CliSeAu: Securing Distributed Java Programs by Cooperative Dynamic Enforcement

Richard Gay, Jinwei Hu, and Heiko Mantel

Department of Computer Science, TU Darmstadt, Germany
{gay,hu,mantel}@mais.informatik.tu-darmstadt.de

Abstract CliSeAu is a novel tool for hardening distributed Java programs. CliSeAu takes as input a specification of the desired properties and a Java bytecode target program, i.e. the format in which Java programs are usually provided. CliSeAu returns hardened Java bytecode that provides the same functionality as the original code, unless this code endangers the desired properties. By monitoring the components of a distributed system in a decentralized and coordinated fashion, our tool CliSeAu is able to enforce a wide range of properties, both effectively and efficiently. In this article, we present the architecture of CliSeAu, explain how the components of a distributed target program are instrumented by CliSeAu, and illustrate at an example application how CliSeAu can be used for securing distributed programs.

1 Introduction

Dynamic enforcement mechanisms establish security at run-time by monitoring a program's behavior and by intervening before security violations can occur [1–3]. Dynamic enforcement mechanisms are often tailored to a particular purpose. For instance, authentication mechanisms ensure the authenticity of users, access-control mechanisms ensure that only authorized accesses can be performed, and firewalls ensure that only authorized messages can pass a network boundary. Besides such special-purpose security mechanisms, there are also dynamic enforcement mechanisms that can be tailored to a range of security concerns.

Our novel tool CliSeAu belongs to this second class of dynamic enforcement mechanisms. Given a Java bytecode target program and a policy that specifies a user's security requirements, CliSeAu enforces that the requirements are met.

In this respect, CliSeAu is very similar to two well known tools, SASI [4] and Polymer [5], and there are further similarities. Firstly, all three tools aim at securing Java bytecode.[1] Secondly, like in Polymer, security policies in CliSeAu are specified in Java. Thirdly, like Polymer, CliSeAu bases enforcement decisions on observations of a target program's actions at the granularity of method calls. Fourthly, like in Polymer, the possible countermeasures against

[1] There is a second version of SASI for securing x86 machine code.

Published in:
A. Prakash and R. K. Shyamasundar (Eds.): ICISS 2014, LNCS 8880, pp. 378–398, 2014.
© Springer-Verlag Berlin Heidelberg 2014
The original publication is available at www.springerlink.com

policy violations include termination of a target program, suppression or replacement of policy-violating actions, and insertion of additional actions. Finally, all three tools enforce policies by modifying the target program's code. Like SASI, CliSeAu performs this modification statically before a program is run.

A distinctive feature of CliSeAu is the support for enforcing security properties in distributed systems in a coordinated and decentralized fashion. CliSeAu generates an *enforcement capsule* (brief: *EC*) for each component of a distributed program. The granularity of encapsulated components is chosen such that each of them runs at a single *agent*, i.e. at a single active entity of a given distributed system. The local *ECs* at individual agents can be used to make enforcement decisions in a decentralized fashion. Decentralizing decision making in this way avoids the bottleneck and single point of failure that a central decision point would be. Moreover, localizing enforcement decisions increases efficiency by avoiding communication overhead. Purely local, decentralized enforcement, however, has the disadvantage that a smaller range of security properties can be enforced than with centralized enforcement decisions [6]. CliSeAu overcomes this disadvantage by supporting communication and coordination between *ECs*. If needed, enforcement decisions can be delegated by one *EC* to another. There are a few other tools that support decentralized, coordinated enforcement, and we will clarify how they differ from CliSeAu when discussing related work.

Another distinctive feature of CliSeAu is the technique used for combining *ECs* with components of a target program. Parts of the *EC* code are interwoven with the target program using the in-lining technique [4], which is used by SASI and Polymer. Other parts of the *EC* code are placed in a process that runs in parallel with the modified target program. This ensures responsiveness of an *EC*, even if its target program is currently blocked due to a pending enforcement decision or has been terminated due to a policy violation.

The enforcement of security properties with CliSeAu is both effective and efficient. In this article, we illustrate the use of CliSeAu at the example of distributed file storage services. As example policy, we use a Chinese wall policy [7]. This is a prominent example of a security policy that cannot be enforced in a purely local, decentralized fashion [6] and, hence, the communication and coordination between *ECs* is essential for enforcing this policy. We also provide results of an experimental evaluation using three different distributed file storage services as target programs. Our evaluation indicates that the performance overhead caused by CliSeAu is moderate. Preliminary reports on further case studies with CliSeAu in the area of social networks, version control systems, and e-mail clients can be found in the student theses [8], [9], and [10], respectively.

In summary the three main novel contributions of this article are the description of the architecture and implementation of CliSeAu (Sections 3 and 5), the explanation of how CliSeAu combines *ECs* with the components of a target program (Section 4), and the report on the case study and experimental evaluation with distributed file storage systems (Sections 6 and 7).

CliSeAu's source code is available under MIT License at `http://www.mais.informatik.tu-darmstadt.de/CliSeAu.html`.

2 Scope of Applications for CliSeAu

Programs are often developed without having a full understanding yet of the security concerns that might arise when these programs are used. Moreover, even if security aspects have been addressed during program development, a user of the program might not be convinced that this has been done with sufficient rigor. Finally, security requirements might arise from a particular use of a program, while being irrelevant for other uses. In general, it is rather difficult for software engineers to anticipate all security desires that forthcoming users of a program might possibly have. Moreover, being overly conservative during system design regarding security aspects is problematic because security features might be in conflict with other requirements, e.g., regarding functionality or performance and, moreover, can lead to substantial increases of development costs.

Hence, there is a need for solutions that harden programs for given security requirements. This was our motivation for developing CliSeAu as a tool that enables one to force properties onto existing, possibly distributed programs.

CliSeAu can be used by both software developers and software users. In order to apply CliSeAu, one must be able to express security requirements by a Java program (see Section 6 for more details on how this works) and the architecture of the distributed target program must be static.

The class of properties that can be enforced with CliSeAu falls into the class of safety/liveness properties [11, 12]. These are properties that can be expressed in terms of individual possible runs of a system, such that a property is either satisfied or violated by an individual program run. Security requirements that can be expressed by properties within this spectrum are, for instance, "A file may only be read by a user who is permitted to read this file." (confidentiality), "Only programs that are authorized to write a given channel may send messages on this channel." (integrity), and "A payment may only be released if different users from two given groups have confirmed the payment." (separation of duty). Security properties that are outside this spectrum are now commonly referred to as hyper-properties [13] and include, for instance, many information flow properties, as already pointed out in [14]. The limitation to properties falling into the safety/liveness spectrum is shared by many other generic tools for dynamic enforcement, including the aforementioned tools SASI and Polymer.

