A Flexible Security Architecture to Support Third-party Applications on Mobile Devices

Lieven Desmet[†], Wouter Joosen[†], Fabio Massacci[‡], Katsiaryna Naliuka[‡],
Pieter Philippaerts[†], Frank Piessens[†] and Dries Vanoverberghe[†]

Lieven.Desmet@cs.kuleuven.be

[†]DistriNet Research Group, Department of Computer Science
Katholieke Universiteit Leuven, Celestijnlaan 200A, B-3001 Leuven, Belgium

[‡]Department of Information and Communication Technology
Università di Trento, Via Sommarive 14, I-38050 Povo (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.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Access Controls

General Terms

Design, Security

Keywords

security architecture, mobile code

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 contains the public keys of trusted parties.

This security model has a number of drawbacks. First, it is not flexible: applications either run in a restricted sandbox, or have full power. Second, no precise meaning is associated with the signatures of trusted third parties: a signature either means that the application comes from the software factory of the signatory or that the signatory vouches for the software, but there is no clear definition of what guarantees it offers. Hence, device owners trust the third party both for (a) appropriate vetting of applications, and (b) using a suitable notion of good behavior. Incidents [**?**] show that the current security model is inappropriate.

The project *Security of Software and Services for Mobile Systems (S3MS) [**?**]* 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 life-cycle. The project is built on the notion of *Security by Contract*: applications come with contracts describing their security relevant behaviour.

In this paper, we describe the architecture of the run-time environment on the mobile device developed in the context of the S3MS project. The architecture integrates in a very flexible way several state-of-the-art policy enforcement technologies, such as proof-carrying code and inlined reference monitors. In addition, the security architecture offers additional support for application contracts and the security-by-contract paradigm. Thanks to the combination of different enforcement techniques and the support for application contracts, our security architecture is able to provide policy enforcement for legacy applications, as well as architecture-aware applications. However, the latter set of applications result in a smaller run-time performance penalty, which is an important characteristic for resource-restricted environments such as mobile devices. In addition, a first prototype implementation of the proposed security architecture is available for Windows Mobile devices with the .NET Compact framework.

The remainder of the paper is structured as follows. Section **??** provides some background information on the security-by-contract paradigm adopted by the S3MS project, existing policy enforcement techniques, and policy languages. Next, our flexible security architecture for mobile devices is presented in section **??**, and section **??** describes our prototype implementation. In section **??**, the advantages and disadvantages of the presented architecture are discussed. Finally, the presented work is related to existing research, and we offer a conclusion.

2. BACKGROUND

The architecture described in this paper is developed in the context of the European project S3MS [**?**]. In this section, we describe the key notion of *security-by-contract* underlying the S3MS project, and we briefly discuss the policy enforcement techniques and policy languages considered in the project.

2.1 The Security-by-contract Paradigm

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 [**?**], 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.

2.2 Policy Enforcement Techniques

The research community has developed a variety of countermeasures to address the threat of untrusted mobile code. These countermeasures are typically based on run-time monitoring [**?**, **?**, **?**], static analysis [**?**], or a combination of both [**?**, **?**, **?**]. Our run-time environment builds on this preexisting research. We briefly review the technologies supported in our system: cryptographic signatures, inline reference monitoring, proof carrying code and contract-policy matching.

Cryptographic signatures. The simplest way to solve the lack of trust is to use *cryptographic signatures.* 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 the 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. Moreover, what is important is the fact that the decision whether the contract is acceptable or not remains with the end user.

Inline reference monitoring. With *inline reference monitoring* [**?**], 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. The biggest challenge for inline reference monitors is to make sure that the application can not circumvent the inlined security checks.

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* [**?**], 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.

Contract-policy matching. Finally, when application contracts (called application models in [**?**]) are available, *contract-policy matching* [**?**, **?**] is an interesting 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, a matching step 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 more efficient and easier to technically enforce the contract on a particular application instead of enforcing the entire policy.

2.3 Policy Languages

In this paper, we make a clear distinction between application contracts and device policies. Both are security automata, but the first ones are associated with a particular application, while the later ones are associated with a device.

A security automaton [**?**] is a Büchi automaton – the extension of the notion of finite state automaton to infinite inputs. A security automaton specifies the set of acceptable (potentially infinite) sequences of *security relevant events* as the language accepted by the automaton.

In our system, a policy identifies a subset of the methods of the platform API as security relevant methods. Typical examples are the methods to open network connections or to read files. Security relevant events in our system are the invocations of these methods by the untrusted application, as well as the returns from such invocations. Hence, a security automaton specifies the acceptable sequences of method invocations and returns on security relevant methods from the platform API.

Security automata have to be specified by means of a policy language. Our system is designed to support multiple policy languages, including policy languages that support multiple runs of the same application. The actual prototype implementation supports already two languages, briefly introduced in the following paragraphs.

