LOAD TIME SECURITY VERIFICATION. THE CLAIM CHECKER

Olga Gadyatskaya, Eduardo Lostal and
Fabio Massacci

July 2011

Technical Report # DISI-11-471
Version 1.0

Technical Report: Load Time Security Verification. The Claim Checker *

Abstract

Modern multi-application smart cards can become 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 report focuses on the first phase and describes an algorithm for static analysis of the loaded bytecode on Java Card. We also report 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 [4, 13, 14, 19], 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 [9]) 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 [8]. 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 the 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 creation process 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 the official specifications [17].

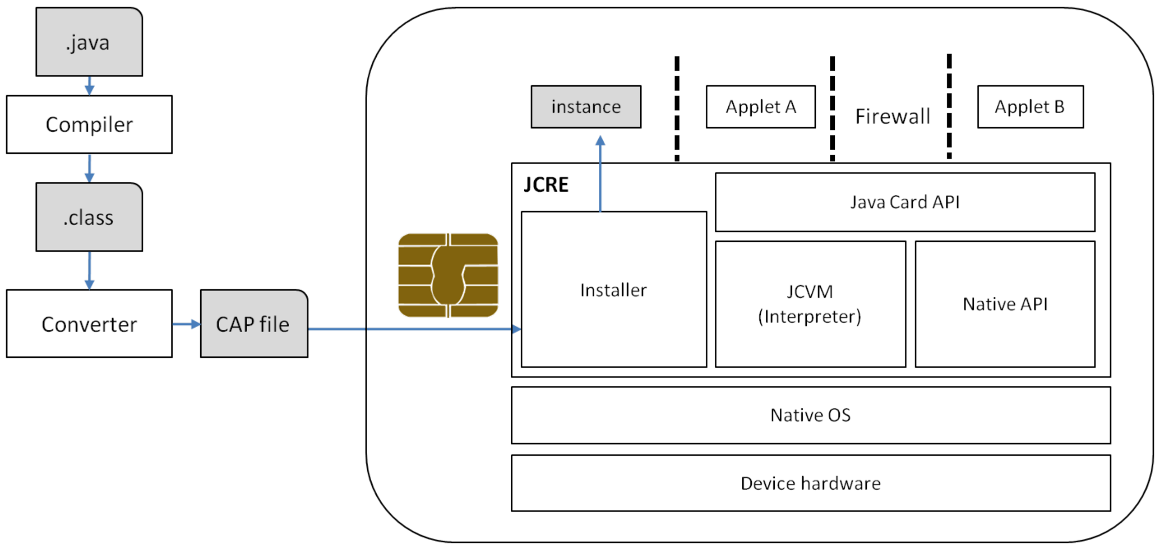


Figure 1: The Java Card Architecture and the Loading Process

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 [6]. 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 only the application code itself, as the verifier will not have access to it.

Applications on Java Card are separated by a firewall and the interactions between ap-

