Enforcing Resource Bound Safety for Mobile SNMP Agents *

Weijiang Yu and Aloysius K. Mok
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712, USA
{*wjyu,mok*}*@cs.utexas.edu*

Abstract

The integration of mobile agents with SNMP creates significant advantages for the management of complex networks. Nevertheless, the security concerns of mobile agent technology limit its acceptance in practice. A key issue is to safeguard resource usage abuse by malicious or buggy mobile agents on the hosting system. This paper describes how the TINMAN architecture, a framework and a suite of tools for enforcing resource safety of mobile code is applied to mobile SNMP agents. TINMAN uses a suite of resource-usage checking tools which consists of a resource bound predictor, a usage certification generator and a verifier at compile-time, and certificate validation and monitoring tools at run-time. This paper shows how TINMAN tools can provide 100% coverage by a combination of off-line static analysis and run-time monitoring in enforcing safety on resource consumption of mobile SNMP agents. Experimental results from the current TINMAN implementation are given.

1 Introduction

The Simple Network Management Protocol (SNMP) is the most widely adopted framework for network and system management in service today. Many network devices have been equipped with an SNMP agent gathering system information. The centralized management architecture of SNMP framework, however, has obvious drawbacks regarding performance, flexibility and scalability [17]. Moreover, the lack of essential support for dynamic network management makes it difficult for SNMP to manage increasingly heterogeneous networks.

To address these problems, recent research activities have started to address the integration of mobile agents with SNMP management [12, 14, 17]. The introduction of the mobile agent technology has demonstrated its flexibility and effectiveness in SNMP management. For example, a mobile agent may behave as a sub-agent of an AgentX (a built-in SNMP protocol) to perform the corresponding network management services. This way, the mobile agent framework can decentralize and automate the SNMP management on the desired network devices.

However, the mobile agent-based SNMP cannot be accepted with confidence by hardware manufacturers and system administrators unless an external mobile agent could be trusted not to breach security policies. One of the serious security concerns is *resource bound safety*: limits on the consumption of system resources such as CPU cycles, memory and network bandwidth by a mobile agent must be observed. Failure to properly delimit resource consumption by untrusted mobile agents may deny legitimate users access to system resources, as is in the case of Denial of Service (DoS) attacks.

Most mobile agent frameworks exploit traditional techniques for authorization, authentication and access control to local resources. These techniques, however, cannot protect the hosting systems from DoS attacks caused by the execution of mobile agents, for example, by buggy or malicious SNMP agents. Therefore, the availability of effective tools for checking, monitoring and enforcing resource bound safety is an essential factor in the acceptance of a mobile SNMP agent framework.

This paper describes an architecture called TINMAN whose goal is to provide a set of tools to efficiently and effectively perform resource bound checks on mobile agents. The goal of this paper is to demonstrate the effectiveness of the TINMAN architecture in achieving resource usage safety for a mobile agent-based SNMP framework.

The key idea of our approach is to endow a mobile agent with formally verifiable certificates describing its resource consumption behavior. Ideally, the hosting system, or a *code recipient*, validates the certificate, and, assuming that the certificate passes validation, grants resources to the

*This work is supported in part by a grant from the US Office of Naval Research under grant number N00014-99-1-0402 and N00014-98-1-0704.

code to complete its execution without any further run-time checks. The difficulty with this approach is that the exact resource usage is often determined dynamically and cannot in general be derived by compile-time static analysis. Our approach is a combination of off-line theorem proving and on-line monitoring. TINMAN performs resource bound prediction off-line and outputs the verification conditions including any unproved assertions by the programmer that are automatically checked at run time. The code recipient, in addition to certificate validation, automatically performs run-time checking on the unproved assertions with relatively small overhead. The off-line and run-time mechanisms together provide *complete coverage* and guarantee resource safety. This mixed strategy applied to resource safety distinguishes our approach from previous work.

A major contribution of our work is that we push the limit of automating fine-grain resource consumption checks by the adoption of a set of tools for resource utilization prediction, certificate generation and validation, and run-time event monitoring.

An overview of the TINMAN architecture is presented in the next section. Section 3 describes how to use TINMAN tools to ascertain the resource consumption safety of mobile SNMP agents. It includes a detailed description of resource prediction, certificate generation and validation and on-line checks using an example SNMP agent. Experimental results are given in section 4. Section 5 discusses related work. Concluding remarks and future work are in section 6.

2 Architecture Overview of TINMAN

In this section, we give a general overview of the TINMAN architecture. The application of TINMAN to a concrete mobile agent-based SNMP will be demonstrated in the next section.

