Security by Contract on the .NET Platform

Lieven Desmet†, Wouter Joosen†, Fabio Massacci‡, Pieter Philippaerts†,
Frank Piessens†, Ida Siahaan‡ and Dries Vanoverberghe†

Pieter.Philippaerts@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

Over the last few years, the success of GPS-enabled PDAs has finally instigated a breakthrough of mobile devices. Many people now already have a device that can connect to the internet and run untrusted code, typically a cell-phone or PDA. Having such a large interconnected and powerful computing base presents some new security issues. In order to counter new threats, the traditional security architectures need to be overhauled to support a new and more flexible way of securely executing mobile code.

This article describes the concept of *security-by-contract* (SxC) and its implementation on the .NET platform. This new model allows users to guarantee that an untrusted application remains within the boundaries of acceptable behavior, as defined by the user herself. A number of different techniques will be presented that can be employed to enforce this behavior.

In order to support the SxC paradigm, some new steps can be introduced in the application development process. In addition to building an application, developers can create an application contract and securely bind this contract to the application. The application deployment process supports legacy applications developed without such contracts, but it can support more advanced enforcement technologies for those applications that are SxC aware.

1. INTRODUCTION

The ubiquitousness of mobile devices has given birth to a whole new spectrum of mobile applications. These applications can be downloaded on the fly, and are often used for only a short period of time. One example is an electronic tourist guide. When a tourist arrives in a historically important city, she might want to download an application that can lead her car to the different touristic hotspots. This electronic guide can be distributed in public places, like the local train station or a nearby airport. Of course, all this means that the tourist guide needs to be able to communicate with the navigation system in order to function properly. This interaction is necessary for this sort of application, but it's not something that should be enabled for every application that the user downloads to her device.

As mobile applications become more prevalent, a small percentage of these applications will be malicious in nature. It is up to the security architecture of a mobile device to protect users from these malicious applications, but the current implementation fails to do so. The .NET Compact Framework offers no protection for mobile code whatsoever. The Code Access Security (CAS, [9]) system that is implemented in the full version of the .NET framework, is not supported on the compact framework. The only defense that stands between a user and malicious code, is the Windows CE security architecture [8]. But this too is far from sufficient to adequately protect users from harmful code.

The security system of Windows CE can run in different modes, depending on the needs of the user. In the more secure 'two-tier' mode, applications can be run in a *trusted* or *partially trusted* context. In a trusted context, an application runs unrestricted, whereas in a partially trusted context it is prohibited from accessing a predefined set of sensitive services, thus limiting the amount of damage it can do. When an application is started, the operating system checks the digital signature on the executable. If the signature traces back to a certificate in the trusted certificate store, the application is executed in a trusted context. Likewise, if the signature traces back to a certificate in the untrusted certificate store, the application is executed in a partially trusted context. If the signer is unknown, if the signature is invalid, or if no signature is present, the operating system will ask the user whether the application should be executed. When the user indicates that the system should run the application, it is executed in a partially trusted context.

This signature-based system doesn't work well in the case of roaming mobile code. The first problem is that the decision of allowing an application to run or not, is too difficult for a user to make. She would like to run an application as long as the application doesn't misbehave or doesn't violate some kind of policy. But she is in no position to know what the downloaded application exactly does, so she cannot make an educated decision to allow or disallow the execution of a program. A second problem is that certifying an application by a trusted third party is rather expensive. Many of these mobile application developers are small companies that do not have the resources to certify their applications. A third, and perhaps most damning, problem is that these digital signatures do not have a precise meaning in the context of security. They confer some degree of trust about the origin of the software, but they say nothing about how trustworthy the application is. Cases are already known where malware was signed by a commercial trusted third party [11]. This malware would have no problems passing through the Windows CE security architecture, without a user noticing anything.