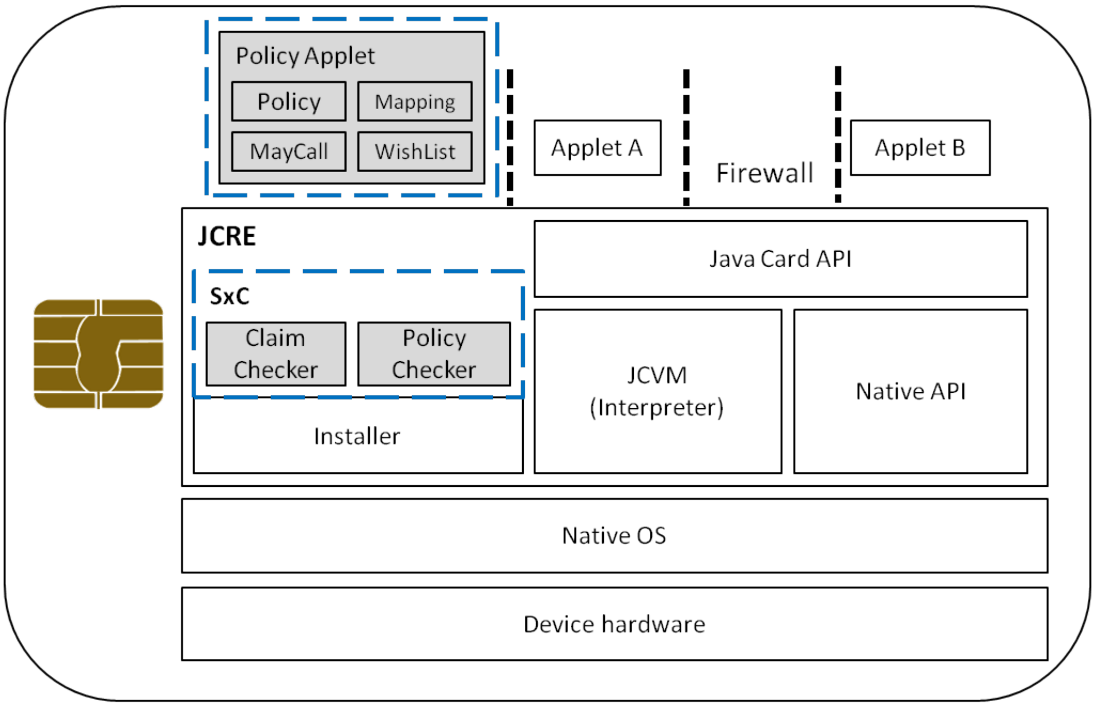


Figure 2: The Security-by-Contract Extended Architecture

plets 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 a 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 [6, 17].

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 blue rectangles. More details about the architecture and an 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):

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 evolution:

- *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.

These properties are guaranteed together by the ClaimChecker and the PolicyChecker components [11]. 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. 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 S×C 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 [17]. For the invoked services we rely on the trustworthiness of the Compiler, that has to be compliant with the Java type safety requirements. It is also a crucial assumption that the bytecode is trustworthy and it was not tampered with after the 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 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, which 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 in the same fashion 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 s in the interface I. Further we will sometimes omit an AID A and will refer to a service as a tuple $\langle I, t\rangle$.

We discuss now the CAP files and service invocations details used further in the ClaimChecker algorithm. The JCRE imposes some restrictions on method invocations in the application code [17]. 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 [17, 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.

3.1 The CAP File Details

CAP files are converted Java Card packages ready to be shipped on a smart card. The structure of the CAP files are specified in the JCVM specification [17]. A CAP file consists of several components, some of them are necessary and some are optional. Table 1 contains the list of all CAP file components.

Header component
Directory component
Applet component
Import component
Constant Pool component
Class component
Method component
Static Field component
Reference Location component
Export component
Descriptor component
Debug component //*optional*
Custom component //*optional, one CAP file can contain several Custom components*

Table 1: The CAP file structure

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.

Definition 4.1 *Let $\Delta_{\mathcal{A}}$ be a domain of applications and $\Delta_{\mathcal{S}}$ be a domain of services.* $\mathsf{AppClaim_A}$ *of an application A is a tuple* $\langle \mathsf{Provides_A}, \mathsf{Calls_A} \rangle$*, where* $\mathsf{Provides_A} \subseteq \Delta_{\mathcal{S}}$ *is a set of the services A provides on the card and* $\mathsf{Calls_A} \subseteq \Delta_{\mathcal{S}}$ *is a set of services that A may call during its execution.*

$\mathsf{AppPolicy_A}$ *of an application A is a tuple* $\langle \mathsf{sec.rules_A}, \mathsf{func.rules_A} \rangle$*, where a relation* $\mathsf{sec.rules_A} \subseteq \mathsf{Provides_A} \times \Delta_{\mathcal{A}}$ *defines which applications are authorized to use services of A,* $\mathsf{func.rules_A} \subseteq \Delta_{\mathcal{S}}$ *is a set of services functionally necessary for application A.*

$\mathsf{Contract_A}$ *is a tuple* $\langle \mathsf{AppClaim_A}, \mathsf{AppPolicy_A} \rangle$.

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.

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 and are presented in Table 2.

```
custom_component_info {
    u1 component_tag
    u2 size
    u1 AID_length
    u1 AID[AID_length] }
```

Table 2: Details of the Custom component

```
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 3: Structure of the Custom component Containing Contract

The scheme of the contract is illustrated in Table 3. 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 different structures, that are provided in Table 4 (we use structures and naming that are similar to the ones defined for CAP files [17], there u1 corresponds to 1 byte and 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.

Functionally necessary services are a subset of called services: $\mathsf{func.rules}_\mathsf{A} \subseteq \mathsf{Calls}_\mathsf{A}$, thus just tag necessary services among the called ones. The value of funcrules_tag is set to 0x01 if the service should be listed in func.rules. Otherwise the tag value should be 0x00.

