Load Time Security Verification [⋆]

Olga Gadyatskaya, Eduardo Lostal, and Fabio Massacci

DISI, University of Trento, Italy
{surname} @disi.unitn.it

Abstract. Modern multi-application smart cards can be an integrated environment where applications from different providers are loaded on the fly and collaborate in order to facilitate lives of the cardholders. This initiative requires an embedded verification mechanism to ensure that all applications on the card respect the application interactions policy.
The Security-by-Contract approach for loading time verification consists of two phases. During the first phase the loaded code is verified to be compliant with the supplied contract. Then, during the second phase the contract is matched with the smart card security policy. The paper focuses on the first phase and describes an algorithm for static analysis of the loaded bytecode on Java Card. The paper also reports about implementation of this algorithm that can be embedded on a real smart card.

1 Introduction

Multi-application smart cards are an appealing business scenario for both smart card vendors and smart card holders. Applications interacting on such cards can share sensitive data and collaborate, while the access to the data is protected by the tamper-resistant integrated circuit environment. In order to enable such cards a security mechanism is needed which can ensure that policies of each application provider are satisfied on the card. Though a lot of proposals for access control and information flow policies enforcement for smart cards exist [2], [9], [10], [12], they fall short when the cards can evolve. The scenario of a dynamic and unexpected post-issuance evolution of a smart card in the field, when applications from potentially unknown providers can be loaded or removed, is novel and not yet treated comprehensively.

For a dynamic scenario, traditionally, run-time monitoring is the preferred solution. But smart cards do not have enough computational capabilities for implementing complex run-time checks. Thus the proposal to adapt the Security-by-Contract approach (initially developed for mobile devices [4]) for smart cards appeared. In the Security-by-Contract (S×C) approach each application supplies on the card its contract, which is a formal description of the application behavior. The contract is verified to be compliant with the application code, and then the system can ensure that the contract matches the security policy of the card.

[⋆] Work partially supported by the EU under grant EU-FP7-FET-IP-Secure Change.

The S×C framework deployed on the card consists of two main components integrated with the card manager. These two components are the ClaimChecker and the PolicyChecker. The ClaimChecker performs extraction of the contract and verifies that it is compliant with the application code. Then the PolicyChecker ensures that the security policy of the card is compliant with the contract. This component is also responsible for updating the security policy after each evolution of the card and maintaining it across updates. A proof-of-concept implementation of the PolicyChecker component is described in [3]. The PolicyChecker prototype was developed in a form of an application installable and runnable on a smart card, thus this prototype demonstrated feasibility of the embedded PolicyChecker implementation.

The loading time verification mechanism for secure application interactions requires a careful investigation of multi-application smart card platforms. We have chosen to focus on the Java Card technology as one of the current leaders for open multi-application smart cards implementation. We present in Section 2 a brief overview of this technology and then we outline the S×C solution for Java Card (Section 2.2) emphasizing the changes to the platform. The Java Card internals are discussed more deeply in Section 3. In this section we focus on the loading process and the run-time environment. We then concentrate on the application contracts in Section 4, discussing the contract can be created and the mechanism to deliver it securely on the card.

In this paper we propose an algorithm for the ClaimChecker component of the S×C framework for the Java Card technology (Section 5). The ClaimChecker parses the bytecode loaded on the card, extracts the contract and compares it with the actual code of the application. The ClaimChecker component is an intricate part of the S×C framework, because its implementation requires access to the loaded application code. We report about implementation of the ClaimChecker algorithm in C. For on-card prototypes it is important that they have small memory footprints. We therefore present the memory usage statistics (for EEPROM and RAM) that demonstrates feasibility of the approach (Section 6). The related work is discussed in Section 7 and we conclude with Section 8.

The main contributions of our current work are:

- The specification of the application contracts;
- The algorithm for the ClaimChecker component of the S×C framework;
- The implementation of the algorithm in C demonstrating that the algorithm can be embedded onto an actual smart card chip.

2 The S×C Architecture for the Java Card Platform Evolution

Java Card is a popular middleware for multi-application smart cards that allows post-issuance installation and deletion of applications. Application providers develop *applets* (Java Card applications) in a subset of the Java language. This subset is object-oriented, but misses some traditional Java data types and features. Full description of the Java Card language is provided in [11].

