Abstracting Security Specifications in Building Survivable Systems

J. Jenny Li and Mark Segal
Telcordia Technologies (formerly Bellcore)
445 South Street,
Morristown NJ 07960-6438 USA
Email: jjli@bellcore.com; Phone: (973)829-4753; Fax: (973)829-5981

Abstract

We have designed a specification-based intrusion detection and prevention infrastructure for building survivable information systems[1]. In that work, we specify security-related behaviors declaratively in a high-level language called Auditing Specification Language (ASL). This specification is then compiled into optimized programs for efficient detection and prevention of computer and network intrusions. Our method is efficient and powerful in intrusion prevention, detection and isolation. This paper intends to automate the process of obtaining ASL specifications. The automation has many advantages: 1) It reduces the chances of human errors; 2) It adapts quickly to new attacks; 3) It reduces the cost in training ASL programmers; 4) It provides solid theoretical proving of the completeness of the specification; and 5) It allows the infrastructure to detect previously unknown attacks.

This paper considers the case when the specification of the software behavior is expressed in a formalism based on communicating extended finite state machines. The specification language SDL is used as a concrete example. The automation approach includes 3 steps: 1) identifying invariants in the vulnerable services, 2) back tracking to find the usage of the negation of the invariants, and 3) expressing the negation usage in ADL using data structures defined in the original program.

We are experimenting this approach on a small service-provider system. We will describe the settings for the experiments in the later part of the paper. The analysis of the experimental results is in progress. One future research direction is to use the generated ASL specification to catch more sophisticated attacks such as the ones involving more than one host.

Keywords

security, survivability, specification invariants, intrusion detection, SDL, communicating extended finite state machine (CEFSM), Audit Specification Language (ASL).

1. Introduction

With the growing usage of network technology in major economic sectors, such as energy, transportation, telecommunication, banking and commerce, network survivability becomes a critical issue. Survivability is the ability of the system to continue to perform its critical functions in a timely manner even in the face of large-scale failures or coordinated, malicious attacks. Several techniques for intrusion detection have been developed recently, such as [2], [3] and [4]. One direction of recent research is specification-based attack detection methods. We have designed a new approach that combines attack prevention, detection, and isolation techniques. The technique is based on a specification of security related behaviors given in a high-level language, called Auditing Specification Language. The detection and prevention are carried out on-line in real-time, as the software program executes. The ASL specification describes the attacks on the system-call and package level. This specification is obtained manually from the software program, its behavior specification documentations and its attack models. The infrastructure of our approach is given in Figure 1:

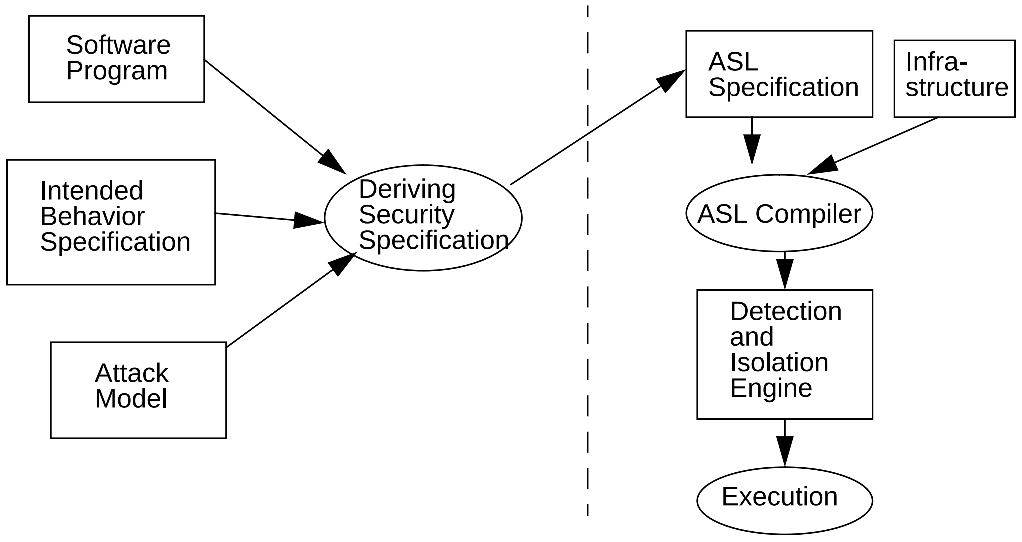
Software Program
Intended Behavior Specification
Attack Model
Deriving Security Specification
ASL Specification
Infra-structure
ASL Compiler
Detection and Isolation Engine
Execution

FIGURE 1: Infrastructure of the Survivable Active Network

The current status of this work is that the right-hand side of the dashed line is fully automated and the left side is done manually, i.e. the derivation of the security specification is now rely solely on a system security administrator who is familiar with intended behavior of a program (as can be determined from its manual pages or other documentation) as well as specific known vulnerabilities obtainable from sources such as attack advisories. This paper presents an approach to automate the derivation. Such automation offers a number of benefits. It reduces the chances of human errors that inject malicious code into ASL specification that will eventually be incorporated into the software execution infrastructure. It reduces the cost in training ASL programmers. It provides solid theoretically provable measurement of the completeness of the security requirement specification instead of a human ad-hoc construction. It allows the infrastructure to detect future unknown types of attacks.