```
provides_info {
    u1 interface_token
    u1 service_token }
calls_info {
    u1 interface_token
    u1 service_token
    u1 server_AID[16]
    u1 funcrules_tag }
secrules_info {
    u1 client_AID[16]
    u1 secrules_applet_count
    secrules_applet_info  secrules_applet[secrules_applet_count] }
secrules_applet_info {
    u1 interface_token
    u1 service_token }
```

Table 4: Contract Attributes Structures in the CAP File

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.

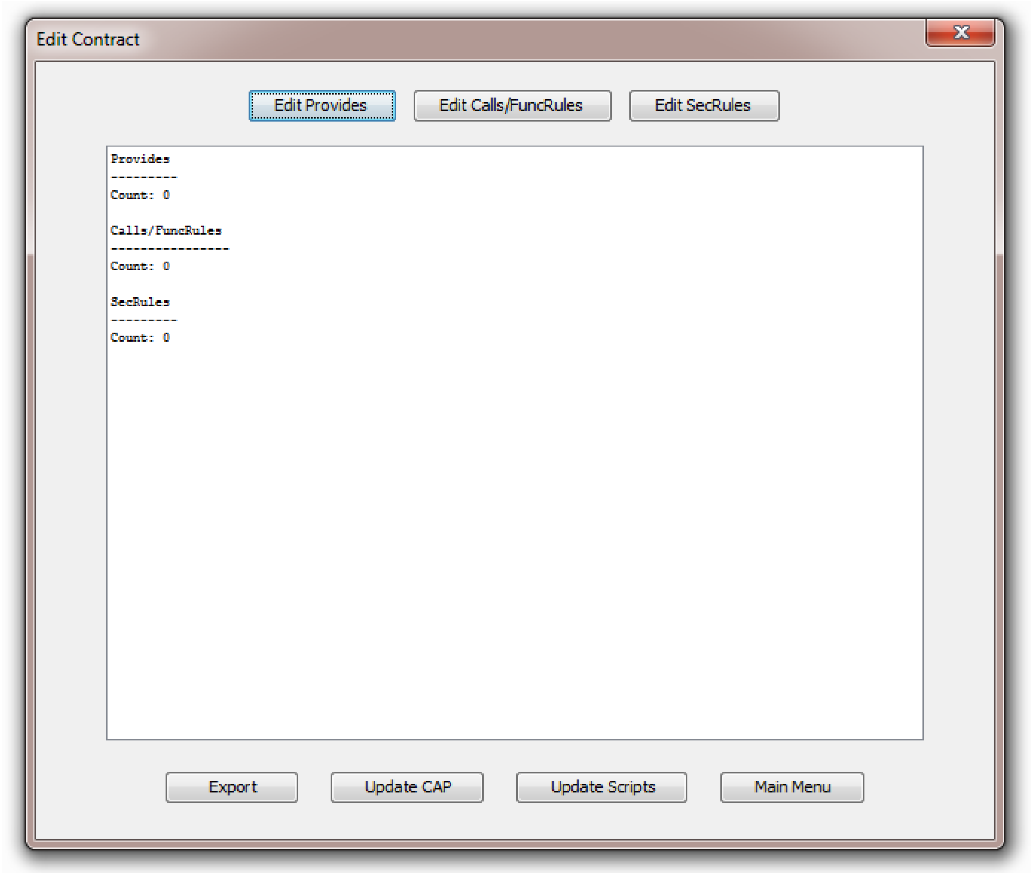
- *Authorization Rules.* An authorization rule is listed in the sec.rules set as a pair containing the service details (defined as a provided service) and the authorized client package AID. Thus the structure is the same as for a called service, with a difference that no tag for functionality is needed: $\langle AID, \mathsf{I}, \mathsf{t} \rangle$. 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 3, 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).

The CAP modifier GUI main window is presented on Figure 3, it depicts an empty contract and the options that users of the CAP modifier have. The user can choose to add services to Provides, Calls/func.rules and sec.rules sets, then the dialog will appear where the user can insert the necessary AID and tokens. When the contract is ready it can be saved for future usage. The contract can also be embedded into the chosen CAP file, and then the CAP modifier can generate the scripts necessary to communicate the CAP file to the card.

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 $\mathsf{Provides}_\mathsf{A}$ exist in package A and the services from $\mathsf{Calls}_\mathsf{A}$ 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.



