TINMAN: A Resource Bound Security Checking System for Mobile Code*

Aloysius K. Mok and Weijiang Yu

Department of Computer Sciences, University of Texas at Austin
Austin, Texas 78712 USA
{mok,wjyu}@cs.utexas.edu

Abstract. Resource security pertains to the prevention of unauthorized usage of system resources that may not directly cause corruption or leakage of information. A common breach of resource security is the class of attacks called DoS (Denial of Service) attacks. This paper proposes an architecture called TINMAN whose goal is to efficiently and effectively safeguard resource security for mobile source code written in C. We couple resource usage checks at the programming language level and at the run-time system level. This is achieved by the generation of a resource skeleton from source code. This resource skeleton abstracts the resource consumption behavior of the program which is validated by means of a resource usage certificate that is derived from proof generation. TINMAN uses resource-usage checking tools to generate proof obligations required of the resource usage certificate and provides full coverage by monitoring any essential property not guaranteed by the certificates. We shall describe the architecture of TINMAN and give some experimental results of the preliminary TINMAN implementation.

1 Introduction

Mobile codes are becoming widely deployed to make wide-area networks extensible and flexible by adding functionality and programmability into the nodes of the network. Mobile code which is written on one computer and executed on another, can be transmitted in the form of either binary or source code. The former includes Java Applet, ActiveX, or executables for specific platforms. The latter includes applications written in high-level languages like C, scripts such as shell files or JavaScript embedded in HTML files and specific languages designed for *active network* [1] such as PLAN [2] and Smart Packets [3].

Not surprisingly, there is increasing demand on host systems to provide security mechanisms for shielding users from the damages caused by executing untrustworthy mobile code. One of the serious concerns of mobile code security is *resource bound security*. Resource bound security pertains to resource consumption limits and the prevention of unauthorized use or access to system

* 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.

resources that may not directly cause corruption or leakage of protected information. Failure to properly delimit resource consumption by untrusted mobile code may deny legitimate users access to system resources, as is in the case of Denial Of Service (DoS) attacks.

In this paper, we propose TINMAN, a platform-independent architecture whose goal is to efficiently and effectively perform resource bound security checks on mobile code. Although the TINMAN architecture is not restricted to mobile code developed in a particular language, we analyze mobile C code in our prototype system since many mobile programs are written in C and lack security checks. For example, a user might try to extend his system software with an external software component, or download and install open source software and applications that are often in the form of C source code, from both trusted and untrusted sources. Thus, we want to validate the TINMAN architecture first for C code; we believe that the TINMAN architecture should be applicable directly to the mobile code in scripts and byte-code as well.

Most of the work on mobile code security has not addressed resource bound security or is not effective when it is applied to a general-purpose languages such as C. For example, Java Sandbox [5], Smart Packets [3] and PLAN for PLANet strive to prevent abuse by mobile code at the programming language level by limiting a user's ability to gain access to system resources through language design. This usually comes at the price of reduced expressiveness of the programming language and may turn off programmers who are unwilling to adopt a restrictive language just for their mobile applications. Also, these approaches cannot guarantee the resource bound consumption which is essential to the hosting systems; for instance, they cannot detect buggy or malicious code that may fall into an infinite loop and use all the CPU cycles and possibly all system memory. An alternative approach to attaining resource bound security is to enforce certain *security policies* that restrict the resource utilization of mobile code execution at the run-time system level. Run-time checking of resource usage alleviates the drawback of strict reliance on language-level mechanisms but it is costly, and certain methods of access control do not apply since mobile code is not stored on the computer it is executed on. In addition, not all these systems provide resource utilization prediction and thus resource-usage security policies are difficult to enforce.

We believe that in order to ensure resource bound security for mobile source code, every hosting system should be endowed with a capability to accept usage specification (via policy-based resource bounds) and to monitor (via policy enforcement) the execution of untrusted mobile programs. This policy specification to enforcement linkage should be established in a way that cannot be spoofed, or else the system should not be trusted to transport mobile code. Our approach is to endow mobile source code with a verifiable certificate that specifies its resource consumption behavior. The key idea underlying the TINMAN architecture, however, is the recognition that 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) and that a combination of compile-time (off-line) and run-time (on-line) techniques is needed to link policy specification to enforcement. The real hard issue is to determine the relative roles of off-line vs. on-line analysis. To answer this question, we adopt the following principle:

"Check the verifiable. Monitor the unverifiable."

TINMAN applies the above principle. Resource bound is analyzed and verified off-line from user-supplied information and security policy, to the extent that it is practical to do so, and security enforcement is performed by coupling language-level and run-time system mechanisms to monitor what cannot be established by off-line analysis. The off-line and run-time mechanisms together provide complete coverage and guarantee resource bound security. TINMAN advances the state of the art by pushing the limit in the automation of fine-grain resource consumption security checks, by deploying a suite of tools for resource utilization prediction, certificate generation and validation, and run-time event matching.

