worm Traffic" and five alerts for "WEB-MISC Bad HTTP/1.1 Request".

Then, another experiment is conducted by performing a simple mutation of the Slapper worm content: replacing the string "TERM=xterm" with "TERM=linux" and "GET / HTTP/1.1" (the banner grabbing routine) with "GET

	Snort Signature	Alert Message
1	TERM=xterm	MISC OpenSSL Worm Traffic
2	GET / HTTP/1.1	WEB-MISC Bad HTTP/1.1 Request, Potentially Worm Attack

Table 3. Snort Signatures for the Slapper Worm

/ HTTP/10". The same vGround is used to experiment with the modified Slapper worm. Once the contents are mutated, no alert is generated by snort from any worm propagation trace. Other recent work [45] has also confirmed the in-effectiveness of content fingerprints under content mutation attacks.
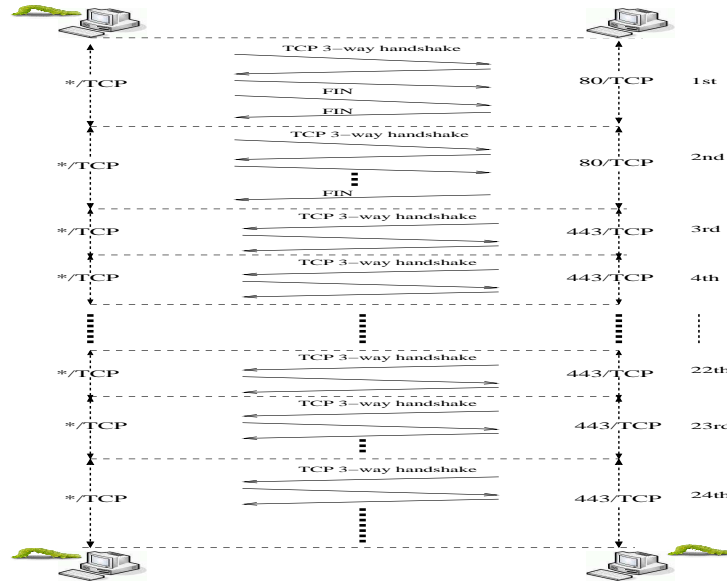

Figure 9. The Behavior Footprint of the Slapper Worm

Behavioral footprinting demonstrates its robustness under this attack. In both cases, sneeze is able to identify the same behavioral sequence of the Slapper worm. As visualized in Figure 9, the slapper worm firstly opens a normal TCP connection ($C_1 F_1 \overleftarrow{F_1}$) against port 80 checking the reachability of remote host; It then issues an invalid HTTP GET request ($C_2 \overleftarrow{F_2}$, *half-close*; containing the second content signature used in snort) to grab the server banner and query the version of web server; Later on, it further establishes 20 simultaneous plain TCP connections ($\prod_{i=3}^{22} C_i$, *opened* without any payload and never shutdown) on 443 port to prepare for the two following exploitations (C_{23}, C_{24}). Finally, a flurry ($> 10,000$) of short packets (1 byte in payload) can be observed for the C_{24} TCP connection.

4.4.2 A Traffic-Encryption Attack

In this experiment, we examine the robustness when the whole worm traffic is encrypted.

As pointed out in [37], the original Slapper worm is propagated through the transmission of a *uuencoded* version of the unencrypted worm source code. In this experiment, a synthesized Slapper variant is first instructed to encrypt the worm source file before propagation and then it is instructed to decrypt the file before compiling it and executing the

Figure 10. N-Gram Analysis on the Original Slapper Worm

Figure 11. N-Gram Analysis of the Slapper Worm Variant with Encrypted Transmission

worm binary in the infected victim. N-Gram analysis (counting the frequency of n-length combinations of bytes) is performed over two infection instances: one for the original Slapper worm with transfer of unencrypted worm source (shown in Figure 10) and the other for the Slapper worm variant with encrypted source (shown in Figure 11).