The remainder of the paper is organized as follows: We present an overview and the rationale of the approach in Section two. Section three illustrates through examples the detailed steps of the approach. Section four describes an application example. Section five gives some concluding observations.

2. Rationale of the Approach

The objective of this work is to investigate methods of obtaining security behavior requirements from the specification of the software application. We denote the set of behaviors defined by the software specification as **R**, and the set of behaviors implemented as **I**. There are two kinds of errors: one is that some features are specified but not implemented and the other is the opposite. Because attackers only take advantage of the implemented behaviors, we do not consider the former case. Therefore we can assume that the implementation has realized all the behaviors specified, i.e. $\mathbf{I} \supseteq \mathbf{R}$. Furthermore, one direction of the security research is to detect security vulnerability in software specifications, for example [5]. Since this direction is not the focus of our research, we assume that such research has been so successful that they have detected and removed all security vulnerabilities, i.e. we can assume that the specified behaviors are all secure.

We define an operation “-” on sets which removes all the elements in one set from the other set. We have *set1* - *set2* = *set3* iff for each $i \in set3 \rightarrow i \in set1$ && $i \notin set2$. We define a **failure** as a behavior "*b*" such that "*b*" is a failure iff $b \in$ **I** - **R**. Thus, the set of failures **F** has **F** = **I** - **R**.

A **security-violation behavior** (**svb**) is a behavior that can cause security damage to the information system, including breaking down the software application or other application, breaking down network infrastructure, breaking down network operation, stealing proprietary information, etc. Suppose the set of security-violation behavior is **V**. Because we assume that specified behaviors are secure, i.e. insecure behaviors are not specified, we have $\forall i\ i \in \mathbf{V} \rightarrow i \notin R$, in other words, we have $\mathbf{V} \subseteq \mathbf{F}$. The goal of survivable active network research is to detect the **svb**'s on line and to prevent them from causing damage.

In our approach, we use the specification of **V** in ASL to detect security attacks on line. During the target system execution time, our infrastructure observes its behavior, i.e. its event sequences and checks whether the behavior belongs to **V**. If so, a preventive action is activated. Overall, the goal of this paper's research is to find an automatic or semi-automatic approach for deriving **V** from **R** and **I**, as well as attacking models. The relationships among the set elements are illustrated in the following Figure 2:

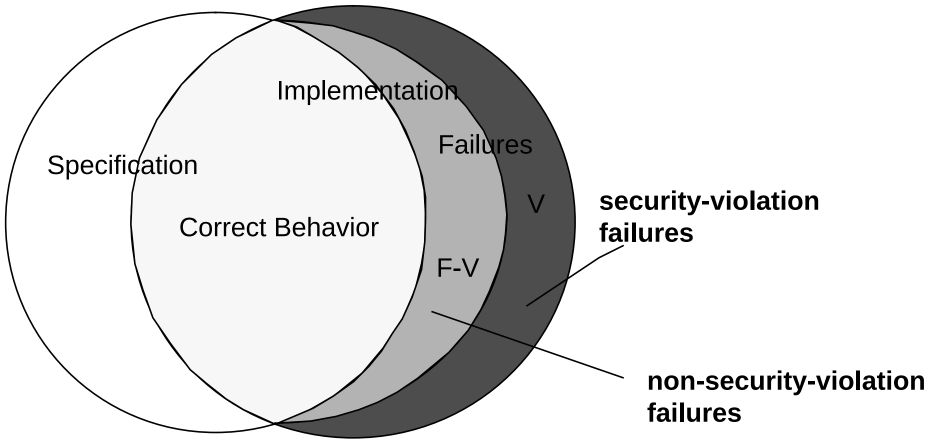

FIGURE 2. Problem Definition

Figure 2 shows that failures include two parts, security-related (**V**) and non-security-related (**F** - **V**). This means that not all failed behavior can cause security damage. The specification and the implementation behaviors are given. The unknown is the boundary between **V** and **F** - **V**. From the observation of the attack models, we noticed that attacks often appear to be legitimate actions with the violation of security invariants, such as the limitation on buffer size or the limitation on system call usages. Based on this observation, we assume that all behaviors violating specification security invariants are security-violation behaviors, i.e. $b \in \mathbf{V}$ iff *b* violates the security invariants of the specification. Now we transfer the problem of obtaining specification for **V** to the problem of identifying specification security invariants whose negations define the dark shade (V) in the diagram.

Suppose the behavior defined by the security invariants of the specification is collected in a set **Q**. **Q** is the implemented behaviors that are not security-violation behaviors. It includes the correct behaviors and failures that are not security sensitive. Based on the definitions, we have the following two facts: 1) $\mathbf{Q} \supseteq \mathbf{R}$; and 2) **Q** - **R** = **F** - **V**.