The TINMAN system ensures the resource safety of mobile codes by detecting potentially malicious mobile code that may consume inordinate amounts of resources such as CPU, memory and network bandwidth, as manifested in DoS attacks. There are two key issues regarding resource safety: resource safety policy and resource safety enforcement. *Resource safety policy* spells out resource usage constraints that mobile agents must observe in order to maintain permission to execute on a host. In TINMAN, the resource safety policy is established by a set of specifications which consists of three parts: resource usage for each service provided by the hosting system, resource limit for each active session (*i.e.*, execution of a mobile agent), and the interpretation of the specification by a proof system consisting of axioms and inference rules. *Resource safety enforcement* pertains to the authorization of resource usage and the limitation of actual resource usage by a program. It prevents a mobile agent from violating resource safety policy by on-line monitoring and dynamic analysis.

TINMAN performs resource safety enforcement by coupling language level and run-time system level mechanisms. At the language level, a mobile agent is analyzed and converted into a *certified program*, *i.e.*, a program with a certificate on resource bounds. The certificate, once formally validated, guarantees that the program will not violate a resource safety property.

In general, it is not possible to guarantee resource bound by only checking the certificate without further run-time checks since exact resource usage cannot in general be derived *a priori* by compile-time static analysis, unless, say, the halting problem is decidable, which it is not. Our approach is to combine off-line verification and on-line checking by the run-time system. It is akin to constructing a proof on resource usage for every computation, where part of the proof is done off-line and the unverified part is checked on-line as the particular computation unfolds.

Figure 1 shows the architecture of the TINMAN system. The following describes the typical steps taken to ensure resource safety of a piece of mobile agent code by the functions of the main components of TINMAN, as illustrated in Figure 1.

- *Resource prediction*. The resource bound of the code is predicted off-line with user-input information, if necessary. This is done interactively with a suite of compiler tools modified for the purpose while allowing the programmer to enter assertions that can only be established on-line (*e.g.*, run-time environment-specific parameters) or cannot be established automatically by the tools. The output is a set of annotations and assertions, called the *resource skeleton* which captures the resource usage behavior of the code, to be checked against the resource safety policy.

- *Resource specification generation*. The resource skeleton is automatically translated into a specification consisting of a group of predicates in a formal logic of a proof system.

- *Certificate generation*. The specification is secured by applying the resource safety policy, with the help of a proof system [11]. The output is a proof of the resource usage bounds of the resource skeleton, called the *resource usage certificate*. The code with resource skeleton and certificate are now ready to be downloaded by a remote host.

- *Resource skeleton validation*. A remote host (code recipient) validates the resource skeleton that has been inserted off-line, and checks for any inconsistency with the imported code, and any violation against the safety policy on resource usage.

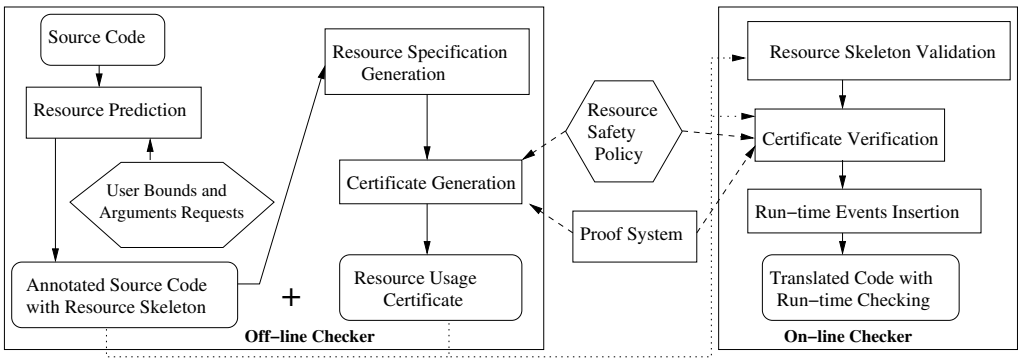

Figure 1. The Architecture of TINMAN for Resource Bound Safety for Mobile Code

- *Certificate verification*. If no violation to resource safety policy is detected, the resource usage certificate is verified by a proof checker. If it is found to be valid, the code is safe to be installed and executed. We emphasize that the validation on-line is done by proof-checking which is much more doable than proof-generation that is done off-line.

- *Run-time events insertion*. Assertions whose validity the off-line analyzer fails to provide a proof are monitored on-line. This is done by having the program annotations automatically converted into event generator code that notifies the run-time system on the progress of the mobile code execution. The run-time resource usage monitor matches the events generated at run time with the behavior inferred from the resource skeleton.

Architecturally, the first three steps are done on the code producer site, grouped into an *off-line checker*, and the last three steps are done by the code recipient, grouped into an *on-line checker*. The resource prediction module requires potentially the intervention of the programmer, and so does the certificate generation module in case a certificate cannot be automatically generated due to the overly complicated program structure of the source code. The user of a host where mobile code runs needs to manually configure the policy rules on resource limits such as maximum running time and memory usage of a mobile agent process. All other modules are automatically done by TINMAN tools. In other words, the programmer of the mobile agent code helps to create the certificate for resource safety off-line since this can be done by taking advantage of computational resources that are usually more abundant off-line.