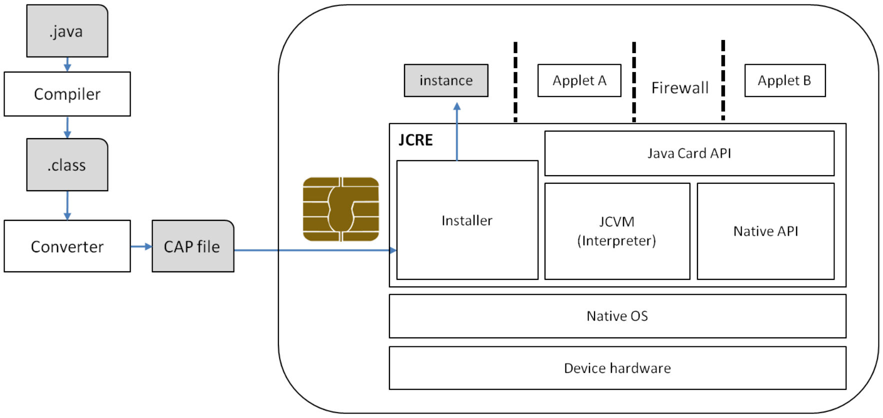

Fig. 1. The Java Card Architecture and the Loading Process

Currently smart cards in the field run on the Java Card version 2.2.2, thus our proposal supports this version. Also a new specification for Java Card 3.0 is published, but its developments are currently frozen due to, among all, security concerns. However, the S×C approach we advocate in the future can be ported also for the third generation of Java Cards.

2.1 The Java Card Platform Architecture and the Loading Process

Figure 1 presents the architecture of a chip with the Java Card platform installed and the application loading process. The architecture comprises several layers which include device hardware, an embedded operating system (native OS), the Java Card run-time environment (JCRE) and the applications installed on top of it [11]. Important parts of the JCRE are the Java Card virtual machine (JCVM) (its Interpreter part) and the Installer, which is an entity responsible for post-issuance loading and installation of applications.

Applets are supplied on the card in packages. The source code of a package is converted by the application providers into class files and then (using a Converter which is actually an off-card part of the JCVM) into a CAP file. The CAP file is transmitted onto a smart card, where it is processed, linked and transformed into a platform-specific executable format (defined by the platform developer). Application providers do not need to consider different on-card executable formats, as they are just required to supply a correct (compliant with the Java Card specifications) CAP file. Then, upon finalization of the linking process, an applet instance is installed.

One of the main technical obstacles for the verifier running on Java Card is unavailability of the application code (in a known format of a CAP file) for reverification purposes after linking. Thus the application policy cannot be stored within the application code itself, as the verifier will not have access to it later.

Applications on Java Card are separated by a firewall and the interactions between applets from different packages are mediated by the JCRE. If two applets belong to different packages, their *contexts* are different, and the Java Card firewall confines applet's actions to its designated context. Thus, normally, an applet can reach only objects belonging to its own context. The only applet's objects accessible through the firewall are methods of specific *shareable interfaces*, also called *services*. A shareable interface is an interface that extends `javacard.framework.Shareable`.

If an application A implements some services, it is called a *server*. An application B that tries to call any of these services is called a *client*. A typical scenario of service usage starts with a client's request to the JCRE for a reference to A's object (that is implementing the necessary shareable interface). The firewall passes this request to application A, which decides if the reference can be granted or not. If the decision is positive, the reference is passed through the firewall and is stored by the client for further usage. The client can now invoke any method declared in the shareable interface which is implemented by the referenced object. During invocation of a service a context switch will occur, thus allowing invocation of a method of the application A from a method of the application B. A call to any other method, not belonging to the shareable interface, will be stopped by the Java Card firewall [11].

As all applet interactions inside one package are not controlled by the firewall and due to the fact that a package is loaded in one pass (thus it is not possible to load a malicious applet in one package with an honest one), we consider that one package contains only one applet and there is an one-to-one correspondence between packages and applications.

2.2 Security-by-Contract for Java Cards

The Security-by-Contract framework for smart cards provides an extension of the Java Card architecture with two main components: the ClaimChecker and the PolicyChecker. The loading time verification process is performed by these components. Another addition to the platform is the Policy applet. The applet appears due to the fact that only applications can allocate space in EEPROM (mutable persistent memory), that is the only type of memory suitable to store the security policy across updates. We have solved the issues of the application code unavailability after linking by storing the security policy (that incorporates each installed application policy) in a separate accessible Policy applet.

Figure 2 depicts the proposed architecture, the additions to the JCRE are in long dashed rectangles. More details about the architecture and its implementation are given in Section 6.

