RULE-BASED MONITORS AND POLICY INVARIANTS FOR GUARANTEEING MOBILE CODE SECURITY

Sanna Mäkelä, Sami Mäkelä, Ville Leppänen

University of Turku, Department of Information Technology, Finland
e-mails: samitu@utu.fi, sajuma@utu.fi, ville.leppanen@utu.fi

Abstract: We consider ensuring the security of executed mobile code by applying runtime monitoring. Of the many approaches for code security, the runtime monitoring approach is perhaps the most general and flexible. We have formerly implemented a rule-based language for describing runtime security policies, and now we discuss the verification of those policies. A security policy can be considered as a specification that restricts the execution of a program in some way. These restrictions can be connected to the program state and the execution history. In this paper, we introduce invariant expressions for our security monitor descriptions, and describe a methodology for proving that the monitor preserves its invariant. Our invariant expressions describe the true meaning of security monitor and relate the monitor state to the execution history and current state of the monitored program. The advantage of our approach is that we can prove *specific* monitors to guarantee *all* monitored programs to preserve such properties that cannot in general be effectively proved or disproved of all possible executions of any program.

Key words: software security, runtime monitoring, policy invariants

1. INTRODUCTION

In this paper, we discuss the verification of a code runtime monitoring system, and present invariants for describing the meaning of policies and the corresponding monitors in a rule-based language. Monitoring code is one approach for verifying code security. This approach is particularly applicable for mobile code, because the access to the source code is unnecessary.

We consider mobile code as any program code that is downloaded for execution from some (untrusted) source. The execution platform can be mobile phone but here we do not consider any specific platform. In the literature (see e.g. [12, 2, 7]), there appears to be three kinds of approaches for mobile code security: (a) providing a proof of security properties along with the code, (b) establishing an authority for

certifying the safety of mobile code, and (c) running mobile code under some runtime monitoring system in the target platform.

Proof-carrying code (PCC) [12] and Abstraction-carrying code [2, 7] follow the approach (a). PCC can work well in certain limited settings, but in general there are problems with PCC, since it is impossible to prove many kind of relevant runtime properties concerning any program. This is due to so-called Rice's theorem [15], related to computability of partial functions, stating that there exists no effective method for calculating any non-trivial property. This has strong consequences when applied to static analysis of pro- grams, e.g. to proving properties concerning programs. For example, it is not possible to effectively either prove or disprove for all programs using functions A() and B() that in any execution of the program there exists one call of B() between each two calls of A(). Naturally, it is possible to analytically prove/disprove that for some programs but not for all programs. It is not possible to effectively try all possible executions of any given program. Although the PCC approach suffers from consequences of Rice's theorem, it is not clear how severe these consequences are in practice.

In the approach (b), the downloaded mobile code comes as cryptographically signed by some trusted authority. The properties are thus guaranteed by the signing authority. The mobile phone industry appears to have chosen approach (b) for providing trusted mobile code, e.g. the Symbian operating system (of smartphones, since version 9.0) [6, 10]. However, even a signed code can include errors and weaknesses, since the code quality is at least partially based on human inspections. Moreover, signing is mainly a guarantee of code quality, and quality is not really related to what a consumer might consider allowed or disallowed. For example, when a person borrows her/his mobile phone to another person, one might temporarily wish to deny applications to use certain resources completely or partially (i.e. applications should be run under some temporary security policy).

When a code runtime monitoring system (c) is used, the code behaviour is controlled by a (security) monitor that is described by some (security) policy. A security policy can be considered as a specification that restricts the execution of a program in some way. These restrictions can be connected to the program state and the execution history. Methods to express security policies that can be checked during execution are previously studied extensively. We make a short overview of such research in the following section.

A common way for describing and checking security policies is to utilize languages based on the idea of using an automaton for comparing steps of program executions to the current security policy. In a policy violation, an operation can, for instance, be rejected, or the execution can be terminated.

We follow the approach (c). We have developed a monitoring language called MPL (Modular Policy Language) [11]. MPL descriptions express simple automata based monitors, where security sensitive calls captured. Because our language is

intended for real-world applications, our monitors can maintain and take conditional actions based on memory values, which represent (attribute, value) pairs describing the state of the MPL monitor. As a result, we use the term rule instead of state transition in our language.

In this paper, we focus on the use of invariants for expressing the meaning of runtime monitors. Since we have not discovered other efforts to apply invariants in this context, we suppose being first to operate with them.

We present policy invariant and monitor invariant for rule-based languages, and demonstrate the use of these invariants by verifying a sample security policy. We aim to show that it is relatively easy to write invariants for proving rule-base monitors. While policy invariants specify policies, monitor invariants specify the implementation of policies. We define monitor invariant as a formally specified predicate that is true before and after applying any policy rule, and sufficiently describes the duty of a policy for being able to expose all possible policy violations. We aim to show that rule-based policies are not only easy to understand but also provable. Proving is done by showing that all rules sustain the invariants. We make a suggestion that the policy verification could be at least semi-automatic in well-specified circumstances.