The guiding principle of TINMAN is to do as much work off-line as possible and leave the hosting code recipient to do as little as possible while ensuring complete coverage. Accordingly, compliance with the resource security policy by mobile agents is to be achieved by having the code producer and the recipient cooperate in verifying what is possible off-line and monitoring on-line what has not been verified off-line.

3 TINMAN Application: Mobile SNMP Agents

The TINMAN architecture is a general framework for securing the resource usage bounds of mobile code, and it is applicable to a variety of applications and platforms. In this section, we describe how the TINMAN tools are used to enforce resource bound safety for untrusted and potentially malicious mobile SNMP agents.

We use the open source NET-SNMP toolkit [5] as the framework for writing mobile SNMP agents. NET-SNMP (previously known as UCD-SNMP) is an implementation of all three SNMP protocols (v1, v2c, v3) as well as AgentX, which includes a master agent and libraries to develop sub-agents. In a NET-SNMP-based SNMP management framework that supports mobile agents, a mobile SNMP agent may behave as an AgentX sub-agent, migrating between network nodes to acquire information from or provide services to a local host.

Example: A Mobile SNMP Agent for Single Scalar Registration. For clarity of explanation, we use a simple agent that comes with the NET-SNMP package. The agent, called "scalar agent" creates some scalar registrations at a local host which allows some simple variables to be accessed via SNMP. The source code of the example is shown in Figure 2. The main() code establishes the connection between the local host and the agent itself. It attaches itself to the master agent using the AgentX protocol, initializes agent library and MIB code, and falls into a while loop to poll SNMP operations. The init_scalar_int() code is the implementation of a user-defined SNMP service. It registers a variable "example1" as accessible and makes it writable. When the scalar agent arrives at a node where it is scheduled to run, it performs the requests for manipulation on the

```
int  main (int argc, char **argv) {
B1:  snmp_enable_stderrlog();
       ds_set_boolean(DS_APPLICATION_ID,
              DS_AGENT_ROLE, 1);
       init_agent("scalar");
U2:  init_scalar_int();
B3:  init_snmp("scalar");
      keep_running = 1;
      signal(SIGTERM, stop_server);
      signal(SIGINT, stop_server);
L4:  while(keep_running) {
      B5:   agent_check_and_process(1);
      }
B6:  snmp_shutdown("scalar");
       exit(0);
}
RETSIGTYPE stop_server(int a) {
   keep_running = 0;
}
```

```
void  init_scalar_int(void) {

B7:  int   example1 = 42;  /* default value */
     oid my_registration_oid[] =
       {1,3,6,1,4,1,8072,2,1,1,0};
     DEBUGMSGTL(("example_scalar_int",
       "Initalizing example scalar int.\
        Default value = %d\n",
        example1));
     netsnmp_register_int_instance(
       "my example int variable",
        my_registration_oid,
        OID_LENGTH(my_registration_oid),
        &example1, NULL);
    DEBUGMSGTL(("example_scalar_int",
     "Done initalizing scalar int\n"));
}
```

Figure 2. Source Code of *Scalar SNMP Agent*

variable "example1" that have been sent to the node (the master agent) from the outside.

In the following subsections, we use the scalar agent to illustrate the design of the off-line checker and its usage by programmers of mobile agents, and the use of the on-line checker invoked at the hosting system. We show that TINMAN provides full coverage and guarantees resource safety of mobile SNMP agents.

3.1 Resource Bound Prediction for SNMP Agents

The off-line checker provides tools for the programmer of a mobile SNMP agent to perform resource bound prediction and to provide resource usage certification for the agent code. The output of the resource analysis as well as a usage certificate as the evidence for the code's safety with regard to resource consumption, are transferred together with the mobile agent code.

In our current implementation, only timing analysis and live memory-demand analysis are performed. We are investigating the extension of usage bound prediction to other system resources such as disk I/O and static network bandwidth consumption. In general, the approach we present in this paper may be applied in the same way to safeguard the usage of the variety of system resources, with more efforts on analysis of the relevant system calls and their parameters (*e.g.* fork() and system() calls for thread and process analysis), and relevant safety policies definitions.

Resource usage prediction is done at the source code level. First, we modified the Broadway Compiler [4], a source-to-source translator for ANSI C to parse the source code and output a flow graph with data and control information for the resource analysis modules. The resource usage of a mobile agent is mainly decided by its program structure (statements, branches, loops etc.) and the resource usage of the library services calls (*e.g.* SNMP APIs) invoked.

For the part of program structure analysis, a resource schema approach is used to perform timing analysis on a variety of programming constructs, henceforth referred as to *tasks*. Timing analysis tasks are defined at different granularities such as basic blocks, loops and service routines. The task information of the scalar agent, as shown in Figure 2, was generated by the program flow analysis. Each task in the flow graph is labeled with its type (*e.g.* L = Loop task) followed by a global counter value.