In this paper, we focus on the system design and implementation of the TINMAN architecture. The next section describes the methodology of system design. The architecture of TINMAN is given in Section 3, followed by some details of the off-line checker and on-line checker. Some experimental results are given in Section 7. Section 8 compares TINMAN with related work. Concluding remarks and future work are in Section 9.

2 Methodology

As noted in the previous section, there are two aspects to the resource security problem: policy and enforcement. *Resource security policy* establishes resource usage constraints that mobile programs must satisfy. In TINMAN, the security policy to be satisfied by mobile program is given by a specification in three parts: (1) resource usage for each service that may be called by a program and provided by a hosting system; (2) resource limit for each program; (3) a proof system consisting of axioms and inference rules for the interpretation of the specification.

Resource security enforcement prevents a mobile program from violating the resource security policy. It pertains to the authorization of resource usage and the limitation of actual resource usage by a program. Enforcement may be performed dynamically, by monitoring the real-time resource utilization of a mobile program at run time. Complete reliance on dynamic enforcement may be too expensive in general as to be practical. An alternative way to perform enforcement is to check the certification of resource bounds for a certified program, similar to Proof Carrying Code (PCC) [6] and Typed Assembly Language (TAL) [7].

Ideally, if certification is trustworthy, the code is considered resource usage safe. The system may then grant resources to the program to complete its execution, and there is no run-time checking (as in PCC). This is in general not possible for resource bound security since exact resource usage is usually determined dynamically and cannot in general be derived by compile-time static analysis. Our approach is to a combine off-line verification and on-line monitoring.

TINMAN deploys an off-line analysis tool that attempts to derive tight resource usage bounds for user-supplied codes that may be loaded and run on hosting systems. In an ideal world, the off-line analyzer would be able to derive an exact resource usage bound and output a correctness proof of the bound which constitutes the *usage certificate*. Short of getting an exact bound, a tight bound may be obtained interactively with the programmer giving help in the form of assertions about the program's behavior.

An on-line analysis tool resides on the code recipient where the code is run. We note that if verification is possible at compile-time, all the on-line analyzer needs to do is to check the usage certificate, and dispense with monitoring. Since checking a proof or validating a usage certificate via, say, a trusted certification authority is in general much easier than deriving the proof from scratch (just as it may take exponential time to solve an NP-complete problem but only polynomial time to check the correctness of a solution), the run-time overhead is relatively small. In the case not all verification conditions can be verified at compile-time or if programmer input is required, the off-line analyzer also outputs all the assertions, and the on-line checker will automatically monitor the validity of the assertions at run time. In this way, a network node determines with certainty that a piece of mobile code is compliant with the resource security policy.

3 System Architecture of TINMAN

Figure 1 shows the TINMAN architecture and the dependencies among the system's components. We now give a high-level description of each component and explain how they fit together to perform resource security check for mobile code.

3.1 Off-Line Checker

The goals of the off-line checker are to provide programmers with a tool for resource bound prediction and usage certificate generation for their mobile program. As shown in Figure 2, resource usage prediction involves *timing analysis*

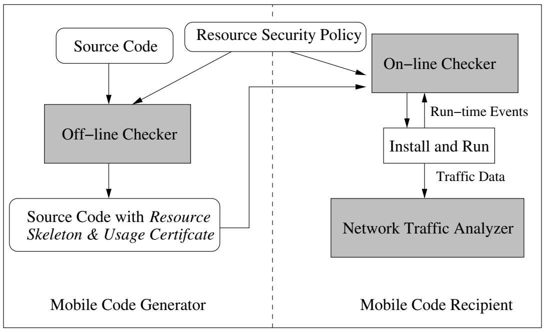


Fig. 1. TINMAN Architecture

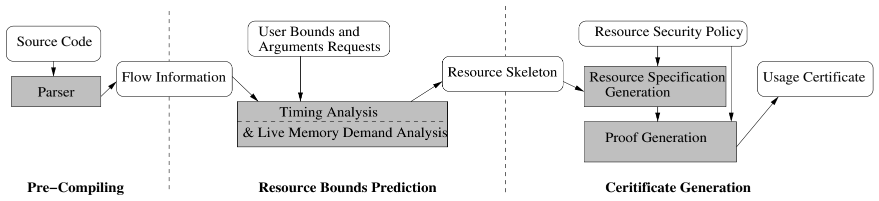


Fig. 2. Overview of Static Resource Bound Prediction and Verification

and *live-memory demand analysis*. The output is a set of annotations and assertions we call a *resource skeleton*. The resource skeleton is parameterized such that the exact values of the bounds can be determined at the code recipient site. A *resource specification generation* component automatically translates the resource skeleton into a specification consisting of a group of predicates that will be verified by a *proof generation* module to produce a usage certificate.