After reviewing related work in Section 2, we discuss invariant based security policy verification in Section 3, and an example of using this methodology is given in Section 4. The example is related to the usage of sockets (in mobile phone context it can be related to WLAN usage). Some conclusions are drawn in Section 5.

2. RELATED RESEARCH

The standard implementation of Java contains a security manager, which monitors the executed programs. Java's solution is static in the sense that the security manager's functionality is statically embedded into a large method set of certain library classes meaning that only the behaviour of those classes/methods can be controlled by security manager descriptions. Recently, extending the functionality of Java's security manager in certain settings is considered in [16, 17]. Interestingly, the extension also deals with execution history based access control but it does not consider invariants. The general possibilities and restrictions of runtime monitoring in general are studied by Schneider in [13]. Sekar et al. [14] have developed Model Carrying Code (MCC) and studied such automata based descriptions applied e.g. to system calls in Unix.

We have previously described a modular policy language and a compiler for it in [8, 11]. Our language enables describing rule-based security monitors, and the descriptions are translated to AspectJ. Previously aspect-based security monitor descriptions are studied and developed e.g. in the form of Polymer language [4], in

the tracematch system [3] and Monitoring-Oriented Programming (MOP) [5]. Of these systems, we choose to discuss Polymer and Tracematch, because they have similar characters that we use in our system. Polymer is shortly described in Section 2.1 and Tracematch in Section 2.2. Besides the mentioned runtime monitoring studied, there exist lots of other related studies.

2.1. Polymer

Polymer is described as "a language and system for enforcing centralized security policies on untrusted Java applications" [4]. Polymer follows the idea to separate the security policy from the main application. The separation is implemented using aspects. The idea of Polymer is to ensure the security of a code by modifying its behaviour at runtime. The modifying operations are based on Edit Automata [9], and consist of sequence truncation, insertion of new actions, and suppression of actions.

Specifying a security policy (i.e. a program monitor) in Polymer [4] requires (1) the decision procedure how to react on security-sensitive operations (2) the security state that can be used to sustain the information of the activity of the application during execution, and (3) methods for updating the security state of the policy. The decision procedure of Polymer returns one of a number of security suggestions (e.g. raise an exception). The security state of Edit Automata represents the state of policy automata. In Polymer policies, there can be found few parameters describing the state of a procedure such as cancelling an action, and other parameters describing other data values such as file names. These parameters can be considered to construct the security state of Edit Automata.

Polymer has a formal semantics that mostly concentrates on guaranteeing type safety. When a program is typed in the right way, its execution succeeds. The Polymer semantics is not useful for verifying policies, since it is focused on the features of the language. However, the semantics proves that a policy implementation follows the given principles.

2.2. Tracematch

Aspects are actually only extra code around the observed code. They are confined to the current action, and, therefore, cannot be used directly to observe the history of computation. Tracematches [3] are history-based language features that make it possible to trigger the execution of the observing code by specifying a regular pattern of events in a computation trace. A tracematch defines a pattern and a code block to be run when the current trace matches that pattern. The pattern language consists of regular expressions over events. These expressions can contain free variables.

Tracematch has a declarative semantics that can be used also for defining policies. The semantics leads to a declarative implementation of monitors. The

correctness of monitor implementations can be proven using operational semantics. If a policy can be defined using regular expressions, Tracematch can be used to monitor the policy. Other kinds of policies cannot be defined using Tracematch.

3. SECURITY VERIFICATION IN MODULAR POLICY LANGUAGE

When desired program behaviour is expressed as an invariant concerning the remembered variable values, it is essentially straightforward to verify the policy and its implementation. For discussing security verification in our rule-based language, we first present types and functions for demonstrating program execution process in Section 3.1. In Section 3.2, we shortly introduce policy invariant for specifying a security policy. Rule-based monitors for implementing policies are discussed in Section 3.3. A monitor invariant specifies the implementation of a policy (i.e. monitor). It associates the history data related to the program execution to the values of monitor variables. Monitor invariant is presented in Section 3.4. Finally, we study proving of policies to preserve their invariants in Section 3.5. The aim of our presentation is to demonstrate that invariants can rather easily be proved to guarantee the secure execution of a code in the context of such rule-based policies as MPL policies.

3.1. Program Execution

To describe our ideas, we do not have to describe all details of program execution. It is enough to consider *security related operations*. These operations are assumed to have pre- and post-conditions that describe their behaviour. The pre- and post-conditions are defined using *security related program state* or security related program history. Other operations are assumed to have no effect to security related program state.

