# A flexible security architecture to support third-party applications on mobile devices

Dries Vanoverberghe, Pieter Philippaerts, Lieven Desmet, Wouter Joosen, Frank Piessens
DistriNet Research Group
Katholieke Universiteit Leuven, Belgium

Katsiaryna Naliuka, Fabio Massacci
Università di Trento, Italy

## ABSTRACT

The problem of supporting the secure execution of potentially malicious third-party applications has received a considerable amount of attention in the past decade. In this paper we describe a security architecture for mobile devices that supports the flexible integration of a variety of advanced technologies for such secure execution of applications, including run-time monitoring, static verification and proof-carrying code. The architecture also supports the execution of legacy applications that have not been developed to take advantage of our architecture, though it can provide better performance and additional services for applications that are architecture-aware. The proposed architecture has been implemented on a Windows Mobile device with the .NET Compact Framework. It offers a substantial security benefit compared to the standard (state-of-practice) security architecture of such devices, even for legacy applications.

## 1. INTRODUCTION

Mobile phones and PDA's have evolved over the past years to become general purpose computation platforms. Many of these devices support downloading third party applications built on either the .NET Compact Framework, or Java Micro Edition. However, supporting applications from potentially untrustworthy sources comes with a serious risk: malicious or buggy applications on a phone can lead to denial of service, loss of money, leaking of confidential information on the device and so forth.

Current devices already provide certain countermeasures against these threats, with support for sandboxing and code signing. The key idea is that unsigned code is severely limited in what it can do on the device, i.e. it runs in a strict sandbox. Code that is signed by a trusted party can break out of the sandbox. The device has a keystore that can be configured with the public keys of trusted parties.

This security model has a number of serious shortcomings. First, it is not flexible: applications either run in a restricted sandbox, or have full power. Second, there is no clear meaning associated with the signatures of trusted third parties: a signature means the application is "well-behaved", but there is no clear definition of what this means. Hence, device owners trust the third party both for (a) appropriate vetting of applications, and (b) using a suitable notion of good behavior. Incidents [10] show that the current security model is inappropriate.

The project *Security of Software and Services for Mobile Systems (S3MS) [11]* is a research project under the 6th Framework Programme of the European Commission that addresses the shortcomings of the current security model, by integrating a variety of existing and newly-developed security technologies into all the phases of the mobile applications lifecycle. In this paper, we describe the architecture of the run-time environment on the mobile device. It has been developed in the context of the S3MS project.

A key ingredient in the S3MS approach is the notion of "security-by-contract" to protect mobile applications . Mobile applications can possibly come with a *security contract* that specifies their security-relevant behavior. Technically, a contract is a security automaton in the sense of Schneider and Erlingsson [3], and it specifies an upper bound on the security-relevant behavior of the application: the sequences of security-relevant events that an application can generate are all in the language accepted by the security automaton. Mobile devices are equipped with a *security policy*, a security automaton that specifies the behavior that is considered acceptable by the device owner. The key task of the S3MS device run-time environment is to ensure that all applications will comply with the device security policy. To achieve this, the run-time can make use of the contract associated with the application (if it has one), and of a variety of policy enforcement technologies. This paper describes the architecture, design and implementation of this run-time environment, and discusses its advantages and disadvantages with respect to the current security model for mobile devices.

The remainder of the paper is structured as follows. Section 2 provides some background information on policy languages and on policy enforcement techniques. Next, our flexible security architecture is presented in section 3, and section 4 highlights some of the design decision in our prototype implementation. In section 5, the advantages and disadvantages of the presented architecture are discussed,

and the presented work is related to existing research in section 6. Finally, section 7 summarizes the contributions of this paper.

# 2. BACKGROUND

The research community has developed a variety of flexible countermeasures to addressing the threat of untrusted mobile code. These countermeasures are typically based on run-time monitoring [3, 1, 4], static analysis [8], or a combination of both [15, 5, 13]. Our run-time environment builds on this pre-existing research. This section briefly describes the most important building blocks: policy languages and policy enforcement techniques.

## 2.1 Policy languages

Device policies and application contracts are security automata, and such automata have to be specified by means of a policy language. Our system is designed to support multiple policy languages, and the actual prototype implementation supports the following two languages.