This paper focuses on the ClaimChecker component, that is responsible for contract-code matching. Thus only the application loading scenario is relevant for the ClaimChecker, as during the application removal the code has already been verified to be compliant with the contract. The workflow of the loading scenario follows (only the actions relevant to the S×C process are listed):

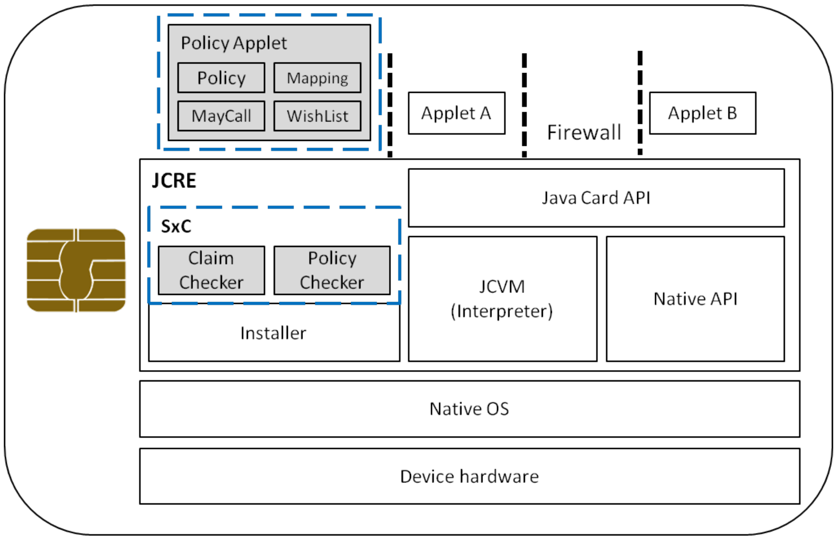

Fig. 2. The Security-by-Contract Extended Architecture

1. New package B is loaded (CAP file is transmitted to the card, the Installer receives it and saves into the modifiable memory);
2. The Installer retrieves the current security policy from the Policy applet and invokes the ClaimChecker;
3. The ClaimChecker gets the contract from the CAP file and runs the verification algorithm;
4. If the ClaimChecker succeeds, it invokes the PolicyChecker and sends it the pointer to the contract;
5. The PolicyChecker gets the security policy and runs the contract-policy compliance algorithm;
6. If the PolicyChecker succeeds, it communicates the update to the security policy;
7. If the ClaimChecker and the PolicyChecker succeeded, B is linked and stored in the persistent memory, and the card security policy is updated to include its contract. Otherwise, B is rejected and removed from the memory.

The S×C framework verifies that the following two properties will be satisfied on the card after any accepted change:

- *Service Invocation Security*: If an application A calls during its execution a service s of an application B, then B has authorized A to access s in B's security policy;
- *Available Functionality*: If an application A declared that it needs a service s of an application B in order to be functional, then the service s is indeed provided by B.

The formal proof of these properties established on the Java Card platform by the S×C framework relied on the fact of existence of a sound ClaimChecker algorithm [6]. In fact, the ClaimChecker component is the corner stone of the S×C framework, and it's specification and implementation were the key tasks while building the framework.

2.3 Threats to Validity of the SxC Approach

The S×C approach and the guarantees it provides are ensured with the certain assumptions made. Obviously, soundness of the framework algorithms relies on the correct implementation of the JCRE and the JCVM, and we assume they are in full compliance with the specifications [11]. For the invoked services we rely on the trustworthiness of the Compiler, that has to be compliant with the Java type safety requirements. We also assume that the bytecode was not tampered with after compilation and conversion.

For the provided services, we rely on the trustworthiness of the servers. Indeed, in the S×C paradigm provision of a service requires a commitment to implement the necessary shared object and to provide a correct object reference in response to a request from any client. The server has to rely on the loading time verification by the S×C framework and it should not use the access control mechanisms embedded into the code. We also have to assume the correctness of the server implementation.

The S×C framework enforces access control for direct services usage. We would like to mention that the current access control enforcement on Java Card is embedded into the application code. Traditionally, the server will receive an AID of the client requesting its service from the JCRE and check that this client is authorized before granting it the reference to the object (that can implement multiple services). Once the object reference is received, the client can access all the services within this object and it can also leak the object reference to other parties. The S×C framework checks the authorizations for each service access, thus the object reference leaks are no longer a security threat.

3 The Java Card Internals

We now present the Java Card platform details that were used to build the S×C framework and to guarantee the security it enforces. In order to realize the application interaction scenario the client has necessarily to import the shareable interface of the server and to obtain the *Export file* of the server, that lists shared interfaces and services and contains their tokens. The server's Export file is necessary for conversion of the client's package into a CAP file. In a CAP file all methods are referred to by their tokens, thus during conversion from class files into a CAP file the client needs to know correct tokens for services it invokes from other applications. As shareable interfaces and Export files do not contain any implementation, it is safe to distribute them.