Before discussing security policies and our monitoring system, we present the applicable types and functions needed for demonstrating the execution process in an appropriate way. Let us first examine types presented Figure 1. Here, we consider that *program state* represents the current data in the memory of the program under execution, and the next call to be executed. A *call* includes the object, for which the method call is targeted, and the called method with its arguments because of object-oriented programming. While a *call event* consist of the next call to be executed and the program state before an execution step, the *return event* consists of the executed call, return value, and program execution state after the step.

$$call = method \times target\ object \times arguments$$
$$call\ event = program\ state \times call$$
$$return\ event = program\ state \times call \times return\ value$$

Figure 1. Types

In each program state, the current *call event* is received using the function *get call*. The concerned call can be the executed using the function *perform call* that takes the call event as an argument, and retrieves a return event. We can evaluate the effects of performing a call by using the functions *precondition* and *postcondition*. If the precondition is true just before a call is performed, the post-condition exposes the effects of the call. Note that if a security related operation calls other security related operations that are not separately monitored, their effect must be included in the pre- and post-condition of the operation that calls them. Lastly, in pre- and post-conditions, we apply the concept of *history*. This concept represents the monitored matters that take place in the execution environment but are not stored in the program state.

get call : *program state* → *call event*
perform call : *call event* → *return event*
precondition : *call event* × *history* → *Boolean*
postcondition : *call event* × *(initial) history state* ×
return event × *history (end) state* → *boolean*

3.2. Security Policy and Policy Invariant

When a program is executed, we may have a need to watch over its behaviour. A security policy can be considered as a specification that restricts the execution of a program in a desired way. These restrictions can be connected to the program state and the execution history, and we may want to apply different policies based on e.g. program type, authors, and users.

An invariant is a formally specified predicate that is true before and after any operation. We present *policy invariant* that can be used to specify a security policy. This function describes the meaning of a policy in terms of the state and execution history of a program (Section 3.1). It is true before and after any executed method call.

policy invariant : *program state* × *history* → *boolean*

For instance, assume that we provide an account interface to a program that was loaded from the Internet. We want to guarantee that the balance on a certain account is always greater or equal than zero. The number of the account is 111-888. Here, we can specify the policy invariant by utilizing an abstract function balance that returns the balance of the account number that is given as an argument:

policy invariant *: I(s, hs) = balance(hs, ”111 − 888”) ≥ 0*

An effort to break a policy invariant is the same as a policy violation. Halting the program in such a situation guarantees that the invariant does not become invalid.

3.3. Rule-Based Monitors

The idea of monitoring code is to guarantee that the execution complies with the policy specifications. It can be thought that a security monitor is developed to enforce a policy invariant in monitored programs. It simply captures the given security sensitive method calls (dealing with monitored system resources), and checks the validity of each captured call before applying the call. If a call is estimated to be not secure, execution can be e.g. halted or an exception thrown. An underlying idea is to describe rules for defining a security monitor corresponding to the policy. In our framework, the rules are first written by using Modular Policy Language (MPL) descriptions [8, 11], then compiled to (AspectJ) aspects, and executed within the code.

Security policies are implemented by specifying corresponding runtime monitors for guarding the usage of certain specified system resources. A policy is used to express execution restrictions for every method that deals with some monitored system resource, e.g. the usage of socket connections. A rule-based monitor is used to implement the policy. In practice, monitors define a set of variables and a set of guarded rules. The purpose of each *guarding rule* is to guard a system resource against illegal usage by the application.

A guarded rule names a *target method call* and a *condition* that is a Boolean valued expression referring to the actual parameter values of the call and variable values remembered by the program and monitor. For presenting our monitoring algorithm, we use types and functions shown in Figure 1, and additionally specify function's *condition* and *effects*. *Monitor state* type consists of the variable values of a monitor. The variables are required to maintain information about the matters that are not necessarily recorded into the state of the program but are considered in the policy. In other words, monitor state corresponds to the concept of history discussed in Sections 3.1 and 3.2.

Before executing a guarded method call, the condition of that method call is checked. If a call is estimated to be secure in the given circumstances, the condition returns true. In this case, there is no policy violation), and the rule can have some effects on the variable values of the monitor after the guarded method call is executed. Function effects updates monitor state to be analogous to the execution history. An effect is simply a conditional update operation regarding some set of variables (and their remembered values). In the other case, the condition returns false, and the execution of the whole application is prevented from continuing (as it does not respect the policy).

condition: $monitor\ state \times call\ event \rightarrow Boolean$
effects: $monitor\ state \times return\ event \rightarrow monitor\ state$

A monitoring algorithm is basically an infinite loop that receives a guarded call, checks a condition related to it, performs the call, and implements the effects on the monitor variables. The monitoring algorithm can be described as follows:

```
program state s = initial state;
monitor state ms = initial monitor state;
while true :
        call event c = get call(s);
        if not condition(ms, c) then halt;
        return event r = perform call(c);
        ms = effects(ms, r);
        s = r.state;
```

Monitor descriptions in MPL contain the criterion (i.e. the class of objects, the method) based on which the determined method calls are captured. Further, rules contain the parts *cond* and *effects* for decision making and updating information. An example of a monitor for the sample policy invariant presented in Section 3.2 is shown in Figure 2. Here, the account management takes place through Connection object, which provides only a limited number of operations for the client. For instance, the account number is not available through this class, and it must, therefore, be contained in monitor variables.

variables
accountTable: Connection → String

rules
Connection: boolean pay(double sum)
cond *this.getAccountBalance()≥sum ∨¬accountTable(this).equals("111−888")*
effects none

Account: Connection setCurrentConnection(String ac, String pwd)
cond *ac ≠ null ∧ pwd ≠ null*
effects *(this.checkPassword(ac,pwd) ∧accountTable(result) = ac)*
∨(¬this.checkPassword(ac,pwd) ∧result = null)

Figure 2: An Account Balance Monitor

Implicitly all methods not specified by rules have true as their condition and they have no effects (onto the variable values remembered by the monitor). Notice that the conditional update operation can be used e.g. to collect information related to only certain calls of the monitored method (for other calls the effect is 'empty').