### 2.1.1 The ConSpec language

ConSpec is based on the semantics of security automata [12]. The ConSpec specification includes the scope of the policy, the definition of state variables (that provide a set of states of the automaton) and security-relevant events.

The scope specifies whether the policy applies to a single run of each application (scope `Session`), saves information between multiple runs of the same application (scope `Multisession`) or gathers events from the entire system (scope `Global`). If the scope of the policy is `Global` or `Multisession` then persistent state variables can be defined that are accessible from different processes.

Security-relevant events are defined by a full signature of an API method and a time modifier, which indicates whether the monitor must be notified about the event before or after the execution of the method call. In the latter case the return value of the method can be considered in further reasoning.

Each event is accompanied with a sequence of guards that specify the conditions, under which the event is allowed. These conditions can involve state variables or parameters of the event itself. Each guard triggers an update block that may assign new values to the state variables. If no updates are required this must be specified by using keyword `skip`. If no guard condition is satisfied and there is no `ELSE` block to handle this case then the security-relevant event violates the policy, and the program must be terminated.

For an example of a ConSpec policy see Fig. 2.1.1.

### 2.1.2 The 2D-LTL language

An alternative to ConSpec is the 2D-LTL policy language [6], a temporal logic language based upon a bi-dimensional model of execution. One dimension is a sequence of states of execution inside each run (session) of the application, and another one is formed by the global sequence of sessions themselves ordered by their start time.

Correspondingly, the temporal operators of the language can be split into two categories: local and global ones. Local operators apply to the sequence of states inside the session, for instance, "previously local" operator ($Y_L$) refers to the previous state in the same session, while "previously global" ($Y_G$) points to the state in a previous session. Other temporal operators are "once locally" ($O_L$) – in some past state of this session, "once globally" ($O_G$) – in some previous session, "historically local" ($H_L$) – in all past states of this session, "historically global" ($H_G$) – in all previous sessions etc.

```
SCOPE Session

SECURITY STATE
    bool opened=FALSE

AFTER System.IO.File.OpenRead(string filename)
PERFORM
    TRUE -> opened=TRUE

BEFORE System.Net.WebRequest.Create(string url)
PERFORM
    not (url.StartsWith("http")) -> skip;
    not opened -> skip;
```

**Figure 1: ConSpec policy "No creating HTTP connections after a local file has been accessed"**

```
LET StartHTTPConnection DEF
  BEFORE System.Net.WebRequest.Create(string url)
  WITH url.StartsWith("http://")
END

LET FileOpen DEF
  AFTER System.IO.File.OpenRead(string filename)
END
```

**Figure 2: Definition of 2D-LTL predicates**

To write a 2D-LTL formula, propositional and temporal operators are applied to the *predicates*. Predicates are arbitrary boolean functions depending on states of execution. They give us some information about the state. In our framework we support two kinds of predicates: those that become true when a security-relevant API call has just executed or is about to execute (close to ConSpec security-relevant events), and those that depend on environmental parameters.

For instance, the policy "Application is not allowed to start a connection if it has opened the local files **in this session**" can be expressed as

$$H_G\,(\texttt{StartHTTPConnection} \rightarrow \neg O_L\,(\texttt{FileOpen}))$$

where predicate `StartHTTPConnection` corresponds to starting a connection and `FileOpen` – to opening file for reading (for an example of how predicates are linked to the actual API see Fig. 2.1.2). Another example: to express the policy "Application is not allowed to start a connection if it has opened the local files **in any session**" one needs the following formula:

$$H_G\,(\texttt{StartHTTPConnection}) \rightarrow \neg O_G O_L\,(\texttt{FileOpen})\,.$$

## 2.2 Policy enforcement techniques

Our system supports a wide variety of policy enforcement techniques, and is designed to be extensible with new techniques. The prototype implementation supports the following enforcement techniques:

#### 2.2.1 Inlined reference monitoring

With Inline Reference Monitoring [3], a program rewriter inserts security checks inside an untrusted application. When the application is executed, these checks monitor the behavior of the application and prevent it from violating the policy.

The key advantage of this approach is that it does not require changes in the runtime system or the trusted system libraries. It is an easy way to secure an application when it has not been developed with a security policy in mind or when all other techniques have failed.