Once mobile code is analyzed by the off-line checker, it is ready to be sent to the recipient site.

3.2 On-Line Checker

The on-line checker validates the annotations inserted by the off-line checker, and detects any violation against the security policy by the imported mobile code before or during its execution. A typical on-line checker session involves resource usage bound calculation, resource skeleton verification, usage certificate validation and run-time assertion monitoring. The on-line checker will be discussed further in Section 6.

3.3 Network Traffic Analyzer

Unlike the CPU and memory, network resource abuse in general may affect entities outside of a host executing the mobile code and as such cannot be readily detected by checking resource usage limits on the system where the mobile programs run. It is difficult to statically check exact network bandwidth usage at the language level because of the dependency on protocol implementation. TINMAN instead prevents such attacks at the sender by performing run-time network traffic analysis in addition to static checks on network-related service routines. Run-time network resource checking is not the subject of focus in this paper, and interested readers are referred to [8] for more details.

4 Resource Bound Prediction

Resource bound prediction is done at the source code level. The off-line checker modifies the Broadway Compiler [9], a source-to-source translator for ANSI C, to parse the source code and output flow information to resource analysis modules. To strengthen the resource prediction results, the off-line checker performs type-checking to assure type safety. We make the following assumptions about the mobile code discussed throughout this paper: No cast, pointer arithmetic, address operation and recursive calls are used. All variables must be declared with types. Variables including structure fields must be initialized before being used. In addition, all array subscripting operations are checked for bound violation. The type-checking is done statically. A failed type-check will result in the termination of the off-line checker in which case no resource skeleton and usage certificate will be generated. We shall formalize the semantics of this subset of ANSI C in section 5.

4.1 Timing Analysis

Timing analysis attempts to determine the worst-case execution time (WCET) of a program. Prediction of WCETs is an extensive research area, and substantial progress has been made over the last decade. Most practical WCET techniques, however, cannot be directly applied here since they are tied to particular target architectures and do not take mobile code into consideration. The off-line checker addresses this issue by combining a source level timing schema approach with a policy-based, parameterized WCET computation method.

The basic idea of the timing schema approach [10] is that the execution time is determined by basic blocks and control structures of a program. For example, the execution time of the assignment statement $A : a \ = b - c$; can be computed as $T(A) = T(b) + T(-) + T(c) + T(a) + T(=)$, where T(X) denotes the execution time of the item X, referred to as an *atomic block*, *i.e.*, basic component in the abstract syntax tree of a program. The execution time for the control statements is computed similarly,

T($if(exp0)\ S1; else\ S2$) = T(exp0) + T(if) + max(T(S1), T(S2)).

T($for(i = 0; i < N; i++)\ stmt;$) = T(i) + T(=) + (N + 1) · (T(i) + T(<) + N · (T(stmt) + T(for) + T(i) + T(++)).

assuming i is not changed in the loop's *stmt*. The timing of compound statement is calculated recursively based on the simple statements.

We note that the above approach fails when a program contains loops whose iteration bounds cannot be directly obtained, or when system service calls are invoked; the latter is common for a mobile application. The off-line checker uses a number of techniques to alleviate these problems.

Loops

For loops, a pre-compiling analysis of the source code discovers constant loop bounds or handles dependencies between loop iteration variables of the nested loops. For example, consider the loop

$while(m < n)\ \{S1; n = n + I0; S2\}$

If both m and n are constants at the entry point, the increment/decrement of n

(i.e. $I0$) is a constant and dominates the next loop iteration, and the number of loop iterations can be statically determined. Therefore, the accurate loop bound can be calculated using constant propagation and the dominance relationship[1] analysis. For a nested loop where the number of iterations in the inner loop depends on the value of the index in the outer loop, we rely on techniques similar to [12] to give a tight prediction on loop iterations.

For loops where loop bounds are difficult to deduce automatically (or even do not exist as in an infinite loop) but are straightforward for a programmer to deduce, our tool will ask the programmer to input the asserted loop bound which will be monitored.

System Service Calls

The execution time of system service calls in a program is not known without information of the remote system where the program runs. Our approach is to specify the resource usage of a service call in a policy rule in the form of pre and post conditions defined in a formal logic (to be described in detail in section 5). The resource consumed by a service is parameterized given the ranges of its arguments. Exact values are determined at the remote site. This approach is flexible since the policy is configurable for a specific platform and run-time environment. As a result, only parameter information and the system call itself are need in the resource skeleton by the off-line checker.