Tokens are used by the JCRE for linking on the card similarly as Unicode strings are used for linking in standard Java class files. A service s can be identified as a tuple $\langle A, I, t\rangle$, where A is a unique application identifier (AID) of the package that provides the service s, I is a token for a shareable interface where the service is defined and t is a token for the method in the interface I. Further we will sometimes omit an AID and will refer to a service as a tuple $\langle I, t\rangle$.

We discuss now the CAP files and service invocation details used further in the ClaimChecker algorithm. The JCRE imposes some restrictions on method invocations in the application code [11]. Only the opcode `invokeinterface` in the code allows to perform the desired context switch. Thus, in order to collect all potential service invocations we need to analyze the bytecode and infer from the `invokeinterface` instructions possible services to be called.

Opcode "`invokeinterface` *nargs* I t" has 3 (explicit) operands, as defined in the JCVM specification [11, Sec. 7.5.54]. Operand *nargs* defines a number of invoked method arguments (plus 1), operand I provides an index in the Constant Pool component where the structure at this index should correspond to a reference to an interface class and operand t is an interface method token for the method to be invoked. Meanwhile, the stack before execution of the opcode `invokeinterface` *nargs* I t should contain on its top an object reference R, followed on the operand stack by $nargs{-}1$ words of arguments.

Intuitively, while analyzing the code, we could try to track the object references on the stack, thus inferring all possible objects of the server that could be referenced by the applet during `invokeinterface` opcode execution. But unfortunately, it is only the server's code that defines which objects it will provide and to whom. It is even possible the server is not yet on the card when the client is loaded (and it could never arrive). Thus our analysis can be only as precise as the tokens provided in the client's code.

4 Application Contract

Let $A.s$ be a service s declared in a package A. The contract consists of two parts: a *claim* and a *policy*. AppClaim specifies provided (Provides set) and invoked (Calls set) services. We say that the service $A.s$ is provided if applet A is loaded and service s exists in its code. Service $B.m$ is invoked by A if A may try to invoke $B.m$ during its execution. The AppClaim will be verified for compliance with the bytecode (the CAP file) by the ClaimChecker.

The application policy AppPolicy contains authorizations for services access (sec.rules set) and functionally necessary services (func.rules set). We say a service is necessary if a client will not be functional without this service on board. The AppPolicy lists applet's requirements for the smart card platform and other applications loaded on it.

Thus the application contract has the following structure: $\mathsf{Contract} = \langle \mathsf{AppClaim}, \mathsf{AppPolicy}\rangle$, where $\mathsf{AppClaim} = \langle \mathsf{Provides}, \mathsf{Calls}\rangle$ and $\mathsf{AppPolicy} = \langle \mathsf{sec.rules}, \mathsf{func.rules}\rangle$.

A functionally necessary service for applet A is the one which absence on the platform will crash A or make it useless. For example, a transport application normally requires some payment functionality to be available. If a customer will not be able to purchase the tickets, she would prefer not to install the ticketing application from the very beginning. It is required that for every application A $\mathsf{func.rules}_\mathsf{A} \subseteq \mathsf{Calls}_\mathsf{A}$.

An authorization for a service access includes the package AID of the authorized client (the format of an authorization will be discussed further). The access rules have to be specified separately for each service and each client that the server wants to grant access.

4.1 The Contract Delivered on the Card

Contracts can be delivered on the card within Custom components of the CAP files. CAP files carrying Custom components can be recognized by any Java Card Installer, as the Java Card specification requires.

Custom components require to have a tag and an AID. We have defined the tag to be 0xC3 and the AID 0x010203040506C3 (but these can be easily modified). These details of the Custom component and its length are listed in the Descriptor component of the CAP file.

```
contract {
    u2 provides_count
    provides_info  provides[provides_count]
    u2 calls_count )
    calls_infocalls[calls_count]
    u2 secrules_count
    secrules_info  secrules[secrules_count] }
```

Table 1. Structure of the Custom component Containing Contract

The scheme of the contract is illustrated in Table 1. The order of the contract attributes is expected to be: Provides, Calls, sec.rules. Thus we just add the number of corresponding elements before each attribute. Elements of each attribute have specifically defined structures (we use structures and naming that are similar to the ones defined for CAP files [11], there u2 corresponds to 2 bytes). The contract is just a byte array, but specifying structures corresponding to each entry allows us to perform the contract extraction efficiently. More information on the structures is available in the companion technical report [7].