```
Provides
---------
Count: 0

Calls/FuncRules
----------------
Count: 0

SecRules
---------
Count: 0
```

Figure 3: User Interface of CAP modifier

5.1 The Algorithm

Algorithm 5.1 contains the English description of the operations done with the CAP file. The ClaimChecker algorithm processes the CAP file components in order of appearance with a standard Installer. The presented algorithm 5.2-5.3 follows the English description and it 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 [17]. 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.

5.2 Soundness of the ClaimChecker Algorithm

In order to demonstrate security guarantees of the ClaimChecker algorithm 5.2-5.3 we formalize its soundness as *an honest client property* of the accepted application. Informally, an application A is a honest client if all services that can be invoked during an execution of A are listed in the set $\mathsf{Calls}_\mathsf{A}$ of $\mathsf{Contract}_\mathsf{A}$. Thus we introduce a model of application

```
Require: A CAP file.
Ensure: True/False, Contract.
 1: Header Component: get the current package AID
 2: Import Component: get package AIDs of imported packages
 3: Constant Pool Component: get imported interfaces and store them in an array
 4: Method Component: parse bytecode of the methods to identify called services
 5: Export Component: get tokens of shareable interfaces
 6: Descriptor Component: get tokens of provided services
 7: Custom Component: get Contract
 8: The Final Check: return true iff the collected sets match with the Contract
 9: if Not Match then
10:     return False
11: else
12:     return { True, current package AID, Contract}
```

Algorithm 5.1: The Claim Checker Algorithm English Description

execution and formalize an honest client property based on this model. The goal is to prove that for each application accepted by the ClaimChecker this application is an honest client.

A method m of an application A is defined by an array of its instructions. Let $\mathcal{B}_m$ be the set of opcodes of the method m. We will denote by $\mathcal{B}_m$[i] an instruction number i. Bytecode of a method m_j $\mathcal{B}_{m_j}$ has $length_{m_j}$ instructions. Let $\mathcal{B}_A = \bigcup_{j=1..k} \mathcal{B}_{m_j}$ be a bytecode of applet A.

The following definition of a call graph of a method is similar to the definition of a control flow dependencies graph (CFG) from [3], as we build on the bytecode verification techniques. We, however, emphasize that calculation of a set of services invoked by an application during its execution does not require application of the fixed point computation approach.

Definition 5.1 *A call graph CG_m of a method m is a directed graph $\langle V_m, E_m \rangle$, where $|V_m| = length_m + 1$, $\mathcal{B}_m \subset V_m$ and $E_m \subseteq V_m \times V_m$ represents the control dependencies among instructions. Digraph CG_m has one source $\mathcal{B}_m[1]$ and one sink v_{final}, where v_{final} is an auxiliary final instruction added to the call graph CG_m. There is an ark $e_{ij} \in E_m$ between the vertices v_i and v_j if and only if in $\mathcal{B}_m$ the instruction v_i can be executed immediately after the instruction v_j, or v_i is one of the* return *opcodes and v_j is v_{final}.*

We define a successor $succ()$ function for $\mathcal{B}_m$: $v_j = succ(v_i)$ if and only if exists an edge $(v_i, v_j) \in E_m$

Let instruction $v_{curr} = \mathcal{B}_m[\text{i}] = opcode(operands)$ is the instruction to be processed and the environment be $\epsilon_{before}[v_{curr}] = \epsilon$. We define the following processing rules, transforming an environment before execution of an instruction into an environment after execution of an instruction:

- Rule 1: $\frac{\epsilon_{before}[v_{curr}]=\epsilon, v_{curr}=\texttt{invokeinterface}(X,I,t)}{\epsilon_{after}[v_{curr}]:=\epsilon \cup \{\langle I,t \rangle\}}$;