AspectJ allows one to capture actual call parameters and return values, and MPL has the same possibility. MPL references the current object of a captured call by this. A constructor, as a target, is identified with the name new. For updating remembered values, the calculation of a new value can be based on side-effect free method calls to any objects known by the policy. Side-effects are not allowed, since the monitor must not influence the state of the monitored program.

Observe that multiple rules can deal with the same guarded method call: The actual situation must satisfy all the guards. The checking order of guards is not important, since evaluation of truth in conditions is required to be side-effect free. The same does not hold for effects: The effects (their possible condition and the expression defining a value) are evaluated in the order they are defined, and a remembered variable is given a new value before the next (if any) effect is evaluated. Another possibility would be to evaluate all effect expressions at the same time (before updating any variable value) and denying any two updates to deal with the same variable.

3.4. Monitor Invariant

To express the relation between the policy invariant and the monitor variables, we present a *monitor invariant* so that it associates the history data related to the program execution (Section 3.1) to the monitor state (Section 3.3), and sufficiently describes the duty of a monitor for being able to expose all possible policy violations. A monitor invariant is true before and after applying any rule. Whereas the policy invariant (Section 3.2) specifies a policy, the monitor invariant specifies the policy implementation. In other words, a monitor invariant determines what monitor variables actually represent. E.g. a monitor invariant could determine the limits for values of the monitor variables, relations between these variables, and the effect of some program events on these variables with respect to the program execution history.

$$\textit{monitor invariant: history} \times \textit{monitor state} \rightarrow \textit{boolean}$$

Meaningful monitor invariant expressions assign a meaning for the remembered monitor variable values in terms of the program execution history. Forming such an expression is challenging, since we must develop an appropriate function for relating the relevant execution history concepts to the policy specification. Moreover, proving such invariant conditions involves evaluating the effect of captured method calls on the actual execution environment (and the execution history of the monitored program). Thus, proving that a rule preserves an invariant condition reduces to matching the caused effects (changes of stored variable values) with respect to the actual effects on the actual execution environment by the guarded method call.

As an example, we specify an invariant for the monitor shown in Figure 2. Our invariant must comply with the policy invariant presented in Section 3.2. In addition, it must connect monitor state to execution history (i.e. connection history).

monitor invariant :

$$\forall conn : connectionhistory.getAccountNumber(conn) = accountTable(conn) \wedge (accountTable(conn) \neq (\text{"}111-888\text{"}) \vee conn.getAccountBalance() \geq 0)$$

In MPL, applying a rule has two different parts. Whereas the condition of a rule uses the monitor values for checking if the execution of the method call in the given situation is trying to break the invariant, the effects part updates the monitor variables to correctly reflect the state of the actual program execution. Verifying a policy means that based on the effects part and the target method specification, it is checked if any resulting state would or would not comply with the invariant, and based on the condition part, it is checked that all invariant breaking efforts are halted but the other calls are allowed. The example presented in the Section 4 is purported to clarify the matter.

3.5. Method of Proving

Next we discuss the method of proving policy specifications and implementations. For proving that the monitor indeed forces the monitored program to comply with the behaviour, it is enough to prove that the rules preserve the invariant condition (and the monitor is correctly formed from the policy description). The setting is identical to having the invariant as rule's pre- and post-conditions, and then proving that a set of conditional assignment statements preserve the invariant condition. If the evaluated expressions (in guarding conditions as well as in the right hand side of assignments) are side-effect free and have a well-defined formal semantics, one is able to use a theorem prover.

For reasoning, we use types and symbols presented in Figures 1 and 3, and functions presented in Sections 3.1, 3.2, 3.3, and 3.4. Assuming that we have some general program monitoring algorithm (applying MPL policies), provably correct policy enforcement can be implemented using the following phases:

P1: A policy invariant is designed to define the policy that will be enforced.

P2: A corresponding monitor is implemented.

P3: A monitor invariant is developed to relate the policy invariant and the monitor state.

P4: Pre- and post-conditions for the operations in the monitoring algorithm are proven correct.

For proving that invariants are valid, we must treat the pre- and post-conditions of monitored calls as axioms used by rules, find every possible policy state (i.e. values of variables) by applying the rules together with these axioms in

all situations in which a method call can be executed, and, finally, check the resulting states of the policy against the invariants.

policy invariant I;
monitor invariant I';
program state s;
monitor state ms;
history state hs = initial history state;
history state hs' = history end state;
call event c;
return event r;

Figure 3: Symbols for the pre- and post-conditions and the algorithm.

When the monitor is executing some program code, there are relevant and irrelevant calls depending on the security policy. The relevant calls are caught by using the function get call. Before we get a call, the invariant must hold. The function get call does not change anything, but it returns us a call event that consists of the current state of the program and the call that should be executed next. Naturally the invariant must also hold after applying get call. Thus, we have a proof obligation $PO1$:

$$\{\ I(s, hs)\ \}\ c = \text{get call}(s);\ \{\ I(c.state, hs)\ \}$$

Basically this means that non-security related functions do not change the program state in a way that has effects security. Also the unrelated calls do not change the history state that records the security related operations.