In order to enforce a given property, a dynamic enforcement mechanism needs certain capabilities. Firstly, it must be able to anticipate the next action of the target program. Secondly, it must be able to block this action until it is clear whether this action is permissible. Thirdly, it must be able to unblock the action – if the action is permissible – and to impose suitable countermeasures on the target program – if the next action would lead to a violation of the desired property. As mentioned before, CliSeAu encapsulates each component of a target program by an *EC*. Each of these *EC*s runs at a single agent and can observe, block, and unblock the method calls of the target's component that this *EC* supervises. An *EC* can also impose countermeasures on the supervised component. The implementation technique that CliSeAu uses for combining

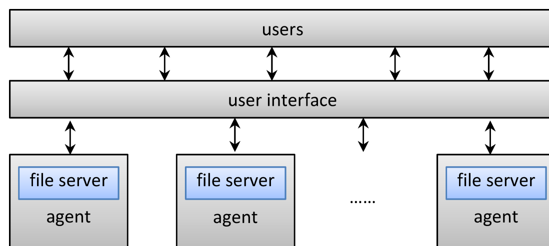

Figure 1: Architecture of a distributed file service.

a target program with the enforcement code ensures that each *EC* has these capabilities (see Section 4 for details on this technique).

Consider, for instance, a file storage service that provides large storage capacities to users. The functionality of a file storage service includes the uploading and downloading of files by users as well as the controlled sharing of files among users. A well known example of such a file storage service is DropBox.[2]

Figure 1 depicts the architecture of a distributed realization of a file storage service. The service is deployed on a collection of distributed machines, each of which hosts a file server program. Users interact with an interface that mitigates their input to the appropriate servers and that communicates the outputs of each file server to the respective users. The mapping of files to servers might be based on criteria like, e.g., geographic proximity in order to ensure low latency. The concrete mapping of files to servers might be hidden from the user.

CliSeAu can be used to secure such a distributed file storage service by encapsulating each file server program with an *EC*. Each *EC* is tailored to a security policy that captures the user's security requirements. An individual *EC* could be tailored, e.g., to a policy requiring that users access files stored at the supervised server only if they are authorized to do so. Moreover, an individual *EC* could also be used to control the sharing of files stored at the given server. However, there are also security requirements that cannot be enforced locally by an individual *EC*. This is the case when the *EC* does not have sufficient information to decide whether an action is permissible or not. For instance, if one wants to limit how much data a user shares within a particular time period then an *EC* needs to know how much data stored at other servers has been shared by this user. Another example are conflicts of interest, where if a user has accessed some file A then he must not access some other file B afterwards, even if in principle she is authorized to access both files. In order to decide whether file B may be accessed, an *EC* needs to know whether the user has already accessed file A at some other server. Conflicts of interest must be respected, e.g., within companies that work for other companies who are competitors.

In our case study and experimental evaluation, we show how CliSeAu can be used to prevent conflicts of interests, expressed by a Chinese Wall policy, in a distributed file storage service. The ability of CliSeAu's *EC*s to communicate

[2] https://www.dropbox.com/

with each other and to coordinate their actions is essentail for CliSeAu's ability to enforce such a Chinese wall policy in a decentralized fashion.

3 Design of CliSeAu

CliSeAu is designed in a modular fashion, following principles of object-oriented design [15,16]. The *ECs* generated by CliSeAu are modeled by UML diagrams that capture different views on CliSeAu. Design patterns employed in the design of CliSeAu include the factory pattern and the strategy pattern [16]. Objects are used to capture the actions of a target program as well as an *EC*'s reactions to such actions.

An *EC*'s reaction to an action is captured by a *decision object*, which corresponds to a particular decision of how to influence the behavior of the target. For example, decisions could range over a fixed set of decision objects that represent the *permission* or the *suppression* of a program action, the *insertion* of additional program actions, or the *termination* of the program. Decision objects can also be more fine-grained and specify further details such as the reason why a decision is made or how the program should be terminated.

An action of a target program is represented by an *event object*, which corresponds to a particular method call by the target program. The fields of such an object capture information about the method call, like actual parameters or the object on which the method is called. How much detail about a method call is stored in the corresponding object, can be chosen by a user of CliSeAu. This allows the user to abstract from details of a method call that are not relevant for her security requirements. For instance, one might represent the program action of sending a particular file by an event object that captures the name of the file and the identifier of the receiver in fields, while abstracting from other information like the name of the protocol by which the file shall be transferred.

In the following, we call an action of the target program (i.e., a method call) *security relevant* if its occurrence might result in a security violation. We also call an action security relevant if whether this actions occurs or doesn't occur affects whether possible future events are deemed security-relevant or not. When applying CliSeAu, one can specify the subset of a program's actions that are security relevant. CliSeAu exploits this information to choose which of a program's actions need to be guarded or tracked.

The abstraction from a target program's security-irrelevant actions and from security-irrelevant details of security-relevant actions both reduce conceptional complexity. This simplifies the specification of security policies and improves performance. Memory is needed only for storing the security-relevant details of security-relevant actions and only security-relevant actions needed to be supervised by CliSeAu.

In the following, we show different views on CliSeAu's *ECs*, focusing on the *ECs*' activities (Section 3.1), the high-level architecture (Section 3.2), and the parametric low-level architecture (Section 3.3). We conclude this section with the architecture of CliSeAu itself (Section 3.4).

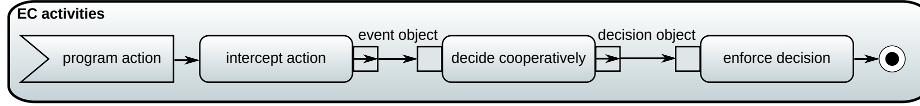

Figure 2: Activities of an *EC* (UML activity diagram)

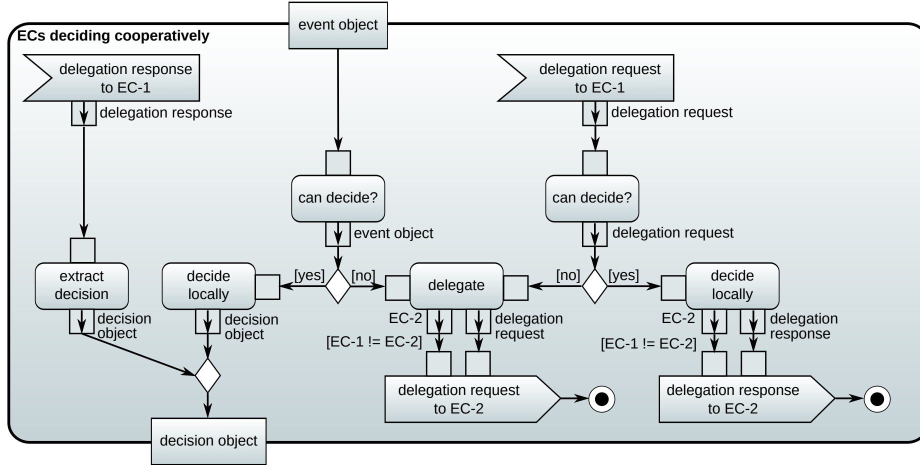

Figure 3: Cooperative deciding by *ECs* (UML activity diagram)

3.1 Activity View of an *EC*

At runtime, an *EC* performs three main activities (Figure 2): *intercepting* the next security-relevant actions of the program, *deciding* about such actions, and *enforcing* the decisions. Intercepting consists of observing the execution of a target program, blocking security-relevant actions until a decision about them has been made, and capturing the respective next security-relevant action by an event object. Enforcing consists either of enabling the currently blocked action of the target program or, alternatively, of forcing a countermeasure on the target.