The basic idea of the timing schema approach [13] is that the program execution time is determined by basic tasks and the control structures of a program. Loops are treated specially to obtain their resource bounds. For loops whose loop bounds are difficult to deduce automatically (or may not even exist as is in the case of an infinite loop) by static analysis but nevertheless are known to the programmer, the programmer is asked to assert a loop bound which will be automatically monitored on-line.

Considering the possible actions of malicious codes on memory, our efforts in memory usage analysis are focused on memory allocation requests in a program. We combine analysis of memory allocation statements (malloc() and memdup() as commonly used by SNMP agents), path analysis of memory freed, and call chain analysis to achieve this goal.

The resource usage of library services are treated as well. Resource usage of a service is specified in a policy rule which is a pair of pre- and post- condition specifications defined in a formal logic (to be described in the next subsection). The resource consumed by a service is parameterized given the ranges of its arguments. Exact values are determined at the remote host site at run-time. This ap-

proach is more flexible since the policy is configurable for a specific platform and run-time environment. For example, snmp_enable_stderrlog() which is called in main() in Figure 2 is an SNMP agent library service. Its exact execution time and memory allocation is determined at the hosting system site, and defined as an axiom in the TINMAN proof system.

For a user-defined function, its resource usage is obtained by summing all its basic blocks, loops, library services calls and user-defined functions.

The timing and memory bound information obtained by either automatic analysis or from programmer input are maintained by using a *resource skeleton*. Basically, a resource skeleton can be viewed as an abstraction of a program with regard to its resource consumption, and will be used by the usage certificate generator and the on-line checker. It contains the execution time and memory allocation for each task, path information of the program, loop bounds, and parameterized resource usage for services.

```
int   main (int argc, char **argv) {
/*@ Entry T[t0] M[m0]*/
B1: {...}
/*@B1: T[Entry + tsnmp_enable_stderrlog + tds_set_boolean + tinit_agent+3]
        M[Entry + msnmp_enable_stderrlog + mds_set_boolean + minit_agent]*/
U2: {...}
/*@U2: T[B7]  M[B7]*/
B3: {...}
/*@B3: T[U2 + tinit_snmp + 2*tsignal + 7]
         M[U2 + minit_snmp + 2*signal] */
/*@L4lb = 1000 */
L4: {...}
/*@L4: T[B3 + L4lb*(tagent_check_and_process+1) + 1]
        M[B3 + L4lb*magent_check_and_process]*/
B6: {...}
/*@B6: T[L4 + tsnmp_shutdown + texit + 1]
        M[L4 + msnmp_shutdown]*/
}
void init_scalar_int(void) {
/*@Entry */
B7: {...}
/*@B7: T[Entry + 13 + 2*tDEBUGMSGTL + tnetsnmp_register_int_instance]
        M[Entry + 2*mDEBUGMSGTL + mnetsnmp_register_int_instance]*/
}
```

Figure 3. Resource Skeleton for code in Figure 2

Results of resource usage analysis on the scalar agent are shown in Figure 3. Resource annotations for each task are enclosed in "/*@ ...*/", which can be identified by the on-line validator and be ignored by a standard compiler. T[exp] and M[exp] represent the time and memory-demand bounds of the corresponding task. For example, the annotation for task B3: T[U2 + tinit_snmp + 2*tsignal + 7] means the execution time to the point after B3 is the sum of T[U2], T[init_snmp], 2*T[signal] and 7, where T[U2] is obtained recursively from the corresponding annotation for U2, and the constant 7 is the total number of atomic blocks involved in B3. We observe from the above that a task is enclosed by a pair of predicates on the resource usage, and this arrangement can be treated like a Hoare triple. In the next subsection, we shall expand on this idea into a process of formal certificate generation.

3.2 Usage Certificate Generation

The resource skeleton contains information about the resource usage bound of a mobile agent. At the stage of usage certificate generation, the off-line checker produces a certificate for the resource skeleton as a proof for its validity. Specifically, it translates the annotations into a set of predicates coded in a formal logic of a proof system, and then proves them within the proof system.

Our basic strategy for establishing the correctness of certificates is to use an extended Hoare Logic. For a task T, an extended Hoare triple for T is in the form $\{$P$\}$ T $\{$Q$\}$, where assertion P is the pre-condition, and assertion Q is the post-condition that holds if P is true and T terminates.