In summary, if we assume that **I** is the complete set of the behaviors and we define the negation $\neg x$ as **I** - *x* when $x \subseteq \mathbf{I}$, then we have the following 4 observations: 1) **R** and **I** are known; 2)

$\mathbf{R} \subseteq \mathbf{I}$; 3) $\mathbf{R} \subseteq \mathbf{Q}$, where **Q** is defined by security invariants which can be derived from **R**; and 4) $\mathbf{V} = \neg\mathbf{Q}$. This proves that the negation usage of the security invariants defines the set V. That is, in theory, our approach of identifying security invariants will be able to detect all security-violation behaviors if all our assumptions are met. Our goal now is to derive **V** from security invariants abstracted from **R** and **I**. The next section elaborates our method of achieving this goal.

3. Automated Security Specification Derivation

This paper considers the case where the intended behavior of the software with possible vulnerability is specified in SDL (Specification and Description Language). SDL is an International Telecommunication Union (ITU) standard, based on Communicating Extended Finite State Machine (CEFSM) model. A CEFSM model includes the definitions of EFSMs and Channels, i.e. **CEFSM** = (**EFSM**s, **C**). Each EFSM is defined as a finite state machine (FSM) with the addition of variables to its states.

In our survivable information system infrastructure, each process behavior is observed independently in most cases. Therefore, we can assume that the SDL specification of the program includes only one EFSM with channels communicating in and out of the EFSM.

Each **FSM** is a quintuple = ($\mathbf{S}$, $\mathbf{S_0}$, **I**, **O**, **T**), where **S** is the set of states, $\mathbf{S_0}$ is the set of initial states, **I** is the set of input events and enabling conditions, **O** is the set of action tasks and output events, and the **T** maps an input set and the old state to tasks and/or outputs and a new state. Each state in an FSM is defined by a variable, state name. Extended FSMs are those defined with additional variables to states. Each state in an EFSM is defined by a set of variables, including state names. The transition **T** of an EFSM becomes: [$<v_0, v_1..., v_n>$+*input**, *task**;*output**+$<v_0', v_1',... v_n'>$], where v_0 and v_0' are the names of the states, $<v_1, v_2..., v_n>$ and $<v_1', v_2',... v_n'>$ are the values of extended variables, n is the number of variables, "+" means coexistence, ";" means sequence of events such as tasks and outputs, and "[,]" denotes a sequenced pair.

We define "**symbolic states**" as the states with different v_0 value, while ignoring the values of the rest of the variables and "**actual states**" as the states with any different values in any of the variables. For example, state <*state1*, *1*, *2*> and state <*state1*, *2*, *2*> are the same symbolic state as plotted in an EFSM diagram, but different actual states when representing the EFSM as a finite state machine. Based on the definitions, we know that the number of symbolic state is $|v_0|$, and the number of actual states is $|v_0|\ |v_1|\ ...\ |v_n|$, where "$|x|$" is the number of values of variable x. We refer "an EFSM state" to a symbolic state. All "states" are symbolic states in the rest of the text, unless otherwise stated.

One research of automatic identification of invariants on a state machine is given in [6]. Their work considers the automatic generation of global state invariants of a specification given in a tabular form with one input transition. Our method automatically generates both local and global invariants, i.e. properties that hold in a set of reachable states of an EFSM. The EFSM allows multiple inputs for each transition. We also allow inputs to be enabling conditions, not just triggering events. Their method used additional information such as environmental constraints and assumptions in deriving state invariants. We use intrusion models for our additional information when needed.

There is no standard definition of intrusion models. One such model is given in an intrusion requirement specification discussed in [7]. Their work mentioned such a specification but did not give a concrete example of such a document. We assume that we obtain the attack model from attack advisories, mailing lists and hacker web sites and have it written in English text with a set of statements, **St***. Each statement has a standard format of **St** = Action(Object). For example, we have attack models: overflow(buffer) meaning buffer-overflow attacks and write(/etc/passwd) meaning the attacks of writing a protected file /etc/passwd with altered authorization. We use this assumption to simplify our intrusion model. The issue of specifying intrusion models needs further research. The attack model is used to identify vulnerable variables and constraints to be considered in deriving state security invariants of an EFSM specification.

In order to achieve the goal of identifying all security invariants for a set of reachable states, our method first searches through all the reachable states from the initial state to find the invariants. It then reduces the number of the searched states. As the number of the searched states reduced, the number of invariants increases. Eventually, when it searches through all various combination sets of states, it reaches a full set of all local and global invariants. Suppose the set of the states being searched is **U**, then $\mathbf{U} \subseteq \mathbf{S}$. When **U** = **S**, the invariants are global.

To illustrate the derivation of invariants, consider an EFSM Z describing the service of managing a set of system devices as given in the example SDL specification of Figure 3.