#### 2.2.2 Proof carrying code

An alternative way to enforce a security policy is to statically verify that an application does not violate this policy. On the one hand, static verification has the benefit that there is no overhead at runtime. On the other hand, it often needs guidance from a developer (e.g. by means of annotations) and the techniques for performing the static verification (such as theorem proving) can be too heavy for mobile devices.

Therefore, with Proof Carrying Code [8], the static verification produces a proof that the application satisfies a policy. In this way, the verification can be done by the developer, or by an expert in the field. The application is distributed together with the proof. Before allowing the execution of an application, a proof-checker verifies that the proof is correct for the application. Because proof-checking is usually much more efficient than making the proof, this step becomes feasible on mobile devices.

#### 2.2.3 Cryptographically signed code

Applications transmitted over unsecure connections can not be trusted. Cryptographic signatures are an easy way to solve this problem. The application is signed, and is distributed along with this signature. After receiving this application, the signature can be used to verify the source and integrity of application.

Traditionally, when a third party signs an application, it means that this third party claims the application is well-behaved. Adding the notion of a contract, as is done in the S3MS approach, allows us to add more meaning to claims on well-behavior. A signature on the application and the contract means that the third party claims that the application respects the supplied contract. In addition, the decision whether the contract is acceptable or not remains with the end user.

#### 2.2.4 Contract matching

Matching the application contract with the device policy is a straightforward approach to decide whether or not the contract is acceptable. When deploying an application with a contract, the contract acts as an intermediate between the application and the security policy of the device. First, *contract matching* checks whether all security-relevant behavior allowed by the contract is also allowed by the policy. If this is the case, all other enforcement techniques can be used to make sure that the application complies to the contract.

Besides decoupling the application from the policy, the contract matching allows the contracts to be much simpler than the policy. Therefore, it may be easier to technically enforce the contract on a particular application instead of enforcing the entire policy.

# 3. SYSTEM ARCHITECTURE

## 3.1 Overview

The S3MS security architecture is built upon the notion of "security-by-contract". Mobile devices are configured with a security policy, specifying an upper bound on the security-relevant behavior of mobile applications. In addition, applications can be distributed with a security contract, specifying their security-relevant behavior. Our security architecture supports the notion of application contracts and device policies, and provides an extensible framework for on-device policy enforcement.

Three key scenarios are identified: policy management and distribution, application deployment and loading, and execution monitoring and run-time enforcement.

**Policy management and distribution** This scenario is responsible for the management of different device policies, and their distribution and deployment onto mobile devices.

**Application deployment and loading** This scenario is responsible for verifying the compliance of a particular application with the mobile device policy before this application is executed.

**Execution monitoring and run-time enforcement** This scenario is responsible for monitoring and enforcing the adherence of a running application to the policy of the mobile device.

## 3.2 Deployment view

The three scenario's operate on two different platforms: on the platform of the policy provider and on the mobile device.

**Policy provider.** Within the S3MS security architecture, the policies are managed off-device by the *Policy Provider* and a specific policy can be pushed to a particular device. The policy provider could for instance be a company that supplies its employees with mobile devices, but wishes to enforce a uniform policy on all these devices. Or it could be an advanced end-user that owns his own device and manages the policy using a PC that can be connected to his device.

**Mobile device.** The mobile device stores the policy and is responsible for deploying and loading applications. If necessary, the mobile device also applies execution monitoring and run-time enforcement to achieve secure execution of the application, i.e. conforming the device policy.

A classical security infrastructure that supports secure communication is underpinning the policy provider and the mobile devices. The policy provider, for instance, is connected to mobile devices through secure links, which guarantee the authenticity and integrity of the communication. Similarly, if the mobile device is using external, trusted services for more intensive computations, these trusted services are also contacted through secure links.

The underlying security infrastructure does however not provide trust relationships between the application provider

and the mobile device, nor do provider and mobiles devices necessarily share a trusted third party. The presented security architecture therefore is also applicable to legacy applications.

Figure 7 in the appendix shows an architectural overview of the device, and of the software entities that are involved in the three scenarios. In the following sections, each of the scenarios is discussed in more detail, and the different software entities are identified.