4.2 Live Memory Demand Analysis

Accurate memory allocation prediction is a non-trivial task, and usually requires considerable cost [13]. The memory used by a mobile code consists of three spaces: stack space, dynamically-loaded code, and heap space. Stack space consumption is largely due to recursive calls which are currently prohibited in the current prototype of TINMAN. On the other hand, the size of a piece of mobile code is generally small and it is relatively trivial to compute the heap memory size of the mobile code. The heap space, however, is exploited by dangling pointers and may be allocated by malicious programmers. TINMAN analyzes the heap space allocation, referred to as the *live memory demand* at the language level. Considering the possible actions of malicious codes on memory, our efforts are focused on memory allocation requests in a program.

Consider for example, the memory demand of an assignment statement S:

$$head = (Node*)malloc(sizeof(Node) * exp);$$

is M(S) = sizeof(Node) * Value(exp). The sizeof(Node) can be determined at the entry point of S, while exp may not. In addition, the memory demand computation may be complicated if the $malloc()$ statement is in a loop, and/or the paired free statement is not properly given. Taking all these into account, we adopt a general approach for live memory demand prediction in the following steps: (1) Analysis of memory allocation statements: All memory allocation statements

[1] A *dominates* B iff all paths from the start node to B intersects A [11].

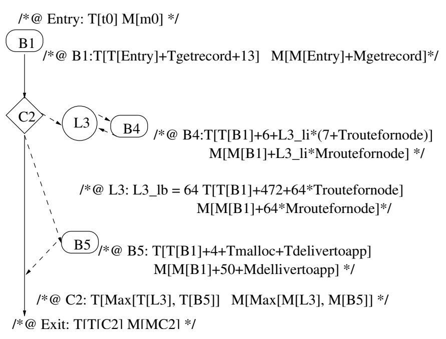
are identified such as *malloc*(), *calloc*(), *memalign*(). The size (in bytes) of maximum memory request is either calculated automatically using flow analysis or provided by the programmer if it cannot be statically bounded, for example, the value of *exp* in *S* depends on program input data. (2) Path analysis of memory freed: The live memory demand may be overestimated if we simply sum all memory allocation requests. We check the dominance relationship between pairs of memory allocation and free statements, and within the same basic blocks or compound statements to tighten the live memory bound (3) Call chain analysis: The demanded memory size is increased throughout the call chain. Therefore, we compute recursively the memory demand at the points of interest, such as loops and procedure calls.

4.3 Resource Skeleton

The timing and memory bound information obtained by either automatic analysis or from programmer input need to be maintained for further use. We use a *resource skeleton* to annotate the information to help bound the resources. Basically, a resource skeleton can be viewed as an abstraction of a program in regard to its resource consumption. The resource skeleton of a program also makes it possible for the on-line checker which is invoked by the code recipient to detect violations against the resource security policy.

The Resource skeleton is created along with the construction of the flow graph of a program derived from its syntax tree. A node in a flow graph is called a *task*. Task types include basic blocks, conditional statements, loops, user-defined procedure calls and system service calls. The tasks of basic blocks and service calls are *primitive tasks*. The flow graph is hierarchical which means a program is a sequence of tasks, and each task (except primitive tasks) can be expended into a flow graph. To reduce unnecessary annotations and proof generation based on these annotations, a sequence of primitive tasks are combined in the same resource skeleton. We illustrate the construction of resource skeleton with an example, shown in Figure 3.

```
B1:  /* look up forwarding record */
     n = getrecord(group, sender);
     m = &n;
     m–>time = n.time – 10;
C2:  /* must find it to continue */
     if (m–>nodes != NULL){
L3:    /* send a copy every way */
       for (i = 0; i < m–>length; i++)
B4:        routefornode(&n, m–>nodes[i]);
     }
     else {
B5:      m–>node = (Node*)malloc(sizeof(Node)*10);
       /* or deliver to application */
         delivertoapp(&n, dpt);
     }
```



```
/*@ Entry: T[t0] M[m0] */
/*@ B1:T[T[Entry]+Tgetrecord+13]   M[M[Entry]+Mgetrecord]*/
/*@ B4:T[T[B1]+6+L3_li*(7+Troutefornode)]
        M[M[B1]+L3_li*Mroutefornode] */
/*@ L3: L3_lb = 64 T[T[B1]+472+64*Troutefornode]
        M[M[B1]+64*Mroutefornode]*/
/*@ B5: T[T[B1]+4+Tmalloc+Tdelivertoapp]
        M[M[B1]+50+Mdellivertoapp] */
/*@ C2: T[Max[T[L3], T[B5]]   M[Max[M[L3], M[B5]] */
/*@ Exit: T[T[C2] M[MC2] */
```