When a security related call is caught by get call, we must check the pre- and post-conditions related to the call. The function condition checks that the precondition of the current call c in the current state is true, and that the invariant still holds after the call is executed. The former requirement must be satisfied, because otherwise the behaviour of the method is unspecified, and we do not know if the invariant is going to be broken. The latter evaluation is done based on the post-condition of the method call. The function condition does not change the program state or the policy state, but if it is not true, the program execution is halted. Thus, we have a proof obligation $PO2$ for each call c:

$$\{\ I(c.\text{state}, hs) \wedge I'(hs, ms)\ \}$$
$$b = \text{condition}(ms, c);$$
$$\{\ b \Rightarrow (\text{precondition}(c, hs) \wedge (\forall hs', r.\text{postcondition}(c, hs, r, hs') \Rightarrow I(r.\text{state}, hs')))\ \}$$

When the function condition returns true, the call is executed by using the function perform call. Generally, perform call changes the program history and the program state as defined in the post-condition. Thus, we have a proof obligation $PO3$ for each c:

{ precondition(c, hs) }
r = perform call(c);
{ $\exists hs'$.postcondition(c, hs, r, hs') }

After the method call is performed, the function 'effects' changes the state of the monitor to be consistent with the history state. Thus, we have a proof obligation *PO4* for each c:

{ $I(r.\text{state}, hs') \wedge I(c.\text{state}, hs) \wedge I'(hs, ms) \wedge$ postcondition(c, hs, r, hs') }
ms = effects(ms, r);
{ $I'(hs', ms)$ }

Since 'get call' and 'perform call' can have access on the program state but not the monitor state, their pre- and post-conditions do not refer to the monitor invariant. On the other hand, the pre- and post-conditions of functions 'condition' and 'effects' know both of the invariants. However, if we have precisely specified method calls, proving a policy is quite simple. We know that function 'get call' preserves the invariant in any case, and 'perform call' preserves that because of the post-condition of 'condition'. Therefore, we only need to check the post-conditions of 'condition' and 'effects' in practice.

Based on the above analysis, there are proof obligations *PO1* - *PO4* that must be checked for the verification of policies. The proof obligations are conditions on the pre- and post-conditions for the parts of rules (i.e. for each condition, perform call, and effect), and for stepping to the next call in the system. Finally, the actual monitoring algorithm is shown in Figure 4 with a proof of correctness. If all of the conditions are true, the monitoring algorithm works in the right way and the invariants are not broken. The proof obligations naturally follow from the algorithm.

4. AN EXAMPLE: SPECIFICATION AND USE OF INVARIANTS

In this section, we consider proving the correctness of the example policy shown in Figure 5. The monitor guards a low-level system resource, socket streams. Considering the presented policy enforcement phases P1 – P4, we need to specify the policy invariant (P1) and the monitor invariant (P3) for the example policy (P2). The invariant specifications are shown in Section 4.1. We present the description of the execution environment referred by these invariants and the specification of monitored method calls in Section 4.2. Finally, we show an example of proving the proof obligations *PO1*, . . ., *PO4* in Section 4.3. For *PO2*, . . ., *PO4*, we study only the most complex rule related to the call of method OutputStreamWriter.write (String s, int off, int len)in the context of sockets.

program state s = initial state;
monitor state ms = initial monitor state;

while true:

{ $I(s, hs) \wedge I'(hs, ms)$ } // *The same as the loop invariant.*
c = get call(s); // *Get a call event.*
// *By definition, the invariant is not broken and the program state // is not changed.*
{ $I(c.\text{state}, hs)$ }
if not condition(ms, c) **then halt;**
// *If condition is true, it also implies that the precondition of the // method call is true.*
{ $I(c.\text{state}, hs) \wedge$ precondition(c, hs)$\wedge$
$(\forall hs', r.\text{postcondition}(c, hs, r, hs') \Rightarrow I(r.\text{state}, hs'))$ }
r = perform call(c);
{ $\exists hs'.I(r.\text{state}, hs') \wedge$ postcondition(c, hs, r, hs') }
// *Effect updates the monitor state so that all the needed // information is stored.*
ms = effects(ms, r);
{ $I'(hs', ms)$ }
// *After the call has been performed, the history state is changed.*
$hs = hs'$;
$s = r.\text{state}$;
// *The invariant holds at the end of the loop.*
{ $I(s, hs) \wedge I'(hs, ms)$ }

Figure 4: Monitoring algorithm, with the proof of correctness.

4.1. Invariants for the sample policy in Figure 5

For the specification of invariants, we benefit a couple of sets that represent history data. First of them contains the streams that have been opened for a socket connection in the history state. The second one, *wrap*, represents the pairs of objects about which the first one is the wrapper of the second one (in the history state). This means that the functionality of the first object is directly or indirectly based on that of the second object. The third set, *writes*, contains stream-integer pairs in which integer represents the number of bytes that have been sent in the history state. The last one, *hasType*, contains object-type pairs that are, in some reason, wanted to be remembered in the history state.