The European FP6 project *Security of Software and Services for Mobile Systems* (S3MS) has addressed these issues by working out a security-by-contract (SxC) paradigm for the development, deployment and execution of mobile applications. SxC supports a number of different technologies that verify or enforce security policies. In the case of the tourist guide, the user could run the guide and enforce a policy where the guide can access the navigation system, but cannot access other resources such as the network or the file system. When a malicious application tries to access one of these restricted resources, the SxC system will prevent it from doing so.

Taking full benefit of this new paradigm does require some changes to the development and deployment process. The development cycle must be modified to include the creation and maintenance of application contracts. Then, during deployment, the necessary checks must be incorporated to make sure that the application contract is compatible with the system policy, and to verify that the contract hasn't been tampered with. However, the paradigm also supports legacy applications without contracts.

2. SECURITY BY CONTRACT

The concepts of system policies and application contracts are of paramount importance in a SxC system. This section will give a definition of a policy and a contract. Furthermore, the updated software development life cycle is presented, with a key focus on the differences between the traditional and the new life cycle.

This section discusses SxC at a conceptual level. More technical detail will follow in later sections.

2.1 Policies versus contracts

Loosely speaking, a system policy is a set of rules to which an application must comply. These rules typically limit the access of an application to a specific part of the system API. For instance, there could be a set of rules to prohibit applications from accessing the network or to limit the access to the file system. These accesses are also called *security-related events* (SREs). One could think of a policy as an upper-bound description of what applications are allowed to do. System policies are defined by the owner of a mobile device.

Application contracts are very similar, but instead of defining the upper-bound of what an application *can* do, it describes the upper-bound of what the application *will* do. It's the 'worst case' scenario of security-related behavior of an application. The contract is typically designed by the application developer and is shipped together with the application as metadata.

If we go back to the tourist guide example, a contract for this application could be: *"This application will only access the navigation system and show a window on the screen."* It specifies what the application will do. The device, to which the application is deployed, could have the following policy: *"An application cannot access the network, and cannot access the file system, except for its installation directory. It is free to use other resources."* In this case, the application contract would be compatible with the system policy, and the application should be allowed to run.

2.2 The software development life cycle

To take full advantage of this new paradigm, applications have to be developed with SxC in mind. This means that some changes occur in the typical *Develop-Deploy-Run* application life cycle. Figure 1 shows an updated version of the application development life cycle.

The first step to develop an SxC compliant application, is to create a contract to which the application will adhere. Remember that the contract represents the security-related behavior of an application and specifies the upper-bound of calls made to SREs. Designing a contract requires intimate knowledge of the inner workings of the application, so it's typically done by a (lead-)developer or technical analyst. Some mobile phone operators, companies or other authorities may choose to publish contract templates that can then be used as a basis for new application contracts. Once the initial version of the contract has been specified, the application development can begin. During the development, the contract can be revised and changed when needed.

After the application development, the contract must somehow be linked to the application code in a tamper-proof way. One straightforward method to do this, is by having a trusted third party inspect the application source code and the contract. If they can guarantee that the application will not violate the contract, they sign a combined hash of the application and the contract. Another way to link the contract and the code, is by generating a formal, verifiable proof that the application complies with the contract, and adding it to the application metadata container. This concept is called *proof-carrying code* [10]. When this step is completed, the application is ready to be deployed. The application is distributed together with its contract and optionally other metadata such as a digital signature from a third party or a proof.

When the program is deployed on a mobile device, the SxC framework checks whether the application contract is compatible with the device policy. This process is called *matching.* What essentially happens is that the SxC framework checks whether the security behavior described in the contract is a subset of the security behavior allowed by the policy. If it is, the application is allowed to run as-is. If the contract is not a subset of the policy, the application is treated as an application without a contract. We will discuss in the next section how such applications are monitored for compliance with the policy at run time.

Matching is one example of a *policy enforcement technology.* It ensures that whenever an application is run, it complies with the rules dictated by the system policy. If it cannot make this assurance, the matching fails. Other types of enforcement technologies can be used to give a similar guarantee.

