A Method for Certifying Code in Trust-By-Policy-Adherence

Guo-Sun Zeng
Department of Computer Science and Technology, Tongji University, Shanghai 201804, China
Email: gszeng@sina.com

Li Li
Department of Computer Science and Technology, Tongji University, Shanghai 201804, China
Email: snopy-xj@163.org

Abstract—This paper proposes and details the notion of trust by policy adherence (TBPA), meaning that code can be certified on the basis of its security-related behaviors rather than its origins and integrity. We describe the overall life cycle of code in this setting, and propose a detailed method whereby a program’s policy adherence can be verified. We suggest enforcing security policies over code by means of aspect-oriented programming (AOP). Based on the characteristics of AOP programs, we model security policies and a verification process using alternating temporal logic. This method can be used to verify whether a given program complies with a wide range of security policies, including both safety and liveness policies. It can also verify whether the original program is affected by policy execution. We argue that TBPA provides a suitable semantic framework for certifying code, and represents a step forward from trusted code toward trustworthy code.

Index Terms—Trust, policy adherence, Aspect-oriented programming, Alternating Time Temporal Logic

I. INTRODUCTION

In highly interconnected and distributed network computing environments, it is becoming more and more necessary to build trust into network entities such as users, platforms, and especially some programs. The purpose of this movement is to make certain that programs will not do something harmful.

Trusted computing technologies based on TPMs [1] (trusted platform modules) try to build trust into applications by supplying an identity certificate and confirming the integrity of the code. However, integrity and identity only reveal who is responsible for the code and that the code was not altered by attackers. No additional security is attached to this information. Essentially, consumers must accept the code “as-is”, relinquishing the possibility of making decisions based on their security requirements. For example, in principle one would like to trust code that can prohibit the send operation after reading secure data, not just code that comes from a famous software factory. Code producers, on the other hand, cannot declare that their work will comply with a given security specification. Because the level of security is built into the TPM hardware, so the decision is out of their hands. As a consequence, they may find it hard to convince consumers that their code will not do anything harmful.

If a producer could guarantee that its code will comply with certain security specifications, however, then consumers would be free to choose the code or not according to their security requirements.

A. Contributions

We propose in this paper the notion of trust by policy adherence (TBPA): the certificate should not just certify the origin of the code, but bind the code to a specific security policy. Loosely speaking, a security policy is a behavior specification for the code. It can define rules for access control, memory use, secure web connections, privacy protection, and so on. In this way, for each program installed, either the platform or the user could choose among a range of security policies avouched by the certificate.

This paper describes the overall life cycle of code in a setting of trust-by-policy adherence, explains how security policies can be enforced on the code by aspect-oriented programming [2], and provides a formal method of verifying policy adherence. We argue that security policies provide the necessary semantics for certifying code, thus representing an important step in the transition from trusted code to trustworthy code.

In the next section, we briefly introduce the technology of executing a security policy over code and describe the basics of a TBPA scenario. In Section 3, we present a concrete programming paradigm for security policy realization. The next two sections (4 and 5) describe a verification method for policy adherence. A discussion of related work and some conclusions end the paper.

Supported by the china 863 program under grant of 2009AA012201, the National Natural Science Foundation of China under grant of 90718015, the joint of NSFC and Microsoft Asia Research under grant of 60970155, the Ph.D. Programs Foundation of Ministry of Education under grant of 20090072110035.
II. THE TRUST BY POLICY ADHERENCE USAGE MODEL

Definition 2.1 (security policy). A security policy [3, 4] is a formal, complete specification of acceptable behavior for applications to be executed on the platform, in matters concerning relevant security actions.

Definition 2.2 (safety policy). A security policy specifying that “nothing bad ever happens” is called a safety policy [3, 4].

Definition 2.3 (Liveness policy). A liveness policy [3, 4] states that nothing irremediably bad happens, or that good things will happen eventually.