We further assume that *oS* and *oSW* are of type OutputStream and OutputStreamWriter. The policy invariant determines that at most a limited number (i.e. limit) of bytes can be written into sockets during the execution of a program.

The policy invariant is trivial – it is based on the execution history described in Section 4.2. The first 6 conditions in the monitor invariant are related to the tracking information. The 7th condition is the true purpose of the policy whereas the

8th condition sets up a connection between the execution history *hs* and the variable bytes.

4.2. Description of the environment and monitored operations

In the following, we give a short description of the execution history referred in Figure 5 and pre- and post-conditions of monitored methods in which the effect of the methods on history (interests) is considered. A notation is needed for the compact presentation of the operations that change the state of history.

History state consists of several fields from set F. As usual, we denote $hs.f$ for the value of a field f in state hs. We denote $hs.f \leftarrow x$ for a history state that is the hs, for setting a field, that is

$$hs' = hs.f \leftarrow x \equiv \forall f' \in F.f' = f \Rightarrow hs'.f = x \wedge f \neq f' \Rightarrow hs'.f = hs.f$$

policy invariant
$\sum numberOfBytes(hs, oS) \leq limit$ for $oS \in hs.forsocket$ where *limit* is a constant value.
monitor invariant
$(\neg(\exists oS : oS \in hs.forsocket \wedge oS \notin oSs)) \wedge$
$(\neg(\exists oS : oS \notin hs.forsocket \wedge oS \in oSs)) \wedge$
$(\neg(\exists oSW, oS : (oSW, oS) \in hs.wrap \wedge oS \in oSs \wedge oSW \notin oSWs)) \wedge$
$(\neg(\exists oSW, oS : (oSW, oS) \in hs.wrap \wedge oS \notin oSs \wedge oSW \in oSWs)) \wedge$
$(0 \leq bytes \leq limit,$ where *limit* is the same as *limit* in the policy invariant) $\wedge$
$(bytes = \sum numberOfBytes(hs, oS)$ for $oS \in hs.forsocket$ **)**
variables
limit : *int* = 10000;
bytes : *int* = 0;
oSs : *setofOutputStream* = {};
oSWs : *setofOutputStreamWriter* = {};
rules
Socket : *OutputStream getOutputStream*()
cond none effects $oSs = oSs + result$;

OutputStreamWriter :
OutputStreamWriter new(*OutputStream oS*)
cond $oS \neq null$ **effects if** $(oS \in oSs) oSWs = oSWs + result$;

OutputStreamWriter :
void write(*String s, int off, int len*)
cond $(\neg(this \in oSWs) \vee (bytes + len > limit) \wedge (s \neq null \wedge len <= s.length() - off \wedge 0 \leq off < s.length()))$
effects if $((this \in oSWs) \wedge (len > 0)) bytes = bytes + len$;

Figure 5: A Socket Writing Monitor.

First, the following operations are related to the history of security related operations:

$addSocketStream(hs, oS) = hs.forSocket \leftarrow hs.forSocket \cup \{oS\}$

$addWrapping(hs, wrapper, wrapped) = hs.wrap \leftarrow hs.wrap \cup \{(wrapper, wrapped)\}$

$addObject(hs, object, type) = hs.hasType \leftarrow hs.hasType \cup \{(object, type)\}$

$numberOfBytes(hs, stream) = max\{n|(s, n) \in hs.writes \wedge s = stream\}$

$write(hs, stream, num) = hs.writes \leftarrow hs.writes \cup \{(stream, num + nOfBytes(hs, stream)\}$

$type(hs, o) = \{o|(o, t) \in hs.hasType \wedge t = type\}$

The first operation describes adding a new socket stream to the history. The second adds wrapping information to the history, and the third stores objects with their types to the history. nOfBytes returns the number of bytes written to a stream. The write operation represents the number of bytes written to the stream in one time. Lastly, type returns the set of objects with the given type.

Next, we present the monitored operations with their pre- and post-conditions. Notice how the machine OutputStreamWriter defines the post-condition of write in terms of the execution history (the precondition of write is slightly different than that of the actual Java class):

$\{true\}$
oS = Socket.getOutputStream()
$\{hs = addObject(addSocketStream(hs_0, oS), oS, OutputStream)\}$

$\{out \neq null\}$
oSw = new OutputStreamWriter(out)
$\{hs = addObject(addWrapping(hs_0, oSw, oS), oS, OutputStreamWriter)\}$

$\{s \neq null \wedge len \leq s.length() - off \wedge 0 \leq off < s.length() \wedge$
$\exists oS \in type(hs_0, OutputStream) : hs_0.wraps(oSw, oS)\}$
oSw.write(s, off, len)
$\{(len > 0 \Rightarrow hs = write(hs_0, oS, len)) \wedge (len \leq 0 \Rightarrow hs = hs_0)\}$