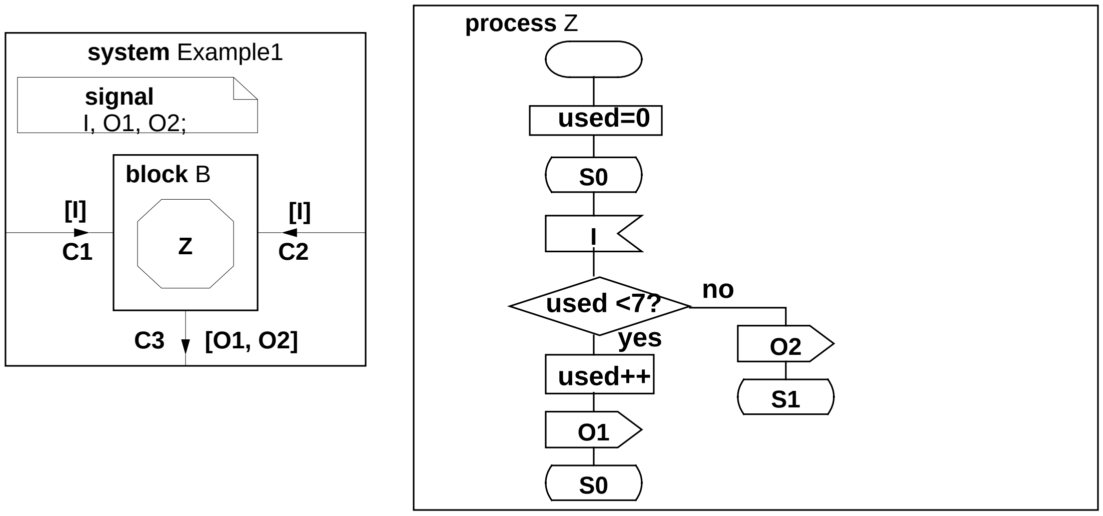

FIGURE 3. An Example Software Specification

In Figure 3, we assume that message *I* comes from the network and the output message O_1 and O_2 are system calls. The specification has two states **S**= $\{s_0, s_1\}$. We started with **U** = **S** = $\{s_0, s_1\}$. First we identify all the variables in the EFSM. This might not be obvious because it involves also the default variables in the SDL semantics. For example, the message "*I*" implies a sender identification. This suggests that the security invariant derivation algorithm must search through sets **S**, $\mathbf{S_0}$, **I**, **O**, and **T** to identify variables, including SDL default variables. It also replaces constants with constant names, i.e. making them variables. For example, the variables for the EFSM Z are {*message-I*, *state-name*, *used*, *total-device*(=7), *message-O1*, *message-O2*, *sender-id*}. We store these variables in a set **V**. The second step of the algorithm is to find out the invariants for **U** (global invariants when **U** = **S**) by examining the constraints on each of the variables and the relationships among the variables. This includes checking constraints on every single element of **V**, every two elements of **V**, and until every *n* element of **V** where *n* is the size of **V**. From the attack model, we know that one kind of attack is buffer overflow, so we construct one kind of security invariants involving one variable to be:

- size-of(*message-I*) < maximum-buffer-size;

• size-of(*message-O1*) < maximum-buffer-size; and

• size-of(*message-O2*) < maximum-buffer-size.

The relationships among the variables include:

• number-of(*message-O1*) = *used*; and

• number-of(*sender-id*) = number-of(*message-O1*) + number-of(*message-O2*).

The third part of the algorithm is to repeat the second part by subtracting a state from U. For instance, we subtract s_1 from **U** to obtain the invariants for state s_0 as: *state-name* = s_0, *used* < 7, *message-O1* = **true**. Similarly, we have invariants for state *s1* as: *state-name* = s_1, *used* = 7, *message-O2* = **true**. Note that we did not generate the redundant invariant of *message-O2* = **false**.

When **U** = ∅, we have obtained all the local and global invariants. The following, Figure 4, shows the algorithm of deriving state security invariants for a security requirement specification.

```
state-invariant-derivation (IN: S, S0, I, O, T; OUT: IV) {
    V = {state-name};
    foreach i in I do V = V ∪ {i};
    foreach o in O do V = V ∪ {o};
    foreach [s+It, Tt; Ot+s'] in T do {
        foreach task-name(v1, ..., vm) in Tt do V = V ∪ {v1, ..., vm} }
    U = S;
    for (k = 0; k <= |U|; k++) do {
        foreach k of u1..uk in U do {
            Usearch = U - {u1..uk};
            i = |V|;
            for (j = 1; j <= i; j++) do {
                foreach j of v1..vj in V do IV=IV ∪ constraint(v1..vj, Usearch); }
        }
    }
    return(IV);
}
```

* The procedure, constraint(), constructs invariants based on attack models.

FIGURE 4. State Security Invariant Derivation Algorithm

The complexity of the algorithm is $|\mathbf{S}|^2 e^{|\mathbf{V}|}$. It appears to be quite high. This is acceptable because this abstraction is done off-line, which will not affect the speed of intrusion detection and prevention. Moving the high complexity part of our infrastructure to the off-line system is part of our strategy in improving the speed of the on-line detection engine. Furthermore, a high complexity algorithm is still far faster than a manual approach.

The negation of the invariants gives us possible ways of attacking the system even if the attack is not previously defined in any attack models. For example, one kind of attack is the violation of the invariant: number-of(*message-O1*) = used, i.e. making it to be number-of(*message-O1*) < *used*. Now we backtrack and find the usage path of the service to reach the negation of the invariant. In this example, both *Message-O1* and "*used*" are modified and applied on the same SDL specification branch. There are two ways to change the value of "*used*", one is by race condition, i.e. changing the variable by the other process before the process moving to send out message O_1, and the other is to use message(*I*) buffer being overflow to modify the "*used*" variable. The specification of these two behaviors allows us to catch the attacks and prevent them, even if such attacks have never been observed before. The ability to detect future unknown attacks is a significant improvement of our approach.