2.3 Variants of the development life cycle

The scenario in the previous section showed how an appli-

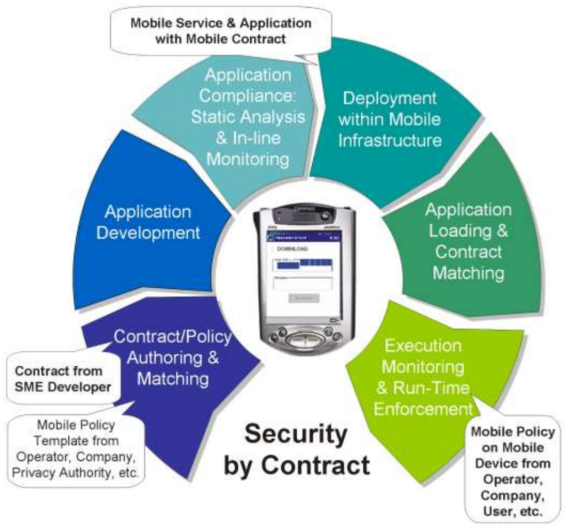
Mobile Service & Application with Mobile Contract
Application Compliance: Static Analysis & In-line Monitoring
Deployment within Mobile Infrastructure
Application Development
Application Loading & Contract Matching
Contract/Policy Authoring & Matching
Execution Monitoring & Run-Time Enforcement
Contract from SME Developer
Mobile Policy Template from Operator, Company, Privacy Authority, etc.
Security by Contract
Mobile Policy on Mobile Device from Operator, Company, User, etc.

Figure 1: The application development life cycle

```
SCOPE Session
SECURITY STATE
    int bytesSent = 0;

BEFORE int sent = System.Net.Sockets.Socket.Send
        (byte[] array)
PERFORM
    array == null ->
    bytesSent + array.Length <= 1000 ->

AFTER int sent = System.Net.Sockets.Socket.Send
        (byte[] array)
PERFORM
    true ->  bytesSent += sent;
```

Figure 2: "A CONSPEC policy, limiting the network data transfer"

cation with SxC metadata would be produced and deployed to a mobile device. There is however an important need to also support the deployment of applications that are not developed with SxC in mind. This backwards compatibility is a make-or-break feature for the system.

When an application without a contract arrives on the mobile device, there is no possibility to check for policy compliance through matching. No metadata is associated with the application that can prove that it does not break the system policy. A solution for this problem is to enforce the system policy through run time checking.

One example of a run time policy enforcement technology is *inlining*. During the inlining process, the application is modified to intercept and monitor all the calls to SREs. When the monitor notices that the application is about to break the policy, the call to the SRE that causes the policy to be broken is canceled.

The strength of run time checking is that it can be used to integrate non-SxC aware applications into the SxC process. A result of having this component in the system is that it is usable as is, without having to update a plethora of existing mobile applications.

A second variant of the development life cycle is where an existing application is made SxC-aware. It can sometimes be difficult to compose a contract for an application that may have been written years ago by a number of different developers. Instead of investing a lot of time (and money) into finding out which rules apply to the legacy application, an inlining-based alternative could be a solution.

The key idea is to use an inlining technique as described in the first part of this section. However, instead of inlining the application when it is loaded, the application is inlined by the developer. After being inlined, the application can be certified by a trusted third party.

There are a number of benefits of this approach, compared to on-device inlining. A first advantage is that the parts of the contract that can be manually verified by the trusted third party do not have to be inlined into the application. For instance, imagine that an existing application should comply with the policy *"An application cannot access the network, and cannot write more than 1000 bytes to the hard disk."* A third party can easily verify whether the application will break the first part of the policy. If the application contains no network-related code, it will not break this part of the policy. The second part of the policy may be harder to verify, if the application does contain some file-related code but if it is unclear how many bytes are actually written to the hard disk. In this case, the developer could inline the application with a monitor that only enforces the second part of the policy, but the application will nevertheless be certified for the full policy.

Inlining large applications on a mobile device can be time consuming. Going through the inlining process before deploying the application eliminates this problem. It speeds up the application loading process, which is a second advantage.