For example, the extended Hoare triple for task B1 in Figure 2 and Figure 3 is

$$\text{PRE1: } \{\mathbf{now} = t0 \wedge \mathbf{mem} = m0 \wedge \mathbf{terminate}\}$$
$$\text{B1}$$
$$\text{POS1:}\{\mathbf{now} <= t0 + tsnmp\_enable\_stderrlog + tds\_set\_boolean + tinit\_agent + 3 \wedge \mathbf{mem} <= m0 + msnmp\_enable\_stderrlog + mds\_set\_boolean + minit\_agent \wedge \mathbf{terminate}\}$$

where **now** and **mem** represent the time value and size of allocated memory, with initial values of *t0* and *m0*, respectively. The boolean variable **terminate** is a program termination indicator. These logical assertions are translated from the corresponding resource skeleton into tasks, for instance, PRE1 from annotation of Entry and POST1 from annotation of B1 in the example. The pre-condition of a user-defined-function task U2 is POS1 (*i.e.*, the post-condition of B1), and the post-condition of U2 (converted from T[B7] and M[B7]) is

$$\text{POS7: } \{\mathbf{now} <= T[B1] + 13 + 2 * tDEBUGMSGTL + tnetsnmp\_register\_int\_instance \wedge \mathbf{mem} <= M[B1] + 2 * mDEBUGMSGTL + mnetsnmp\_register\_int\_instance \wedge \mathbf{terminate}\}$$

where Entry is replaced by B1 at the calling point.

The specifications for loops and service calls may contain extra predicates on loop bounds and ranges of parameters to services that are derived from the resource skeleton and resource safety policy rules. The following example shows the specification for task L4. It contains a predicate on the loop bound of its precondition PRE4.

PRE4: $\{\mathbf{now} <= T[U2] + tinit\_snmp + 2 * tsignal + 7 \wedge L4lb = 1000 \wedge \mathbf{mem} <= M[U2] + minit\_snmp + 2 * msignal \wedge \mathbf{terminate}\}$

L4

POS4: $\{\mathbf{now} <= T[B3] + L4lb * (tagent\_check\_and\_process + 1) + 1 \wedge \mathbf{mem} <= M[B3] + L4lb * magent\_check\_and\_process \wedge \mathbf{terminate}\}$

The source code of the mobile agent with its resource skeleton is converted into tasks and a set of assertions, referred to as *resource specifications*. The specifications, as well as the theory for their proof generation are then formalized for use by the theorem prover PVS (Prototype Verification System) [11]. PVS supports a specification language based on typed higher-order logic, and provides tools to allow mechanized checking on reasoning about expressions in the logic.

The formalization consists of two parts. First, in order to formulate resource specification into the PVS specification language, programs are identified with their semantics, *i.e.*, the relations on states. A *state* contains a mapping of program variables to values, current time, allocated memory, and termination indicator. We developed an extended and modified version of the construction rules defined in Jozef Hooman's framework [7]. All tasks are defined with regard to their resource specifications.

Second, the proof system is constructed with respect to resource specification by formalizing programming constructs (*i.e.* tasks). The timing analysis tasks are axiomatized by inference rules and axioms. The rules in the framework are proved independently and published as part of the resource security policies as well. Discussion of the complete definition of program constructs and the corresponding set of axioms and inferences rules are outside the scope of this paper. Interested users are referred to [1] for more details.

A resource specification in PVS is automatically obtained by applying the program construction rules as a result. The conversion from source code and its resource skeleton to the specification is invisible to the programmer, and the specification itself is only used for internal purpose. Figure 4 shows a PVS specification fragment of the scalar agent for task B1 and L4. Each task consists of pre- and post-conditions on its resource usage ($ASRT_i$), task constructs (bb1 and loop1) and theories in the form of Hoare Triples (cor1 and cor4) that are ready to be proved.

The proof of the specifications is performed by the certificate generation tool. The proof module in TINMAN is constructed based on a set of proof strategies, an automatic theorem proving mechanism provided in PVS by composing proof steps into strategies. Several strategies are developed for a variety of program constructs including basic blocks, loops, and conditional branches. Using these proof strategies, the resource specification for an annotated program is proved automatically. PVS outputs the proof into a text file which constitutes the usage certificate for the program.

```
IMPLIES (P1, P2: [State–>bool]) : [State–>bool] =
      (LAMBDA s : P1(s) IMPLIES P2 (s));
spec (P, Q) : program = (LAMBDA s0, s1: P(s0) IMPLIES Q(s1))
%task B1: basic block bb1
ASRT1: pred[State] = (LAMBDA s :
    now(s) = t0 AND
    mem(s) = m0 AND term(s))
ASRT2: pred[State] = (LAMBDA s :
    now(s)<=t0 + tsnmp_enable_stderrlog +
      tds_set_boolean + tinit_agent+3 AND
    mem(s) <= m0 + msnmp_enable_stderrlog +
      mds_set_boolean + minit_agent AND term(s))
bb1  : program = seq (seq (seq(srvc(tsnmp_enable_stderrlog,
                          msnmp_enable_stderrlog),
                       srvc(tds_set_boolean,
                          mds_set_booleans)),
                  srvc(tinit_agent,minit_agent)), bb(3))
cor1 : LEMMA bb1 => spec ( ASRT1, ASRT2)

%task L4: loop task loop1
ASRT4: pred[State] = (LAMBDA s :
    val(s)(inter_lb1) = 0 AND L4lb = 1000 AND
    now(s) <= T[U2] + tinit_snmp + 2*tsignal + 7 AND
    mem(s) <= M[U2] + minit_snmp + 2*msignal AND term(s))
ASRT5: pred[State] = (LAMBDA s :
    now(s) <= T[B3]+L4lb*(tagent_check_and_process+1)+1 AND
    mem(s) <= M[B3]+L4lb*magent_check_and_process AND term(s))
lb1axiom : AXIOM  val(s)(inter_lb1) <= L4lb AND
             val(s)(inter_lb1) >= 0
loop1init  : program = lbop(inter_lb1, (LAMBDA s: 0))
loop1cond  : pred[State] = (LAMBDA s: val(s)(inter_lb1)<L4lb)
loop1body  : program = seq( srvc(tagent_check_and_process,
                          magent_check_and_process),
          lbop(inter_lb1,(LAMBDA s: val(s)(inter_lb1)+ 1)))
loop1    : program = while(loop1cond, 1 , 0, loop1body)
cor4     : LEMMA loop1 => spec(ASRT4, ASRT5)
```

