Enhancing System-Called-Based Intrusion Detection with Protocol Context

Anyi Liu*, Xuxian Jiang†, Jing Jin*, Feng Mao‡, and Jim X. Chen*
*Department of Computer Science
George Mason University, Fairfax VA 22030
Email: aliu1, jjin3, jchen@gmu.edu
†North Carolina State University
Raleigh, NC 27695
Email: jiang@cs.ncsu.edu
‡EMC
Santa Clara, CA 95054
Email: fengmao@acm.org

***Abstract*—Building an accurate program model is challenging but vital for the development of an effective host-based intrusion detection system (IDS). The model should be designed to precisely reveal the intrinsic semantic logic of a program, which not only contains control-flows (e.g., system call sequences), but also data-flows as well as their inter-dependency. However, most existing intrusion detection models consider either control-flows or data-flows, but *not* both or their interweaved dependency, leading to inaccurate or incomplete program modeling. In this paper, we present a semantic flow-based model that seamlessly integrates control-flows, data-flows, as well as their inter-dependency, thus greatly improving the precision and completeness when modeling program behavior. More specifically, the semantic flow model describes program behavior in terms of basic *semantic units*, each of which semantically captures one essential aspect of a program's behavior. The relationship among these semantic units can be further obtained by applying the protocol knowledge behind the (server) program. We show that the integrated semantic flow model enables earlier detection and prevention of many attacks than existing approaches.**

***Keywords*-Intrusion detection; System calls; Protocol specification; Context.**

I. Introduction

Building an accurate program model is challenging but vital for the development of an effective host-based intrusion detection system (IDS). A strict model will likely generate alerts with high false positives while a loose model might not detect any advanced evasive attacks. To improve the detection accuracy, a number of models [4], [5], [12] have been proposed to precisely capture the intrinsic semantic logic of a program. Particularly, due to the efficiency and convenience in collecting system call logs as well as rich semantics of collected logs, system calls have been widely leveraged to build program models. For example, Forrest et al. [7] uses normal system call sequences to model program behavior and considers any violation as an intrusion; Gao et al. [5] applies a gray-box approach to reconstruct program execution graph and is able to detect anomaly system call sequences when any inconsistency is observed; Sekar et al. [1] leverages system call arguments to obtain a model that describes the inherent data-flow dependency.

From another perspective, note that a program's semantic logic usually contains control-flows (e.g., system call sequences), data-flows (e.g., system call argument relationships), and their inter-dependency. However, existing techniques consider either control-flows or data-flows, but *not* both, resulting in an inaccurate or incomplete program modeling. This weakness could be potentially exploited by advanced attackers to avoid their detection. For example, Wagner et al. [13] demonstrates that the mimicry attack can effectively evade the detection from system call sequence-based models and related IDSes.

To address the weakness, we present a new semantic flow-based model that naturally integrates control-flows, data-flows, and their inter-dependency. Different from previous program models, the semantic flow model describes program behavior in terms of basic *semantic units*. With collected system call sequences, arguments, as well as related run-time context information, each semantic unit semantically describes one essential aspect of a program's behavior. In addition, with the protocol knowledge behind the (server) program, the interweaved dependency among these semantic units can be naturally extracted and modeled. For example, the possible *data-control* relation describes the dependency from system call arguments to subsequent system calls and the *data-data* relation reveals the inherent semantic dependency among different system call arguments. Specifically, when compared with existing approaches, our semantic flow approach has the following three key advantages: (1) Logical integration of control-flows and data-flows. (2) Protocol-aware semantic analysis. (3) Early and accurate detection.

We have applied the semantic flow model to characterize most popular server programs (e.g., `httpd` and `ftpd`). For each one of them, we are able to observe those basic semantic units and then construct their semantic relations. The experimental results with real world attacks, including both control-flow and data-flow exploits, show that the semantic flow model can immediately detect them once any violation to the normal semantic flow model occurs, resulting in much earlier detection and prevention than existing approaches. We believe that the semantic flow model holds great promise for more precise and complete host-based intrusion detection.