Functionally necessary services are a subset of called services, thus we just tag necessary services among the called ones. The value of specific funcrules_tag is set to 0x01 if the service should be listed in func.rules. Otherwise the tag value should be 0x00.

4.2 Contract Population

Now we discuss how to populate the contract and embed it into the CAP file. Following are the rules for contract population.

- *Provided Services.* A service is required to be listed in the Provides set if it is a method of an interface extending Shareable. A service is listed in Provides array as a pair $\langle \mathsf{I}, \mathsf{t} \rangle$, where I is the Export file token for shareable interface and t is the Export file token for the method (1 byte each).
- *Called and Functionally Necessary Services.* An application provider should list a service (belonging to another package) in the Calls set, if an invocation of this service is present in the code of the applet. A service from a package with AID XXX is listed in the contract as $\langle XXX, \mathsf{I}, \mathsf{t}, \mathsf{funcrules\_tag} \rangle$, where funcrules_tag tags if this service is also functionally necessary or not. For optimization purposes, the Calls set is then restructured to separate services provided by different servers. The AIDs are space-consuming objects (can take up to 16 bytes) and avoiding their repetitions where possible can bring significant space savings.
- *Authorization Rules.* An authorization rule is listed in the sec.rules set as a pair containing the service details (defined as a provided service) and the authorized client package AID. Thus the structure is the same as for a called service, with a difference that no tag for functionality is needed: $\langle AID, \mathsf{I}, \mathsf{t} \rangle$. Then the same optimization strategy as for called services is applied.

The CAP file is in fact a JAR archive with a known structure. In order to embed the contract created by these rules and in compliance with the structure from Table 1, our CAP modifier takes the CAP file generated with the standard Java Card tools and appends the Contract Custom component within it, modifying the Descriptor component accordingly (as the specification requires).

5 The Claim Checker Algorithm

The ClaimChecker component is responsible for verification of the contract and the bytecode compliance. Thus it has to establish that the services from $\mathsf{Provides_A}$ exist in package A and the services from $\mathsf{Calls_A}$ are indeed the only services that A can try to invoke in its bytecode. The details of the service invocation instructions were already discussed in Section 3. The goal of the ClaimChecker algorithm is to collect for each `invokeinterface` opcode the method index t and the Constant Pool index I. Then we can compare the collected set with the set Calls of the contract. We emphasize that operands of the `invokeinterface` opcode are known at the time of conversion into a CAP file and thus are available directly in the bytecode. All methods of the application are provided in the Method Component of the application's CAP file, an entry for each method contains an array of its bytecodes. Exported shareable interfaces are listed in the Export component of the CAP file and flagged in the Class component. The strategy for the ClaimChecker is to ensure that each service listed in the Provides set is meaningful and no other provided services exist.

5.1 The Algorithm

The ClaimChecker Algorithm 5.1 processes the CAP file components in order of appearance with a standard Installer, the comments on the steps of the algorithm are inlined. The presented algorithm is a script for an actual implementation of the ClaimChecker. The received CAP file is a byte array, but it is structured accordingly to the CAP file specification [11]. Thus the algorithm refers directly to items (fields) of the structures defined in the CAP file specification, such as CONSTANT_Classref_info structure or Interface_info structure. The algorithm also uses variable-length arrays and arrays of tuples, that do not exist on a smart card. The actual implementation explores just constant-length byte arrays. The function offset(b) is used in the algorithm, that serves as a pointer and returns a structure S which is provided at the given offset b.

Soundness of the algorithm for the service invocation security (in assumption of a correct JCRE implementation) follows from the fact that only invokeinterface opcode allows the JCRE to switch the context, thus any application can only use this opcode to invoke services. Thus the ClaimChecker will accept only the applications that have declared the invoked services set Calls honestly. We discuss the soundness proof in more details in the companion technical report [7].

6 Implementation of the Claim Checker

We have implemented full S×C prototype in C, as it is a standard language for smart card platform components implementation. In this section we will give an overview of the prototype architecture and implementation details, and then we will focus on the ClaimChecker component implementation and present the memory usage statistics.

The main C components of the S×C prototype are:

SxCInstaller This component is an interface with the Installer. SxCInstaller calls the ClaimChecker that in a positive case (contract and bytecode are compliant) will return the address of the contract in the Contract Custom Component of the CAP file being loaded. The SxCInstaller also comprises (for memory saving reasons) the PolicyChecker component. Any negative result either in the ClaimChecker or PolicyChecker algorithms or errors during parsing of the CAP file are propagated as *false* to the SxCInstaller, that returns a *boolean* to the Installer.