4.3. Proving a rule

In this section, we give an example how to use invariants to prove our sample policy. The invariant of this policy is valid when $limit - n$ bytes have been sent via socket connections, where $0 \leq n \leq limit$. Proving of a rule is based on the method presented in Section 3.5. Although we limit our detailed observation here only to one rule that is essentially related to the policy, it would be easy to check the other rules

similarly. Moreover, if we try to achieve the policy enforcement, it is easy to see that all output stream writers for socket connections are really in *oSWs*, and that other kind of connections are not in *oSWs*. We first catch the output stream *oS* for a socket connection. $oS \in oSs$ after operation Socket.getOutputStream(). Next, when we call the constructor OutputStreamWriter(*oS*), wraps(*oSW, oS*), and $oS \in oSs$. Thus $oSW \in oSWs$. Since $(oSW \in oSWs \wedge wraps(oSW, oS)) \Rightarrow (oS \in oSs)$, In the other cases, $(oSW \notin oSWs)$ and the writing operation is never halted.

Next, we give an example how to prove the rule guarding the method OutputStreamWriter.write(String s, int off, int len).

We use the following shorthand notations.

$ms = monitor\ state$
$ms.bytes = limit - n$
$args = (s, off, len)$
$cmd = (OutputStreamWriter.write, this, args)$

PO1**: get call(state)**
$\{ I(st, hs) \}$
$c = get\ call(st) = (state, cmd)$
$\{ I(c.state, hs) \}$

The invariant holds, because the function get call does not execute any methods that could break the invariant. It is presumed that the precondition is true.

PO3**: perform call(call event)**
$\{ precondition(c, hs) \}$
$r = perform\ call(c);$
$\{ \exists hs'.postcondition(c, hs, r, hs') \}$

The invariant holds, since the pre-condition and post-condition are checked by the condition before the method call is performed. It is presumed that the method complies with its specification. The function perform call may change the program state and the execution history.

PO2**: condition(monitor state, call event):**
$\{ I(c.state, hs) \wedge I'(hs, ms) \}$
b = condition(ms, c);
$\{b \Rightarrow (precondition(c, hs) \wedge (\forall hs', r.postcondition(c, hs, r, hs') \Rightarrow I(r.state, hs')))\}$
condition :
$(this \notin ms.oSWs \vee ms.bytes + c.args.len \leq limit)$

$\wedge\, c.args.s \neq null$
$\wedge\, c.args.len \leq c.args.s.length() - c.args.off$
$\wedge\, 0 \leq c.args.off < c.args.s.length())$

We need to consider the following cases:

1. $this \in ms.oSWs \wedge limit \geq ms.bytes + c.args.len$
2. $this \notin ms.oSWs$
3. Otherwise, condition returns false and the program is halted in every case.

Notice that from the monitor invariant, it follows that $this \in ms.oSWs \Leftrightarrow oS \in hs.forsocket$.

The **precondition of the method** is given below. The condition needs to verify that the precondition is satisfied, because otherwise the behaviour of the method is unspecified. For cases 1) and 2) we have as the method precondition:

$\{c.args.s \neq null \wedge c.args.len \leq c.args.s.length()$
$-c.args.off \wedge 0 \leq c.args.off < c.args.s.length() \wedge$
$\exists oS \in type(hs, OutputStream) : hs.wraps(this, oS)\}$

The **postcondition of the method** is
$\{(c.args.len > 0 \Rightarrow hs = write(hs, oS, c.args.len)) \wedge (c.args.len \leq 0 \Rightarrow hs' = hs)\}$

We need to verify that the invariant is satisfied with this post-condition.

1. When $c.args.len \geq 0$, the history is changed only for oS and

for $oS \in hs.forsocket$ $\sum numberOfBytes(hs, oS) = ms.bytes$
thus we get
for $oS \in hs'.forsocket$ $\sum numberOfBytes(hs', oS) = ms.bytes + c.args.len$

which was checked to be less or equal than *limit*. Otherwise $hs' = hs$ (no bytes written).

2. When $oS \notin hs.forsocket$,
for $oS \in hs'.forsocket$ $\sum numberOfBytes(hs', oS) = ms.bytes$.

The invariants hold, since the function *condition* does not change anything. If performing the guarded method call could not break the invariant, the function returns true. In the other case, it returns false, which causes halting the execution of the program.

PO4**: effects(monitor state, return event):**
$\{I(r.state, hs') \wedge I(c.state, hs) \wedge I'(hs, ms) \wedge$ postcondition$(c, hs, r, hs')\}$
// that is the precondition holds

ms' = effects(ms, r);
$\{ I'(hs', ms') \}$
effects :
if $((this \in ms.oSWs) \wedge (c.args.len > 0)) ms.bytes = ms.bytes + c.args.len;$

1. If $c.args.len < 0$, then $ms' = ms$ as in case 2).
   $ms'.bytes = ms.bytes + len$
   $= \sum numberOfBytes(hs, oS) + c.args.len$
   $= \sum numberOfBytes(hs', oS).$

The last equality holds because of the post-condition.

2. Because $oS \notin hs.forsocket, ms' = ms$,
   ms'.bytes = ms.bytes
   $= \sum numberOfBytes(hs, oS) + c.args.len$
   $= \sum numberOfBytes(hs', oS).$

In both of the cases, the monitor invariant holds. The function 'effects' updates only the monitor variables to be consistent with the policy related history data. The policy invariant trivially holds.