 ISBN: 978-1-61208-010-9

II. Related Work

To construct a program behavior-based anomaly detection model, various approaches have been proposed. Starting from the work of Forrest et al. [7], the *black-box* approaches [12], [13] model the normal program behavior (e.g., based on system calls) and then detect intrusions by identifying anomaly within observed system calls. The *white-box* approaches apply static analysis on either source code [9], [14] or binary [6] to build program models. And the *gray-box* approaches further leverage the program runtime information to improve the accuracy of anomaly detection models [2], [4], [5]. Our work is more closely related to data-flow anomaly detection [1], which examines inherent data-flow dependencies among system call arguments to make the model more robust. However, none of the previous works utilizes protocol knowledge behind the modeled program, which inspires our work to fully exploit the semantic meanings of system call arguments and build semantic dependencies among extracted semantic units. Our approach makes one step further and allows to derive more complicated semantic dependencies, e.g., *data* → *control* and *control* → *data* relations. As such, our approach enables the construction of more accurate and complete program models for anomaly detection.

III. An Illustrative Example

In this section, we illustrate the semantic flow model with a representative example, i.e., the Apache web server. For each incoming web request, we can divide the corresponding Apache behavior (or the `httpd` worker daemon) into the following four logical phrases: (1) The Apache server waits for a client request, and prepares a worker thread. (2) The worker thread handles the request and process it. (3) The server generates response for the incoming request. (4) After the response is sent back to the clinet, the network socket used for the communication is closed.

Figure 1 shows the Apache behavior when answering an incoming request, both from a network/OS viewpoint as well as the semantic flow viewpoint. Specifically, Figure 1(a) contains a list of invoked system calls as well as their arguments while Figure 1(b) highlights some inherent dependencies within these system calls and their arguments. Instead of syntactically grouping adjacent system calls into sequences or mining arguments for possible relationships, the semantic flow model aims to leverage the protocol logic that has been implemented by the modeled program to characterize its behavior. In addition, we can verify the program logic by reconstructing the implemented protocol with semantic-sensitive information from observed system calls, arguments, or other run-time context information.

The above example illustrates system calls and arguments are strongly connected. The key to obtain their relationships lies in protocol-aware semantic analysis. Partial analysis on system call sequences or arguments without knowing their semantic implications will lead to incomplete and imprecise program modeling.

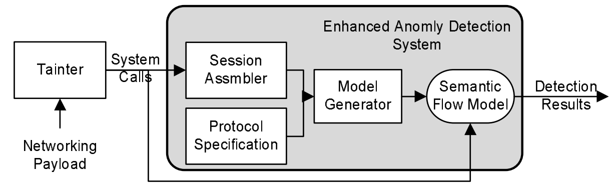


Figure 2. Overview of semantic flow model

IV. Designing Semantic Flow Model

A. Terminologies

In this section, we first define the terminologies that will be used throughout this paper.

- We denote the set of system calls and the set of system call arguments as $\mathcal{C} = \{c_i \mid 1 \leq i \leq m\}$ and $\mathcal{A} = \{a_i \mid 1 \leq i \leq n\}$, respectively. For simplicity, the return value of a system call will be considered as one of its arguments. We also represent the control-flow relation $\mathcal{R}_c$ on $\mathcal{C}$ as $\mathcal{R}_c \subseteq C \times C$ and the data-flow relation $\mathcal{R}_d$ on $\mathcal{A}$ as $\mathcal{R}_d \subseteq A \times A$. Note that existing models that are built upon $\{\mathcal{C}, \mathcal{R}_c\}$ fall into the *control-flow model* category and others built based on $\{\mathcal{A}, \mathcal{R}_d\}$ belong to the *data-flow model* category.
- We log system calls and save them as a record in the form of $sc = \{n, A\}$, in which $sc.n$ is the name of the system call, $sc.A$ is the set of arguments. When processing system calls, we simply consider them as an array sc. An argument $sc[i].a_j \in A$ is assigned by a value and a semantic type, which denoted as $sc[i].a_j.value$, and $sc[i].a_j.type$, respectively.
- The *semantic set* S_{sem} is the super set of system calls and arguments and can be simply represented as $S_{sem} = \mathbf{2}^{\mathcal{C} \cup \mathcal{A}}$. The semantic relation $\mathcal{R}_s$ on S_{sem} is similarly denoted as $\mathcal{R}_s \subseteq S_{sem} \times S_{sem}$. We call models build upon $\{S_{sem}, \mathcal{R}_s\}$ as *semantic flow models*.