As it can be seen the derivation of invariants can also identify race conditions on the requirement specification level. In this example, the checking of the value "*used*" and the sending of the output "O_1" or "O_2" cannot be intercepted by other interleaving processes that changes the value of "used". This race condition can be prevented in the SDL specification because SDL semantics restricts the usage of the value of the variables across the processes. But in the implementation, this race condition must be re-enforce to prevent the attack of a critical service.

At this point we have obtained the security requirement specification on software requirement level. We have to map it to the implementation program of the software to derive an ASL specification of security behaviors. Supposed the O_1 is implemented in a sequence of system calls: **C**. We represent the attack mentioned above in ASL as

- **C** | size-of(*I*) > maximum-buffer-size --> abort; and
- *used*++..**C**.

The first ASL rule means that abort the process if the buffer overflows and the later means *used*++ and *message-O2* are an atomic sequence that their execution are not to be interleaved with the system calls of any other concurrently executing processes. This step of mapping can use software supervision [8] and slicing[9] techniques. This software supervision technique is to execute both versions of the software, its specification and its implementation of the attack scenarios. This slicing technique highlights the execution paths of the scenario in both specification and the implementation. We use an algorithm to map the two paths and find out how O_1, for instance, was implemented in the program. Such implementation will be abstracted out and put into the ASL specification. ASL specification is on the package and system call level while the SDL is written on the requirement level of the software.

In summary, our automatic security invariant abstraction method includes three steps: *1)* identifying security invariants in the vulnerable services, 2) back tracking to find the usage of the negation of the invariants, and 3) expressing the negation usage in ADL using data structures defined in the original program.

4. Experimental Results

We designed an experiment to evaluate our method of automatic derivation of security requirements from software specifications given in SDL. The goal of this experiment is to illustrate the invariant derivation method and to show the feasibility of the method.

4.1 Target Software System

An evaluation of our method was carried out on a service-provider application. The target system was the program that manages the devices used by the required network users. The system consisted of a computer host with a target application program and an operating system kernel. The operating system interfaced the application program through system calls. The application program sensed the arrival and departure of network packages through a hardware interface memory. In this memory, for example, the arrival of request package was continuously reflected in the value of bits in the arrival package segment. Similarly, the setting of a particular bit in another segment would result in the output of a network package. In this way, we can simulate network attacks without using actual networks. This setting is shown in Figure 5.

The system level communication diagram of a simplified specification of the application program is shown in FIGURE 6. This figure shows the device manager process which manages the

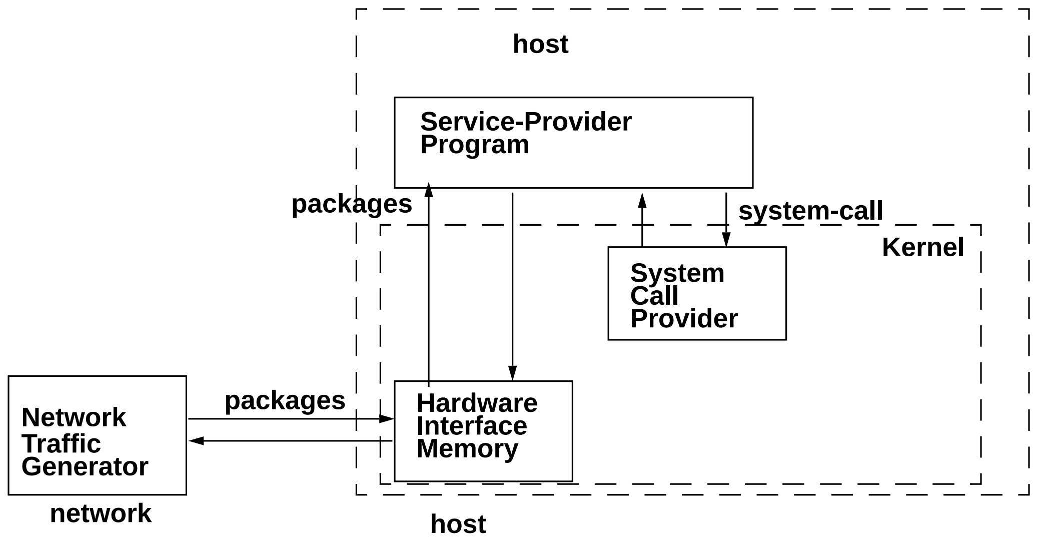

FIGURE 5 Evaluation Environment

sharing of devices among network users. In the full specification, the number of the device was 60, and there was another resource management process, which manages the sharing of another kind of devices. This specification defines the software behavior as seen by the network users.