A final advantage is that the developer can do a quality assurance check on the inlined application. This is useful to weed out subtle bugs and ensure that the inlined application meets the expected quality standard of the developer.

3. POLICIES AND POLICY ENFORCEMENT

The previous sections have given an introduction to SxC and how it can be used to securely execute mobile code. This section will delve a bit deeper into the technical details of the platform, starting with an informal definition of the CONSPEC policy language. This is followed by an overview of some of the different enforcement techniques.

3.1 Informal definition of CONSPEC

To express the different system policies and application contracts, some kind of descriptive language is needed. One such language is the CONSPEC policy and contract language [4]. In this language, rules can be described on so-called security-related events (SRE). In the case of the .NET SxC implementation, these SREs correspond to security-sensitive calls from an application into the .NET base class library. Examples of these calls are methods that open files, network connections or give access to sensitive data.

Figure 2 contains a small example of a CONSPEC policy. The first line in the CONSPEC source sets the *scope* of the contract or policy. The scope defines whether the CONSPEC rules act on a single application instance, on all the instances of a specific application, or on every application in the system. A *session* scope acts on single application instances. Hence, the example of figure 2 that imposes a 1000-byte network limit means that every instance of an application can send at most 1000 bytes. A second type of scope is a *global* scope. Rules in such a global scope act on all applications together. If we modify the example in figure 2 to use a global scope instead of a session scope, the meaning of the rules would become "all applications on the system combined may send up to 1000 bytes over the network". Finally, a third *multi-session* scope is supported. This scope fits in between the global and session scopes. A multi-session scope defines rules for all instances of the same application.

The scope declaration is followed by the security state. This security state contains a definition of all the variables that will be used to store the CONSPEC state. In the example of figure 2, this will be a variable that holds the number of bytes that have already been sent. The ability for a CONSPEC policy to keep track of state is a critical difference between the SxC system and traditional security systems such as code access security. Where CAS can only either allow or deny a call to a specific library function, the SxC system can allow such a call as long as a particular precondition is met.

The security state is followed by one or more clauses. Each clause represents a rule on a security-relevant event. These rules can be evaluated before the SRE is called, after the SRE is called, or when an exception occurs. A clause definition consists of the 'BEFORE', 'AFTER' or 'EXCEPTIONAL' keyword to indicate when the rule should be evaluated, the signature of the SRE on which the rule is defined, and a list of guard/update blocks. The method signature corresponds largely to something a C# programmer would expect.

As the name implies, a guard/update block consists of first a guard and then an update block. The guard is a boolean expression that is evaluated when a rule is being processed. If the guard evaluates to true, the corresponding update block is executed. All state changes that should occur can be incorporated in this update block. When a guard evaluates to true, the evaluation of the following guards (and consequently the potential execution of their corresponding update blocks) is skipped.

If none of the guards evaluates to true, this means the policy does not allow the SRE. For example, in figure 2, if the current state of the policy has bytesSent = 950, then a call to the Send method with an array of length 55 will fail all the guards.

3.2 Overview of enforcement technologies

A number of different enforcement technologies can be used to make sure that an application complies with a device policy. Section 2.2 describes the most common scenario of how a policy can be enforced upon an application, but other options are available. The enforcement architecture is pluggable, so new enforcement modules can be easily integrated and used. This section gives an overview of the most common enforcement technologies.

3.2.1 *Digital signatures*

One way to enforce a policy is to have a trusted third party certify that an application complies to a given policy. This trusted party would have a different public/private key pair for every different policy it certifies. An application developer would then send his application to this trusted party for compliance certification with one of the trusted party's policies. When the application is found to comply with the policy, it gets signed with the key corresponding to that policy. The signature can be contained in the application metadata, or can be embedded in the executable itself (using the Authenticode mechanism).

Notice the subtle difference between the meaning of the digital signature in the SxC system and in the standard Windows CE security architecture. Both systems make use of the exact same mechanism to verify the signature on an application, but on the SxC system, the signature tells more about the application than simply its origin. It certifies that the application will not violate a specific policy. It certifies that this application will not harm the mobile device, whereas a signature in the Windows CE security architecture gives no precisely specified guarantees.