The N-gram analysis on the original Slapper worm trace shows several common strings with a much higher frequency of occurrence than other strings. However, these strings are not the same as the signature adopted in snort to detect Slapper worms. In fact, the signature used in snort "TERM=xterm" only happens twice within the N-Gram analysis. It suggests that the most-recurring content blocks are not necessarily suitable for signature purpose. Once the transmission is encrypted, almost every string has equal probability of occurrence. On the other hand, the sequence $C_1 F_1 \overleftarrow{F_1} C_2 \overleftarrow{F_2} \prod_{i=3}^{22} C_i C_{23} C_{24}$ is exhibited in both the original and the synthesized Slapper worm infections, which demonstrates the applicability of behavioral footprinting even to worms that encrypt their traffic.

Figure 12. A Phylogenetic Tree Built from 20 **Polymorphic Behavioral Sequences of the Slapper Worm Variant**

4.4.3 A Behavior-Polymorphism Attack

The previous two experiments demonstrate the robustness of worm footprints against content-mutation and traffic encryption attacks. In this experiment, we further examine the robustness of worm footprints against a behavior-

polymorphism attack.

Instead of following the behavior sequence shown in its original footprint, a behavior-polymorphic Slapper worm variant is crafted, which is capable of (1) intentionally introducing an arbitrary number of irrelevant or miscellaneous sequences during the infection [13]; (2) intentionally adding a certain random timing delay among any two consecutive infection steps; and (3) intelligently changing the IP address from which to download the attack payload, including the worm replica. However, as restricted by the way to exploit the OpenSSL heap vulnerability (Section 3.3.1), the temporal order in the original behavior sequence has to be maintained to ensure successful infection.

A vGround with 1500 virtual nodes is constructed and all successful infection sessions are recorded for sequence analysis. For brevity and readability, Figure 12 only shows the phylogenetic tree built from collected traces with 20 infection instances. The numbers in the leaf nodes are index numbers from 1 to 20. The values in intermediate nodes indicate normalized similarity ($[0, 1]$) based on Smith-Waterman algorithm (Section 3.2). Lower value indicates higher similarity between the two sub-clusters. The penalty used for each gap through the algorithm is $p = -2$ and the scoring matrix used for Smith-Waterman algorithm is

$$s(i,j) = \begin{cases} 2, & x_i = y_j \\ -1, & otherwise. \end{cases} \tag{4}$$

As we observe, the phylogenetic tree algorithm is still able to extract the most critical part of the original behavior sequence: $\prod_{i=3}^{22} C_i C_{23} C_{24}$, demonstrating the resilience of behavioral footprinting against the behavior-polymorphism attack.

5 Limitations

As a new dimension to characterize self-propagating worms, behavioral footprinting shows great potential in identifying all infection incidents of each real-world worm we have experimented with. However, we would like to point out that behavioral footprinting is proposed to enrich worm characterization along with other dimensions, e.g., content fingerprinting. It alone could lead to either incomplete or inaccurate worm characterization. In the following, we describe current limitations of behavioral footprinting. Such limitations also call for further improvement of this new dimension and the adoption of a multiple-dimensional approach to worm characterization and identification.

Behavior substitution attacks   Our current pairwise alignment algorithm leverages a basic sequence alignment technique, or more specifically, a simple predefined scoring matrix (Section 3.2), to align worm infection sequences where a worm-identifying behavioral footprint is derived. An attacker might intentionally introduce some *substitutable* subsequence, which attempts to corrupt the alignment process while still achieves its goal for infection or propagation. For example, within the Replication phase (Figure 1), different transport channels or even tunneling can

---

[13] We would like to point out that though the worm is able to initiate the connections (e.g., ICMP/TCP/UDP flows) to the victim node, it can not control the *reverse* direction as the victim is not under its control yet before a successful exploitation.

be leveraged to retrieve the worm replica.