```
Require: A CAP file.
Ensure: True/False, Contract.
 1: //Header Component: get the current package AID
 2: byte $CurrentPID[16]$ gets current package AID;
 3: // Import Component: get package AIDs of imported packages
 4: add ⟨imported package ID, internal imported package token (index in the current
    array)⟩ to $ImportedPackages$;
 5: // Constant Pool Component: get imported interfaces
 6: for all elements of the Constant Pool array of the type class_ref do
 7:   if the high bit equals to 1 then
 8:     add ⟨imported package token, external class or interface token, internal class or
        interface token (index in the current array)⟩ to $ImportedInterfaces$;
 9: // Method Component: parse bytecode of the methods to identify called services
10: for each method of the methods[ ] array do
11:   if invokeinterface X Y Z opcode is in the method then
12:     add ⟨internal token of the interface, external token of the method⟩ to
        $InvokedServices$;
13: // Export Component: get tokens of shareable interfaces
14: for $i = 0$ to class_count do
15:   add ⟨offset into the Class component, external interface token⟩ to
      $ExportedInterfaces$;
16: // Descriptor Component: get external tokens of provided services
17: for $i = 0$ to classes_count do
18:   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
19:     // This interface is shareable and its external token was collected
20:     for all methods of this interface do
21:       add ⟨external interface token, method token⟩ to $ListedServices$;
22: // continued on the next page
```

Algorithm 5.2: The Claim Checker Algorithm

- Rule 2: $\frac{\epsilon_{before}[v_{curr}], v_{curr} \neq \texttt{invokeinterface}(operands)}{\epsilon_{after}[v_{curr}] := \epsilon}$

We also need a rule to process transformation of the environment from the current instruction to the successor.

- Rule 3: $\frac{(v_i, v_j) \in E_m \text{ and } \nexists (v_k, v_j) \in E_m \text{ such that } v_i \neq v_k}{\epsilon_{before}[v_j] := \epsilon_{after}[v_i]}$

Specific treatment should get a point where branches merge, which transforms the environment after execution of each branch into an environment before execution of the merging instruction:

- Rule 4: $\frac{(v_{i^1}, v_j), \ldots, (v_{i^k}, v_j) \in E_m}{\epsilon_{before}[v_j] := \bigcup\limits_{l = i^1, \ldots i^k} \epsilon_{after}[v_l]}$

A service invocation trace of a method m, denoted $\mathsf{Trace}(\mathsf{CG_m})$, is an environment $\epsilon_{after}[v_{final}]$ of the sink v_{final} of the call graph CG_m, which is computed using Rules 1-4 starting from $\mathcal{B}_m[1]=\emptyset$.