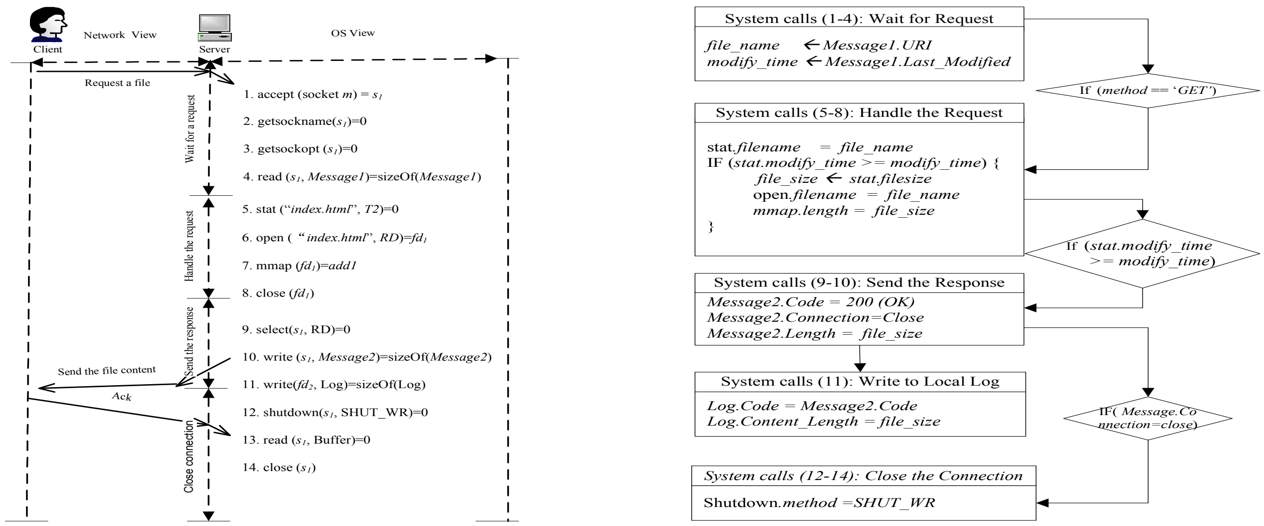
B. System Overview

Figure 2 shows our semantic flow-based intrusion detection model, which has three main components: (1) The *session assembler* propagate tainted networking payload to invoked system calls within a networking session (Section V-A); (2) The *protocol selector* leverages protocol knowledge and matches semantic units with pre-defined protocol specification (Section V-B); (3) The *semantic flow model generator* will reconstruct semantic relations among semantic units and build the program behavior model as the corresponding semantic flow model (Section V-C). The doted line circulated the major components.

V. Methodology

A. Networking Input Propagation

To correlate the networking traffic with system calls and their arguments, we use tain techniques, which have been discussed in [11], [15]. Specifically, we initially *taint* the string in packet payload received by networking-related



(a) Network and OS views

(b) Semantic dependencies between system calls

Figure 1. The simplified network/OS view (left) and the semantic flow (right) of Apache when answering an incoming request. In the OS view, the recorded system calls are sequentially labeled (some of them are omitted for readability). The semantic flow highlights some inherent dependencies among invoked system calls and their arguments.

system calls, such as sys_socket. We also instrument the data movement instructions (e.g., `mov`) and arithmatic/logic instruction (e.g., `add`, `mul`, `and`), such that the tainted string can be propagated through the lifetime of string processing. For a data movement instruction, we check whether the source operand is marked. If yes, we will annotate the destination operand, which can be a register or a memory location, with the source operand's annotation, i.e. its offset in the original message. If the source operand is not marked, we will simply unmark the destination operand. If two marked operands appear in the same instruction, we will union their annotations (e.g., for the `add` operation, the result is the union of the operands if they are both marked).

Then, we need to re-map system call arguments based on *semantic types*, based on the protocol specification. Semantic types are used to more precisely capture the semantic meaning of system call arguments as they cannot be naturally obtained from the original argument types according to the neutral system-wide system call convention. An an example, the first argument of the `open` system call, which originally defined as a *string*, is now redefined as the `Filename` semantic type. Its return value will be similarly redefined as the semantic type `FileDescriptor`, instead of *int*.

```
Name  Filename         Flag    FileDescriptor
open  ``/etc/passwd''  ``RD''   6
```