However, if we re-examine the motivation behind the sequence analysis and consider each behavior substitution as a possible mutation, such attack is reminiscent of the classic challenge faced by biologists on how to optimally align gene sequences under possible mutations. It is interesting to note that two popular scoring matrices used in gene sequence alignment, i.e., PAM [18] and BLOSSOM [18], have been constantly evolved (and are still evolving) to reflect newly-discovered mutations for decades. Similarly, considering the scoring matrix behind our algorithm is primitive as it simply returns 1 if two flows are fully matched, additional efforts are still necessary to refine the scoring matrix. Fortunately, our application domain is different from the original biological domain as a worm usually can not evolve itself at runtime and has relatively limited number of possible substitutions. In addition, a worm capable of substituting its infection steps is likely to be more bloated (e.g., reflected by its replica size) than a compact one. An over-bloated worm is more likely to be detected in the first place.

Behavior-camouflaging attacks Behavioral footprinting is designed to capture a worm's infection steps exposed during its infection. A worm author might attempt to inject fake steps into the infection sequences. After these fake steps have been included in the worm's behavioral footprint, the worm will stop exhibiting these fake steps. As a result, the behavioral footprint will experience a sudden increase in false negatives because a full match against the footprint will fail from now on. The fundamental solution is to identify and remove those fake steps using techniques such as semantic-level analysis [34, 44], which is an on-going, challenging research topic. Another possible approach is to mitigate such attack by adopting partial instead of full footprint matching. However, a trade-off will be made to determine the confidence level of the partial matching to avoid the opposite, namely high false positives. Other dimensions (e.g., content fingerprinting) may provide complementary capability in this case.

6 Related work

Due to the significant threat imposed by self-propagating worms, security researchers have explored various dimensions to first capture worms' uniqueness and then apply them for worm identification.

Among the most notable, content fingerprinting [26, 28, 33, 43] has been widely examined and utilized to derive the most representative content sequences. Realizing the inconvenience in manually extract the content sequences, several systems such as Honeycomb [28], Autograph [26], EarlyBird [43] and Polygraph [33] have been recently proposed to *automate* the content-based signature extraction process. However, a content sequence is only able to detect the worm activity within one infection step or most likely, the exploitation stage (Figure 1). Behavioral footprinting instead is proposed to capture worms' uniqueness during its entire infection session, which nicely complements content fingerprinting (Section 4.3).

Another dimension, anomaly detection [1, 2, 3, 24, 29, 39, 48], leverages the insight that worms are likely to generate anomalous behaviors such as port scanning [24] and failed connection attempts [1, 2, 3], which are *different* from the normal behavior. Though such approach has been demonstrated effective in detecting worm infection, it is

not intended to *identify* worms. In other words, it mainly answers the question "is there a worm infection?", *not* the question "which worm is this?".

Other promising dimensions include vulnerability-specific characterization [1, 32, 47] and semantic-aware taintedness tracking [14, 15, 34, 40, 44]. Shield [47] or similarly Worm Vaccine [1] and Generic Exploit Blocking [32] propose the notion of vulnerability-specific signature and use it to accurately filter out attack flows. TaintCheck [34], Minos [15], Vigilante [14], and other related systems [40, 44] enable the detection of unknown attacks by associating a tag to untrusted information sources and reporting an alert if a tainted instruction is executed. These schemes are generally applicable even to detect unknown attacks or intrusions. While capable of detecting the occurrence of a possible exploitation, they *do not* attempt to characterize the *entire* worm infection process where exploitation is only one of the infection phases.

Different from these dimensions, behavioral footprinting is a new but complementary dimension. Recently, another related behavior-oriented approach [19] is proposed. However, it focuses on the inter-machine propagation pattern (tree) exhibited by worms as well as the similar payload from one machine to another. Moreover, it implicitly *assumes* the existence of worms' behavioral footprints, without justifying the existence and proposing the extraction of worm behavioral footprints, which is the focus of our work.

7   Conclusion

We have presented a new promising dimension, behavioral footprinting, to enrich the worm characterization space. Orthogonal and complementary to existing dimensions, behavioral footprinting characterizes the temporal worm infection process. Efficient and robust algorithms are proposed to accurately and reliably extract worm behavioral footprints. Our experiments with real-world worms, in comparison with the content-based fingerprinting approach, clearly demonstrate the feasibility, uniqueness, and robustness of behavioral footprinting.