The *ECs* generated by CliSeAu can cooperate with each other when making decisions. We capture the individual activities belonging to cooperative deciding in detail in Figure 3. Essentially, four cases can be distinguished:

1. locally making a decision for a locally intercepted event. This case occurs when an event object (top box) is given to the deciding activity by the intercepting activity of Figure 2 and the check whether the *EC* itself *can decide* succeeds. Then the *EC decides locally* and returns the result as a decision object (bottom box).
2. remotely making a decision for a locally intercepted event. This case also occurs when an event object is given to the deciding activity. However, in this case, the check whether the *EC* can decide fails and the *EC delegates* the decision-making to an *EC*-2 by signalling a *delegation request to EC-2*. This

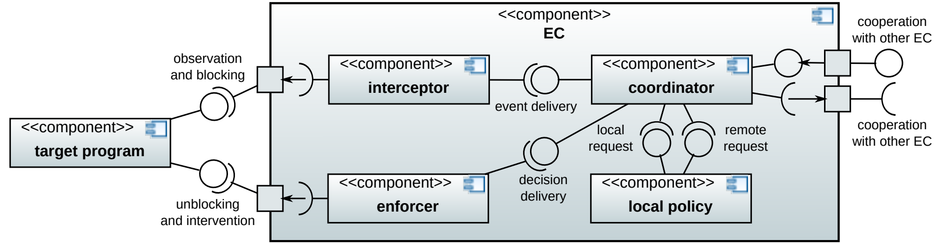

Figure 4: High-level architecture of CliSeAu's *ECs* (UML component diagram)

case leads to a decision object when a *delegation response to EC* (top left) is signalled and the *EC extracts the decision* from the delegation response.

3. locally making a decision for a remotely intercepted event. This case occurs when a delegation request is signalled to the *EC* and the check whether the *EC* itself *can decide* succeeds. Then the *EC* locally decides and signals the decision as a delegation response to an *EC*-2.
4. remotely making a decision for a remotely intercepted event. This case also occurs when a delegation request is signalled to the *EC*. However, in this case, the check whether the *EC* itself can decide fails and the *EC*-2 delegates the decision-making to an *EC*-2 by signalling a delegation request to *EC*-2.

Note that the cooperative deciding activity ends when the *EC* delegated the decision-making for an event object. That is, the activity does not block until a response is signalled, which enables the *EC* to cooperate with remote *ECs* in the meantime.

3.2 High-level Architecture of *ECs*

The high-level architecture of the *ECs* generated by CliSeAu follows the concept of *service automata* [6], according to which a service automaton is an *EC* that features a modular architecture consisting of four particular components: the interceptor, the coordinator, the local policy, and the enforcer. Each of the components is responsible to perform particular activities of the *EC* (see Section 3.1) and uses particular interfaces to interact with the other components of the *EC*. The UML component diagram in Figure 4 visualizes the high-level architecture of CliSeAu's *EC*.

In CliSeAu, the *interceptor* is a component that performs the activity of intercepting attempts of the program to perform security-relevant actions. Furthermore, its purpose is to generate event objects. The component requires an interface to the program by which it *observes and blocks* the program's attempts to perform actions. How this interface is established by combining the *EC* with the target program is the focus of Section 4. The component also requires an interface for the *event delivery*.

The *enforcer* of CliSeAu is a component that enforces decisions. The component provides an interface for the *delivery of enforcement decisions*. It requires an interface for *unblocking and intervening* the program execution.

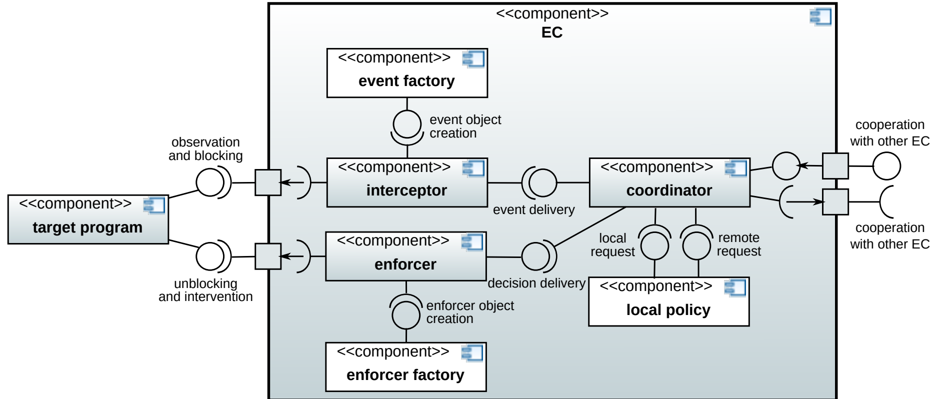

Figure 5: Low-level architecture of an *EC* (UML component diagram)

The *local policy* of CliSeAu is a component that performs the activities of (i) checking in which cases it can make a local decision, (ii) making local decisions, (iii) delegating the decision-making for events, and (iv) extracting decision objects from delegation responses. The component provides an interface for processing *local requests* – in the form of events capturing local program actions – as well as *remote requests* – in the form of delegation requests or delegation responses.

The *coordinator* of CliSeAu is a component that connects the components within one *EC* as well as *ECs* with each other. The component provides an interface for the delivery of events that are to be decided. It requires interfaces for the delivery of enforcement decisions, the delivery of local requests, and the delivery of remote requests. For the *cooperation with other ECs*, an *EC* provides one interface and requires one interface of each other *EC*.

3.3 Parametric Low-Level Architecture of *ECs*

CliSeAu provides a generic *EC*, that is, an *EC* that is parametric in the security policy that the *EC* enforces. The parametricity of CliSeAu's *ECs* is manifested in two kinds of entities: data structures (event objects, decision objects, delegation requests, and delegation responses) as well as active components. The refined architecture of *ECs* in Figure 5 pinpoints the parametric activities to three components (solid white boxes in the figure): the event factory (enabling parametric events), the enforcer factory (enabling parametric countermeasures), and the local policy (enabling parametric deciding and delegation). The remaining components (shaded boxes) are fixed by the *EC*.

Parametric events. An event object corresponds to a particular attempt of the target program to perform a security-relevant action. The concrete type of event objects can be chosen by the user of CliSeAu. That allows the user of CliSeAu

to choose how much detail about a program action is stored in the corresponding event object.

Closely related to event objects in an *EC* is the event factory component, a component that creates event objects. It encapsulates functionality that transforms the details of a program action to the content of an event object. The parametric event factory allows a user of CliSeAu to specify how the information that an event object captures is obtained from a concrete program action. The use of the factory design pattern [16] allows the *EC* architecture to integrate varying concrete event factories despite the fixed interceptor component.