Besides the knowledge of system call convention, we further use protocol specification to extend our knowledge of semantic meaning. We used the technique in [10] to discover protocol formatting specification. In the following, we illustrate the snippet of `SERVICE_REQUEST` specification for the HTTP protocol.

```
<SERVICE_REQUEST>
SYSCALL = Read(FD, BUFFER, RET)
FD = %Accepted_Socket
BUFFER =((GENERAL_HEADER|REQUEST_HEADER)\13\10)*\13\10
GENERAL_HEADER = %Method %URI %Dummy\13\10
REQUEST_HEADER = From|Host|If-Match|Last-Modified...=\%VALUE
RET = sizeOf(BUFFER)
```

Recall the *read* system call in the line 4 of Figure 1(a). We can capture its semantic meaning with the above protocol specification. More specifically, the file descriptor equals to the accepted socket number after *accepting* the incoming request. The argument *BUFFER* contains two fields, `GENERAL_HEADER` and `REQUEST_HEADER`, each of them can be further parsed into various sub-fields and eventually casted into more specific semantic types. For example, the `REQUEST_HEADER` field can be analyzed based on the following format:

```
From: Type = Email, Format = %username@%hostname
Host: Type = IP|Host_Name, Format = %{4B}|String
Last-Modified: Type = Date, Format = Timestamp
```

The first line states that the `From` field should be parsed as an email address. The second line specifies that the `Host` field should be defined as an IP address or a host name. The third line is to define the type of `Last-Modified` field as the default timestamp format.

B. Algorithm for Constructing Semantic Units

To describe the high-level functionalities of a networking protocol, we introduce the concept of *user session* $\mathcal{S}$ to represent a execution path of one server program, and *semantic unit* U, which intended to capture one essential aspect of modeled program behavior. As an example, the `accept` and the `close` system call are the starting point and the ending point of the user session shown in Figure 1. Semantic units comprises of a number of system calls, their arguments, as well as return values. In our current implementation, we organize semantic units from adjacent system calls based on whether they share the same file descriptors, filenames, or network sockets. In other words, adjacent system calls that manipulate the same file descriptor, filename, or network socket will be grouped to the same semantic unit. For example, the following system call sequence is a *semantic unit* as the three system calls `open`, `read`, and `close` are used to access a file named *"/etc/passwd"* by referring to the same file descriptor.

```
open("/etc/passwd", RD)=6,
read(6, buf)=123,
close(6)=0
```

Algorithm 1: *SemanticUnitExtraction(sc, U)*

---

input : A system call sc, and the semantic unit array U.
output: The updated array of semantic units U.

begin
  for *i=1* **to** N **do**
    for *j=1* **to** M **do**
      if $sc.a_i.type = U[i].a_j.type$ **and** $sc.a_i.value = U[i].a_j.value$ **then**
        $U[no\_of\_su] = UNION(U[i], sc)$;
        break;
      else
        no_of_su++; instantiate $U[no\_of\_su]$;
        $U[no\_of\_su] = sc$;
        break;
end

With collected system calls, our algorithm *SemanticUnitExtraction(sc, U)* groups them into different semantic units. The algorithm works as follows: It maintains a global variable *no_of_su* (initialized with 0) that keeps the current number of semantic units in $\mathcal{S}$. For each collected system call sc, the algorithm will be check whether it is a member of the existing semantic unit $U[no\_of\_su]$. If yes, it will be added to $U[no\_of\_su]$ (via the UNION(U[i], sc) function) and the global variable remains intact. Otherwise, a new semantic unit will be created and the *no_of_su* will be incremented by 1. We need to point out that adjacent system calls manipulate the same file descriptors, file names, or sockets will be grouped into the same semantic unit. However, not all system calls that manipulate the same file descriptor, filename, or socket will be included into the same semantic unit. This design choice makes the Algorithm 1 easy to implement.

Example 1 We illustrate the algorithm by revisiting the simplified *httpd* case in Section III. First, when the first system call – `accept` – is encountered, it will be included in a new semantic unit U_1. The following three system calls (at line 2-4) will also be grouped into the same semantic unit U_1 as they essentially wait for (and then receive) incoming requests and manipulate the same socket (as the `accept` system call). After that, the *stat* system call at line 5 will start with a new semantic unit U_2 as it is not related to the previous socket, and their main purpose is to handle the request. Moreover, since the following system calls at lines 6-8 handles the same file named *"index.html"* with the *stat* system call, they will join with the second semantic unit because they send back the response to the requesting client. In a similar manner, system calls at line 9-10 (U_3) send back the response to the requesting client; the requesting behavior is locally recorded at line 11 (U_4); and the communication channel is finally shutdown and closed at lines 12-14, U_5).