The upside of using this enforcement technology is that it is relatively simple to implement and use. However, third party certification can be costly, and requires trust in the certifying party.

3.2.2 *Matching*

The operation of matching the application's claim with the platform policy is solved through *language inclusion* [1]. The contract and the policy are interpreted as automata accepting sequences of SRE's. Given two such automata Aut^C (representing the contract) and Aut^P (representing the policy), we have a match when the language accepted by Aut^C (i.e. the execution traces of the application) is a subset of the language accepted by Aut^P (i.e. the acceptable traces for the policy).

For interesting classes of policies, the problem of language inclusion is decidable.

3.2.3 *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 [15], 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.

3.2.4 *Inlining*

A final enforcement technology is policy inlining. During the inlining process, the SxC system goes through the application code and looks for calls to SREs. When such a call is found, the system inserts calls to a monitoring component before and after the SRE. This monitoring component is a programmatic representation of the policy. It keeps track of the policy state and intervenes when an application is about to break the policy. After the inlining process, the application complies with a contract that is equivalent to the system policy.

The biggest advantage of this technique is that it can be used on applications that are deployed without a contract. It can be used as a fall-back mechanism for when the other approaches fail and it can also ensure backwards compatibility. So, even in the case of contract-less applications, the SxC framework can offer guarantees that the application will not violate the system policy. A disadvantage is that the application is modified during the inlining process, which might lead to subtle bugs. Also, the monitoring of the SREs comes with a performance hit. The performance hit is strongly dependent on the complexity of the policy being enforced. However, our preliminary experience with the prototype implementation indicates that the decrease in performance is small.

4. IMPLEMENTATION ON .NET

The SxC system presented in this paper has been implemented on the .NET compact framework as part of the S3MS project [12, 14]. A complementary Java version has also been developed in the context of this project.

To further clarify the architecture and implementation of the SxC system, three common scenarios are presented. In the first scenario, the creation, management and distribution of policies is investigated. The second scenario zooms in on the deployment and loading of an SxC application. And finally, the third scenario explains how the execution monitoring takes place. Figure 3 shows a detailed overview of the architecture, and the important components for each scenario.

4.1 Policy management and distribution

System policies can be written by anyone, but in practice it can be expected that only a few people will actually write policies. Writing policies requires technical skills that are beyond those of most users of mobile devices. It can be anticipated that large companies or mobile phone operators will write policies centrally by their technical staff, and then distribute them to their employees' or customers' mobile devices. Policy writers can use the *Policy Manager* tool to write and edit policies, to prepare the policy for deployment, and to deploy it to a mobile device.

Preparing the policy for deployment means that the policy is converted from a textual policy to different formats that can easily be used by the on-device SxC framework. Multiple formats - also called *policy representations* - for one policy are possible, depending on which enforcement technologies the policy writer would like to support. Different enforcement technologies require different formats. For instance, if the policy has to support matching, a graph representation of the policy must be deployed to the mobile device. Likewise, if the policy writer wants to support inlining, an executable version of the policy must be generated. Our implementation supports multiple policy languages, including the CONSPEC language discussed before, and a temporal logic based language 2D-LTL. Figure 4 shows how textual representations of policies are compiled into different policy representations.

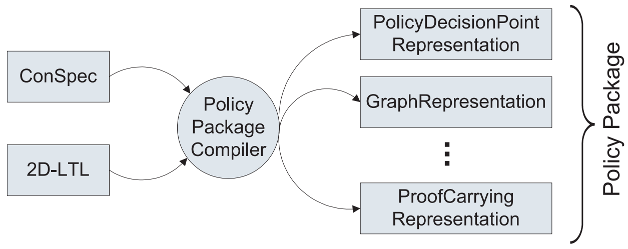

Figure 4: Compilation of a policy package

When the policy is ready, and the different policy representations are generated, it can be deployed to a mobile device. The Policy Manager sends the policy and its representations to the *Persistent Policy Store*. This store is a container for all the policies that have been deployed on the device. Policies are saved on secure data storage, and can only be read by the Persistent Policy Store manager. The *Secure Storage Service* prohibits all access to the policies to other applications. If the device is equipped with a trusted platform module (TPM), this prohibition is enforced by the hardware.