References

[1] Arbor Networks: PeakFlow X. *http://www.arbornetworks.com/products_x.php*.

[2] CounterStorm. *http://www.sysd.com*.

[3] Mazu Networks. *http://www.mazunetworks.com/*.

[4] SANS Institute: Lion worm. *http://www.sans.com/y2k/lion.htm*.

[5] The Honeynet Project. *http://www.honeynet.org*.

[6] VMware. *http://www.vmware.com/*.

[7] Ramen Worms. *http://service1.symantec.com/sarc/sarc.nsf/html/Linux.Ramen.Worm.html*, January 2001.

[8] Enbiei Worms. *http://securityresponse.symantec.com/avcenter/venc/data/w32.blaster.f.worm.html*, September 2003.

[9] MSBlaster Worms. *CERT Advisory CA-2003-20 W32/Blaster Worms http://www.cert.org/advisories/CA-2003-20.html*, August 2003.

[10] SARS Worms. *http://www.patching.net/bbs/viewgooddoc.asp?id=25669&bordid=4*, June 2003.

[11] SoBig Worms. *http://www.cert.org/incident_notes/IN-2003-03.htm*, 2003.

[12] Welchia Worm. *http://securityresponse.symantec.com/avcenter/venc/data/w32.welchia.worm.html*, August 2003.

[13] Sasser Worms. *http://www.microsoft.com/security/incident/sasser.asp*, May 2004.

[14] Manuel Costa, Jon Crowcroft, Miguel Castro, and Antony Rowstron. Can we contain Internet worms? *Proceedings of the Third Workshop on Hot Topics in Networking (HotNets-III)*, November 2004.

[15] Jedidiah R. Crandall and Frederic T. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. *Proceedings of 37th International Symposium on Microarchitecture*, October 2004.

[16] T. Detristan, T. Ulenspiegel, Y. Malcom, and M. Underduk. Polymorphic Shellcode Engine Using Spectrum Analysis. *Phrack Issue 0x3d*, 2003.

[17] J. Dike. User Mode Linux. *http://user-mode-linux.sourceforge.net*.

[18] R Durbin, Sean Eddy, and Anders Krogh. *Biological Sequence Analysis*. Cambridge University Press, ISBN: 0521629713, 1998.

[19] Daniel R. Ellis, John G. Aiken, Kira S. Attwood, and Scott D. Tenaglia. A Behavioral Approach To Worm Detection. *Proceedings of the 2004 ACM workshop on Rapid malcode*, October 2004.

[20] S. Hanks, T. Li, D. Farinacci, and P. Traina. Generic Routing Encapsulation (GRE). *RFC 1701*, October 1994.

[21] S. Hanks, T. Li, D. Farinacci, and P. Traina. Generic Routing Encapsulation over IPv4 networks. *RFC 1702*, October 1994.

[22] Xuxian Jiang and Dongyan Xu. Collapsar: A VM-Based Architecture for Network Attack Detention Center. *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[23] Xuxian Jiang, Dongyan Xu, Helen J. Wang, and Eugene H. Spafford. Virtual Playgrounds for Worm Behavior Investigation. *Proceedings of the 8th RAID, Seattle, USA*, September 2005.

[24] Jaeyeon Jung, Vern Paxson, Arthur W. Berger, and Hari Balakrishnan. Fast Portscan Detection Using Sequential Hypothesis Testing. *Proceedings of IEEE Symposium on Security and Privacy, Oakland, CA*, May 2004.

[25] K2. ADMmutate. *CanSecWest/Core01 Conference, Vancouver http://www.ktwo.ca/ADMmutate-0.8.4.tar.gz*, March 2001.

[26] H. A. Kim and B. Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. *Proceedings of the 13th Usenix Security Symposium (Security 2004), San Diego, CA*, August 2004.