C. Constructing Semantic Specification

Different from previous approaches that solely depend on either control-flow or data-flow relations, a semantic relation flow $\mathcal{R}_s$ covers the inter-dependencies between them. In this paper, we focus our semantic flow relations in three categories: Data → Control, Data → Data, and Control → Data, which illustrate in Table I.

VI. Evaluation

We have implemented a proof-of-concept system that runs on the Fedora 13. The system calls, arguments, and return values are collected with a customized loadable Linux kernel module (LKM). The experiments are performed on a PC with Intel Core 2 Due 2.83GHZ CPU and 2G physical memory.

A. Effectiveness

We evaluate the effectiveness of our approach with a number of real-world attacks that are publicly obtained from [3]. Table II contains the list of five experimented server programs as well as attacks exploiting their vulnerabilities. Within these attacks, two of them are control-flow attacks which directly hijack the control flow of vulnernable programs, while the other three are data-flow attacks that are able to manipulate security-critical data to evade traditional detection techniques. Since server programs of `wu-ftpd` and `ghttpd` are vulnerable to both control-flow and data-flow attacks, we simply use a subscript to differentiate them. For instance, we use *wu-ftpd*$_1$ to represent the control-flow attack and *wu-ftpd*$_2$ to represent the data-flow attack against `wu-ftpd`.

In the following, we use three examples to show that how the three types of semantic relations, i.e., *Data → Control*, and *Data → Data* are used to detect attacks.

Data → Control Violation Detection All versions of *wu-ftpd* before 2.6.1 contain a vulnerability that can be exploited to trigger a heap corruption vulnerability (CVE-2001-0550). The vulnerability is located in the *ftpglob* function, which fails to properly handle the FTP commands and consequently allows remote attackers to execute arbitrary commands via a $\sim\{$ argument [16].

Figure 3(a) shows the related semantic flow specification that will be violated by this attack. More specifically, there exist three related semantic units for the exploited *wu-ftpd* sub-session. The first semantic unit receives the command request from the client and interprets it to be a *CWD* command. The following semantic unit will actually execute the `CWD` command by invoking the `chdir` system call. The return value of `chdir` will determine the *code* field that will be later sent back to the client in the third semantic unit. The *code* field essentially notifies the client whether the operation is successful or not.

Our approach detected this attack when the server sent its response to the client via a *write* system call. Based on the ftp protocol, the raw command `CWD pathname`

 ISBN: 978-1-61208-010-9

Category	Subcategory	Meaning	Example
Data* → *Control	**Single data to control relations**	Relations that a single argument determines follows system calls	In *ftp* protocol, the argument `CWD` determines system call `chdir`
	Multiple data to control relations	Multiple arguments together determine system calls later	The *readfds* and *writefds* arguments of *select* system call determine the following *read* or *write* system call
	Number of loops relations	Relations that arguments determine the number of system calls that will appear later	The argument `st_size` of system call `stat` determines the number of `write` system calls be invoked later
Data* → *Data	**Logical relations**	Relations that a single argument might determine future system calls	The return value of `-13` (meaning *Permission denied*) of `open` determines the error code `304` in the reply buffer.
	Numeric relations	Relations that evaluate two numeric values v_1 and v_2	*LargerThan*(v_1, v_2), *SmallerThan*(v_1, v_2), *EqualTo*(v_1, v_2)
	Timing relations	Relations evaluate two timing values d_1 and d_2	*Before*(d_1, d_2), *After*(d_1, d_2), and *At*(d_1, d_2)
Control* → *Data		Relations determine system calls to system call arguments	The system call `write` determines certain keywords in the reply buffer, such as *Code*, *Connection*, and *Length*

Table I
SEMANTIC RELATIONS R_s IN OUR FRAMEWORK

Program (version)	Reference	Attack description	Program size(KB)	Total # of system calls	# of system calls in attack session	Violation
wu-ftpd$_1$(2.6.1)	CVE-2001-0550	Heap corruption allows execute arbitrary commands via a $\sim\{$ argument to commands	2916	1372	2	*Data* → *Control*
ghttpd$_1$(1.4)	CAN-2001-0820	Long arguments passed to the `Log` function in `util.c` allows attackers to get shell	311	27	20	*Data* → *Control*
wu-ftpd$_2$(2.6.0)	S.Chen et al. [3]	Format string overwrite user ID	2916	15754	8	*Data* → *Data*
ghttpd$_2$(1.4)	S.Chen et al. [3]	Stack overflow to overwrite backup value of `ESI`	311	105	14	*Control* → *Data*
null-httpd(0.5)	S.Chen et al. [3]	Two `POST` commands corrupt CGI-BIN configure string	806	230	72	*Data* → *Data*