One efficient way of enforcing a security policy is by inlining monitor code [5, 6] to produce a self-monitoring program. At compile time, untrusted code or binary executable are automatically rewritten to comply with an external security policy, which can be defined using a policy specification language such as PSLang in SASI [7] or MEDL/PEDL in Java-MaC [8]. Another language, ConSpec [9], is a simplification of PSLang. Its denotation semantics are built upon security automata, so it can describe a safety policy as well as PSLang or MEDL/PEDL. The inline monitors execute safety policies by accepting legal actions and rejecting illegal actions as the program executes. In this situation, the monitors act as invalid execution recognizers. Sometimes, monitors insert control actions into the program in order to execute some kind of liveness policy.

Independent of the research on inline monitoring, aspect-oriented programming [2] (AOP) has been proposed to deal with the tangled and scattered code that can result from crosscutting concerns. Numerous authors [10, 11, 12, 13] have observed that AOP lends itself to the implementation of security enforcement mechanisms such as inline monitors. The approach is quite powerful, having been used to enforce a wide range of important security policies, including some safety and liveness properties [4, 10].

In our usage model (see Fig. 1), code developers incorporate security policy enforcement into their programs using AOP. Then, using the verification method designed in Section 4, their programs can be certified to comply with a security policy designed by the developer’s own company or any trusted third party.

At deployment time, the user checks whether the certificate is correct. As we have said already, this certificate can serves as proof that the code complies with certain security policies. Once the user confirms that the certificate is trustworthy, he/she can check whether the policies adhered to by the code meet their platform’s security requirements. If so, the application can be run. As we see in Figure 1, a key obstacle in the overall workflow of TBPA is verifying the policy adherence of the code. To address this issue, we design an abstract verification model. This will be the subject of section 4.

III. USING AOP TO ENFORCE POLICIES OVER PROGRAMS

Security policies specify various allowed and prohibited execution steps. Below we list some types of security policies for programs:

1) Secure connection policies: for example, the program can only use HTTPS.
2) Authorization policies: for example, only user Jones is permitted to access a file.
3) All files opened by the program need to be closed before the program ends.
4) A policy might prohibit execution of Send operations after a file read operation.

Examples 1, 2, and 4 are safety policies. Example 3 is a liveness policy.

A. Developing a program using AOP

AOP enables the modular implementation of crosscutting concerns. This practice is most often realized in AspectJ [14] languages. The base code of AOP is a primary program handling the core concerns of the application; an aspect is a code fragment that modularizes an orthogonal concern. An aspect weaver is a compiler that integrates the aspects into the base code. Each aspect specifies where and how to inject its own code into the base code. The basic constructors of an aspect include three new concepts:

- Joinpoints: A joinpoint is a location within a program where the aspect weaver can integrate a code fragment, called an advice. An advice can be executed before, after, or even around a join point.
- Pointcuts: A pointcut is a set of join points sharing specific static properties. For instance, in

![Figure 1. Workflow of trust-by-policy adherence](image-url)
AspectJ, pointcuts are defined using quantified Boolean formulas over method names, class names, control flow, and/or lexical scopes. A pointcut can capture specific event occurrences such as method calls, access to attributes, and exceptions.

- **Advice:** advice is a fragment of code that is executed before, after, or around the evaluation of the joinpoint.

Aspect weaving technology based on these constructors allows security-related events to be defined beyond the kernel functions of the AOP program. For example, one aspect could add code that logs every file operation, while another inserts a security authorization mechanism before operations that read secret data.

The following example shows how to translate a security policy into an aspect, and the constraint behaviors present in the base code after aspect weaving.

**Example 1** There is a Separation of Duty policy requiring that the critical() operation be performed only under the endorsement of both the manager() and accountant() operations. We develop a program that adheres to this policy using the AspectJ language. The program is shown in Fig. 2.