For an example, consider again the actions of sending a file. To create the event objects for the action of sending a file, the factory could access the actual parameters to the method call of the action and read the needed information from the arguments. Suppose that two sending actions share the same file name and recipient, but differ in the protocol used for the file transfer (e.g., FTP vs. HTTP) and in the access time-stamp. In this case, the factory could transform the two actions to the same event object, which contains only the fields file name and recipient identifier.

Parametric deciding and delegation. A decision object corresponds to a particular decision of how to influence the behavior of the target program. The concrete type of decision objects can be chosen by the user of CliSeAu. That allows the user of CliSeAu to choose how much technical details about concrete countermeasures must be known when making decisions.

When an *EC* cannot make a decision about an event on its own, it delegates the decision-making to another *EC*. In this process, the *ECs* exchange delegation requests and delegation responses. A delegation request is an object that captures enough information for the delegate *EC* to make a decision or to further delegate the request. Analogously, a delegation response is an object that captures a decision for the receiving *EC*. The concrete types of delegation requests and delegation responses can be chosen by the user of CliSeAu. That allows the user of CliSeAu to choose what information is exchanged between the *ECs*. For instance, a delegation request corresponding to an event object could, in addition to the event object to be decided upon, carry partial information about the delegating *EC*'s state.

The local policy is the active component of an *EC* that encapsulates the *EC*'s functionality for deciding and delegating decision-making. The parametric local policy allows a user of CliSeAu to specify for the respective security policy how decisions shall be made and when cooperation between *ECs* shall take place. The use of the strategy design pattern [16] allows the *EC* architecture to integrate varying concrete local policies despite the fixed coordinator component.

Parametric countermeasures. CliSeAu captures countermeasures, i.e., actions that the *EC* performs to prevent security violations, by *enforcer objects* in the *EC*. An enforcer object is an object that encapsulates concrete code whose execution results in actions that prevent the security violations. For instance, an enforcer object can encapsulate code for suppressing an action of the program,

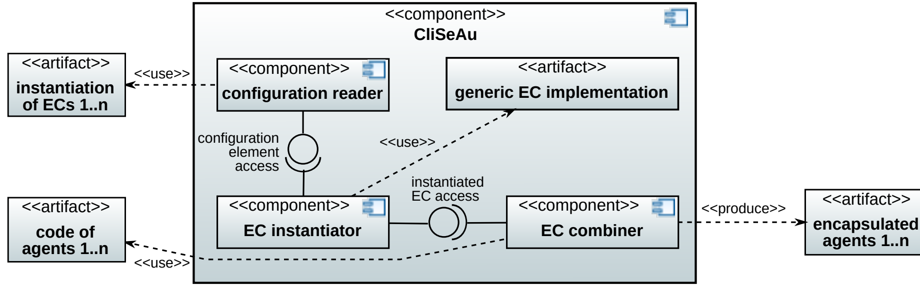

Figure 6: Architecture of CliSeAu (UML component diagram)

for terminating the program, for replacing an action of the program by other actions like the display of an error message, or for allowing an action to execute. The concrete set of possible enforcer objects can be chosen by the user of CliSeAu. This allows one to tailor countermeasures to the concrete target.

Closely related to decision objects and enforcer objects inside an *EC* is the enforcer factory component. An enforcer factory is an object that generates enforcer objects. The factory takes a decision object as input and returns enforcers whose execution could achieve the effect intended by the decision. Being a parametric component, the enforcer factory allows a user of CliSeAu to tailor which decisions result in which countermeasures specificly for the application scenario. For example, for a decision object "permit", the factory could return an enforcer that permits the execution of program actions. For the decision "terminate", the factory could return an enforcer that executes "`System.exit(1)`".

Instantiation of parameters. The parametric components of an *EC* must be *instantiated*, i.e., substituted by concrete instances of the components, before the *EC* can be used for enforcing a concrete security policy on a concrete target. That is, before one can use CliSeAu, one must define a concrete instance for each parametric component of the *EC*.

When using CliSeAu to enforce a security policy in a distributed target program, one must instantiate an *EC* for each of the target's agents. All *ECs* for the target must share at least the same instantiation of the delegation request and delegation response objects, such the *ECs* can cooperate with each other. The remaining parameters of the *ECs* can be instantiated tailored to the respective agents of the target. For instance, decisions may be enforced differently at the individual agents. In this case, the instantiated *ECs* would comprise different enforcer factories and enforcer objects. However, for a distributed target consisting of replications of one and the same agent, all parameter instances may also be shared among the individual *ECs*.

3.4 Architecture of CliSeAu

Figure 6 shows the high-level architecture of CliSeAu. This architecture consists of three main components. The *configuration reader* consumes as input

the instantiations of the *EC* parameters. It passes these parameters to the *EC* instantiator. The *EC instantiator* takes CliSeAu's generic *EC* as well as the parameters of the *EC* and instantiates the *EC* based on the parameters. The *EC combiner* takes an instantiated *EC* as well as the code of an agent and combines the two. This combination establishes the interfaces between the program and the interceptor and enforcer that are described in Section 3.2. The technique for combining the code of an agent with an *EC* is described in Section 4.

When using CliSeAu to enforce a security policy on a distributed target, each agent of the distributed target is combined with an *EC*. This *EC* intercepts the security-relevant events of the agent, participates in cooperatively deciding for security-relevant events, and enforces decisions on the agent. To enforce a security policy on a distributed target, CliSeAu therefore takes instantiations of the *EC* parameters for *each* agent and all agents of the target as input.

The result of applying CliSeAu to a distributed target with a given instantiation of the *ECs* is a set of encapsulated agents. The *ECs* at the encapsulated agents then together enforce the security policy encoded by the instantiation.

4 Technique for Combining *ECs* with Targets

For combining the components of a distributed target program with *ECs* generated by CliSeAu, CliSeAu applies a technique that we describe in the following. The combination of an *EC* with an agent consists of two parts: rewriting the code of the agent as well as creating a separate program that shall, at the runtime of the agent, run in parallel to the agent.

Rewriting the code of the agent serves the purpose of making security-relevant program action's guarded. Being guarded here means that the *EC* makes a check against the policy before the action occurs and runs a countermeasure against the action in case the action would violate the policy. For this rewriting, CliSeAu takes as input a specification of the security-relevant program actions.

As part of the rewriting, CliSeAu places code into the agent that corresponds to the *EC* components for intercepting and the acting. That is, the interceptor, the event factory, the event objects, the enforcer, the enforcer factory, the enforcer objects, and the decisions objects are placed into the code of the agent. In the rewritten code of the agent, the code of the interceptor is placed before each security-relevant action and the enforcer is placed "around" the action in the style of a conditional. The remaining components are added to the code of the agent at a place where they can be invoked by the interceptor or enforcer executor.

The separate program that is created by CliSeAu as part of combining an *EC* with an agent covers the *EC* components for deciding. That is, the separate program contains the coordinator, the local policy, the delegation request/response, and the event and decision objects.[3]

[3] CliSeAu provides a base implementation of the delegation request/response; one can also supply one's own implementation.