Table II
VULNERABLE SERVERS AND REAL-WORLD ATTACKS USED IN OUR EVALUATION

allows the client to change the current working directory to `pathname`. As such, in our semantic flow specification (Figure 3(a)), the semantic unit U_2 will invoke the system call `chdir`. After invoking the `chdir`, the server will notify the client with the return code either `250`(indicating "the CWD command is successful"), or `550`(meansing "No such file or directory").

When considering the actual attack sequence, it violates at least twice our semenatic flow specifications: First, there does not exist a subsequent `chdir` system call. Second, the response message will usually contain return code of `250` or `550`. For previous approaches that detect control injection attacks, the same attack could be detected when the attack invokes the *execve* system call to obtain a command shell (*"/bin/sh"*), which is much later than the detection point by our approach. Figure 3(b) shows the difference between the detection point by our approach and the detection point by other approaches.

***Data* → *Data* Violation Detection** The same *wu-ftpd* server (versions 2.6.0 and earlier) contains another vulnerability, i.e., a format string bug (CVE-2000-0573), which can be exploited with a specially-crafted string to the *SITE EXEC* command. Instead of overwriting the return address on the stack, this attack use format string to overwrote a security-critical variable $pw \rightarrow pw\_uid$ to 0. After that, the attack further established another data connection and issues a *get* command, which essentially invoked the function *getdatasock()* in the wu-ftpd server. Due to the corruption of $pw \rightarrow pw\_uid$, the execution of the function will set the `EUID` of the process to 0, elevating the process privilege to the super-root. As such, an originally non-privileged user is able to access the system with the root privilege. This overall exploitation is a typical data-flow attack [3].

It is interesting to point out that data-flow-based anomaly detection is also able to detect this attack. As discussed in [1], this attack can be detected as a violation of the equality relation between the `seteuid` system call and another `setuid` system call (in function *pass()*). However, the root cause of this attack is that the attacker crafts a format string, in the form as `SITE EXEC` $aaabcd\%.f\%.f\%.f\%...\%d...|\%.8x$, to overwrite $pw \rightarrow pw\_uid$ to 0. And our semantic flow-based detection is able to identify this attack when the *Equal* relation between the file name `execve` invoked and the file name in reply message is been violated, which is earlier than the previous detection point.

```
……
#U1: Receive a client request
U1(Message){
  SWITCH(CMD)
  CASE ( "CWD" ){
    PARAMETER = pathname;
    CALL U2(chdir,PARAMETER);
  }
  ……
#U2: Handle the request
U2(SYSCALL, PARAMETER) {
  IF(chdir(PARAMETER).ret < 0)
  {
/* No such file or directory */
    Code= 550;
  }
  ELSE{
/* The CWD command succeeds */
    Code = 250;
  }
}

#U3: Send the reply
U3(CMD, Code){
  write.Message.cmd = CMD;
  write.Message.code = Code;
}

  ……
```

(a) Partial semantic specification for wu-ftpd

```
#Other normal system calls
......
read(0, "CWD ~{\10./././././.\10.\
10000….\10", 1024) = 7

write(1, "$\10sP\10$", 4) = 4        <- Our detection point
read(0, "3U/AF3E…..", 255) = 72
setreuid(0, 0) = 0

mkdir ("T", 237)=0                    <- Others' detection points
chroot("T")=0
chroot()=0
execve("//bin/sh", addr, 00000000)
```

(b) Logged (attack) system calls and the detection points

Figure 3. A control-flow attack based on the wu-ftpd heap corruption vulnerability (CVE-2001-0550): The system call sequences shown in Figure 3(b) violates the semantic specification in Figure 3(a).