ClaimChecker This component is called by SxCInstaller. It carries out the check for the compliance between the contract and the CAP file. The check is carried out after parsing the CAP file. By means of the functions of the CAPlibrary library for CAP file parsing on-card (discussed further), this component gets the initial address of the components it needs from which it can eventually parse the rest of the components. If the result is positive, the ClaimChecker will return the address of the contract of the application in the Contract Custom component. Any error during parsing or a negative result from the ClaimChecker leads to return of *null*.

We now discuss the implementation of the proposed algorithm 5.1 in C. In order to reduce the amount of RAM memory the prototype uses, instead of copying parts of the CAP file (for example, the delivered contract) we operated with the pointers to the corresponding parts of the CAP file. We have used a set of functions to access the parts of CAP file components, calling it the CAPlibrary library, assuming that for each component we can retrieve its location in the card memory and its size. These functions belong to a standard functionality of the Installer. As we did not have access to an actual smart card platform implementation, we have implemented these functions in C for testing purposes, but we do not include this implementation in the following memory statistics of the prototype.

6.1 The Policy Checker and the Policy Applet Implementation

Due to the lack of space we do not report the details of the PolicyChecker implementation. However, we present the security policy data structures just to give a flavor of this part of the system.

The security policy stored on the card consists of contracts of the currently loaded applications. A contract in the form supplied on the card is a space-consuming structure. Each AID can occupy up to 16 bytes. Therefore, a set of sec.rules with authorizations given for, for instance, 8 applets can occupy up to 144 bytes. We would like to save the space necessary for storing the security policy while making the operations with the contracts (performed by the PolicyChecker for contract-policy compliance check) faster. To do so we have resolved to store the security policy on the card in a bit vectors format. The current data structure for security policy assumes there can be up to 4 loaded applets, each containing up to 8 provided services. Thus the security policy is a known data structure with a fixed format, the bits are taking 0 or 1 depending if the applet is loaded or the service is called/provided. This structure is called Policy in Figure 2 (see the Policy applet structures). The amount of the loaded applets can potentially be modified dynamically (if the 5th applet arrives).

The chosen security policy data structure requires the table on the card that maintains correspondence between the number the applet gets in the on-card security policy structure and the actual AID of the package, and between the provided service token and the number of this service in the policy data structure. We store this correspondence in the Mapping object. The other two objects that are part of the on-card security policy are MayCall list and WishList list. The MayCall list contains the potential future authorizations, necessary for a case when a loaded application carries a security rule for some application not yet on the card. These authorizations have to be stored on the card in the form they were supplied (with the client's AID), thus they are space-consuming objects. The WishList object is a set of services that are called by applications but are not yet on the card, because the server is not yet loaded, or because the current version of the server does not provide this service. The WishList set maintains the AIDs of the service providers and the services as tuples $\langle I, t \rangle$. Again, the WishList entries are space-consuming, as they contain AIDs of desired packages.

The Policy applet has to communicate the security policy of the card to the PolicyChecker component that will run the contract-policy compliance check.

This communication is currently implemented through the APDU buffer, that is a common object for communication for all entities on the card. We have assumed the size of the APDU buffer to be 255 bytes, as it is one of the standard implementations. Thus the full security policy (the Policy, Mapping, WishList and MayCall objects) has to fit within 255 bytes. That is why we have developed such a small security policy object, which is enough to fit only 4 loaded applets, and we have set restrictions on the number of authorizations in the MayCall object and desired services in the WishList object. We are currently investigating if there are better means for communication (in both directions) of the C components and the applets on the card that will allow us to implement a bigger and dynamically scalable policy model.

6.2 Details of the Claim Checker Implementation Memory statistics

We now present an overview of the memory consumption by the ClaimChecker prototype. The most important characteristics for an on-card component are RAM and EEPROM consumption. EEPROM space is required to store the prototype and the necessary data between the card sessions. RAM memory, on the other hand, is used to store the temporary data while the verification is performed. We can consider as an example of a modern smart card chip P5CT072 device from Philips Semiconductors [13]. The chip has 72 KB of EEPROM, 160 KB of ROM and 4608 bytes of RAM. Therefore, we can assume that the verifier embedded on the card should occupy at most 20-30 KB of EEPROM.