```
1: // **Custom Component**: *get Contract*
2: for $j = 0$ to provides_count do
3:    add ⟨external interface token, external method token⟩ to $ContractProvides$;
4: for $j = 0$ to calls_count do
5:    add ⟨external interface token, external method token, AID⟩ to $ContractCalls$;
6:    if funcrules_tag = 0x01 then
7:       add ⟨external interface token, external method token, AID⟩ to
         $ContractFuncrules$;
8: for $j = 0$ to secrules_count do
9:    add ⟨external interface token, external method token, AID⟩ to $ContractSecrules$;
10: // **The Final Check**: *return true iff the collected sets match with the Contract*
11: *Check of called services: construct the same structure as in the contract and check for
    mutual inclusion*
12: for each $\langle I, t, AID\rangle \in ContractCalls$ do
13:    add $\langle I, t, P\rangle$ to $CALLS$ such that $\langle P, AID\rangle \in ImportedPackages$;
14: for each $\langle P, I, cpt\rangle \in ImportedInterfaces$ and $\langle cpt, t\rangle \in InvokedServices$ do
15:    add $\langle P, I, t\rangle$ to $CALLS1$;
16: if $CALLS1 \neq CALLS$ then
17:    return False;
18: else
19:    // *Check for provided services: all services in ContractProvides set have valid
       interface and method tokens*
20:    if $ContractProvides \neq ListedServices$ then
21:       return False
22:    else
23:       return { True, $CurrentPID$, $Contract$}
```

Algorithm 5.3: The Claim Checker Algorithm: part 2

The following Lemma states that extension of a call graph with one vertex and adjacent arcs can only potentially enrich a service invocation trace of a method.

Lemma 1 *If CG_{m_1} and CG_{m_2} are two call graphs such that $V_{m_1} \cup \{v_{new}\} = V_{m_2}$ and $E_{m_1} \cup \{E_{new}\} = V_{m_2}$, where $E_{new} = \{(v_k^1, v_{new}), \dots (v_k^t, v_{new}), (v_{new}, v_k^{t+1}), \dots, (v_{new}, v_k^r)\}$, then $\mathsf{Trace}(\mathsf{CG}_{\mathsf{m}_1}) \subseteq \mathsf{Trace}(\mathsf{CG}_{\mathsf{m}_2})$.*

Proof is by reasoning by cases on possibilities of an added new node and applied rules for computing $\mathsf{Trace}(\mathsf{CG}_{\mathsf{m}_2})$ □.

5.3 Execution Models

Definition 5.2 An execution of a method m *of an applet A, denoted* $\mathsf{ExecPath}(\mathcal{B}_{\mathsf{m}})$*, is a path in the call graph CG_m that starts from the node $\mathcal{B}_m[1]$ and ends at the node v_{final}.*

This definition does not consider cycles in the applet execution. While cycles often happen in real executions, our model does not treat them specifically, as they do not produce environments different from the ones produced by the same opcodes set executed once.

An execution of an applet usually starts from the standard method $\mathsf{process}$. However, it is not the only possibility, because some applet's methods can be invoked as services or remote methods, by the JCRE or during an instance construction. Thus we can assume that any method can be a starting point of an execution. An execution of an application consists of a sequence of its methods. Now we apply the usual word concatenation approach.

A simple bytecode sequence is a sequence of bytecode instructions which correspond to a path in a call graph of method m (possible execution of m $\mathsf{ExecPath}(\mathcal{B}_\mathsf{m})$) or a subpath in this graph. We can refer to each element of a simple bytecode sequence by its number, if $\mathsf{sequence}_\mathcal{B}$ is a simple bytecode sequence, then $\mathsf{sequence}_\mathcal{B}[1]$ is a first instruction in the sequence and $\mathsf{sequence}_\mathcal{B}[\mathrm{i}]$ is an instruction number i.

If $\mathsf{sequence}_\mathcal{B}{}^1$ and $\mathsf{sequence}_\mathcal{B}{}^2$ are simple bytecode sequences, we denote their concatenation as $\mathsf{sequence}_\mathcal{B}{}^1+\mathsf{sequence}_\mathcal{B}{}^2$. If $\mathsf{sequence}_\mathcal{B}{}^1$ contains k instructions and $\mathsf{sequence}_\mathcal{B}{}^2$ contains m instructions, then $\mathsf{sequence}_\mathcal{B}= \mathsf{sequence}_\mathcal{B}{}^1 + \mathsf{sequence}_\mathcal{B}{}^2$ contains $k+m$ instructions, and for $i = 1,\ldots,k$ $\mathsf{sequence}_\mathcal{B}[i] = \mathsf{sequence}_\mathcal{B}{}^1[i]$, for $i = k+1,\ldots,m$ $\mathsf{sequence}_\mathcal{B}[i] = \mathsf{sequence}_\mathcal{B}{}^2[i-k]$. A concatenation of multiple simple sequences is defined analogously.