The base code in block (a) is the primary program. It defines two global variables, pm and pa, and executes three security-related functions: accountant(), manager(), and critical(). According to the policy, critical() can be executed only after both accountant() and manager() have been executed. To execute this policy, we create three aspects. First, aspect Ma defines Pointcut m, which is located at the function manager(). The type of advice is After. Thus, the action defined by aspect Ma is that after manager() has finished, the variable pm is set to “true”. Next, aspect Ac is defined similarly to aspect Ma, but sets pa to true after accountant() has finished. Finally, aspect Cr defines Pointcut c at the function critical(). The type of advice is Before, so the actions defined by aspect Cr take place before the execution of critical(). The code fragment evaluates pm and pa; if both are true, it executes the critical() function and sets pm and pa to false. Otherwise, it throws an exception.

At compilation, the aspect weaver incorporates all three aspects into the base code to produce a policy-adherence program (PA program). Figure 3 displays the code for the resulting PA program.

IV. A VERIFICATION MODEL FOR POLICY ADHERENCE

By virtue of AOP technology, it is easy to develop clear PA programs. The existence of non-declarative advice in aspects, however, makes it even hard for programmers to ensure that programs comply exactly with the intended security policies, and that aspects will have no side effects on the base code. Taking the characteristics of PA programs and the composition reasoning used in aspect verification [15, 16] as starting points, we construct a model to verify that PA programs have the correct property. In this context, “correct” includes two abstract properties: coherence and transparency. Coherence means that the program is guaranteed to adhere to the security policies. Transparency means that any aspects incorporated do not impair the primary function of the base code.

A. Abstract program structure

Alternating-Time Temporal Logic (ATL) and Alternating Transition Systems (ATS) [17, 18] are logical specification tools for an open system. They were proposed to account for questions such as the following: “On a state machine model which describes an open system and its environment, can the system resolve its choices in such a way that the satisfaction of a property is guaranteed no matter how the environment resolves the external choices?” This satisfaction can be viewed as the winning condition of a two-player game between the system and the environment. When the system consists of several components, the question implies a multi-player game where each component of the program, system, and environment is represented by a different player. ATL and ATS can be used to specify this more general setting in addition to the simple example discussed in this paper.

We will use ATL and ATS to specify alterations of the environment, the base code, and the aspects, and then

Figure 2. base code and aspect of example 1

Figure 3. Policy-adherence program after Aspect weaving
analyze the correct property of the program. To begin with, we abstract a PA program as a Turn-Based Alternating Transition System (Turn-based ATS). The concrete definition is as follows:

**Definition 4.1** An PA program structure (PAP structure) can be abstracted as a Turn-based ATS. Expressed as tuple, the PAP Structure is $<\Sigma,Q,P,\pi,\sigma,\delta>$ with the follow components:

1. $\Sigma$ is a set of players: Aspect, the BaseCode, and the Environment.
2. $Q$ is a finite set of states $q$;
3. $\Pi$ is a finite set of propositions $p$;
4. The function $\pi: Q \rightarrow 2^\Pi$ is a labeling function which maps each state $q \in Q$ to a set $\pi(q) \subseteq \Pi$. $\pi(q)$ is the set of propositions that are true when the system is in state $q$.
5. The function $\sigma: Q \rightarrow \sum$ maps a state $q$ to a player $a_q$. It indicates that when the system is in state $q$, it is the turn of player $a_q$ to choose the next execution step of the program. The integer $d_a(q)\geq 1$ is the number of moves available to player $a_q$ at state $q$. We identify the moves with the sequence of numbers $1…d_a(q)$. Thus, for each state $q \in Q$, there is a vector of possible moves: the tuple $<j_1, …, j_k>$ such that $1 \leq j_k \leq d_a(q)$ for each player $a$. For all other players $b \in \Sigma$ at state $q$, $d_b(q)=1$. (This is a way of stating that no other players have a choice of action.)
6. $\delta(q, j_k)$ is a transition function. When $a_q$ chooses action $j_k$, state $q$ will transit to state $q' = \delta(q, j_k) \in Q$.

**B. Definition of formulas specifying the “correct” property**

As we have already emphasized, correct implies both coherence and transparency. A sound PA program complies with the intended security policy. Transparency indicates that the incorporated aspects do not impair the function of the primary program. In this section, we interpret the syntax and semantics of alternating-time temporal logic. We shall then describe the properties of coherence and transparency as ATL formulas. 