As we cannot install the prototype on a card and measure its footprint in the linked state, we explored two metrics for the EEPROM usage: the size of the object files in C and the number of lines of code (LOCs). The ClaimChecker prototype requires 6522 bytes (6.36 KB) to store the object files. The .c file of the ClaimChecker contains 155 LOCs, and the .h file contains 7 LOCs.

RAM usage is also very important, as over-consumption of RAM by the prototype can lead to the denial of service. The higher is the RAM consumption, the less is the level of interoperability of the prototype, because some cards cannot provide a significant amount of RAM for the verifier which has to run in the same time with the Installer. We have used a temporary array of 255 bytes to store the necessary computation data. 255 bytes is a small temporary memory buffer which ensures the highest level of interoperability for the prototype.

7 Related Work

A plethora of works exist for verification of application interactions security on Java Card. Ghindici et al [8] proposed an approach for the information flow verification on small embedded systems. Each application gets a certificate with the information flow signature of each method, and on device these signatures are checked using the proof-carrying-code techniques. The expressive information flow security properties captured the interactions of applications on the platform.

This approach is extremely powerful, but has not yet been demonstrated to be implementable on Java Card.

A lot of papers were dedicated to the static scenarios, when all the applications are known a priori and can be verified using off-card facilities [10], [9], [2], [12]. Dynamic scenarios were considered in [1] and [5]. Avvenuti et al [1] developed the tool JBIFV that was similar to a bytecode verifier and could verify absence of illicit information flows between Java applications. The drawback of this tool in a dynamic scenario is that the applications have to be analyzed locally prior being loaded on the card. Thus the card is not empowered with the ability to make decisions itself.

In the work of Fontaine et al [5] the authors consider the same dynamic scenario as we did and propose an on-card loading time verification approach for transitive control flow policies that can control application collusuons. Their algorithm performs verification while parsing the received CAP file. With respect to [5] our work enforces less stronger policies. However, the S×C approach offers greater flexibility than the transitive control flow policies proposed by Fontaine et al. Indeed, as we have mentioned before, the application code after linking is not available for reverification. Thus the approach by Fontaine et al, that makes the policy compliance verification simultaneously while parsing the bytecode, requires to store a significant amount of additional data related to the invoked methods, what can be a prohibitive requirement for an on-card prototype.

8 Conclusions and Future Work

In the paper we have presented the ClaimChecker component of the S×C framework for the Java Card-based smart cards. This component's duty is to ensure compliance of the applet's contract with its code. The contracts are delivered within the Custom component of the CAP file, and they list provided and called services of the applets and the application providers' policies. We have proposed the structure of the contracts expected by the ClaimChecker in the notation similar to the CAP file contents specification [11].

Once the CAP file is received the ClaimChecker invoked by the Installer component on the card, extracts it and analyzes whether the contract is compliant with the bytecode. Our focus is on the invoked services and we have presented the sound algorithm that can capture the comprehensive list of the called services and match it with the claimed list. The implementation of the algorithm is straight-forward provided that one has access to a smart card platform implementation and knows the necessary APIs to access the CAP file contents.

For the future work we plan to validate the S×C framework implementation within the Secure Change project with the help of Gemalto (an industrial partner in the project). We have implemented the algorithm in C and the memory statistics we have provided ensures that a proof-of-concept embedded implementation is possible. Another interesting direction of the future work is richer contracts. We believe that the perfect trade-off between verification time, richness of the contracts and flexibility of the approach for evolution is yet to be found.