Fig. 3. Example of Resource Skeleton

The task in the flow graph is labeled with its type (e.g. L = Loop task) followed by a global counter value. The solid line marks the execution sequence of related tasks while a broken line means a task expansion. Resource annotation for each task is enclosed in "/*@ ... */" which can be identified by the on-line checker 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 execution time of C2 is the maximum execution time of T[L3] and T[B5], where T[L3] and T[B5] can be recursively computed from the resource annotations for L3 and B5. The loop bound for L3 is provided by the programmer. We note that, for example, the CPU and memory usage bound for B1 are given only with the name of the service call ($Tgetrecord$). The actual bound value calculation is performed at the remote site with instantiated resource security policies.

5 Generating Usage Certificate

The mobile program is transported with a usage certificate since a code recipient cannot trust the resource skeleton which may be corrupted. Our basic strategy to establish the correctness of certificates is to use an assertional programming logic such as Hoare Logic. Our approach is to translate the resource skeleton into a resource safety specification in a formal logic, and generate a certificate by checking for compliance with security policy. In this section, we first introduce the resource specification, using extended Hoare triples. Next, we present a proof system by formulating the formal semantics of a subset of the C language (mentioned in section 4) in terms of a set of axioms and inference rules. The program correctness on satisfying a specification can be proved within the proof system.

5.1 Resource Specification

For a task T, a *resource specification* for T is an extended Hoare triple {P} T {Q}, where assertion P is the precondition, and assertion Q is the postcondition which holds if P is true and T terminates. An assertion is a first-order logic formula which is typically a conjunction of predicates over program variables, time and memory usage and terminational judgment. In order to translate a resource skeleton annotation for task T into an assertion, we first define the type $Time$ to be the nonnegative reals and $Memory$ to be the nonnegative integers as respectively domains of time and memory size values. Three special variables: *now* of type $Time$, *mem* of type $Memory$, and *terminate* of type boolean are introduced. These variables denote the moment, the allocated memory and the termination before, if they are in P, or after, if in Q, the execution of T, respectively. The initial value of *now* is *t0*, and *mem m0*. With this terminology, the resource skeleton can be represented as logical specifications. For example, the specification for task B1 in Figure 3 is

$\{\mathbf{now} = t0 \wedge \mathbf{mem} = m0 \wedge \mathbf{terminate}\}$
B1
$\{\mathbf{now} <= t0 + Tgetrecord + 13 \wedge$
$\mathbf{mem} <= m0 + Mgetrecord \wedge \mathbf{terminate}\}$
And the specification for loop task L3 is as follows. Note that *L3_lb* is the user-provided loop bound.
$\{L3\_lb = 64 \wedge \mathbf{now} <= t0 + Tgetrecord + 17 \wedge \mathbf{mem} <= m0 + Mgetrecord \wedge \mathbf{terminate}\}$
L3
$\{\mathbf{now} <= t0 + Tgetrecord + 472 + 64 \cdot Troutefornode \wedge$
$\mathbf{mem} <= m0 + Mgetrecord + 64 \cdot Mroutefornode \wedge \mathbf{terminate}\}$
It should be pointed out that the specification for a service call contains a precondition, if any, over the ranges of its arguments and expressions of its running time and memory usage. It is published as part of the security policy.

5.2 Proof System

Similar to Jozef Hooman's framework [14] for sequential programs, we construct a proof system for resource specification by formalizing programming constructs, or tasks in our cases. The tasks are axiomatized by inference rules and axioms. The rules in the framework are proved independently and published as part of security resource policies as well.

Clearly, the formalization of resource specification and the proof system requires mechanical support. TINMAN uses the Prototype Verification System (PVS) [15] to implement its logical framework. In order to formulate resource specification into the PVS specification language, our approach is to identify programs 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 construction rules defined in [14]. All tasks are defined with regard to their resource specifications. For example, the task B1 in the previous example is defined in PVS as follows,

```
P0 : [State->bool] =
     ((LAMBDA s : state) : now(s) = t0
     AND mem(s) = m0 AND terminate(s) )
srvc1: program = SRVC(Tgetrecord, Mgetrecord)
bb1  : program = BB(13);
B1   : program = seq(srvc1, bb1);
Q0 : [State->bool] = (LAMBDA s :
     now(s) = t0 + Tgetrecord + 13 AND
     mem(s) = m0 + Mgetrecord AND
     terminate(s))
{P0}B1{Q0} : THEOREM
  ((FORALL s0, s1 : state):
     P0(s0) AND B1(s0, s1) IMPLIES Q0(s1))
```

The complete set of axioms and inferences rules and program constructs definition can be found in [8].