2.3.1 The ConSpec Language

ConSpec [?] is directly based on the notion of security automata, and is similar to Erlingsson's PSLang policy language [?]. Like PSLang, a ConSpec specification includes the definition of state variables (that gives us the set of states of the automaton) and the definition of what state transitions are caused by each of the security relevant events. ConSpec extends PSLang with support for multiple scopes.

A 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: the event can correspond to the start of a method call or the return of a method call. In the latter case the return value of the method can be taken into account when updating the security state.

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 identified 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 application must be terminated.

For a (toy) example of a ConSpec policy see figure **??** [1].

2.3.2 The 2D-LTL Language

An alternative to ConSpec is the 2D-LTL policy language [?], 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.

[1]Because of the string comparison in the policy, the policy is vulnerable to canonicalization attacks. A real-life policy would have to make sure the url is canonicalized before doing the string comparison, but this is irrelevant for this paper.

```
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 local files **in this session**" can be expressed as

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

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. **??**). Another example: to express the policy "Application is not allowed to start a connection if it has opened local files **in any session**" one needs the following formula:

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

3. SYSTEM ARCHITECTURE

In this section, our security architecture for mobile devices is presented. First, subsection **??** enumerates the most important architectural requirements for the security architecture. Next, subsection **??** gives an overview of our architecture, and highlights three important architectural scenario's. The following three subsections discuss some architectural decisions in more detail.

3.1 Architectural Requirements

Before presenting and discussing our flexible security architecture for mobile devices, the most important architectural requirements for the on-device, run-time environment are briefly discussed.

Secure execution of third-party applications The architecture should give high assurance that applications running on top of it can never break the device security policy. This is the key functional requirement of our architecture.

Support for the security-by-contract paradigm The architecture should support the notion of application contracts: by offering support for enforcement techniques built upon the security-by-contract paradigm, discussed in section **??**, the architecture should provide better performance for applications with contracts.

Flexible integration of enforcement techniques The security architecture should seamlessly integrate the set of on-device enforcement techniques discussed in section **??**. In addition, the security architecture should provide a flexible framework for adding, configuring or removing additional on-device enforcement techniques.

Optimized for resource-restricted devices The security architecture needs to be optimized for the use on resource-restricted, mobile devices such as personal digital assistants or smartphones. These device typically have limited memory and processing power, and restricted battery capacity. Also, their communication resources are often limited: the devices can easily go offline, they may have limited bandwidth or their data communication can be quite expensive. The architecture should secure the execution of applications with a minimal performance penalty during the application execution, without compromising security during network disconnectivity.

Compatible with legacy applications To be compatible with existing applications, it is important that the security architecture supports the secure execution of architecture-unaware, legacy applications. Of course, the fact that an application is architecture-unaware may impact performance.

In the following section, an overview of our security architecture for mobile devices is presented. As will be explained further, each of the enumerated architectural requirements has impacted the overall architecture.

3.2 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.

The three main scenarios are: 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 in the case where the previous scenario has decided that this is necessary.

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. It could also 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, loading and running applications. If necessary, the mobile device also applies execution monitoring and run-time enforcement to ensure compliance to the device policy.

A classical security infrastructure for secure communication supports the policy provider and the mobile devices. The policy provider, for instance, connects 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 **??** shows an architectural overview of the device, and of the software entities that are involved in the three scenarios. Each of the software entities is now discussed in more detail in the overview of the three scenarios.

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

THE OLD PICTURE WAS BETTER.

Figure 3: Detailed architecture overview

device over a secure channel. This policy distribution is initiated by the domain administrator[2] 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. Policies are securely stored on the device in the *Persistent Policy Store*.

Figure 4: Distribution of a policy to a mobile device

In our architecture, on-device policies can not be edited. However, the mobile device can manage multiple policies, and switching between such pre-installed policies is supported.

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 *Application 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 section **??**. For example, when an application contract is present, 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 **??**, each of the configured policy enforcement techniques is applied sequentially until some form of 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.

Scenario 3: Execution Monitoring and Run-time Enforcement

Monitoring the application and enforcing the device policy at run-time is completely executed on the device, as shown in figure **??**. The application initiates this scenario by attempting to perform a security relevant method call. In this scenario, the inlined *Execution Monitor* makes sure that the execution of the application is suspended 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.

[2]The domain administrator is not necessarily the end user of the phone. For instance for company phones it can be a company's system administrator.

Figure 6: Execution monitoring

3.3 Policy Representations