4.2 Application deployment and loading

Applications that are deployed to a device can either be aware of the SxC framework or not. If an application is SxC-aware, it can be deployed with extra metadata that can be used by the SxC framework. Examples of such metadata are: a proof that the application complies with a specific policy, a digital signature of a trusted third party, or other data that is required for some policy enforcement technology.

When the *Application Loader* receives a request to execute an application, it sends the application to the *Application Deployer*. The deployer will first check whether the application was already verified before. In the architectural model, this is achieved by checking whether the application is present in the *Certified Application Database*. However, this database is a pure conceptual component. In the actual .NET implementation, the deployer checks whether the application is signed with a key that is managed by the SxC platform on the device. If an application is signed with this key, it has been processed before and it is considered to be compliant.

A non-compliant application is sent to the *Compliance Engine*. The purpose of this engine is to verify that the application adheres to the system policy, by trying every supported verification or enforcement method. These methods are represented by the different *Compliance Modules*. As soon as one of the compliance modules returns a result that indicates that the application conforms with the system policy, the compliance engine signs the application with the SxC key and executes the program. During this verification process, the actual contents of the executable may be changed. This is for instance the case with inlining. Figure 5 contains a graphical representation of this process.

At deployment time of the policy, the policy writer can choose which policy enforcement mechanisms will be supported by the policy. Three enforcement mechanisms have been implemented in our prototype.

The first supported mechanism is the use of digital signatures set by a third party that certify compliance. This