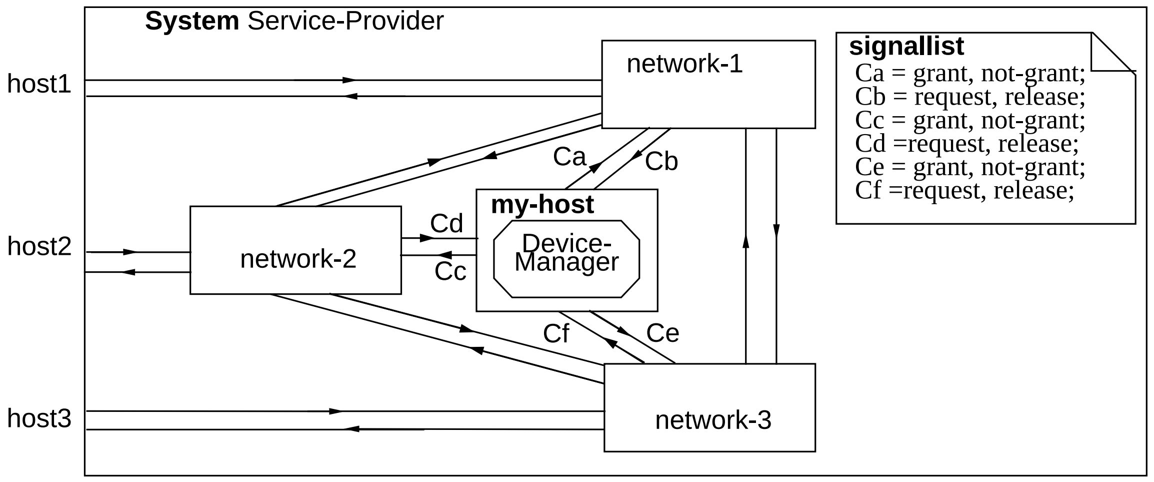

FIGURE 6 Target Application Specification (Simplified)

The behavior of the Device-Manager itself is given in an EFSM of SDL specification as given in Figure 7. This specification is a more complex version of the previous example.

4.2 Generating Invariants

We use our method to generate ASL specification to be used to detect and isolate the attacks to the device management process. The focus and the key step of this method is state security invariant derivation. The specification given in Figure 7 has only one state. The invariants for this state are global invariants. Figure 7 defines an EFSM such that:

- **S** $=\{s_0\}$;

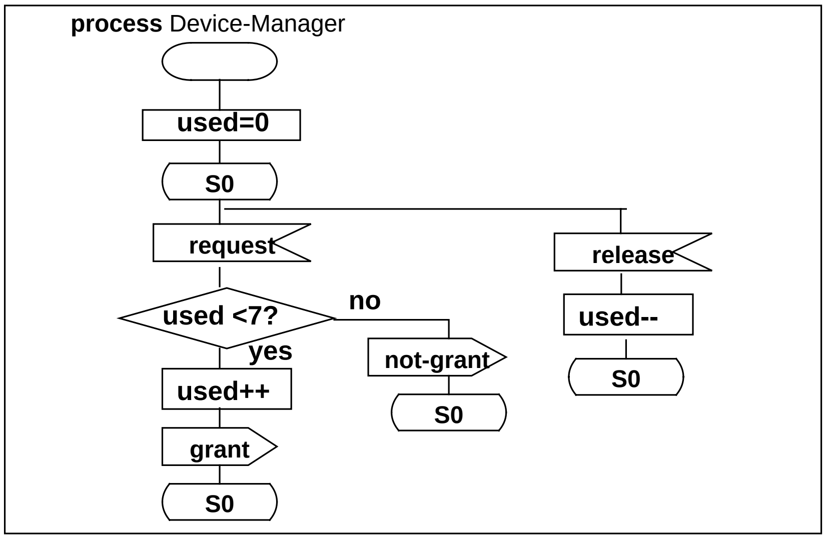

FIGURE 7. An Example Device-Manager Specification

- $\mathbf{S_0} = \{s_0\}$;
- $\mathbf{I} = \{$*request*, *release*$\}$;
- $\mathbf{O} = \{$decision(*used*, 7), ++(*used*), --(*used*), *grant*, *not-grant*$\}$; and
- $\mathbf{T} = \{$[*request*+s_0, decision(*used*, 7, **true**);++(*used*);output(*grant*)+s_0],
  [*request*+s_0, decision(*used*, 7, **false**);output(*not-grant*)+s_0],
  [*release*+s_0, --(*used*)+s_0] $\}$.

The result of the first part of the algorithm is a set of variables used in the EFSM, **V**= {*state-name*, *request*, *release*, *used*, *grant*, *not-grant*}. The following Table 1 shows the resultant state invariants when considering relationships among one to six variables.

Table 1: Invariant Derivation Table

Number of Variables	Security Invariants
1	1. *state-name* = s_0; 2. buffer-size(*request*) <= maximum-buffer-size; 3. buffer-size(*release*) <= maximum-buffer-size; 4. *used* <= 7; 5. number-of(*grant*) < 7;
2	6. number-of(*release*) <= number-of(*grant*); 7. *used* < 7 -> *grant*; 8. *used* = 7 -> *not-grant*;
3	9. number-of(*request*) = number-of(*grant*) + number-of(*not-grant*); 10. number-of(*grant*) <= used + number-of(*release*);
4	none
5	none
6	none

We have obtained 10 invariants for the example EFSM specification. They can be used to find out attacks that can cause the negation of the invariants. Such attack event sequences are then expressed in ASL.