Our architectural requirements ask for flexibility in policy enforcement techniques used, as well as for resource conscious implementations. However, the different enforcement techniques impose different constraints on optimized policy representations, and hence it is hard, or even impossible, to find one optimal representation that is suitable for each technique.

For instance, in a run-time monitor, the decision whether an event is allowed or aborted relies only on the current state of the policy. Therefore, an efficient 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 may be required.

Figure 5: Verifying application/policy compliance

To deal with this problem, our architecture introduces the notion of *policy packages*. A policy package is a set of optimized representations (each geared towards a different enforcement technique) of the same policy.

Figure 7: Compilation of a policy package

The policy provider is responsible for distributing the policy packages to the mobile devices. To do so, he uses a policy compiler to transform a given policy specification (e.g. in ConSpec or 2D-LTL) to a policy package (figure **??**). In a similar vein, representation of application contracts can be optimized towards different application-policy matching algorithms, and hence contracts are supplied in our architecture in the form of a contract package. The design of the policy package and the policy representations is depicted in figure **??**.

Figure 8: Policy package and policy representations

3.4 The Deployment Framework: 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, our architecture includes a very configurable and extensible compliance engine. At deployment time of an application, this compliance engine attempts to ensure the compliance of the application by means of all installed enforcement technologies. Each enforcement technology is represented as a *compliance module*. Each compliance module can select which policy representations present in the installed policy package (see discussion above) it will use.

Figure **??** shows the design of the deployment framework for deployment-time compliance verification. The class diagram includes the *Compliance Engine* and the extensible set of *ComplianceModules*.

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.

The order in which the compliance modules are applied, and their particular configuration is made policy-specific. This policy-specific configuration is part of the policy package. In this way, policies can optimize the deployment process by favoring or excluding compliance modules (e.g. because they are optimal for the given policy, or because they are resource-intensive or inappropriate).

3.5 The Default Enforcement Technology: Inlining of the Policy

Of all enforcement technologies discussed in section **??**, only inlining is suitable for legacy applications, i.e. applications that come with no metadata such as contracts, proofs or signatures. Hence, in our architecture inlining is the fallback technology: when all other compliance modules fail, the inlining compliance module is used to ensure compliance of the application to the device policy.

Several practical implementations of inline reference monitors have been described in the literature [**?**, **?**], and all of them could in principle be integrated in our architecture. Our default compliance module is very similar to existing systems. Two noteworthy deviations are discussed below.

Concurrency and inlined reference monitoring. Since basically all mobile device applications are multi-threaded, our system has to deal with concurrency in inlined reference monitoring.

Erlingsson's seminal implementation of inline reference monitoring [**?**] puts the burden of synchronization on the policy writer, forcing the policy writer to take locks on the policy state where necessary. This is undesirable in our system, as explicit locking in the policy (1) makes composition of policies harder, as it is hard to see what additional synchronization is needed when composing two preexisting policies, and (2) complicates matching as contracts have no synchronization information in them.

Existing systems that do provide support for composition of policies such as Polymer [**?**] explicitly leave dealing with concurrency as future work.

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 design is semantically equivalent to the simple solution, but performs finer grained locking based on a simple partitioning of the policy state in disjoint lock regions.

Caller side versus callee side inlining. When security relevant events are method call invocations and returns, 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. Moreover, on some mobile devices, the platform libraries are in ROM, essentially ruling out callee side inlining for our prototype. Another issue with callee side inlining is that it can cause a cyclic dependency between the library and the policy enforcement assembly.

Because of the above issues, our design uses caller side inlining. This makes achieving complete mediation harder, and our current implementation can not yet handle all applications. We discuss the limitations in more detail in the following section.

4. PROTOTYPE IMPLEMENTATION

Figure 9: Design of the deployment framework

We have implemented a first prototype of this security architecture in the .NET Compact Framework on Windows Mobile 5.

Our prototype includes policy compilers for both ConSpec, a policy specification language based on security automata, as well as 2D-LTL, a bi-dimensional temporal logic language. The compiler outputs a policy representation package that includes two policy representations, one suitable for inlining, and one suitable for signature verification. A third one suitable for matching is currently under development. The corresponding compliance modules that use these representations are also implemented.

Compliance modules need not be aware of the source policy language. For instance, our runtime execution monitor uses the same Policy Decision Point interface irrespectively of the policy specification language used: we can reuse the same inlined reference monitor and instrumentation support with both specification languages.

Our current inliner implementation 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.

A final noteworthy implementation aspect of our prototype is 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 [?]. 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 AND FUTURE WORK

In this section, we offer a brief preliminary evaluation of the presented security architecture based on the architectural requirements set forth in section **??**. But further experiments are needed: we also present a roadmap for a further in-depth evaluation.

Secure execution of third-party applications Our architecture assumes that the individual compliance modules are secure: a buggy or unreliable compliance module can validate an application that does not comply with the device policy. This is a weakness, but the cost of building in redundancy (e.g. requiring two independent compliance modules to validate an application) is too high. Apart from this weakness, our architecture supports high assurance of security through a simple and well-defined compliance validation process, and through the precise definitions of the guarantees offered by security checks. An example is the treatment of cryptographic 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.