## 3.3 Scenario 1: Policy management and distribution

The domain administrator manages device policies off-device on the policy provider platform. To configure a particular device with a given policy, the policy is pushed to the mobile device over a secure channel. This policy distribution is initiated by the domain administrator and executes partially on the policy provider platform and partially on the mobile device. As a result, the policy is stored on the mobile device by the *Policy Manager* and the policy is activated.

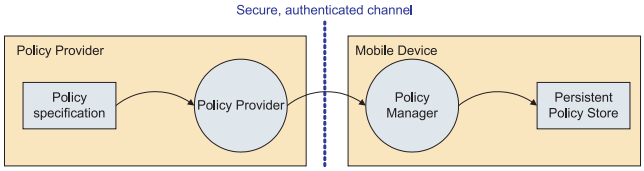


**Figure 3: Distribution of a policy to a mobile device**

## 3.4 Scenario 2: Application deployment and loading

The second scenario executes after downloading or installing the application and before the first execution of the application. The *Deployer* verifies the compliance of the application with the given device policy, and it enables the execution of the application in case of compliance. By default, the execution of an application is disabled in order to ensure that only compliant applications are executed on the mobile device. Compliant applications are recorded in the *Certified Application Database*.

To verify the compliance of the application with the device policy, this scenario applies a flexible combination of the different policy enforcement techniques discussed in subsection 2.2. For example, when an application contract is provided, the compliance can be verified by matching the application contract and the device policy, and by verifying the compliance of the application with the supplied application contract. As shown in figure 4, each of the configured policy enforcement techniques is applied sequentially until the compliance is ensured.

In case of applying an inlined reference monitor, which is the typical fallback scenario in our architecture, this scenario is also responsible for instrumenting the application to enforce the policy at run-time by means of an execution monitor. The execution monitor and run-time enforcement are further explained in scenario 3.

## 3.5 Scenario 3: Execution monitoring and run-time enforcement

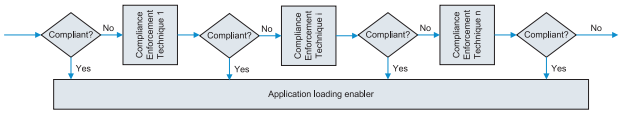


**Figure 4: Verifying application/policy compliance**

Monitoring the application and enforcing the device policy at run-time is completely executed on the device, as shown in figure 5. The application initiates this scenario by attempting to perform a security-relevant operation. By definition, any operation that occurs in the device policy is a security-relevant operation. In this scenario, the inlined *Execution Monitor* makes sure that the execution of the application is halted before and after each security-relevant operation. Based on the policy, the *Policy Decision Point* decides to continue with the execution or to terminate the application. To do so, the Policy Decision Point uses stored policy state, system information parameters (such as the battery level) and parameters supplied with the security-relevant operation. In addition, the Policy Decision Point can also update the policy state.

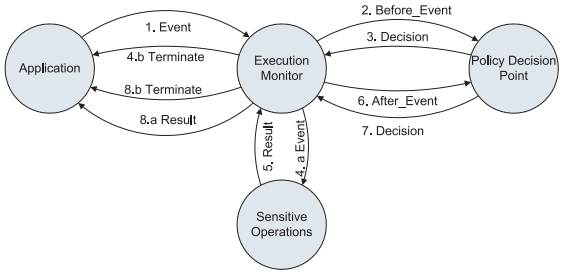


**Figure 5: Execution monitoring**

# 4. DESIGN AND IMPLEMENTATION

We have implemented a first prototype of this security architecture in the .NET Compact Framework on Windows Mobile 5. In this section, the most important design decisions are briefly highlighted.

**Support for a variety of policy enforcement techniques.** In order to achieve a powerful security architecture that can incorporate a variety of state-of-the-art policy enforcement techniques, a very configurable and extensible compliance engine is constructed as depicted in figure 8.

Each compliance verification technology is encapsulated in a *ComplianceModule*. To verify the compliance of an application with a policy, the *Process(Application app)* method is executed on such a compliance module. The boolean result of the method indicates whether or not the compliance verification is successful. As a side effect of executing the process method, the application can be altered (e.g. instrumented with an inline reference monitor). The compliance engine instantiates the different compliance modules and applies them sequentially until the compliance of the application with the policy is ensured.