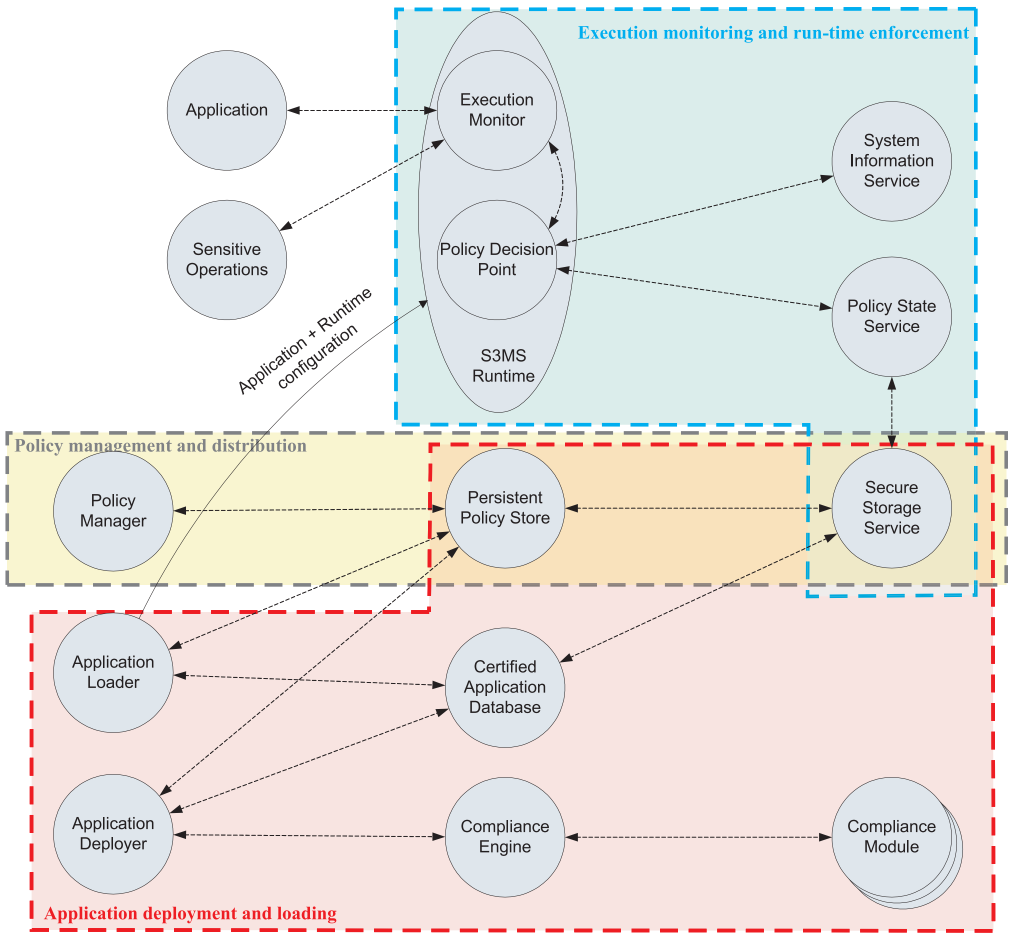

Figure 3: Detailed architecture overview

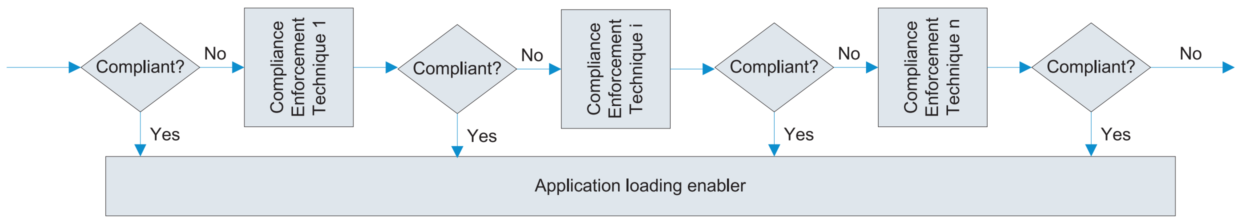
Compliant?
No
Yes
Compliance Enforcement Technique 1
Compliance Enforcement Technique i
Compliance Enforcement Technique n
Application loading enabler

Figure 5: Verifying application/policy compliance

trusted party has a different public/private key pair for every different policy it certifies. If a mobile device administrator wishes to support a policy of this trusted third party, he configures a digital signature compliance module that is initialized with the public key that corresponds to this policy. When a signed application arrives on the device, this compliance module will check the signature against the public key of this third party. If the signature is valid, the application is allowed to run.

The second supported mechanism is matching. At deployment time the target platform checks that the application security claims stated in the contract match with the platform policy. The matching procedure takes as input the application's contract and the mobile platform's policy in a suitable formal representation and then starts a depth first search procedure over the initial state. When a suspect state is reached we have two cases. First, when a suspect state contains an error state of the complemented policy then we report a security policy violation without further ado. Second, when a suspect state does not contain an error state of the complemented policy we start a new depth first search from the suspect state to determine whether it is in a cycle, i.e. it is reachable from itself. If it is we report availability violation.[2, 17]

The third enforcement mechanism that is supported is inlining for run time monitoring. We discuss it as part of the third scenario.

4.3 Execution monitoring and runtime enforcement

The runtime enforcement scenario only comes into play when an application has been inlined. Other enforcement technologies are not active during the execution of the program, because they can guarantee that the application will always comply with the policy before ever running the application. Runtime enforcement takes another approach, and lets applications execute without first formally proving (using either a mathematical proof, or a trust-based proof) that it will not violate the system policy. Instead, the application is instrumented with a monitoring library that can enforce the policy while the application is running.

The inlining process [5, 6, 7, 13, 16] is discussed as a separate scenario, because it is the only enforcement technique that provides backwards compatibility. This technique can be used on applications that are deployed without a contract or any other SxC-related metadata. Figure 6 gives an overview of this process.