Support for the security-by-contract paradigm Applications developed according to the S3MS methodology will come with security contracts, and possibly with metadata (proofs or signatures) that provide evidence for the compliance of the application with the contract. Our architecture includes several compliance modules that can show compliance of such applications with the device policy with considerably less overhead at runtime.

Flexible integration of enforcement techniques The architecture already incorporates different on-device enforcement techniques such as cryptographic signatures, contract/policy matching and on-device inlining of a reference monitor. Thanks to the pluggable compliance modules and the concept of policy packages the framework can easily be extended with additional on-device enforcement techniques. In addition, the policy configuration allows for policy-specific configuration of the different compliance modules, including configuration of the order in which they are applied to applications.

Optimized for resource-restricted devices Thanks to the technology-specific policy representations, the performance of each policy enforcement technique on the device can be optimized: each technology can use its own optimized representation. Moreover, for any given policy, the compliance modules can be ordered such that more efficient ones are tried first. In particular, for security-by-contract aware applications, compliance with the device policy can be checked without runtime overhead for the application.

Compatible with legacy applications Because of the use of a general-applicable fallback compliance module (e.g. on-device inlining of a reference monitor), the architecture can also ensure security for architecture-unaware legacy applications. The coverage does however depend on the specific compliance module used. For example, the inlined reference monitor used in our prototype still has some restrictions on the applications to be inlined: the use of certain delegates and reflection is for instance prohibited in order to ensure full mediation. This can lead to the rejection of a legacy application, even if it complies with the device policy. However, with further improvement of the fallback compliance module, full coverage of legacy applications is achievable with the presented architecture.

Based on this preliminary evaluation, the presented architecture looks promising. However, a more in-depth architectural evaluation and validation is necessary for a more grounded conclusion. We see three important tracks for further evaluation of the presented architecture.

First, *an extensive architectural evaluation* of the proposed architecture is necessary. For instance, an architectural trade-off analysis (such as ATAM [?]) with the different stakeholders involved (such as mobile end users, telecom operators, mobile application developers and vendors, mobile device vendors, security experts, ...), can evaluate and refine several architectural trade-offs. To do so, it is important to have more concrete business scenarios and use cases. Moreover, because of the intrinsic resource limitations of mobile devices, an elaborate performance analysis on the prototype implementation is necessary as a basis for architectural evaluation and optimization.

Second, it is necessary to perform an *end-to-end threat analysis* of the proposed architecture. Based on these results, a risk assessment will identify the most important security risks and will provide additional input for the architectural trade-off analysis.

Third, a *further integration of existing enforcement techniques* in the prototype implementation is needed to validate the flexibility of the deployment framework.

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 **??**. A more complete survey of relevant technologies can be found in one of the deliverables of the S3MS project [?].

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 [?] and PoET/PSlang [?] were pioneering implementations of run-time monitors. Polymer [?] 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 (MCC) [?] is an enforcement technique that is very related to the contract matching based enforcement used in the S3MS project. In MCC, an applications comes with a *model* of its security relevant behavior, and hence models are basically the same as contracts. The MCC paper describes a system design where models are extracted from the application by the code producer. The code consumer uses the model to select a matching policy, and enforces the model at runtime. Mobile [?] is an extension to the .NET Common Intermediate Language that supports certified inline reference monitoring. Certifying compilers [?] use similar techniques like proof carrying code, but they include type system information instead of proofs.

7. CONCLUSION

We 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 security-by-contract-aware applications.

The prototype of our architecture 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.

We are not aware of 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 is actually doable on todays mobile phones and PDA's.

8. REFERENCES

[1] I. Aktug and K. Naliuka. ConSpec – a formal language for policy specification. In *Proceedings of the First International Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM2007)*, September 2007 (accepted).

[2] 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.

[3] N. Dragoni, F. Massacci, K. Naliuka, R. Sebastiani, I. Siahaan, T. Quillinan, I. Matteucci, and C. Schaefer. S3ms deliverable d2.1.4- methodologies and tools for contract matching, April 2007.

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

[5] 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.

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

[7] 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.

[8] R. Kazman, M. Klein, and P. Clements. Atam: Method for architecture evaluation. Technical Report CMU/SEI-2000-TR-004, CMU/SEI, August 2000.

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

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

[11] 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.

[12] 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.

[13] B. Ray. Symbian signing is no protection from spyware. `http://www.theregister.co.uk/2007/05/23/symbian_signed_spyware/`, May 2007.

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

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

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

[17] D. Vanoverberghe, F. Piessens, T. Quillinan, F. Martinelli, and P. Mori. S3ms deliverable d4.1.0/d4.2.0 - run-time compliance state of the art, November 2006.

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