Given the limited resources of mobile devices, it is also important to choose an optimal representation of the policy to do the compliance verification. However, the differences between the technologies make it hard to find one optimal

representation that is suitable for each technique.

For instance, in a runtime monitor, the decision whether an event is allowed or aborted relies only on the current state of the policy. Therefore, the representation for runtime enforcement only contains the current state, and methods for each event that check against the state, and update it. On the other hand, contract/policy matching checks whether or not the behavior allowed by the contract is a subset of the behavior allowed by the policy. For this task, a full graph representation is required.

To counter this problem, we decided to provide a set of optimized representations, called a policy package. The policy provider is responsible to distribute the policy packages to the mobile devices, including suitable representations for the variety of policy enforcement techniques. To do so, we have developed a compiler to transform a given policy specification in ConSpec or 2D-LTL to a policy package (figure 6). Similarly, to use optimized representations to do compliance verification based on the application contract, the contract must be supplied in the form of a consistent policy package.

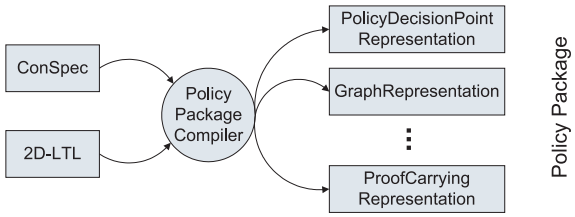


**Figure 6: Compilation of a policy package**

In addition to the flexibility in policy enforcement techniques, our security architecture also supports a variation of the policy specification language. In our prototype, we support both ConSpec, a policy specification language based on security automata, as well as 2D-LTL, a bi-dimensional temporal logic language. Although both policy specifications are necessarily transformed into a different implementation of the Policy Decision Point, the execution monitor uses the same Policy Decision Point interface irrespectively of the policy specification language used. In addition, we are able to reuse the same inlined reference monitor and instrumentation support with both specification languages.

A final decision in our prototype was the way we ensure that only compliant applications can be executed on the mobile device, i.e. only after the application successfully passes the deployment scenario. Instead of maintaining and enforcing a Certified Application Database, we decided to rely on the underlying security model of Windows Mobile 5.0 in our prototype. The *Locked or Third-Party-Signed* configuration in Windows Mobile allows a mobile device to be locked so that only applications signed with a trusted certificate can run [7]. By adding a policy-specific certificate to the trusted key store, and by signing applications with that certificate after successfully passing the deployment scenario, we ensure that non-compliant applications will never be executed on the protected mobile device.

# 5. DISCUSSION

In this section, we shortly discuss the advantages and disadvantages of the presented security architecture. In particular, we discuss the improved semantics of signatures, the concurrency challenges with inline reference monitoring and the difference between caller side and callee side inlining.

**Semantics of signatures.** Our security architecture relies on cryptographic signatures in several places. But a key difference with the use of cryptographic signatures in the current .NET and Java security architectures is the fact that the semantics of a signature in our system are always clearly and unambiguously defined. A signature on an application with a contract means that the trusted third party attests to the fact that the application complies with the contract, and this is a formally defined statement. Similarly, a signature on a policy package attests to the fact that the different policy representations in the package all represent the same security automaton, again a formally defined statement.

**Concurrency and inlined reference monitoring.** Inlined reference monitoring has been developed in a single-threaded context. State of the art systems such as Polymer [1] explicitly leave dealing with concurrency as future work. Since basically all mobile device applications are multi-threaded, our implementation had to deal with the concurrency issues. The conceptually simple solution is to lock the entire security state for the complete duration of a security-relevant method call. However, the performance penalty of this simple solution can be devastating if blocking calls, for instance listening on a socket, are security-relevant. Our current implementation is semantically equivalent to the simple solution, but performs more fine grained locking based on a simple analysis of the policy.

**Caller side versus callee side inlining.** When security relevant events are method calls, the security checks can be inlined in the calling code or in the called code. Both approaches have advantages and disadvantages. With callee side inlining, it is easier to obtain complete mediation, i.e. the assurance that every call is monitored. But callee side inlining typically requires modification of the platform libraries, as some of the method calls that need to be monitored are implemented in these libraries. On some mobile devices, the platform libraries are in ROM, essentially ruling out callee side inlining. Moreover, callee side inlining can cause a cyclic dependency between the library and the policy enforcement assembly.