The centerpiece of the execution monitoring implementation is the monitoring library - also called the *Policy Decision Point* (PDP). This is the component where the policy logic is located. It interprets the current state and the requested action, and makes a decision to either allow or disallow the action. Calls from the application are received by the *Execution Monitor* and passed to the PDP. The PDP then requests the current state from the *Policy State Service* and may optionally request a number of system-specific settings from the *System Information Service.* The PDP can then execute the policy logic, update the state, and possibly disallow the call to the SRE. Figure 6 depicts this process.

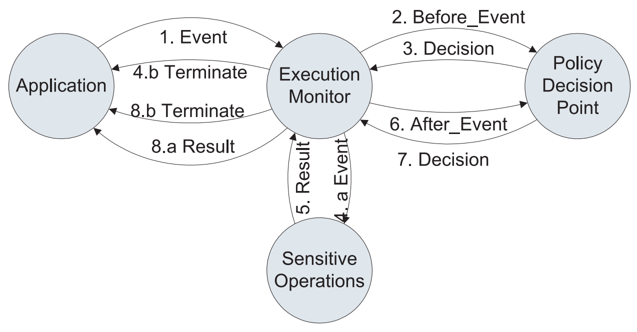

Figure 6: Execution monitoring

When an application arrives on a mobile device, and none of the compliance modules succeeds in verifying that the application does not violate the policy, the application is sent to the inlining module. The inlining module is also a compliance module, but it is always the last compliance module in the list. This is because the inlining process will never fail, but it does have some disadvantages that other enforcement techniques do not have.

In our implementation, only the Execution Monitor is inlined. The Policy Decision Point is kept as a separate component that is called by the Execution Monitor. The inliner opens the executable file, and scans through it looking for security-related calls into the base class library. These calls correspond in our system with CONSPEC SREs. When such a call is found, the inliner inserts new instructions around this call. These instructions call into the policy decision point, which keeps track of the policy state. When, during runtime, the policy decision point notices that an application is going to break the policy, it intervenes and aborts the security-related call.

5. CONCLUSION

We have argued that the classic security architecture of Windows CE is not well adapted to protect users from malicious roaming applications. In particular, the digital signatures used in Windows CE do not offer any guarantee of what the code will actually do. We have introduced the notion of *security-by-contract* and shown that this is superior to the Windows CE security architecture. The SxC architecture can be used to secure untrusted roaming code, and protect users from malicious applications. By defining security contracts for applications during their development, and by matching these contracts with device policies during deployment, compliance with the policy can be verified without incurring run time overhead. Legacy applications, or applications with contracts that do not match, can be executed under the supervision of an execution monitor that will prevent the monitored application from violating the policy.

6. REFERENCES

[1] E. M. Clarke and O. Grumberg and D. A. Peled Model Checking, The MIT Press, 2000.

[2] F. Massacci and I. Siahaan Matching Midlet's Security Claims with a Platform Security Policy using Automata Modulo Theory NORDSEC, 2007.

[3] R. Sekar and V.N. Venkatakrishnan and S. Basu and S. Bhatkar and D.C. DuVarney Model-carrying code: a practical approach for safe execution of untrusted applications In *Proceedings of the 19th ACM symposium on Operating systems principles (SOSP-03)*, pages 15–28, 2003.

[4] 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).

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

[6] 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.

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

[9] MSDN. Code Access Security. `http://msdn2.microsoft.com/en-us/library/930b76w0.aspx`, 2007.

[10] 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.

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

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

[13] D. Vanoverberghe, F. Piessens. Security enforcement aware software development, Elsevier Information & Software Technology, to appear in 2008.

[14] Lieven Desmet, Wouter Joosen, Fabio Massacci, Katsiaryna Naliuka, Pieter Philippaerts, Frank Piessens and Dries Vanoverberghe. A flexible security architecture to support third-party applications on mobile devices. *CSAW '07: Proceedings of the 2007 ACM workshop on Computer security architecture*, pages 19–28, 2007.

[15] 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.

[16] 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.

[17] 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.