Placing the interceptor and the enforcer executor into the code of the agent as guards of code for security-relevant actions serves the purpose of enabling an efficient enforcement of security policies that are expressed at the level of program actions. Alternatives such as intercepting and enforcing within the operating system may allow the *ECs* to enforce the same security policies but incurs overhead for reconstructing program-level actions from operating system-level actions.

Placing the event factory and the enforcer factory into the code of the agent mainly serves the purpose of efficiency: event objects are supposed to be smaller in size than all agent's data related to a program action (e.g., large data structures on which the action operates); hence, transmitting an event object from the agent to the separate program requires less time. A similar argument applies to the placement of the enforcer factory, because decision objects are smaller in size than the enforcer objects. A beneficial, more technical side-effect of this placement of the factory components is that this placement eliminates or reduces the dependency of the separate program on agent-specific data structures.

Placing the coordinator and the local policy into a separate program serves the purpose of effectiveness and efficiency: the separate program runs in parallel to the agent; hence, it remains responsive even when the agent is blocked due to a pending enforcement decision or has been terminated due to a policy violation. More concretely, even in these cases the separate program can receive delegation requests and make decisions or delegate further with the coordinator and local policy. If a blocked agent would delay the operation of the coordinator and local policy, this would impact the efficiency of the enforcement. Worse, if a terminated agent would prevent the coordinator and local policy from operating, then this could prevent an effective enforcement in cases when cooperation is required to make precise decisions.

5 Implementation

The CliSeAu implementation consists of two parts: an implementation of generic *ECs* following the architecture in Figure 5, called SeAu, and a command-line tool following the architecture in Figure 6, called Cli, for instantiating *ECs* and combining the instantiated *ECs* with target agents. The SeAu implementation consists of Java classes for the fixed components of CliSeAu's *ECs* and Java interfaces and abstract classes for the parametric components. Cli takes as input the instantiation of the SeAu *ECs* in the form of a configuration file and produces instantiated *ECs*. An example configuration is given in Section 6. For modifying the Java bytecode of the agents of a target program according to the technique presented in Section 4, Cli uses AspectJ [17] as a back-end.

6 Case Study

We have applied CliSeAu to a distributed file storage service.We built the service by ourselves, following the architecture of distributed programs depicted in Figure 1 of Section 2. Our service uses off-the-shelf file servers: DRS [18],

(a) actions, events and event factory

```
1 pointcut FileAccess(ftpdControl control, File file > boolean):
2   call(boolean eventDownloadFilePre(File))
3     && target(control) && args(file);
4 class AccessEvent implements AbstractEvent {
5     String user, company, COI; }
6 class AccessEventFactory {
7   AccessEvent fromFileAccess(ftpdControl control, File file) {
8     return new AccessEvent(getUser(control),
9       getCompany(file), getCOI(file)); } }
```

(b) local policy

```
class ChineseWallPolicy extends LocalPolicy {
  PolicyResult decideEvent(AccessEvent event) {
    if (locallyResponsibleFor(event)) return getChineseWallDecision(event);
    else return new Delegation(whoIsResponsible(event), event); } }
```

(c) decisions and enforcer factory

```
1 enum BinaryDecision implements AbstractDecision { PERMIT, REJECT }
2 class SuppressionEnforcerFactory implements EnforcerFactory{
3   Enforcer fromDecision(final AbstractDecision d) {
4     BinaryDecision bd = (BinaryDecision) d;
5     switch (bd.decision) {
6       case PERMIT: return new PermittingEnforcer();
7       case REJECT: return new SuppressingEnforcer(); } } }
```

(d) CLISEAU configuration

```
1 cfg.agents              = srv1, srv2, srv3, ...
2 srv1.code               = AnomicFTPD.jar
3 srv1.address            = srv1.example.com
4 srv1.localPolicy        = ChineseWallPolicy
5 srv1.pointcuts          = FileAccess.pc
6 srv1.eventFactory       = AccessEventFactory
7 srv1.enforcerFactory    = SuppressionEnforcerFactory
8 # parameters for srv2, ... are defiend similarly and are omitted here
```

Figure 7: An instantiation of CLISEAU

AnomicFTPD [19], and Simple-ftpd [20]. Function-wise, our service allows users to upload, download, and share files.

Security-wise, our service only supports user authentication. However, other more specific security requirements may also arise. Consider for instance that the storage service is used in an enterprise setting like in a bank. According to [21], an employee of the bank may not access files from the bank's two client companies that have conflicts of interests. In general, such a requirement is captured by *Chinese Wall policies* [7]: no single user may access files that belong to two companies bearing conflicts of interests.

To enforce a security requirement that is not obeyed in our service, like a Chinese Wall policy, we need to employ some security mechanism. CLISEAU can be used to generate such a mechanism by performing the following steps: (1) define security-relevant actions, event objects, and event factory; (2) define

the *ECs*' local policies; (3) define decision objects, enforcer objects, and enforcer factory; (4) assemble the above to a configuration for CliSeAu.

Following these 4 steps, we actually provide an instantiation for CliSeAu. In turn, CliSeAu uses this instantiation to generate *ECs* and combine them with the file servers of our service. In this way, our service is hardened with the enforcement of the Chinese Wall policy. Now we explain in detail how to construct an instantiation for our service with AnomicFTPD as the file servers.

Security-relevant actions, event objects, and event factory. In order to enforce a security requirement on a program, we first identify the program's security-relevant actions. For the Chinese Wall policy that we want to enforce on our service, the actions are method calls whose execution corresponds to users' file accesses. For the AnomicFTPD file server, we find that file download boils down to a call of the method eventDownloadFilePre of an ftpdControl class with a File parameter. Therefore we use the pointcut in Figure 7 (a) (Lines 1-3) to specify that calls of the method eventDownloadFilePre are security-relevant and shall be intercepted.[4] From an intercepted method call, an event object shall be created. Observe that the Chinese Wall policy shall define which company each file belongs to and a COI (conflicts-of-interests) relation on the set of companies. As such, the event object should capture the user who attempts to access a file, the company that the file belongs to, and the involved COI relationships. Figure 7 (a) (Lines 4-5) shows the event object AccessEvent, which has three fields: user, company and COI. In order to construct AccessEvents, we use the AccessEventFactory in Figure 7 (a) (Lines 6-9). In this event factory, the fromFileAccess method uses the actual parameters of the intercepted method call and extracts the needed information to create an object of AccessEvent.

Local policy. Next we define the local policy component of an *EC*. The local policy shall make decisions about security-relevant actions and about delegation of decision-making. For our service, we construct the local policy component named ChineseWallPolicy in Figure 7 (b). ChineseWallPolicy checks whether it should decide upon an input event by the method locallyResponsibleFor. If it is the case, then a decision is computed by the method getChineseWallDecision. Otherwise ChineseWallPolicy delegates the decision-making for the event to a remote *EC* by the method whoIsResponsible. We implement the methods locallyResponsibleFor and whoIsResponsible with the guarantee that accesses to conflicting files are always decided by the same *EC* (i.e., its local policy component). The implementation of getChineseWallDecision checks whether or not a user trying to access a file has already accessed a conflicting file before. A method for deciding delegation requests is defined analogously to decideEvent and thus omitted here.