Our current implementation thus uses caller side inlining. Because caller side inlining needs to find the target of a method call statically, it is harder to ensure complete mediation. Therefore, we impose some restrictions on the programs that are monitored: in the current prototype we disallow for instance the use of delegates when these delegates cross the boundary of the untrusted application, and the use of reflection. In addition, to deal with virtual methods, our inliner inserts an additional run-time check to dispatch a security-relevant call to the appropriate Policy Decision Point method, based on the dynamic type of the object.

# 6. RELATED WORK

There is a huge body of related work that deals with specific policy enforcement technologies for untrusted applications. This research area is too broad to discuss here. Some of the key technologies were briefly discussed in section 2.2. A more complete survey of relevant technologies can be found in one of the deliverables of the S3MS project [14].

Even more closely related are those research projects that have designed and implemented working systems building on one or more of the technologies discussed above. Naccio [4]

and PoET/PSlang [2] were pioneering implementations of run-time monitors. Polymer [1] is also based mainly on run-time monitoring, but the policy that is enforced can depend on the signatures that are present on the code. Model-carrying code [13] is an interesting application of proof carrying code in the domain of untrusted mobile code security. Mobile [5] is an extension to the .NET Common Intermediate Language that supports certified inline reference monitoring. Certifying compilers [9] use similar techniques like proof carrying code, but they include type system information instead of proofs.

We are not aware of any other research projects that are designing and implementing a code security architecture on a mobile device. So our system seems to be the first evidence that a flexible combination of code security technologies can be supported on todays mobile phones and PDA's.

# 7. CONCLUSION

This paper proposed a flexible security architecture for mobile devices built upon the notion of "security-by-contract". In a very extensible way, the architecture integrates a variety of state-of-the art technologies for secure execution of mobile applications, and supports different policy specification languages. In addition, the proposed architecture also supports the secure execution of legacy applications, although a better run-time performance is achieved for architecture-aware applications.

In addition, the paper reports on the experiences with a working prototype implementation of the proposed architecture. The prototype has been implemented on a Windows Mobile 5 device with the .NET Compact Framework, and includes already several compliance verification techniques and two policy specification languages. This paper highlights the most important design decisions that have been taken in this prototype implementation. It discusses the advantages of the proposed security architecture relative to the standard security architecture of mobile devices.

# 8. REFERENCES


[1] L. Bauer, J. Ligatti, and D. Walker. Composing security policies with Polymer. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 305–314, June 2005.

[2] U. Erlingsson. *The inlined reference monitor approach to security policy enforcement.* PhD thesis, Cornell University, 2004. Adviser-Fred B. Schneider.

[3] U. Erlingsson and F. B. Schneider. Irm enforcement of java stack inspection. In *SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy*, page 246, Washington, DC, USA, 2000. IEEE Computer Society.

[4] D. Evans and A. Twyman. Flexible policy-directed code safety. In *IEEE Symposium on Security and Privacy*, pages 32–45, 1999.

[5] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Certified in-lined reference monitoring on .net. In *PLAS '06: Proceedings of the 2006 workshop on Programming languages and analysis for security*, pages 7–16, New York, NY, USA, 2006. ACM Press.

[6] F. Massacci and K. Naliuka. Multi-session security monitoring for mobile code. Technical Report DIT-06-067, UNITN, 2006.

[7] MSDN. Windows mobile 5.0 application security. http://msdn2.microsoft.com/en-us/library/ms839681.aspx, May 2005.

[8] G. C. Necula. Proof-carrying code. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, New York, NY, USA, 1997. ACM Press.

[9] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *Proceedings of the 1998 ACM SIGPLAN Conference on Prgramming Language Design and Implementation (PLDI)*, pages 333–344, 1998.

[10] B. Ray. Symbian signing is no protection from spyware. http://www.theregister.co.uk/2007/05/23/symbian_signed_spyw
May 2007.

[11] S3MS. Security of software and services for mobile systems. http://www.s3ms.org/, 2007.

[12] F. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.

[13] R. Sekar, V. Venkatakrishnan, S. Basu, S. Bhatkar, and D. DuVarney. Model-carrying code: A practical approach for safe execution of untrusted applications, 2003.