4.3 Usage of Negation of Invariants

The identification of the cause of the security invariant negations is now done manually because it requires human intelligence. We will set up a knowledge-based software-system to accomplish this in the future.

Change the value of the state-name can cause the negation of the invariant 1 of Table 1. This kind of attack could bring down the entire system because only state s_0 can handle the “request” and “release” signals. Any usage, i.e. a sequence of events, that modifies the state-name must be detected as an attack and prevented. The usage of the other invariant negations can also be obtained similarly. Table 2 gives the results.

Table 2: Cause of Invariant Negation

Invariant ID	Invariant Negation	Usage
1	state-name s0;	change state-name in the behavior sequences
2	buffer-size(request) > maximum-buffer-size;	the length of the package request exceeds the maximum buffer size.
3	buffer-size(release) > maximum-buffer-size;	the length of the package release exceeds the maximum buffer size.
4	used > 7;	use a device that is already used.
5	number-of(grant) > 7;	unauthorized usage of the device.
6	number-of(release) > number-of(grant);	steal a device by faking a release, achieved by sending in an invalid release network package.
7	used > 7 -> grant;	assign a device to more than one user.
8	used < 7 -> not-grant;	taking up a device while others need it.
9	number-of(request) < number-of(grant) + number-of(not-grant);	1. steal a device without request for it. 2. some request’s are not checked.
10	number-of(grant) > used + number-of(release);	Listen in to other people’s usage of the device.

We use ASL to specify the security requirements describing the event sequences that cause negation of the invariants. This specification is not given here due to the space limitation.

4.4 Security Requirements in ASL

We used supervision and slicing techniques to map the specification on the software requirement level to the implementation level. As an example, suppose the device-manager is implemented in C as given in Figure 8.

```
void ttrx_manager (const int tm_mq)
{
        LIST * ttrxlist;
        char * filename;
        int scanner[SLOTS];
        ttrxlist = Read (filename = "ttrx.list");
        if (!ttrxlist) error ("failed reading %s", filename);
        for (;;) {
                MSG msg;
                int ttrxcard;
                if (Receive (tm_mq, &msg, All, 0) == -1) error ("ttrx_manager: Receive() failed");
                switch (msg.type) {
                case TTRx_Get:
                        ttrxcard = RemoveOne (ttrxlist);
                        if (ttrxcard == -1)
                                if (Send (tm_mq, msg.src_mq, Normal, TTRx_Busy, 0) == -1) error ("ttrx_manager: Send() failed");
                        else {
                                int ts_mq;
                                if (Send (tm_mq, msg.src_mq, Normal, TTRx, 1, trxcard) == -1)
                                        error ("ttrx_manager: Send() failed");
                                if (!Spawn ("ttrx_scanner", &ts_mq)) {
                                        ttrx_scanner (ts_mq, msg.src_mq, ttrxcard);
                                        Die (ts_mq);
                                }
                                scanner[SLOT(ttrxcard)] = ts_mq;
                        }
                        break;
                    case TTRx_Free:
                        ttrxcard = msg.par[0];
                        Add (ttrxlist, ttrxcard);
                        if (Send (tm_mq, scanner[SLOT(ttrxcard)], Normal, TTRx_Terminate, 0) == -1)
                          error ("ttrx_manager: Send() failed");
                          Wait ();
                        break;
                    default:
                        error0 ("ttrx_manager: ignoring message `%s'", msgname(msg.type));
                        break;
}}}
int Send (int src_mq, int dst_mq, int msgtype, int type, int extra, ...)
{
        MSG msg;
        msg.msgtype = msgtype;
        msg.src_mq = src_mq;
        msg.type = type;
        if (extra > 0) {
                va_list arg;
                int count = 0;
                 va_start (arg, extra);
                for (count = 0; count < extra; count++) msg.par[count] = va_arg (arg, int);
                va_end (arg); }
        return (msgsnd (dst_mq, MSGBUF(&msg), MSGSIZ(extra), 0));
}
int Receive (int src_mq, MSG * msg, int msgtype, int flags)
{
        return (msgrcv (src_mq, MSGBUF(msg), MSGSIZMAX, msgtype, flags));
}
```

FIGURE 8. A Simplified C Implmentation of Device Manager

The ASL specification of the security requirements of each invariants is given in the Table 3.

Table 3: Security Requirements

Invariant ID	ADL Specification of Security Requirements
1	Since there is only one state, there is no state variable in the implementation.