**Definition 4.2 (Strategy)** Given a structure $S = <\Sigma,Q,\Pi,\pi,\sigma,\delta>$, a strategy for player $a \in \Sigma$ is a function $f_a: Q \rightarrow 2^\Sigma$ such that for all $\lambda \in Q$ and $\lambda = \lambda(q, f_a(\lambda)) \in d_a(q)$.

The strategy of player “a” constraints the possible executions, and those possible executions is path chosen by player $a$.

**Definition 4.3 (Path)** Given a state $q \in Q$, a set $A \subseteq \Sigma$ of players, and a set $F_A(f_a; a \in A)$ of strategies for the players in $A$, we define the path from $q$ to be the set out$(q; F_A)$ of computations $\lambda$ that the players in $A$ enforce as they follow the strategies in $F_A$. That is, a given computation $\lambda=q_0,q_1,q_2…$ is in out$(q; F_A)$ if $q_0=q$ and for all positions $i \geq 0$, there is a move $j_i$ such that $1 \leq j_i \leq d_a(q)\{0,1\}$ for all players $a \in A$, and (2) $\delta(q, j_i) = q_{i+1}$.

**Definition 4.4 (ATL Syntax)** An ATL formula is one of the following:

1) $P$, proposition $p \in \Pi$;
2) $\neg \phi$ or $\phi \lor \phi$, where $\phi$, $\phi_1$ and $\phi_2$ are all ATL formulas;
3) $\langle A \rangle \diamond \phi$, $\langle A \rangle \Box \phi$, $\langle A \rangle \circ \phi$, or $\langle \langle A \rangle \rangle \phi$, $\langle \langle A \rangle \rangle \circ \phi$, $\langle \langle A \rangle \rangle \Box \phi$ , where $A \subseteq \Sigma$ is a set of players and $\phi$, $\phi_1$ and $\phi_2$ are all ATL formulas.

The operator $\langle \langle \rangle \rangle$ is a path quantifier. The symbols $\phi$ ("next"), $\Box$ ("always"), $\diamond$ ("eventually"), and $U$ ("until") are temporal operators. $\langle A \rangle \phi$ represents the path chosen by the players in set $A$. The Quantifier $\langle \langle \rangle \rangle$ also has a dual form $[\langle \rangle]$. While formally $\langle A \rangle \phi$ means that the players in $A$ can cooperate to make $\phi$ true, $[\langle A \rangle] \phi$ means that the players in $A$ cannot cooperate to make $\phi$ false.

**Definition 4.5 (Semantics of ATL)** The satisfaction relation $\models$ defined as follows:

1) $q \models \phi$ iff $p \in \Pi$ and $p \subseteq \mathcal{T}(q)$;
2) $q \models \neg \phi$ iff $q \not\models \phi$;
3) $q \models \phi \lor \phi_1$ iff $q \models \phi$ or $q \models \phi_2$;
4) $q \models \langle \langle A \rangle \rangle \circ \phi$ iff there exists a set of strategies $F_A$, one for each player in $A$, such that for all computations $\lambda \in \text{out}(q, F_A)$, we have $\lambda(1) \models \phi$;
5) $q \models \langle \langle A \rangle \rangle \Box \phi$ iff there exists a set of strategies $F_A$, one for each player in $A$, such that for all computations $\lambda \in \text{out}(q, F_A)$ and all $i \geq 0$, we have $\lambda[i] \models \phi$;
6) $q \models \langle \langle A \rangle \rangle \circ \phi \lor U \phi_2$ iff there exists a set of strategies $F_A$, one for each player in $A$, such that for all computations $\lambda \in \text{out}(q, F_A)$ and all $i \geq 0$, we have $\lambda[i] \models \phi_2$, and for all positions $j$, $1 \leq j < i$, we have $\lambda[i] \models \phi$.

Based on the above definitions of ATL logic, we can now formulate the correct property of a PAP structure.