[4] The security-relevant actions depend on the interpretation of "access": users access files by (1) only downloading them or (2) by either downloading or uploading. Figure 7 (a) (Lines 1-3) is defined for case (1). In case (2), we could define a similar specification but with the pointcut extended to match method calls for file uploads.

Decision objects, enforcer objects, and enforcer factory. We choose a decision for enforcing the Chinese Wall policy to be either permitting an access or rejecting the access; the BinaryDecision object in Figure 7 (c) (Line 1) captures this choice of decision. Corresponding to the two decision values are two enforcers: PermittingEnforcer and SuppressingEnforcer, which allows an intercepted method call to execute and suppresses the call, respectively. These two enforcers are provided by CliSeAu. The SuppressionEnforcerFactory of Figure 7 (c) (Lines 2-7) turns a reject decision into a SuppressingEnforcer object and a permit decision into a PermittingEnforcer object.

Configurations. Finally, we provide the configuration in Figure 7 (d) for CliSeAu. The configuration declares which agents exit (Line 1), which programs the agents run (Line 2), and the agents' addresses (Line 3). The configuration also assembles references to the previously described parts of the instantiation (Lines 4-7).

Figure 7 constitutes an instantiation of CliSeAu, which hardens our file storage service with a system-wide enforcement of Chinese Wall policy. CliSeAu allows us to address individually the aspects of an instantiation: how to intercept security-relevant actions, how to decide and possibly delegate, and how to enforce decisions. When using CliSeAu, we can focus on these aspects and, for instance, we need not be concerned about exchanging messages between distributed *ECs* or instrumenting the executables of the program. In particular, the deciding can be defined at a more abstract level (here based on AccessEvents) than the level of program actions (here involving the program-specific data type ftpdControl).

7 Performance Evaluation

Securing a program with CliSeAu necessarily results in some reduction on the program's runtime performance. Our evalution focuses on the run-time overhead of the enforcement.

Experimental setup. We evaluated CliSeAu with the distributed file storage service introduced in Section 6. The service has 3 variants, depending on which file servers are used. In our experiments, the service consisted of 10 file servers, all of which are either DRS [18], AnomicFTPD [19], or Simple-ftpd [20]. We refer to them as *DRS service*, *AnomicFTPD service*, and *Simple-ftpd service*, respectively. We used CliSeAu to enforce the Chinese Wall policy (i.e., no single user may access files bearing conflicts of interest), as described in Section 6.

We conducted all experiments on a 2.5 GHz dual CPU laptop running Gentoo Linux with Kernel 3.6.11, OpenJDK 6, and AspectJ 1.6.12. All servers of each service were run on the same machine. We chose this setup because in our experiments, we are interested in the overhead introduced by the implementation of CliSeAu. By using local network connections, we factor out the overhead introduced by a real network, as this overhead originates from CliSeAu-independent aspects such as network topology and network load distribution. We leave experiments in real network settings as future work.

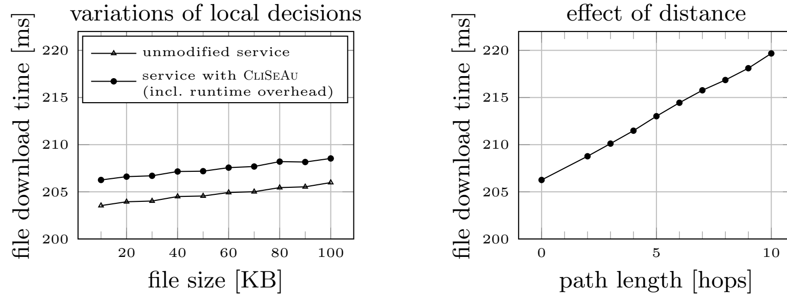

Figure 8: AnomicFTPD runtime evaluation

Run-time overhead. We evaluated the impact of CliSeAu on the services' run-time performance of file downloads from the perspective of a service user: we measured the duration from the moment the user made a file download request till the moment the user obtained the file. For DRS, we used a modified DRS client to access files and measured inside the client the durations of the accesses; the modification was done to measure the time. For AnomicFTPD and Simple-ftpd, we accessed files and measured durations with a self-written FTP client implemented based on the Apache Commons Net library. In both clients, time was taken using the System.nanoTime API method of Java.

For each service, we varied both the size of the requested files and the number of hops taken in the cooperation between the *ECs* for making enforcement decisions. Figures 8–10 show the results, which are averaged over the measurements of 2500 independent experiments.

The diagram on the left-hand side of Figure 8 shows the absolute time required for downloading files of different size from the AnomicFTPD service. With the unmodified service, downloads took from about 203.5 ms to 206 ms, depending on the file size. As Figure 8 (lhs) shows, the time is roughly linear in the file size. On the other hand, the service secured with CliSeAu used time ranging from about 206 ms to 208.5 ms. Still, download time remains linear in the file size (see Figure 8, lhs). As Figure 9 (lhs) shows, the absolute runtime overhead caused by CliSeAu ranged from about 2.5 ms to 2.75 ms. This corresponds to a relative overhead of about 1.3%. We consider this performance overhead reasonable for the security enforcement it is traded for.

We conducted the same experiments on the DRS service and the Simple-ftpd service as on the AnomicFTPD service. Figure 9 shows the results. The absolute overhead is less than 3 ms. For Simple-ftpd and AnomicFTPD, the overhead is roughly constant regardless of the changes in file size. For DRS, the overhead was relatively more unstable; the reason for this remains unclear to us. Still, the variation stays in a limited range from 1.9 ms to 2.8 ms.

Figure 8 (rhs) shows the absolute time required for downloading files of 10 kilobytes from the AnomicFTPD service when varying the number of hops taken in the cooperation between the *ECs*. We obtained up to 10 hops by letting the local policy implementations to not directly delegate to the responsible *EC* but