5.3 Proof Generation

A PVS specification for resource skeleton is obtained by applying the program construction rules. To prove the specifications, one would like to construct proofs interactively using the PVS prover system. For a mobile program, however, we aim at generating a proof or certificate as automatically as possible. PVS provides a mechanism for automatic theorem proving by composing proof steps into proof strategies. We note that the specification for a different type of program constructs may require different proof strategies.

For primitive constructs like basic block tasks and service call tasks, we have defined a strategy that performs a sequence of built-in proof steps. In this strategy, the theory definition including the task definitions and assertions is expended by automatic rewrite rules, and then Skolemization and Decision procedure are invoked repeatedly until the theory is proved. For example, the theorem {P0}B1{Q0} in the previous example is proved using this strategy by first auto-rewriting P0, Q0, B1, seq, srvc1, and bb1, and then repeatedly invoking prover commands `ASSERT` and `SKOSIMP*` until the theorem is proved.

The proof strategy for proving a choice task specification, say $\{P\}\textbf{if}\ b\ \textbf{then}\ T1$ $\textbf{else}\ T2\ \{Q\}$, is more complicated than the primitive tasks. The **Rule 2** for choice tasks in the proof system illustrates the general steps of the strategy. Briefly, we need first to prove the two corollaries, $\{P \wedge b\}\ COND(tb, mb);\ T1\ \{Q\}$, and $\{P \wedge \neg b\}\ COND(tb, mb);\ T2\ \{Q\}$ by applying some other strategies and then invoke the prover command `LEMMA Rule 2` and the quantifier rule. The construction of those strategies and other proof steps in the strategy for choice tasks are non-trivial, and will not be discussed any further here.

The strategy for sequential tasks and loop tasks are constructed similarly in consideration of the corresponding rules in proof system. The strategy for loop task, however, involves a loop invariant and an assertion that holds if the loop terminates. In order to generate them automatically, we simplify a loop invariant by introducing an auxiliary loop index, say li for a loop L, where $li \in [0, loopbound_L]$ and is increased by 1 in each loop iteration. The loop invariant is constructed with li and the specification (precondition and postcondition) of task L, since we only need to assure the correctness of the resource bound, not what the program actually does.

Using these proof strategies, the resource specification for an annotated program is proved automatically. PVS outputs the proof onto a text file which constitutes the usage certificate for the program. However, due to the large size of the proofs written in the built-in PVS specification logic, we further shorten the detailed usage certificate by only keeping the strategies and related parameters required to produce the proof. We refer the shortened usage certificate as a *certificate skeleton*. Finally, the annotated mobile program only requires the certificate skeleton to be transferred to the remote site.

6 On-Line Checker

The on-line checker performs limited static verification by validating the supplied resource skeleton and usage certificate, and it detects any violation against security policy on resource utilization limits.

Figure 4 shows the major steps of static on-line verification. We note that the calculation of actual resource bound, given instantiated policies on service calls is performed before the validation of the resource skeleton. It enables detection of any violation against a resource usage limit at an early stage since if there is a violation, the mobile program will not be trusted, and no further verification is warranted.

The purpose of the resource skeleton validation is to check for consistency between the source code and the resource skeleton. Annotations are inserted at the appropriate points. Specifically, the tasks, programmer-provided information if necessary for loop bound, memory allocations and service call arguments are annotated, and there are no further manually inserted annotations.

The resource specification generation is the same as that in the off-line checker. It outputs a specification consisting of predicates on the resource skeleton. The full usage certificate is restored from the certificate skeleton. The proof checker verifies the supplied usage certificate to conform with the specifications within the PVS system. In our implementation, the proof checking is as simple as a validation run of PVS in a *batch mode* which automatically reruns all proofs in the usage certificate. An invalid usage certificate will generate errors which can be caught by the proof checker by examining the run log file.

After the verification of the resource skeleton and validation of usage certificate, the only untrusted part are the user-provided assertions on loop bounds and the ranges of function arguments. The on-line checker needs to monitor these values at run-time. In order to do this, the on-line checker translates them into related assertions for checking the range of the values of untrusted data. A run-time exception will be raised if any violation of policy is detected. Dynamic resource utilization monitoring is a tried concept used by much previous

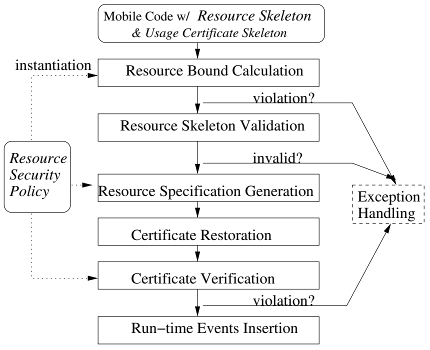

Mobile Code w/ *Resource Skeleton*
& Usage Certificate Skeleton
instantiation
Resource Bound Calculation
violation?
Resource Skeleton Validation
invalid?
Resource Security Policy
Resource Specification Generation
Exception Handling
Certificate Restoration
Certificate Verification
violation?
Run−time Events Insertion

Fig. 4. Structure of Static On-line Verification

work. Our approach avoids much of the dynamic resource utilization monitoring since the resource bound safety is guaranteed by static verification, and run-time checking is required only on a few programmer-provided annotations.