References

1. M. Avvenuti, C. Bernardeschi, and N. De Francesco. Java bytecode verification for secure information flow. *SIGPLAN Not.*, 38:20–27, December 2003.
2. P. Bieber, J. Cazin, V. Wiels, G. Zanon, P. Girard, and J-L. Lanet. Checking secure interactions of smart card applets: Extended version. *J. of Comp. Sec.*, 10(4):369–398, 2002.
3. N. Dragoni, E. Lostal, O. Gadyatskaya, F. Massacci, and F. Paci. A load time Policy Checker for open multi-application smart cards. In *Proceedings of the 2011 IEEE International Symposium on Policies for Distributed Systems and Networks.*
4. N. Dragoni, F. Massacci, K. Naliuka, and I. Siahaan. Security-by-Contract: towards a semantics for digital signatures on mobile code. In *Proc. of EuroPKI-07*, volume 4582 of *LNCS*, pages 297 – 312. Springer-Verlag, 2007.
5. A. Fontaine, S. Hym, and I. Simplot-Ryl. On-device control flow verification for java programs. In *Engineering Secure Software and Systems*, volume 6542 of *Lecture Notes in Computer Science*, pages 43–57. Springer Berlin / Heidelberg, 2011.
6. A. Fontaine, S. Hym, I. Simplot-Ryl, O. Gadyatskaya, F. Massacci, F. Paci, J. Jurgens, and M. Ochoa. D6.3 Compositional technique to verify adaptive security at loading time on device. *SecureChange EU project public deliverable, www.securechange.eu*, 2010.
7. O. Gadyatskaya, E. Lostal, and F. Massacci. Load time security verification. The Claim Checker. Technical Report DISI-11-471. On the web at http://eprints.biblio.unitn.it.
8. D. Ghindici and I. Simplot-Ryl. On practical information flow policies for java-enabled multiapplication smart cards. In *Proceedings of CARDIS 2008*, volume 5189 of *LNCS*, pages 32–47. Springer-Verlag, 2008.
9. P. Girard. Which security policy for multiplication smart cards? In *USENIX Workshop on Smartcard Technology.* USENIX Association, 1999.
10. M. Huisman, D. Gurov, C. Sprenger, and G. Chugunov. Checking absence of illicit applet interactions: a case study. In *FASE'04*, volume 2984 of *LNCS*, pages 84–98. Springer-Verlag, 2004.
11. Sun Microsystems. Virtual Machine and Runtime Environment. Java Card^{TM} platform. Specification 2.2.2, Sun Microsystems, 2006.
12. G. Schellhorn, W. Reif, A. Schairer, P. Karger, V. Austel, and D. Toll. Verification of a formal security model for multiapplicative smart cards. In *ESORICS'00*, volume 1895 of *LNCS*. Springer-Verlag, 2000.
13. Philips Semiconductors. P5CT072 Secure Dual Interface PKI Smart Card Controller. On the web at http://www.usmartcards.com/images/pdfs/pdf-199.pdf.

```
Require: A CAP file.
Ensure: True/False, Contract.
//Header Component: get the current package AID
byte $CurrentPID[16]$ gets current package AID;
// Import Component: get package AIDs of imported packages
add ⟨imported package ID, internal imported package token (index in the current array)⟩ to $ImportedPackages$;
// Constant Pool Component: get imported interfaces
for all elements of the Constant Pool array of the type class_ref do
    if the high bit equals to 1 then
        add ⟨imported package token, external class or interface token, internal class or interface token (index in the current array)⟩ to $ImportedInterfaces$;
// Method Component: parse bytecode of the methods to identify called services
for each method of the methods[ ] array do
    if invokeinterface X Y Z opcode is in the method then
        add ⟨internal token of the interface, external token of the method⟩ to $InvokedServices$;
// Export Component: get tokens of shareable interfaces
for $i = 0$ to class_count do
    add ⟨offset into the Class component, external interface token⟩ to $ExportedInterfaces$;
// Descriptor Component: get external tokens of provided services
for $i = 0$ to classes_count do
    if classes[i] has a flag ACC_INTERFACE = 0x40 AND exists $\langle int\_offset, I\rangle \in ExportedInterfaces$ such that int_offset = classes[i].this_class_ref then
        // This interface is shareable and its external token was collected
        for all methods of this interface do
            add ⟨external interface token, method token⟩ to $ListedServices$;
// Custom Component: get Contract
for $j = 0$ to provides_count do
    add ⟨external interface token, external method token⟩ to $ContractProvides$;
for $j = 0$ to calls_count do
    add ⟨external interface token, external method token, AID⟩ to $ContractCalls$;
    if funcrules_tag = 0x01 then
        add ⟨external interface token, external method token, AID⟩ to $ContractFuncrules$;
for $j = 0$ to secrules_count do
    add ⟨external interface token, external method token, AID⟩ to $ContractSecrules$;
// The Final Check: return true iff the collected sets match with the Contract
Check of called services: construct the same structure as in the contract and check for mutual inclusion
for each $\langle I, t, AID\rangle \in ContractCalls$ do
    add $\langle I, t, P\rangle$ to $CALLS$ such that $\langle P, AID\rangle \in ImportedPackages$;
for each $\langle P, I, cpt\rangle \in ImportedInterfaces$ and $\langle cpt, t\rangle \in InvokedServices$ do
    add $\langle P, I, t\rangle$ to $CALLS1$;
if $CALLS1 \neq CALLS$ then
    return False;
else
    // Check for provided services: all services in ContractProvides set have valid interface and method tokens
    if $ContractProvides \neq ListedServices$ then
        return False
    else
        return {True, $CurrentPID$, $Contract$}
```

Algorithm 5.1: The Claim Checker Algorithm