A bytecode sequence is a simple bytecode sequence or a concatenation of simple bytecode sequences: $\mathsf{sequence}_\mathcal{B}= \mathsf{sequence}_\mathcal{B}{}^1 + \ldots + \mathsf{sequence}_\mathcal{B}{}^n$, where $n \geq 1$ and $\mathsf{sequence}_\mathcal{B}{}^i$ is a simple bytecode sequence. A bytecode sequence model corresponds to a sequence of method executions. When a method m_1 is being executed, another method m_2 can be invoked. Thus part of a call graph path of m_1 will be traversed, then a path in the invoked method m_2 call graph will be traversed, and then the execution of m_1 will be resumed. This execution can be represented as a concatenation of three simple bytecode sequences: $\mathsf{sequence}_\mathcal{B}{}^{\mathsf{m_1}^1} + \mathsf{sequence}_\mathcal{B}{}^{\mathsf{m_2}} + \mathsf{sequence}_\mathcal{B}{}^{\mathsf{m_1}^2}$, where $\mathsf{sequence}_\mathcal{B}{}^{\mathsf{m_1}^1} + \mathsf{sequence}_\mathcal{B}{}^{\mathsf{m_1}^2}$ is a simple sequence corresponding to a path in CG_{m_1}.

We now define a modified version of Rule 3 that will be used for operating with bytecode sequences.

- Rule 3': $\frac{\epsilon_{after}[\mathsf{sequence}_\mathcal{B}[\mathsf{j}]]=\epsilon}{\epsilon_{before}[\mathsf{sequence}_\mathcal{B}[\mathsf{j}+1]]:=\epsilon}, j > 0$

A service invocation trace of a simple bytecode sequence $\mathsf{sequence}_\mathcal{B}$ is a set $\mathsf{Trace}(\mathsf{sequence}_\mathcal{B}) = \epsilon_{after}[\mathsf{sequence}_\mathcal{B}[\mathsf{k}]]$ where $\mathsf{sequence}_\mathcal{B}[\mathrm{k}]$ is the last element of $\mathsf{sequence}_\mathcal{B}$ and $\epsilon_{before}[\mathsf{sequence}_\mathcal{B}[1]] = \{\emptyset\}$. Informally, a service invocation trace of a simple bytecode sequence $\mathsf{sequence}_\mathcal{B}$ is a set of service invocations that appear in the $\mathsf{sequence}_\mathcal{B}$.

Only Rules 1, 2 and 3' are used to obtain a service invocation trace, because a bytecode sequence is a path (or subpath) in a call graph, hence merging of branches is not required.

A service invocation trace of a bytecode sequence $\mathsf{sequence}_\mathcal{B}$ is a union of all simple bytecode sequences $\mathsf{sequence}_\mathcal{B}$ consists of. If $\mathsf{sequence}_\mathcal{B}= \mathsf{sequence}_\mathcal{B}{}^1 + \mathsf{sequence}_\mathcal{B}{}^2 + \mathsf{sequence}_\mathcal{B}{}^\mathsf{k}$, then $\mathsf{Trace}(\mathsf{sequence}_\mathcal{B}) = \bigcup\limits_{i=1..k} \mathsf{Trace}(\mathsf{sequence}_\mathcal{B}{}^\mathsf{i})$.

We say that two bytecode sequences $\mathsf{sequence}_\mathcal{B}$ and $\mathsf{sequence}_\mathcal{B}{}^1$ are *equivalent with respect to service invocations* if their service invocations traces are equal: $\mathsf{Trace}(\mathsf{sequence}_\mathcal{B}) = \mathsf{Trace}(\mathsf{sequence}_\mathcal{B}{}^1)$.

An execution of an applet consists of several methods executions. Some of them are executed sequentially and some are nested. It is generally impossible to follow a call graph of an application in order to capture all possible scenarios of executions. For example, if an application A is being analyzed and it can invoke another application B during execution of a method $A.m$. B, in turn, can invoke one of A's services (or a transitive call through more applets can be performed), resulting in an invocation of a method $A.s$ indirectly from $A.m$. Without the code of B it is not possible to infer which methods of A can be called from any method $A.m$ that has invocations of some services of B in it. However, obviously all possible executions of an applet A can be represented as a general sequence of A's methods invocations. The following lemma provides a basis for reasoning about application execution model.

Lemma 2 (Simple bytecode sequences transpositions equivalence) *If a bytecode sequence* $\mathsf{sequence}_{\mathcal{B}}{}^{A} = \mathsf{sequence}_{\mathcal{B}}{}^{\mathsf{m_1}^1} + \mathsf{sequence}_{\mathcal{B}}{}^{\mathsf{m_2}} + \mathsf{sequence}_{\mathcal{B}}{}^{\mathsf{m_1}^2}$ *and a bytecode sequence* $\mathsf{sequence}_{\mathcal{B}}{}^{B} = \mathsf{sequence}_{\mathcal{B}}{}^{\mathsf{m_1}} + \mathsf{sequence}_{\mathcal{B}}{}^{\mathsf{m_2}}$ *such that* $\mathsf{sequence}_{\mathcal{B}}{}^{\mathsf{m_1}^1} + \mathsf{sequence}_{\mathcal{B}}{}^{\mathsf{m_1}^2} = \mathsf{sequence}_{\mathcal{B}}{}^{\mathsf{m_1}}$*, where* $\mathsf{sequence}_{\mathcal{B}}{}^{\mathsf{m_1}^1}, \mathsf{sequence}_{\mathcal{B}}{}^{\mathsf{m_1}^2}$,