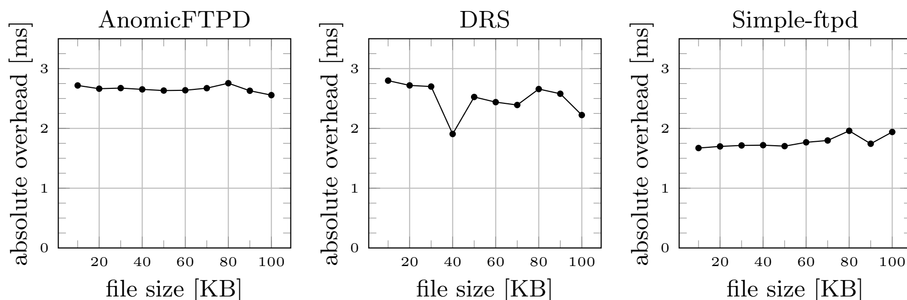

Figure 9: Absolute runtime overhead for different file sizes

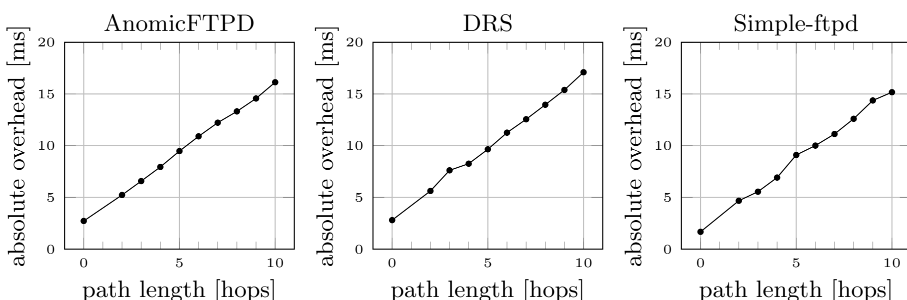

Figure 10: Absolute runtime overhead for different path lengths (effect of distance)

delegate to a number of other *ECs* before. This setting reflects the cases where responsible *ECs* are not directly reachable from a delegating *EC* or where more than one *EC* share the information for deciding about an event. In our experiments, the download time ranged from about 206 ms to 220 ms, corresponding to overhead between about 2.7 ms and 16.1 ms (see Figure 10, lhs). The overhead grows almost linearly with the number of hops at approximately 1.34 ms per hop. Our experiments with DRS and Simple-ftpd show very similar results; see the diagrams in the middle of and on the right hand side of Figure 10, respectively.

Summary. The *ECs* gengerated by CLISEAU caused moderate runtime overhead for our file storage service: For file download, the overhead was about 3 ms when the *ECs* could make decisions locally. When coopeerative decision-making was needed, the overhead increased linearly with the number of the hops between the *ECs* involved in the cooperation. This linearity is encouraging for deploying CLISEAU-generated *ECs* in a real-world setting like the Internet where a distributed program may have a larger number of agents and thus of hops for cooperation among *ECs*.

8 Related Work

As described in the introduction, CliSeAu follows the line of SASI [4] and Polymer [5]. SASI is a tool for generating *ECs* for Java bytecode programs as well as for x86 executables. The *ECs* generated by SASI either permit the occurrence of a security-relevant action or terminate the target otherwise. CliSeAu also allows one to specify enforcers that use termination as a countermeasure. However, CliSeAu additionally allows one to specify countermeasures corresponding to the suppression, insertion or replacement of security-relevant actions.

Polymer is a tool for generating *ECs* for Java bytecode programs. The policies that a user provides to Polymer can define so-called abstract actions, Java classes whose instances can match a set of program instructions. Furthermore, Polymer allows a policy to be composed from several subordinate policies; in such a composition, the superior policy queries its subordinate policies for their policy suggestions and combines these to obtain its own suggestion. Only when a suggestion is accepted, its corresponding countermeasure is executed. For the countermeasures, Polymer supports the insertion and replacement of actions, throwing a security exception, as well as to termination of the target. Considering a non-distributed setting, Polymer and CliSeAu support the same observable program operations (method calls), the same expressiveness in the decision-making (Java code), and the same kind of countermeasures. Therefore, the class of properties enforceable with CliSeAu is the same as for Polymer. Conceptually, Polymer's abstract actions are very similar to the combination of CliSeAu's event objects and event factories. Polymer's suggestions, in turn bear a similarity to CliSeAu's decision objects. However, in Polymer the layer between suggestions and their corresponding countermeasures serves the purpose of policy composition while CliSeAu's layer between decision objects and countermeasures (as enforcer objects) reduces the dependency between the local policy and technical details of the target program.

Further tools for generating *ECs* for Java bytecode programs include, for example, JavaMOP [22]. A particular characteristic of JavaMOP is the generation of efficient *ECs* for properties on parametric program actions. The focus of JavaMOP's efficiency efforts is enforcing properties on individual Java objects of the target program, which are realized by binding the objects of the target program to individual monitors for the decision-making. In contrast, with CliSeAu, we sacrifice this kind of optimization for the sake of an abstraction layer that maps program entities to entities at the policy-level. The latter shall then be usable by a distributed *EC* in the decision-making for system-wide security requirements.

Tools that are specifically tailored to distributed systems include, for example, Moses [23], DiAna [24], and Porscha [25]. Moses is a tool for dynamic enforcement for distributed Java programs. Technically, Moses is implemented as a middleware that is to be used by the agents of target programs for the coordination among themselves. Moses aims at enforcing properties, called laws, on the coordination between agents. The policies of Moses enforce such properties at the level of agent communication by delivering, blocking, or modifying exchanged messages. CliSeAu differs from Moses in two main directions. First, CliSeAu

can intercept and intervene not only communication operations of agents but also computation operations of a single agent, like the file accesses in our example of a distributed file storage service. Second, CliSeAu can be applied to arbitrary Java programs and does not rely on the program to be built upon a particular middleware. This allows CliSeAu to enforce security requirements also on programs that have not been designed with an enforcement by CliSeAu in mind, such as the targets of our experimental evaluation (Section 7).

DiAna is a tool for monitoring temporal properties on the state of distributed Java programs. These programs are assumed to be built on a monitoring library. In this sense, DiAna is similar to Moses. DiAna's *ECs* intercept the communication operations between the agents of the target and exchange information among each other by piggy-backing the information on the messages exchanged between the agents. That is, DiAna's *ECs* perform coordinated decentralized monitoring. CliSeAu differs from DiAna in two main directions: first, CliSeAu does not rely on the target to be built upon a particular library and, second, a *EC* generated by CliSeAu is able to intercept and coordinate its enforcement for program actions beyond agent communication.

Porscha [25] is an *EC* for enforcing digital rights management policies on Android smart phones. Porscha *ECs* are placed in the runtime environment of the target. Also, the *ECs* exchange information about the policy that affects the data exchanged by the agents of the target. Porscha and CliSeAu have in common that they support coordinated decentralized enforcement. However, a key difference is that the *ECs* generated by CliSeAu can coordinate their enforcement by themselves, without relying on the intercepted communication actions of agents. That is, the *ECs* communicate and cooperate in a proactive way, regardless of whether and when the agents of a distributed program communicate with each other. This allows the *ECs* to enforce security in the scenario of Section 6, in which cooperation is required also for file access events that do not correspond to data exchange between agents of the storage service.

9 Conclusion

We presented the tool CliSeAu for securing distributed Java programs. CliSeAu uses cooperative dynamic mechanisms to enforce system-wide security requirements and allows to instantiate the mechanism for different programs and security requirements. We showed a case study of CliSeAu on a distributed file storage service and performed experimental evaluation on the example service. The experimental results demonstrate that the enforcement mechanisms provided by CliSeAu incur moderate runtime overhead.

Acknowledgments We thank Sarah Ereth, Steffen Lortz, and Artem Starostin for their feedback on our research. We are also grateful to Cédric Fournet and Joshua Guttman for inspiring discussions at early stages of our research project. This work was partially funded by CASED (`www.cased.de`) and by the DFG (German research foundation) under the project FM-SecEng in the Computer Science Action Program (MA 3326/1-3).