In this section, we have proved that a rule preserves the policy invariant and the monitor invariant. When we know that all rules preserve the invariant, we can say that the monitor enforces the policy. However, since we know that all output stream writers for socket connections are really in *oSWs* and that other kind of connections are not in *oSWs*, we can easily conclude that our rules preserve the invariants.

5. CONCLUSIONS

We have demonstrated using invariants for proving security policies and made an effort to show that invariant based proving with rule-based monitoring languages is relatively easy. We have presented the policy invariant for the policy specification and the monitor invariant for the policy implementation. Both of these invariants are formally specified. The policy invariant is a predicate that expresses the meaning of a policy in terms of the state and execution history of a program. It is true before and after any executed method call. The monitor invariant is a predicate that associates the history data related to the program execution to the monitor variables, and is true before and after applying any policy rule.

Since the policy related execution history is stored in the monitor variables, the policy and monitor invariants can be used when proving that the monitor is really enforcing the policy. A four phase policy enforcement method was given and analysed. As a practical case, we showed how the presented proof obligations *PO1 – PO4* can be verified in practice.

One possible area of future work is using theorem provers to prove correctness of monitors. To accomplish this, the security related procedures and the policy and monitor invariants have to be specified formally in a language that is understood by some theorem prover. Then the proof obligations can be verified by the prover. This can be used as a lightweight approach for proving certain kind of security properties of programs.

REFERENCES

[1] Abrial, J.-R. *The B-Book, Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] Albert, E. et al. Abstraction-Carrying Code: a Model for Mobile Code Safety. *New Generation Computing,* vol. 26, pp.171-204, Springer, 2008.

[3] Allan, C. et al. Adding Trace Matching with Free Variables to AspectJ. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005*, pp.345-364, October 2005.

[4] Bauer, L. et al. Run-Time Enforcement of Nonsafety Policies. In *ACM Transactions on Information and Systems Security*, 12(3), Article 19, January 2009. ACM Press.

[5] O'Nell Meredith, P. et al. An overview of the MOP runtime verification framework. *Int J Softw Tools Technol Transfer* (2012) 14:249–289.

[6] Heath, C. *Symbian OS Platform Security: Software Development Using the Symbian OS Security Architecture*. Wiley, 2006.

[7] Hermenegildo, M. et al. Abstraction Carrying Code and Resource-Awareness. In *PPDP '05: Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 1–11, New York, NY, USA, 2005. ACM Press.

[8] Karlstedt, T. et al. Embedding Rule-Based Security Monitors into Java Programs. In *Proceedings of IEEE 32nd Annual International Computer Software & Applications Conference, COMPSAC'08*, pages 20–27, 2008.

[9] Ligatti, J. et al. Edit Automata: Enforcement Mechanisms for Run-time Security Policies. *Int. Journal of Information Security*, 4(1–2):2–16, 2005.

[10] Leavitt, N. Mobile phones: the next frontier for hackers?, *Computer,* 38:4, 20-23, IEEE, 2005.

[11] Leppänen, V., J-M. Mäkelä, Security Monitors for Java Programs with MPL, *International Journal on Information Technologies and Security* 4 (1), pp. 35-50, 2012.

[12] Necula, G.C. Proof-carrying code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN- SIGACT Symposium on Principles of Programming Languages*, pages 106–119, 1997.

[13] Schneider, F.B. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[14] Sekar, R. et al. Model-Carrying Code (MCC): a new paradigm for mobile-code security. In *Proceedings of the 2001 Workshop on New Security Paradigms, NSPW'01*, pages 23–30, New York, NY, USA, 2001. ACM Press.

[15] Rice, H.G. Classes of Recursively Enumerable Sets and Their Decision Problems. *Trans. Amer. Math. Soc.*, vol. 74, pages 358–366, 1953.

[16] Martinelli F., P. Mori. Enhancing Java Security with History Based Access Control. In *Foundations of Security Analysis and Design (FOSAD 2006/2007)*, LNCS 4677, Springer Verlag, pages 135–159, 2007.

[17] Ion, I. et al. Extending the Java Virtual Machine to Enforce Fine-Grained Security Policies in Mobile Devices. In *Proceedings of 23rd Annual Computer Security Applications Conference (ACSAC 2007)*, IEEE Computer Society, pages 233–242, 2007.

Information about the authors:

Sanna Mäkelä (formely Tuohimaa) – She is a PhD student of University of Turku, Department of Information Technology. Her research interests have focused on software security and architectures.

Sami Mäkelä – Mäkelä is currently finishing his PhD studies. His PhD thesis work has focused on software metrics and correctness issues. He has also participated into security assessment work related to Finnish electronic voting system.

Ville Leppänen – PhD, works current as a software engineering professor in the University of Turku, Finland. He has over 100 scientific publications. His research interests are related broadly to software engineering ranging from software security and quality to engineering methodologies and practices and from tools to programming language, parallelism and algorithmic design topics. In the security domain he has led a (Finnish) Ministry of Defense funded research project on software diversification techniques, and is currently site leader of large Cyber Trust project.

Manuscript received on 15 April 2015