2	Module RequestOverflow() {
	msgrcv (src_mq, MSGBUF(msg), MSGSIZMAX, msgtype, flags) |
	(msg.type = TTRx_Get) && (strlen(msg) > MSGSIZMAX) -> abort();}

3	Module ReleaseOverflow() {
	msgrcv (src_mq, MSGBUF(msg), MSGSIZMAX, msgtype, flags) |
	(msg.type = TTRx_Free) && (strlen(msg) > MSGSIZMAX) -> abort();}

4	Send (tm_mq, msg.src_mq, Normal, TTRx, 1, trxcard) | ttrxcard = -1 ->abort();
5	Spawn (“ttrx_scanner”, &ts_mq) | numberof(“ttrx_scanner”) > 7 ->abort();
6	Send (tm_mq, msg.src_mq, Normal, TTRx, 1, trxcard); (Receive(TTRx_Free)||Send)*; | numberof(send(TTRx)) < numberof(Receice(TTRx_Free)) -> abort();
7	(send(TTRx);spawn)*| numberof(send(TTRx) > numberof(spawn) ->abort();
8	Send (tm_mq, msg.src_mq, Normal, TTRx_Busy, 0) | ttrxcard != -1 ->abort();
9	1. (Receive(TTRx_Get);spawn)* | numberof(spawn) > numberof(Receive(TTRx_Get)) ->abort(); 2. i=0(send_from_source | i= i+1)*; j=0(Receive (tm_mq, &msg, All, 0)|j++)* |i < j ->abort();
10	Spawn (“ttrx_scanner1”, &ts_mq1); Spawn (“ttrx_scanner2”, &ts_mq2) | ((ttrx_scanner1 != ttrx_scanner2) && (ts_mq1 = ts_mq2)) -> abort();

Table 3 gives all possible security requirements on the device manager. This experiment shows that our method is able to derive ADL specifications for intrusion detection and isolation, even when we do not know how the attacks are carried out. The specification defines what can be done to the system as opposed to how to attack the system. In this way, this complete set of ADL specifications will be able to detect any attacks falling inside our attack model. We do not have to worry about racing with the attackers unless they come up with new attack models.

4.5 Computational Cost

Two issues need to be further proved are the completeness of the state invariants and the redundancy among them. In our experimental example, we did not face the problem of redundancy because we discard logical duplicates during the invariant construction. We are also able to prove the completeness of the invariants taking advantage of our assumption that we only have two attack models: buffer overflow and race condition.

The computational complexity of the experimental example lies in the number of the variables. Since there is only one state, the computational cost of the invariant derivation algorithm is $e^{|\mathbf{V}|}$ where |V| is the number of the variables, which is 6 in this case. As it can be seen, we did not experience much slow down in this case.

The second issue of the computational cost is the efficiency of the generated ASL code. Since the focus of this paper is on invariant derivation, the results of efficiency experiments will be reported elsewhere.

5. Concluding Observations

This paper presents a method for abstracting security requirements in ASL from a software behavioral specification. This method includes three steps: 1) identifying invariants in the vulnerable services, 2) back tracking to find the usage of the negation of the invariants, and 3) expressing the negation usage in ADL using data structures defined in the original program. The focus of this paper is on the step 1.

Our method is based on the assumption that all behaviors violating security invariants are security attacks. We need to further investigate whether this assumption is realistic.

Our invariant derivation method may face two well-known problems: one is the completeness problem and the other redundancy problem. We overcome these two problems in our experiments by using a small EFSM specification with one symbolic state and a small number of attack models. One future research direction of this work is to extend our algorithm to more general cases.

Other future work lies in the area of intrusion detection. We would like to make our method less depend on the attack models, such that the resulted ASL specification will be able to detect any intrusions including the kinds that are not known at current stage, i.e. not defined in known attack models.

Other research direction includes extending our infrastructure to more complicated attacks such as those involving multiple coordinating hosts.

Acknowledgments

We would like to thank R. Sekar, T. Bowen and H. Liu for their participation and valuable comments at various stages of this work.

References

[1] R. Sekar, Y. Cai, and M. Segal, “A Specification-Based Approach for Building Survivable Systems”, NISSC98, pp338-347.

[2] D. Anderson, T. Lunt, H. Javitz, A. Tamaru, and A. Valdes, “Next Generation Intrusion Detection Expert System (NIDES): A Summary”, SRI-CSL-95-07, SRI International, 1995.

[3] C. Ko, G. Fink and K. Levitt, “Automated Detection of Vulnerabilities in Privileged Programs by Execution Monitoring”, Computer Security Application Conference, 1994.

[4] A. Kosoresow and S. Hofmeyr, “Intrusion Detection via System Call Traces”, IEEE Software, vol. 14, no. 5, Sept-Oct, 1997.

[5] L. C. Paulson, “Proving properties of security protocols by induction”, Proc. 10th Computer Security Foundations Workshop, pp70-83, June 1997.

[6] R. Jeffords and C. Heitmeyer, “Automatic Generation of State Invariants from Requirements Specifications”, Proc. of Sixth Int’l Symp. on Foundations of Software Engineering (FSE-6), 1998.

[7] R.C. Linger, N.R. Mead, and H.F. Lipson, “Requirements Definition for Survivable Network Systems”, ICRE98, pp14-23.

[8] M. Hlady, R. Kovacevic, J. J. Li. et. al, “*An Approach to Automatic Detection of Software Failures*” Proc. IEEE 6th International Symposium on Software Reliability Engineering (ISSRE), pp314-323, Oct. 1995.

[9] Norman Wilde and Michael Scully, “Software Reconnaissance: Mapping Program Features to Code”, *Journal of Software Maintenance: Research and Practice,* Vol 7, No. 1, 1995, pp49-62.