[27] Oleg Kolesnikov and Wenke Lee. Advanced Polymorphic Worms: Evading IDS by Blending in with Normal Traffic. *http://www.cc.gatech.edu/ ok/w/ok_pw.pdf*.

[28] Christian Kreibich and Jon Crowcroft. Honeycomb: Creating Intrusion Detection Signatures Using Honeypots. *ACM SIGCOMM Computer Communication Review*, January 2004.

[29] Michael E. Locasto, Janak J. Parekh, Angelos D. Keromytis, and Salvatore J. Stolfo. Towards Collaborative Security and P2P Intrusion Detection. *Proceedings of the 2005 IEEE Workshop on Information Assurance and Security, United States Military Academy, West Point, NY*, June 2005.

[30] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. The Spread of the Sapphire/Slammer Worm. *IEEE Security and Privacy, 1(4)*, July 2003.

[31] D. Moore, C. Shannon, and J. Brown. Code-Red: A Case Study on the Spread and Victims of an Internet Worm. *In Proceedings of the ACM Internet Measurement Workshop*, November 2002.

[32] Carey Nachenberg. From AntiVirus to AntiWorm: A New Strategy for A New Threat Landscape. *Invited talke in ACM Workshop on Rapid Malcode (WORM 2004)*, October 2004.

[33] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically Generating Signatures for Polymorphic Worms. *Proceedings of IEEE Symposium on Security and Privacy, Oakland, CA*, May 2005.

[34] James Newsome and Dawn Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS'05)*, February 2005.

[35] WL Nyhan. Behavioral phenotypes in organic genetic disease. *Pediatric Research 6:1-9*, 1972.

[36] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks, 31(23-24):2345-2463*, 1999.

[37] Frederic Perriot and Peter Szor. An Analysis of the Slapper Worm Exploit. *Symantec White Paper http://securityresponse.symantec.com/avcenter/reference/analysis.slapper.worm.pdf*.

[38] N. Provos. A Virtual Honeypot Framework. *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[39] Svetlana Radosavac. Detection and Classification of Network Intrusions using Hidden Markov Models. *Master's Thesis, Institute for System Research, University of Maryland, http://techreports.isr.umd.edu/reports/2003/MS_2003-1.pdf*, May 2002.

[40] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel Roy, and Tudor Leu. A Dynamic Technique for Eliminating Buffer Overflow Vulnerabilities (and Other Memory Errors) . *Proceedings of the 2004 Annual Computer Security Applications Conference, Tucson, AZ*, December 2004.

[41] M. Roesch. Snort: Lightweight Intrusion Detection For Networks. *Proceedings of the 13th Systems Administration Conference(LISA), Seattle, Washington*, November 1999.

[42] M. Sedalo. Jempiscodes: Polymorphic shellcode generator. *http://securitylab.ru/tools/services/download/?ID=36712*, 2003.

[43] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated Worm Fingerprinting. *Proceedings of the ACM/USENIX Symposium on Operating System Design and Implementation, San Francisco, CA*, December 2004.

[44] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure Program Execution via Dynamic Information Flow Tracking. *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2004.

[45] Giovanni Vigna, Will Robertson, and Davide Balzarotti. Testing Intrusion Detection Signatures Using Mutant Exploits. *Proceedings of the ACM Conference on Computer and Communication Security (ACM CCS) 21-30 Washington, DC*, October 2004.

[46] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, Fidelity and Containment in the Potemkin Virtual Honeyfarm. *Proceedings of ACM Symposium on Operating Systems Principles (SOSP 2005)*, October 2005.

[47] Helen J. Wang, Chuanxiong Guo, Daniel R. Simon, and Alf Zugenmaier. Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits. *SIGCOMM 2004*, September 2004.

[48] Ke Wang and Salvatore J. Stolfo. Anomalous Payload-based Network Intrusion Detection. *Proceedings of the 7th International Symposium on Recent Advances in Intrusion Detection (RAID 2004), Sophia Antipolis, French Riviera, France*, September 2004.