Figure 4. A Fragment of PVS Specification for Figure 3

The corresponding proofs to Figure 4 are given in Figure 5. Program constructs such as bb (basic blocks), seq (sequence of tasks) and srvc (services calls) are expanded, and basic PVS proof rules such as ASSERT and SKOSIMP* are applied to complete the proofs. The proof strategy for loop task L4 is more complicated than that of basic block task B1 in that an inference rule (LEMMA "rulewhile") for loops and a loop invariant (loop1inv) are involved in the proof generation.

The size of the proof, however, may be as large as, for this example, three times the source code. In order to reduce its size, we further simplify the usage certificate by only keeping the strategies and related parameters for reconstructing the proof. As a result, the above proof fragment is reduced to the form of

```
cor1:  SB bb1 ASRT1 ASRT2
cor4:  SL loop1 loop1inv loop1ceterm
```

where SB and SL stand for the strategies for basic tasks and

loop tasks, respectively.

```
(|cor1| "" (EXPAND "bb1")
 (("" (EXPAND "ASRT1") (("" (EXPAND "ASRT2")
  (("" (EXPAND "seq") (("" (EXPAND "spec")
   (("" (EXPAND "bb") (("" (EXPAND "srvc")
   (("" (EXPAND "=>") (("" (ASSERT)
     (("" (SKOSIMP*) (("" (ASSERT)
     (("" (SKOSIMP*) (("" (ASSERT)
     (("" (SKOSIMP*) (("" (ASSERT) NIL NIL)) .... NIL))

(|cor4| "" (LEMMA "rulewhile")
 (("" (EXPAND "loop1") (("" (INST?)
  (("" (GROUND) (("" (INST + "loop1inv" "loop1cterm")
  (("" (HIDE 2) (("" (LEMMA "lb1axiom")
  (("" (SPLIT)
   (("1" (GRIND) NIL NIL)
    ("2" (EXPAND "=>") (("2" (SKOSIMP) (("2" (GRIND) NIL... NIL)
    ("3" (EXPAND "=>") (("3" (SKOSIMP) (("3" (GRIND) NIL... NIL)
    ("4" (GRIND) NIL NIL)) ... NIL))
```

Figure 5. The Proof Fragment for Figure 4

3.3 On-line Validation and Run-time Check for SNMP Agents

Upon receiving a mobile SNMP agent, a hosting system confirms the agent has a resource skeleton and a simplified usage certificate. Then the on-line checker is invoked to perform safety enforcement by taking advantage of the guarantees of the off-line checker and by run-time monitoring.

At the static verification stage, the on-line checker first validates the supplied resource skeleton and usage certificate, and detects any violation against safety policy on resource utilization limits. In the case of the scalar agent, the on-line checker checks that all resource annotations ($/*$@B1...$*/$,... $/*$@B7 ... $*/$) are correctly inserted and the loop bound for L4 is provided by the user as well. Given the validity of the annotations, the on-line checker calculates the actual resource bound with instantiated policies on service calls. In other words, the variables of tsnmp_enable_stderrlog, msnmp_enable_stderrlog, tds_set_boolean, etc., are replaced with actual values in the hosting system. The resource usage bound of the post-condition (*i.e.*, POS5) of main() code is checked against the corresponding rule for a session of a mobile agent.

If no violation is found, the complete usage certificate is restored from the certificate skeleton, and the proof checker verifies the supplied usage certificate against the specifications within the PVS system. The proof checking procedure is as simple as a validation run of PVS in *batch mode* and automatically reruns all the proofs in the usage certificate. An invalid usage certificate will generate errors which will be caught by the proof checker by examining the run log file.