**Definition 4.6 (Coherence)** Coherence means that the PA program should adhere to the intended policy. That is to say, in the framework of ATL all possible paths decided by the strategies of two players, Aspect and BaseCode should satisfy each policy requirement $\phi$, no matter how the Environment chooses. Expressed as an ATL formula, one would write $\phi = \langle \langle \text{Aspect, Basecode} \rangle \rangle \phi$. $\phi_s$ would be described differently according to the concrete policy:

1) For a safety policy, $\phi_s$ is a statement that bad things will never happen, or that some property should always hold. For example, if the policy requires that action1 and action2 never happen concurrently on the path chose by Aspect and BaseCode, then one would write: $\langle \langle \rangle \rangle \Box \langle \langle \text{Aspect, Basecode} \rangle \rangle \neg \langle \langle \text{action1 \land action2} \rangle \rangle$
2) For a liveness policy, $\phi_s$ is a statement that something will happen eventually. For example, if all opened files should be closed before the end of the program, one would write $\langle \langle \rangle \rangle \langle \langle \text{OpenFiles} \rightarrow \langle \langle \text{Aspect, Basecode} \rangle \rangle \Box \text{Close} \rangle$. 

© 2011 ACADEMY PUBLISHER
3) If a policy can be decomposed into several sub-policies, then ϕ, includes all the child formulas.

**Definition 4.7 (Transparency)** Transparency means that the Aspect player should not impair the character or behavior of the primary program (its semantics and functionality). That is to say, within the PAP structure, all paths decided by the strategy of BaseCode should satisfy the core requirements ϕ of the primary program no matter how the other players choose. Expressed as an ATL formula, one would write ϕ = (BaseCode)ϕ. ϕ would be described differently according to the character of the original program; it might describe a functionality of the base code, or some desired properties of the base code.

C. Verifying the "correct" property by model checking

In the framework of ATL, the model checking problem consists of computing winning strategies, which also testify to the correctness of the game structure. We can use the general ATL model-checking algorithm on the coherence and transparency formulas defined above. That is, given a PAP structure and specific coherence and transparency ATL formulas, this algorithm can compute winning strategies. If winning strategies exist for the Aspect and BaseCode players, we conclude that the PA program is guaranteed to have the correct property. Otherwise, there exists a winning strategy for the Environment which serves as a counter-example. In this situation, we can conclude that the PA program does not have the correct property.

V. AN EXAMPLE

The following example demonstrates the effectiveness of our verification model.

**Example 2** A program module makes use of the point class, which has a method move, to move a point on canvas. The program interface contains a canvas, two numeric text fields where the user can fill in x and y coordinates, and press "ok" button to move the point to the specified location on the canvas. The program only reads the text fields; it does not write to them. Their values are therefore determined by the environment alone. We assume that the text fields only accept non-negative coordinates.

For clarity of display, we add a constraint policy on this program’s executions. When the point’s location is too close to the origin, e.g. 0<\(x<5\) and 0<\(y<5\), the base code multiplies the coordinate by a significant factor, in this case 10. This factor is stored in the variable scaleFactor. Conversely, if \(x \geq 5\) or \(y \geq 5\), the coordinate is left alone. For the display to continue working properly, scaleFactor must be reset to 1 at the end of the program. This policy is encoded as an aspect named **adjustscale**. The code is shown in Fig. 4. Henceforth, the constraints are written as (x, y) < 5 and (x, y) ≥ 5 for simplicity.

A. PAP structure

Adopting the method proposed in section 4, we construct the PAP Structure of the code in Fig. 4. There are three players: Environment, BaseCode, and Aspect.