References

[1] Schneider, F.B.: Enforceable Security Policies. Transactions on Information and System Security **3**(1) (2000) 30–50
[2] Fong, P.W.L.: Access Control By Tracking Shallow Execution History. In: IEEE Symposium on Security and Privacy, IEEE Computer Society (2004) 43–55
[3] Ligatti, J., Bauer, L., Walker, D.: Edit Automata: Enforcement Mechanisms for Run-time Security Policies. IJIS **4**(1–2) (2005) 2–16
[4] Erlingsson, U., Schneider, F.B.: SASI Enforcement of Security Policies: A Retrospective. In: Proceedings of the 2nd NSPW, ACM (2000) 87–95
[5] Bauer, L., Ligatti, J., Walker, D.: Composing Expressive Runtime Security Policies. Transactions on Software Engineering and Methodology **18**(3) (2009)
[6] Gay, R., Mantel, H., Sprick, B.: Service Automata. In: Proceedings of the 8th International Workshop on Formal Aspects of Security and Trust. LNCS 7140, Springer (2012) 148–163
[7] Brewer, D.F., Nash, M.J.: The Chinese Wall Security Policy. In: Proceedings of the IEEE Symposium on Security and Privacy. (1989) 206–214
[8] Mazaheri, S.: Race conditions in distributed enforcement at the example of online social networks. Bachelor thesis, TU Darmstadt (2012)
[9] Scheurer, D.: Enforcing Datalog Policies with Service Automata on Distributed Version Control Systems. Bachelor thesis, TU Darmstadt (2013)
[10] Wendel, F.: An evaluation of delegation strategies for coordinated enforcement. Bachelor thesis, TU Darmstadt (2012)
[11] Lamport, L.: Proving the Correctness of Multiprocess Programs. IEEE Transactions on Software Engineering **3**(2) (1977) 125–143
[12] Alpern, B., Schneider, F.B.: Defining Liveness. Information Processing Letters **21** (1985) 181–185
[13] Clarkson, M.R., Schneider, F.B.: Hyperproperties. Journal of Computer Security **18**(6) (2010) 1157–1210
[14] McLean, J.D.: Security Models. In Marciniak, J., ed.: Encyclopedia of Software Engineering. John Wiley & Sons, Inc. (1994)
[15] Booch, G., Maksimchuk, R.A., Engle, M.W., Young, B.J., Connallen, J., Houston, K.A.: Object-oriented Analysis and Design with Applications. third edn. (2007)
[16] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)
[17] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In: Proceedings of the 15th ECOOP. LNCS 2072, Springer (2001) 327–353
[18] : DRS. `http://www.octagonsoftware.com/home/mark/DRS/` (1999)
[19] : AnomicFTPD v0.94. `http://anomic.de/AnomicFTPServer/` (2009)
[20] : simple-ftpd. `https://github.com/rath/simple-ftpd` (2010)
[21] : PUBLIC LAW 107 - 204 - SARBANES-OXLEY ACT OF 2002
[22] Chen, F., Roşu, G.: MOP: An Efficient and Generic Runtime Verification Framework. In: Proceedings of the 22nd OOPSLA, ACM (2007) 569–588
[23] Minsky, N.H., Ungureanu, V.: Law-governed Interaction: a Coordination and Control Mechanism for Heterogeneous Distributed Systems. ACM Transactions on Software Engineering Methodology **9**(3) (2000) 273–305
[24] Sen, K., Vardhan, A., Agha, G., Roşu, G.: Efficient Decentralized Monitoring of Safety in Distributed Systems. In: Proceedings of the 26th ICSE. (2004) 418–427
[25] Ongtang, M., Butler, K.R., McDaniel, P.D.: Porscha: Policy Oriented Secure Content Handling in Android. In: ACSAC. (2010) 221–230

```
config      ::= keyvalue | config keyvalue
keyvalue    ::= cfg.agents = agentnames | agentname.address = DomainOrIP
              | agentname.classkey = JavaClassName | agentname.filekey = FileName
agentnames ::= agentname | agentnames , agentname
classkey    ::= eventFactory | enforcerFactory | localPolicy
filekey     ::= intActComp | decComp | code | pointcuts
```

Figure 11: Syntax of CliSeAu configurations in BNF

```
1  interface AbstractEvent extends Serializable {}
2  interface AbstractDecision extends Serializable,PolicyResult {}
3  interface DelegationReqResp extends Serializable,PolicyResult {}
4  interface EventFactory {}
5  interface EnforcerFactory {
6    static Enforcer fromDecision(AbstractDecision ad); }
7  interface Enforcer {
8    boolean suppress();
9    void before();
10   void after();
11   Object getReturnValue(Class c); }
12 abstract class LocalPolicy {
13   /* fields, getters, setters, and constructor omitted */
14   abstract PolicyResult decideEvent(AbstractEvent ev);
15   abstract PolicyResult decideRequest(DelegationReqResp dr); }
```

Listing 1: Interfaces for the Java parameters of CliSeAu configurations

A CliSeAu Implementation Details

In this section, we provide additional details about the implementation of CliSeAu. This serves as a reference for the configuration of CliSeAu in concrete settings.

The syntax of configuration files is shown in Figure 11. The `cfg.agents` key declares a list of unique names for agents. Each *EC* of an agent is configured by eight keys, prefixed by the instance's name: (1) `address` defines the network address through which an *EC*'s coordinator can be reached by other *ECs*; (2) `eventFactory` defines the class name of the event factory; (3) `enforcerFactory` defines the class name of the enforcer factory; (4) `localPolicy` defines the class name of the local policy; (5) `pointcuts` specifies the name of a file, in which security-relevant actions are specified; (6) `intActComp` specifies the name of a JAR file that holds the compiled code of the parametric components for intercepting and acting; (7) `decComp` specifies the name of a JAR file that holds the compiled code of the parametric components for deciding; (8) `code` specifies the name of a JAR file that holds the bytecode of the agent's original implementation.

The parametric components of SeAu (see Figure 5) must be Java classes implementing the interfaces and base classes of Listing 1. AbstractEvents, AbstractDecisions, and DelegationReqResps – i.e., the datatypes exchanged between the components,

must implement the Serializable interface as they are serialized for transmission via network connections. An EventFactory must provide one factory method for each method whose call corresponds to a security-relevant action of the target agent. An EnforcerFactory uses a single factory method, fromDecision, for producing enforcers corresponding to decisions. An Enforcer must implement whether a security-relevant action shall be suppressed, must specify code to execute before and after the security-relevant action, and must provide an alternate return value for suppressed method calls. Instantiations of the local policy parameter must extend the abstract LocalPolicy class of Listing 1. That is, a local policy component of an *EC* must implement two methods: The decideEvent method takes an event object and must return a decision object or a delegation request (both derived from PolicyResult). The decideRequest method takes a delegation request or delegation response and must return a decision object, a delegation request, or a delegation response.