7 Experimental Results

In this section, we present some experimental results obtained by using our tools. We have translated the following two mobile programs used by other research groups [1, 4] into C. The multicast_subscribe installs forwarding pointers in each router (member of a group) it traverses so it can receive messages sent to the group. It has six services calls, one loop and six tasks. The multicast_data simply routes itself along a distribution tree, and it has five services calls, two loops and eight tasks.

We first measured the code size augments due to insertion of annotations and the usage certificate. The results are shown in Table 1. We note that the increased size depends on the number of tasks in a program, but is not directly related to the size of the program. This is because, for example, multicast_data has more complicated control structure and more tasks. It also explains the reason that multicast_data has a larger usage certificate. We also observed that the certificate size is significantly decreased by up to 94.3% by using a certificate skeleton.

Table 2 shows the cost of off-line certificate generation and on-line annotation verification and certificate checking. The certificates of both example programs are generated completely automatically. However, off-line certificate validation and on-line certificate check result in an order of magnitude slower than cer-

Table 1. Code Size with Annotations (in bytes)

Program	multicast _subscribe	multicast _data
Original Size	1508	1113
Resource Skeleton	316	462
Certificate Skeleton	569	1087
Full Certificate	9963	14595
Increase(%)	58.7	139.2

Table 2. Cost of off-line checker and on-line checker

Program	multicast _subscribe	multicast _data
Specification Generation	36.5ms	58.5ms
Off-line Cert. Construction + Validation	0.83s + 11.06s	1.45s + 23.85s
On-line Anno. Verification	62.9ms	51.1ms
On-line Cert. Check	11.09s	22.72s

Table 3. Resource Utilization Measurement

Program	Predicted WCET	Observed WCET (w/o ann.)	Observed WCET (w/ ann.)	Pess.	Predicted Mem(B)	Actual Mem(B)	Pess.
multicast_subscribe	74.08ms	57.56ms	81.73ms	29.8%	932	804	15.9%
multicast_data	1456ms	1363ms	1503ms	6.8%	12800	8192	56.3%

tificate construction. The overhead is two-fold. First, a certificate consists of PVS rules that are interpreted by the PVS prover interactively. Second, current proof strategy for loops and compound choice tasks involves catch-all prover commands like `GRIND` which is timing consuming. Table 2 also indicates that annotation verification time (e.g resource usage calculation and annotation check) is quite small and negligible compared with certificate check time.

Table 3 shows the timing and memory analysis results of the programs. The loop bounds of multicast_data are automatically calculated while the bound of multicast_subscribe are given by a programmer. The pessimism of multicast_data is only 6.8% because we have obtained a tight loop bound. On the contrary, the actual loop counts in multicast_subscribe depend on the conditions of inner break statements. Therefore, it is very conservative and is off by 29.8%. 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.

The pessimism of memory analysis is caused by the actual execution path that affects memory allocation statements. For example, in multicast_data, the *malloc*() inside a loop is only executed upon satisfaction of some condition, and its total execution time is also decided by the loop bound. The experiments show, however, the actual memory allocated does not exceed the predicted live memory demand that is essential to our goal of resource security.

8 Related Work

Previous resource usage safety efforts for mobile code usually enforce the security policy in the run-time system to limit the resource utilization of mobile code. Smart Packets [3] checks the CPU and memory usage of active packets written in Sprocket and enforces limits on the number of instructions executed, amount of memory used, and access to MIB variables. The KeyNote in PLANet has a similar mechanism [16]. These active network systems do not provide tools to do source-code-level checking concerning resource usage, and have significant restrictions on languages features and thus limit the expressiveness of mobile code. Some researchers have developed extensions for more expressive security policies. For instance, Naccio [17] specifies security policies for Java and Win32 using a specification language. Java programs are transformed to call wrapper functions instead of the original library code in order to enforce safety policies.

In PCC, a code producer creates a formal safety proof to prove adherence to the safety rules that guarantee safe behavior of programs. A remote host depends on proof validation techniques to check that the proof is valid so that the foreign code is safe to execute. The difficulty of generating proofs for large programs and more interesting policies are the difficulties in the application of PCC. 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. Unlike PCC, TINMAN concentrates on resource security assurance in high-level to prevent DoS-like attacks and buggy or malicious codes with infinite loops or improper arguments to services calls that are not addressed by previous work. The resource security policy is flexible and configurable at 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.