[14] D. Vanoverberghe, F. Piessens, T. Quillinan, F. Martinelli, and P. Mori. D4.1.0/d4.2.0 - run-time compliance state of the art, November 2006.

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


# APPENDIX

The appendix includes two additional overview figures. Figure 7 shows the overview of the flexible security architecture for mobile device as presented in section 3. Figure 8 shows the detailed design of the deployment-time compliance verification. The class diagram includes, among other, the extensible set of compliance modules and policy representations as discussed in section 4.

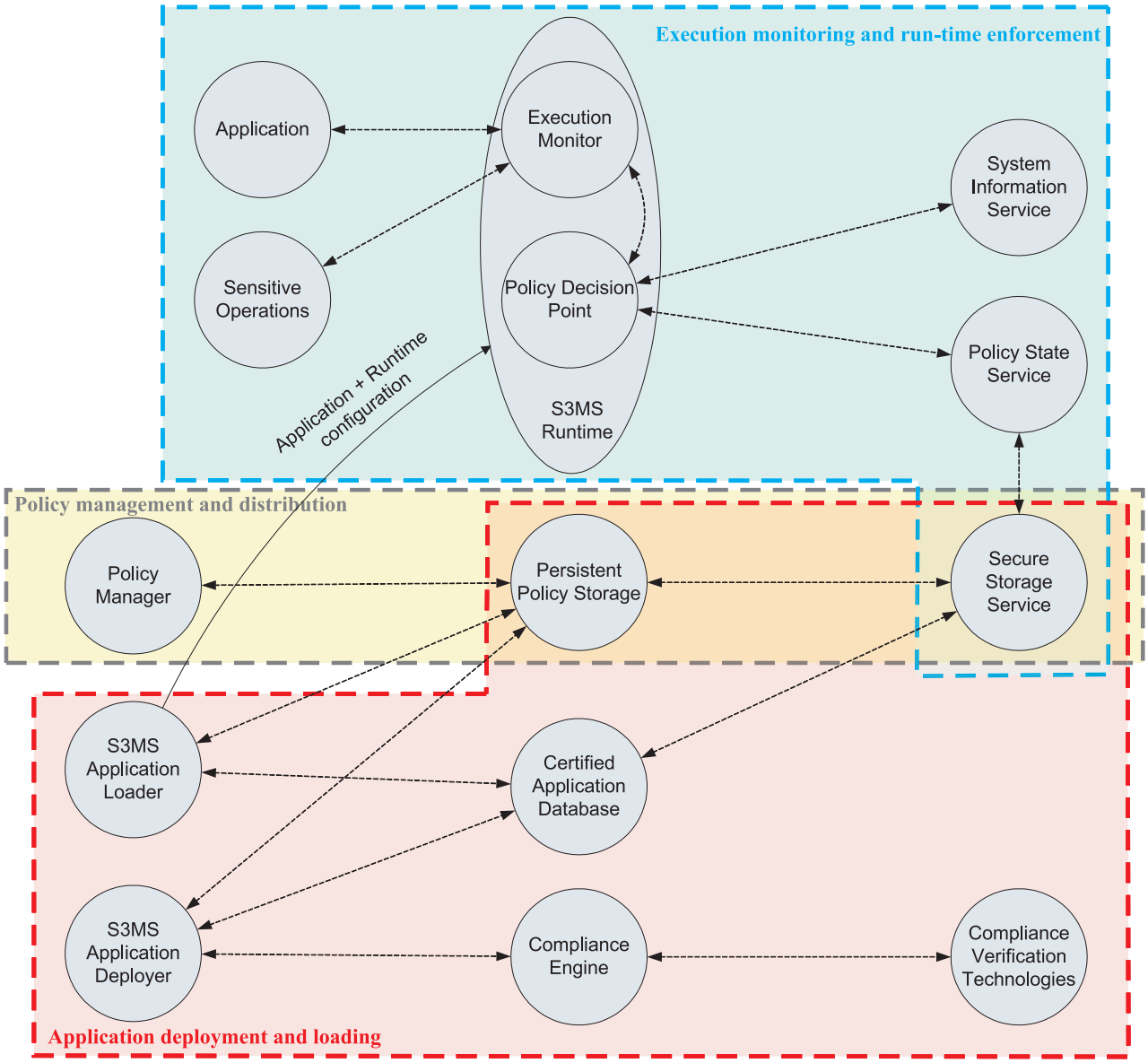


**Figure 7: Architecture overview**

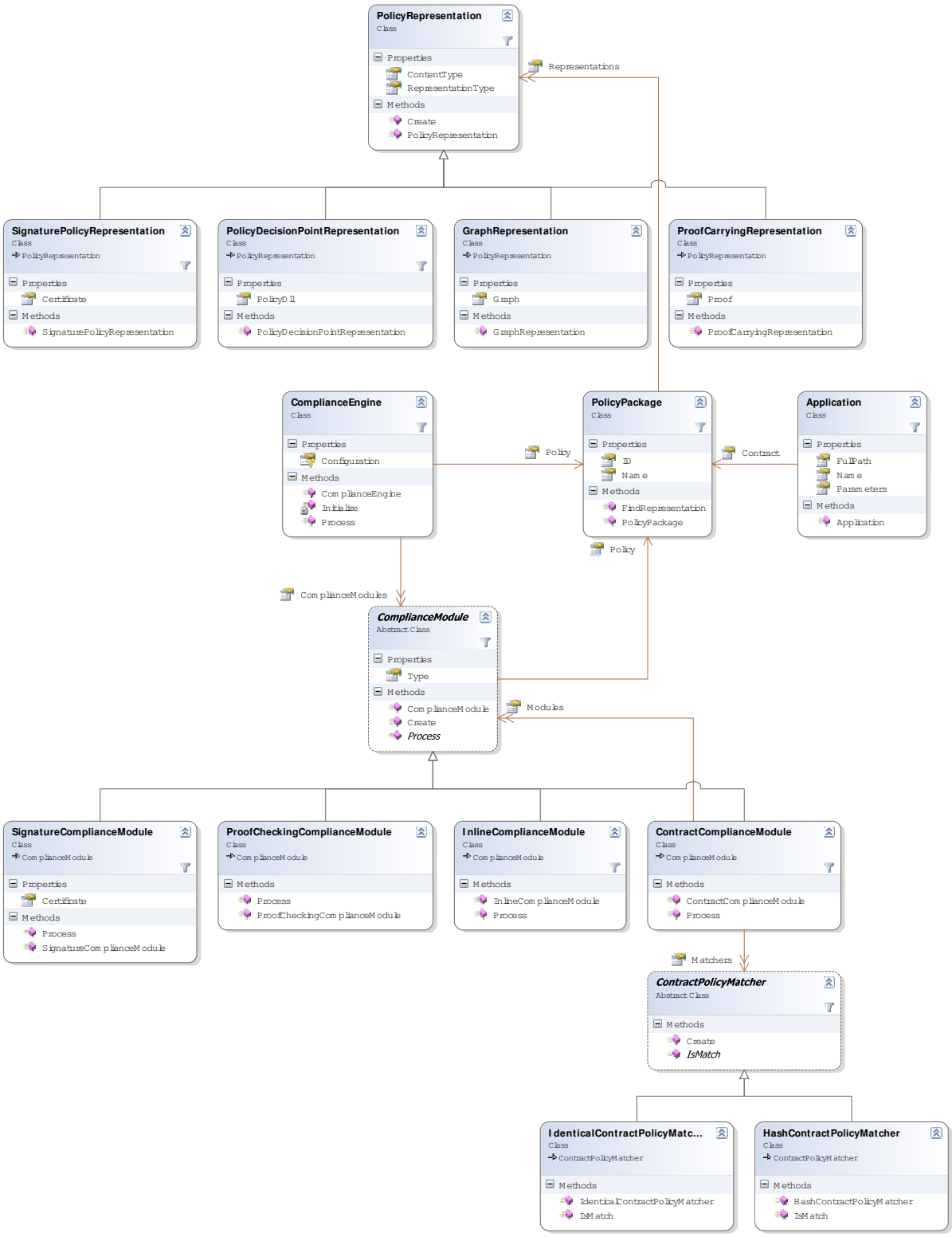


**Figure 8: Design of the deployment-time compliance verification**