The untrusted user-provided assertions, if any, on loop bounds and function argument ranges are translated into appropriate assertions for checking the value range of untrusted data and communication primitives: UNIX signals and FIFO queues in our current implementation. The while loop in the scalar agent is changed to the following code.

```
...
_TINMAN_scope(SCOPE_LOOP_IN);
int internal_lb = 0;
while(keep_running) {
      _TINMAN_assert(internal_lb++<1000);
      agent_check_and_process(1);
}
_TINMAN_scope(SCOPE_LOOP_OUT);
...
```

A run-time exception will be raised if any assertion is violated. Whereas dynamic resource utilization monitoring has been used by previous work, our approach avoids much of the work since the resource bound safety is guaranteed by static verification and run-time checking is required only on programmer-provided annotations.

4 Experimental Results

TINMAN has been implemented by a set of tools that support resource bound prediction and certificate generation and validation. In this section, we shall present some experimental results obtained by using our tools. We use the example AgentX sub-agents (a.k.a mobile agents) provided by the NET-SNMP packages as the benchmark programs. They are typical mobile agents for SNMP management. A summary of the programs' properties is given in Table 1.

Program	#Services	#Tasks
scalar_int	6	6
data_set	20	6
delayed_instance	26	23
notification	14	7

Table 1. Program properties of tested mobile code

We have extended the security policies for mobile agent-based NET-SNMP applications by defining their resource usage of SNMP APIs (library functions). There are eight agent modules with more than two hundreds APIs, and even more SNMP APIs other than agent modules that could be used in mobile agent applications. Currently we only tested all of the twenty-five NET-SNMP APIs that are called in the benchmark. The code executes on a Intel Pentium III running Debian Linux and were analyzed and monitored with regard to their execution time and memory allocation. The benchmark agents were submitted for analysis to the TINMAN tools which produced resource bound annotations and a usage certificate for each agent, given the loop

Program	Source Code	Resource Skeleton	Increase	Full Certificate	Certificate Skeleton	Decrease
scalar_int	1818	2154	18.5%	4242	319	92.5%
data_set	3472	4432	27.6%	4244	321	92.4%
delayed_instance	5340	6979	30.6%	13673	1078	92.1%
notification	2590	3112	20.1%	4946	349	92.9%

Table 2. Size of Resource Skeleton and Certificate (in bytes)

Program	Specification Generation	Off-line Certificate Construction + Validation	On-line Annotation Verification	On-line Certificate Check
scalar_int	21.0ms	0.93s + 10.08s	28ms	10.09s
data_set	21.7ms	1.28s + 10.05s	44ms	10.11s
delayed_instance	57.2ms	2.87s + 30.51s	103ms	32.1s
notification	30.5ms	1.31s + 11.25s	27ms	11.54s

Table 3. Cost of off-line checker and on-line checker

bound in main() as 1000. The on-line checker validated an agent by applying the extended policies on services and agent sessions.

The purpose of the experiments is to demonstrate the effectiveness and performance of TINMAN system applied to mobile SNMP agents. The summary of the experimental results and our observations are given as follows:

- Code size augmentation due to insertion of the annotations and generation of usage certificates depends on the control structure and the number of tasks of a program. As shown in Table 2, the programs with resource skeletons are between 18.5% and 30.6% larger than the original ones. We also observed that the certificate size is significantly decreased by up to 92.9% by using a certificate skeleton. The size of a certificate skeleton is less than 13.7% of total size of the annotated program.

- The certificates of all example programs are generated completely automatically. As shown in Table 3, certificate validation (verification of resource usage bounds) is an order of magnitude slower than certificate construction (conversion of resource skeleton to PVS-friendly formal specification). The overhead is two-fold. First, a certificate consists of PVS rules that are interpreted by the PVS prover interactively. Second, the current proof strategy for loops and compound choice tasks involves catch-all prover commands like GRIND which is timing consuming. An interesting finding is that these SNMP agents follow a common programming style (program structures, a series of SNMP APIs calls, actions to operation requests for MIBs, etc.). So developing new proof strategies for certain coding patterns will improve the (automatic) proof generation and validation.

- The overhead of running a mobile program with annotations comes from run-time monitoring of programmer-provided information and the communication between the annotated program and the on-line checker during the execution. All of them have small monitoring overheads.

5 Related Work

Some novel frameworks on integration of SNMP management with mobile agents have been proposed [17, 12, 14] by past authors. However, they did not provide resource security at the application level to deter malicious mobile agent from mounting DoS attacks. On the other hand, some researchers have developed resource management systems for mobile agents at the operating systems level [3]. In Bellavista's work [3], the resource usage of mobile agents is monitored and controlled on-line. Certain security policies are enforced to restrict resource utilization of mobile agent execution. This approach may incur significant run-time overhead which is an issue in resource-critical system like SNMP network devices.

The SNMP management in active network environments has attracted quite a bit of attention in recent years. Extant resource safety efforts for mobile agents usually fall into two levels: at the programming language level and at the run-time system level. For example, Smart Packets [15] checks the CPU and memory usage of active packets written in Sprocket and enforces the limits on the number of instructions executed, amount of memory used, and access to MIB variables. The KeyNote in PLANet has a similar mechanism [2, 6]. This usually comes at the price of reduced expressiveness of the programming language (*e.g.* Sprocket, PLAN) and may turn off programmers who are unwilling to adopt a restrictive language just for their mobile SNMP applications. In addition, not all these systems provide resource utilization prediction and thus resource-usage safety policies are difficult to enforce at run-time.