**Class Point**

```java
{ int x, y;
    int scaleFactor=1;
    public int getX() {return x}
    public int getY() {return y}
    public coordinate(int a, int b) {this.x=a; this.y=b;}
    public void move (int nx, int ny)
     { x=nx*scaleFactor; y=ny*scaleFactor; }
}
```

While ( ! Cancel )

```java
{ if (Ok)
    p.coordinate (getX(),getY())
    p. move (x , y)
}

( Base Code)

aspect adjustscale

{ Pointcut m(Point p):
    execution(void Point.move(int,int))
    &targets(p);
    Before m()
     { if ((p.x<5)&&(p.y<5))
        {p.scaleFactor=10;}
    After m()
     {p.scaleFactor=1;}
}
( Aspect)
```

Figure 4. Code of Example 2

Environment sets the inputs of the program and presses the button. BaseCode executes the move function. Aspect executes the constraint policy. Henceforth we use the characters e, b and a to represent the three players. Figure 5 represents our PAP Structure visually using state transition diagrams.

On the bottom of Fig. 5, we label lines of code with 0, 1, 2, m1 or m2. On the top of Fig 5, each circle with label inside describe which propositions hold in that state, such as, at root state, propositions ¬L and pre hold.. Those propositions are abstractions of the concrete program states, which are defined in Table 1. The symbol e, b or a beside a circle indicates the active player at that state. For example, the symbol beside the root state is e0, meaning that Environment is the active player at program point 0.

Assume that the program is at point 0, and that both coordinates (x,y) are currently greater than or equal to 5. The Environment can choose between entering coordinates (x, y) ≥ 5, entering coordinates (x, y)<5, or pressing the ok button.
If the ok button is pressed, the program will execute the right-hand subtree. BaseCode gets the values of the coordinates, then attempts to advance the program counter to point 2. Aspect interposes itself again and sets the program to point $m1$. Since $(x, y) \geq 5$, Aspect does not increase the $scaleFactor$ value and returns control to BaseCode at point 2. After the move function is finished, Aspect interposes itself again and sets $scaleFactor$ to 1.

If the Environment enters coordinates in the range $0 < (x, y) < 5$ and then presses the button, Aspect would set $scaleFactor=10$ at point $m1$ but change its value back to 0 if $(x, y) < 5$ and the button is pressed, on all the path choose by Aspect and BaseCode, there is no state satisfy move with $scaleFactor=10$. Hence, we conclude that the program in Fig. 4 has the correct property.

B. Coherence formulas and verification

1) According to the policy, the coefficient $scaleFactor$ should be reset to 1 eventually. This is a liveness policy. Thus, we have the following formula:

$$\langle p \rangle \diamond \langle \langle \text{move} \rightarrow \langle \langle a, b \rangle \rangle \rangle \diamond \neg L$$

Whenever the move function is finished, Aspect and BaseCode should cooperate to restore $scaleFactor$ to its normal value before the program terminates.

2) According to the security policy, program execution should always obey the following logic: if $(x, y) < 5$, then the move function is executed with $scaleFactor=10$; if $(x, y) \geq 5$, then the move function is executed with $scaleFactor=1$. Thus, we have formulas:

$$a) \quad \langle \langle \text{move} \rangle \rangle \diamond ((\neg \text{pre} \land \text{ok}) \rightarrow \langle \langle a, b \rangle \rangle \diamond \neg (L \land \text{move})): \text{whenever } (x, y) < 5 \text{ and the button is pressed, on all the path choose by Aspect and BaseCode, there is no state satisfy move with } scaleFactor=1$$

$$b) \quad \langle \langle \text{move} \rangle \rangle \diamond ((\neg \text{pre} \land \text{ok}) \rightarrow \langle \langle a, b \rangle \rangle \diamond \neg (L \land \text{move})): \text{whenever } (x, y) \geq 5 \text{ and the button is pressed, on all the path choose by Aspect and BaseCode, there is no state satisfy move with } scaleFactor=10.$$
then press the button again. The corresponding execution path is (S1→S2→S4→S6→S8→S1→S3→S5→S7). The first time the button is pressed, the \( \neg Pre \) proposition holds, so the program will execute along the left-hand subtree. During this stage, scaleFactor is set to 10 (at S6).