9 Conclusions

In this paper we have presented TINMAN, a resource bound security checking system for mobile code. The system detects malicious mobile source code that, once installed and executed, may consume inordinate amounts of resources such as CPU, memory and network bandwidth, as is common in DoS (Denial-Of-Service) attacks.

TINMAN provides multiple levels of protection on resource security at both compile time and run time, at both the source-code level as well as run-time system level. It has been implemented by a set of tools that support resource bound prediction and certificate generation and validation. TINMAN exploits programmer input but does not depend on it for ensuring resource security; incorrect programmer input about resource bounds will be checked and detected against the resource skeleton associated with the mobile code, and the usage certificate is validated against given resource security policies. An on-line checker tool is used to detect malicious modification of resource bound annotations and certificate validation. This enables any violation of resource utilization with respect to security policy to be detected as early as possible before the execution of the mobile code. Together, the off-line and on-line checkers provide complete coverage, following the guideline of proving what can be verified and monitoring what cannot be verified.

In this paper, we consider mobile programs written in C with active network benchmarks. However, our framework is extensible and applicable, in an even simpler style, for other programming languages for mobile applications such as Javascript of which source codes are usually embedded in a HTML file. By gaining experience in TINMAN, we shall hopefully be able to customize the framework for some version of byte-code for C. Our plan is to use the script from the verifier session as the usage certificate so that the proof can be checked on-line efficiently.

References

[1] Wetherall, D., Guttag, J., Tennenhouse, D.: ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. IEEE OPENARCH (1998) 117–129 178, 190

[2] Hicks, M. W., Kakkar, P., Moore, J. T., Gunter, C. A., Nettles, S.: PLAN: A Packet Language for Active Networks. International Conference on Functional Programming (1998) 86–93 178

[3] Schwartz, B., Jackson, A. W., Strayer, W. T., Zhou W., Rockwell, D., Partridge, C.: Smart Packets: Applying Active Networks to Network Management. ACM Transactions on Computer Systems 18:1 (2000) 67–88 178, 179, 191

[4] Kornblum, J., Raz, D., Shavitt, Y.: The Active Process Interaction with its Environment. Lucent Technology (1999) 190

[5] Fritzinger, J. S., Mueller, M.: Java Security. Sun Microsystems white paper (1996) 179

[6] Necula, G. C.: Proof-Carrying Code. POPL'97 (1997) 106–119 180

[7] Morrisett, G., Walker, D., Crary, K., Glew, N.: From System F to Typed Assembly Language. IEEE Symposium on Principles of Programming Languages (1998) 180

[8] TINMAN Project. `http://www.cs.utexas.edu/wjyu/tinman` 182, 187

[9] Guyer, S. Z., Jiménez, D. A., Lin, C.: Using C-Breeze. Department of Computer Sciences, The University of Texas, (2002). `http://www.cs.utexas.edu/users/lin/cbz` 182

[10] Park, C. Y., Shaw, A. C.: Experiments with a program timing tool based on source-level timing schema. Computer J 25:5 (1991) 48–57 183

[11] Aho, A. V., Sethi, R., Ullman, J. D.: Compilers: Principles, Techniques, and Tools. Addison Wesley (1986) 184

[12] Healy, C., Sjdin, M., Rustagi, V., Whalley, D.: Bounding Loop Iterations for Timing Analysis. IEEE Real-Time Applications Symposium (RTAS'98) (1998) 12–21 184

[13] Unnikrishnan, L., Stoller, S. D., Liu, Y. A.: Automatic accurate stack space and heap space analysis for high-level languages. Computer Science Department, Indiana University TR 538 (2000) 184

[14] Hooman, J.: Correctness of Real Time Systems by Construction. FTRTFTS: Formal Techniques in Real-Time and Fault-Tolerant Systems LNCS 863, Springer-Verlag, (1994) 19–40 187

[15] Owre, S., Rushby, J., Shankar, N.: PVS: A prototype verification system. 11th International Conference on Automated Deduction, Lecture Notes in Artificial Intelligence, Springer Verlag (1992) 748–752 187

[16] Alexander, D. S., Anagnostakis, K. G., Arbaugh, W. A., Keromytis, A. D., Smith, J. M.: The Price of Safety in an Active Network. University of Pennsylvania MS-CIS-99-02 (1999) 191

[17] Evans, D., Twyman, A.: Flexible Policy-directed Code Safety. The IEEE Symposium on Research in Security and Privacy, Research in Security and Privacy IEEE Computer Society Press (1999) 32–45 191