Our certified-code approach is similar in spirit to the

Proof Carrying Code (PCC) [10] and Typed Assembly Language (TAL) [9, 8]. In practice, PCC has been used to verify low-level safety properties, and it does not address the resource bound security problems in terms of resource behavior prediction and program termination. Walker [16] has developed an extension of TAL that derives the type system from security automata which can specify and enforce expressive security policies. Unlike PCC and TAL, TINMAN concentrates on resource security assurance at a higher level to prevent DoS attacks by buggy or malicious codes with infinite loops or improper arguments to services calls. More importantly, we provide full coverage by means of a combination of off-line verification and on-line monitoring. We want the resource security policy to be flexible and configurable at the code recipient site. In addition, the proof system constructed using the PVS system makes it easier for proof construction and validation for more complicated mobile applications.

6 Conclusions

In this paper, we explain how the TINMAN architecture for securing resource safety is applied to mobile SNMP agents. We demonstrate the effectiveness and efficiency of the TINMAN suite of tools which provide complete coverage on enforcing resource usage bound limits of a mobile SNMP agent. This is achieved by cooperation between the programmer of mobile agent code and the hosting code recipient. The division of labor is in such a way that the programmer can interactively provides certificates for resource safety off-line and the code recipient can efficiently validate the certificates against resource bound safety policy on-line.

TINMAN is a platform-independent architecture and does not require modifying the underlying library and operating system. It makes TINMAN applicable to a variety of mobile SNMP agents frameworks by extending the existing policy rules for SNMP APIs. Moreover, our certified-code approach of static checking a resource usage certificate avoids much of performance overheads, a prime concern for SNMP management in network nodes.

We recognize, however, that many mobile codes are shipped in a "bytecode" format to the code recipient. With more experience gained from applying the TINMAN architecture, we hope to be able to customize this framework for some versions of byte-code for C and for Java.

References

[1] TINMAN project. www.cs.utexas.edu/wjyu/tinman/tutorial.

[2] D. S. Alexander, K. G. Anagnostakis, W. A. Arbaugh, A. D. Keromytis, and J. M. Smith. The Price of Safety in an Active Network. Technical Report MS-CIS-99-02, University of Pennsylvania, January 1999.

[3] P. Bellavista, A. Corradi, and C. Stefanelli. How to monitor and control resource usage in mobile agent systems. In *3 rd IEEE Int. Symp. on Distributed Objects and Applications (DOA'01)*, Italy, Sep 2001.

[4] S. Z. Guyer, D. A. Jimnez, and C. Lin. Using C-Breeze. Technical report, Department of Computer Sciences, The University of Texas, January 2001. http://www.cs.utexas.edu/users/lin/cbz/index.html.

[5] W. Hardaker. The net-snmp project. http://netsnmp. sourceforge.net.

[6] M. W. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. PLAN: A packet language for active networks. In *International Conference on Functional Programming*, pages 86–93, 1998.

[7] J. Hooman. Correctness of real time systems by construction. In *FTRTFTS: Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 19–40. LNCS 863, Springer-Verlag, 1994.

[8] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. TALx86: A Realistic Typed Assembly Language. In *Second ACM Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, GA, May 1999.

[9] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *IEEE Symposium on Principles of Programming Languages*, January 1998.

[10] G. C. Necula. Proof-Carrying Code. In *POPL'97*, pages 106–119, Paris, France, January 1997.

[11] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In *11th International Conference on Automated Deduction*, pages 748–752. Lecture Notes in Artificial Intelligence, Springer Verlag, 1992.

[12] B. Pagurek, Y. Wang, and T. White. Integration of mobile agents with snmp: Why and how. In *2000 IEEE/IFIP Network Operations and Management Symposium*, pages 609–622, Honolulu, Hawaii, USA, April 2000.

[13] C. Park and A. C. Shaw. Experiments with a program timing tool based on source-level timing schema. *Computer*, 25(5):48–57, May 1991.

[14] A. Puliafito and O. Tomarchio. Using mobile agents to implement flexible network management strategies. *Computer Communication Journal*, 23(8):708–719, April 2000.

[15] B. Schwartz, A. W. Jackson, W. T. Strayer, W. Zhou, D. Rockwell, and C. Partridge. Smart Packets: Applying Active Networks to Network Management. *ACM Transactions on Computer Systems*, 18(1):67–88, 2000.

[16] D. Walker. A type system for expressive security policies. In *Symposium on Principles of Programming Languages*, pages 254–267, 2000.

[17] M. Zapf and K. Herrmann. Decentralized snmp management with mobile agents. In *IFIP/IEEE Conference on Integrated Network Management*, pages 623–635, Boston, May 1999.