The program then accepts coordinates (7,8), which satisfy the Pre proposition. The program should execute along the right-hand subtree, but there will be a state that can satisfy both L and move. Therefore, Formula 2.b does not hold. Environment has succeeded in forcing the program to execute Move with scaleFactor=10 when the Pre proposition is true. Thus, formula 2.d also does not hold. We conclude that there must be some problem with the Aspect, as the program does not adhere to the policies.

VI. RELATED WORK AND CONCLUSION

Once a program has been verified to comply with a given security policy, the software company or a trusted third party will issue a certificate to this effect. Code consumers can then choose to trust the code or not on this basis.

Language-based approaches to computer security have employed two major strategies for enforcing security policies over untrusted code: low-level type systems and execution monitoring.

Low-level type systems can enforce security policies involving important program invariants such as memory and control. Proof-carrying code (PCC) [19] generalizes the type-safety approach by providing explicit proof of safety (code safety). The PCC approach launched the idea that untrusted code should be accompanied by information that aids in verifying its safety. The code consumer uses a specialized application to check that the proofs provided are valid, and hence the code is safe to execute. Such proofs can be automatically generated by a certifying compiler [20] based on a static analysis of the producer code. The traditional approach to PCC based on type theory is problematic in that it usually enforces fixed-type security policies that are encoded into the type system or proof logic itself. The security policies therefore cannot be changed without changing the type system or certifying compiler.

Execution monitoring is an established technique for enforcing a wide range of policies over programs. For efficiency, execution monitoring is often implemented in the form of in-lined reference monitors [6]. Researchers have devised many techniques for proving that a program with in-line monitors obeys the safety policies.

Mobile [21] is a certifying in-lined reference monitoring system for the Microsoft .NET framework. It rewrites .NET CLI binaries according to a declarative security policy specification, producing a proof of policy-adherence in the form of typing annotations in an effect-based type system. These proofs can be verified by a type-checker to guarantee policy-adherence of code with in-line monitors.

Aktuga et al. [22] designed a two-level class file annotation scheme using Floyd-style program logic for Java bytecode, characterizing two key properties: (i) that the program adheres to a given policy, and (ii) that the program has an embedded monitor for this policy. They sketch a simple in-lining algorithm, and show how the two-level annotations can be completed to produce a fully annotated program. This method establishes the mediation property, meaning that in-lined programs are guaranteed to adhere to the intended policy. Furthermore, the validity of the code can be efficiently checked using an annotation checker based on the weakest precondition. This work is preparing the ground for on-device checking of policy adherence in a proof-carrying code setting.

The methods developed by Hamlen and Aktuga only certify that a program with in-lined monitors adheres to certain safety policies. Neither can establish the transparency property, which would ensure that the monitors have no ill effects on the original program.

In this paper, we have tried to deal with the verification of policy adherence in a different way. Firstly, we enforce security policies over a program by means of aspect-oriented programming (AOP). Secondly, based on the characteristics of AOP, we abstract the execution structure of the program using alternating-time temporal logic (ATL). In this framework we can devise formulas that characterize the coherence and transparency properties. Finally, by checking the validity of the ATL formulas within the abstracted structure, we can determine whether the program complies with the security policies and whether execution of a policy affects the original functionality. Together, the two
conclusions attest to the correctness of the program. This method can prove that programs comply with a wide range of security policies, not just safety policies.

This method establishes trust by policy adherence, and provides a semantic framework for certifying code on this basis. It represents a step forward from trusted code toward trustworthy code.

REFERENCES


GuoSun Zeng, born in 1964, received his Ph.D degree in computer science and technology from ShangHaiJiaoTong University, ShangHai, china, in 1999. Since 2000, He is a professor of institute of electronic and information engineering of Tongli University, ShangHai, China. His areas of research include software verification and security.

Li Li, born in 1977, received the B.S. degree and M.S. degree in computer application technology from XinJiang University, XinJiang, China in 2001 and 2004 respectively. Since 2006, she has been a Ph.D. Candidate for software theory from TongJi University. From 2006 to now, her main research interests include trusted computing and code safety.