VII. Conclusion

In this paper, we have presented a semantic flow-based host intrusion detection model that seamlessly integrates control-flow and data-flow dependencies. When compared with existing approaches, which only focus on control-flows, or data-flows but not both, our approach greatly improves the accuracy and completeness of the obtained program behavior models. An efficient algorithm is presented to accurately extract basic semantic units, each of which characterizes an essential aspect of the modeled program behavior, and then obtain the semantic dependencies among them. Our experimental results show that our model enables earlier detection and prevention of many attacks than existing approaches and holds great promise for more precise and complete host-based intrusion detection.

Acknowledgment

We would like to thank Zhiqiang Lin from Purdue University for providing the source code of AutoFormat, the networking protocol reverse engineering tool.

References

[1] S. Bhatkar, A. Chaturvedi, and R. Sekar. "Dataflow Anomaly Detection". In *S&P '06: Proceedings of 2006 IEEE Symposium on Security and Privacy*, Oakland, CA, USA. pp. 48-62. May 2006.

[2] M. D. Bond, V. S. Kathryn, S. McKinley, V. Shmatikov. "Efficient Context-Sensitive Detection of Real-World Semantic Attacks". In *PLAS '10: Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*. pp. 1-10. ACM, 2010

[3] S. Chen, J. Xu, E. Sezer, P. Gauriar and R. Iyer. "Non-Control-Data Attacks Are Realistic Threats". In *USENIX Security '05: Proceedings of the 14th USENIX Security Symposium*, Baltimore, MD, USA. pp. 177-192. August 2005.

[4] H. Feng, J. Giffin, Y. Huang, S. Jha, W. Lee, and B. P. Miller. "Formalizing sensitivity in static analysis for intrusion detection". In *S&P '04: Proceedings of 2004 IEEE Symposium on Security and Privacy*, Oakland, California, USA, pp. 194-208. May 2004.

[5] D. Gao, M.K. Reiter and D. Song. "Gray-Box Extraction of Execution Graphs for Anomaly Detection". In *CCS '04: Proceedings of the 11th ACM Conference on Computer and Communications Security*. Washington, DC, pp. 318-329. October 2004

[6] J. Giffin, S. Jha, and B. Miller. "Efficient contextsensitive Intrusion Detection". In *NDSS '04: Proceedings of The 11th Annual Network and Distributed System Security Symposium*, San Diego, CA.

[7] S. A. Hofmeyr, S. Forrest, and A. Somayaji. "Intrusion Detection Using Sequences of System Calls". Journal of Computer Security (JCS), 6(3):151-180, 1998.

[8] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. "On the Detection of Anomalous System Call Arguments". *In ESORICS'03: Proceedings of the 8th European Symposium on Research in Computer Security*. Gjovik, Norway, pp. 101-118. October 2003

[9] L. C. Lam and T. Chiueh. "Automatic Extraction of Accurate Application-specific Sandboxing Policy". *In RAID '04: Proceedings of the 7th International Symposium on Recent Advances in Intrusion Detection*. French Riviera, France. pp. 1-20.

[10] Z. Lin, X. Jiang, D. Xu, and X. Zhang. "Automatic Protocol Format Reverse Engineering Through Context-Aware Monitored Execution". *In NDSS '08: Proceedings of the 15th Network and Distributed System Security Symposium*, San Diego, CA, February 2008

[11] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, D. Song. "A Symbolic Execution Framework for JavaScript". *In S&P '10: Proceedings of IEEE Symposium on Security and Privacy*, Oakland, California, USA, May 2010.

[12] R. Sekar, M. Bendre, P. Bollineni and D. Dhurjati. "A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors". *In S&P '01: Proceedings of 2001 IEEE Symposium on Security and Privacy*, Oakland, California, USA. pp. 144re-155. May 2001.

[13] D. Wagner and P. Soto. "Mimicry attacks on host-based intrusion detection systems". *In CCS'02: Proceedings of the 9th ACM Conference on Computer and Communications Security*, Washington, DC, USA. pp. 255-264. Noverber 2002.

[14] D. Wagner and D. Dean. "Intrusion Detection via Static Analysis". *In S&P'01: Proceedings of 2001 IEEE Symposium on Security and Privacy*, Oakland, California. May 2001.

[15] H. Yin, D. Song, E. Manuel, C. Kruegel, and E. Kirda. "Panorama: Capturing system-wide information flow for malware detection and analysis". *In CCS'07: Proceedings of the 14th ACM Conferences on Computer and Communication Security*, pp. 116-127. October 2007.

[16] Wu-ftpd vulnerability. http://www.securityfocus.com/bid/3581/references (June 2, 2011)