$\mathsf{sequence}_{\mathcal{B}}{}^{\mathsf{m_2}}$ *and* $\mathsf{sequence}_{\mathcal{B}}{}^{\mathsf{m_1}}$ *are simple bytecode sequences, then* $\mathsf{Trace}(\mathsf{sequence}_{\mathcal{B}}{}^{\mathsf{A}}) = \mathsf{Trace}(\mathsf{sequence}_{\mathcal{B}}{}^{\mathsf{B}})$

Proof follows from definitions of a bytecode sequence and a service invocation trace. □

Lemma 2 allows to consider as a model of applet execution (related to services invocations) sequential composition of several methods rather than nested composition.

An execution of an application A is a bytecode sequence $\mathsf{sequence}_{\mathcal{B}}(\mathrm{A}) = \mathsf{sequence}_{\mathcal{B}}{}^{\mathsf{m_1}} + \cdots + \mathsf{sequence}_{\mathcal{B}}{}^{\mathsf{m_n}}$, where $\mathsf{sequence}_{\mathcal{B}}{}^{\mathsf{m_i}}$ is an execution $\mathsf{ExecPath}(\mathcal{B}_{\mathsf{m_i}})$ of a method $A.m_i$.

We now define *a property of an honest client* which represents formally guarantees that the $\mathsf{ClaimChecker}$ Algorithm 5.2-5.3 provides.

Definition 5.3 *An application A satisfies an honest client property if for any execution of A* $\mathsf{sequence}_{\mathcal{B}}(A)$ *a service invocation trace* $\mathsf{Trace}(\mathsf{sequence}_{\mathcal{B}}(\mathsf{A})) \subseteq \mathsf{Calls}_{\mathsf{A}}$.

Theorem 5.1 *If the* $\mathsf{ClaimChecker}$ *Algorithm 5.2-5.3 returned True for an applet A, then A satisfies an honest client property.*

As the algorithm 5.2-5.3 returned True on the CAP file of A, the collected invoked services set equals $\mathsf{Calls}_{\mathsf{A}}$. We now prove that for any execution of A $\mathsf{sequence}_{\mathcal{B}}(\mathrm{A})$ its service invocation trace $\mathsf{Trace}(\mathsf{sequence}_{\mathcal{B}}(\mathsf{A})) \subseteq \mathsf{Calls}_{\mathsf{A}}$.

Let $\mathsf{sequence}_{\mathcal{B}}(\mathrm{A}) = \mathsf{sequence}_{\mathcal{B}}{}^{\mathsf{m_1}} + \cdots + \mathsf{sequence}_{\mathcal{B}}{}^{\mathsf{m_n}}$ is an execution of A. Thus we have to prove that for any m_i $\mathsf{Trace}(\mathsf{ExecPath}(\mathcal{B}_{\mathsf{m_i}})) \subseteq \mathsf{Calls}_{\mathsf{A}}$. Applying Lemma 1 it follows that $\mathsf{Trace}(\mathsf{ExecPath}(\mathcal{B}_{\mathsf{m_i}})) \subseteq \mathsf{Trace}(\mathsf{CG}_{\mathsf{m_i}})$.

For any element (a service) $\langle I, t\rangle \in \mathsf{Trace}(\mathsf{CG}_{\mathsf{m_i}})$ it can appear in the environment after execution of an instruction if and only if the instruction is `invokeinterface` *nargs I t*. Thus $\langle I, t\rangle \in \mathsf{Calls}_{\mathsf{A}}$ (in an appropriate format, because of the specification of the $\mathsf{ClaimChecker}$ algorithm 5.2-5.3). It follows that A satisfies an honest client property. □

6 Implementation of the Claim Checker

We have implemented full S×C prototype in C (except the Policy applet, that is in Java Card), 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.

An overall schema of the prototype (C part), in the shape of a component diagram, can be observed in Figure 4. The main components 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 ClaimCheckeror 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.*

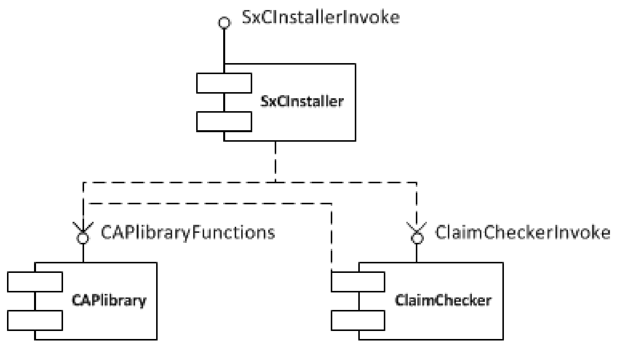


Figure 4: The Component Diagram

We now discuss the implementation of the proposed algorithm 5.2-5.3 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 are standard part of the Installer functionality. As we did not have access to an actual smart card platform implementations, we have implemented these functions for testing purposes, but we do not include this implementation in the following memory statistics of the prototype. Table 5 contains the list of the CAPlibrary functions, the purpose of each function is deducible from its name. We note that the length of each component can be also inferred from the CAP file itself, but we have introduced the length functions as they are very handy for the prototype. The actual implementation of the length functions for testing purposes infers the size of each component from the CAP file. The implementation of the pointer functions (the ones that return the reference to the beginning of the components) for testing purposes operates on the CAP file as a byte array.

```
getHeaderComponentRef();
getHeaderComponentLength();
getImportComponentRef();
getImportComponentLength();
getConstantPoolComponentRef();
getConstantPoolComponentLength();
getMethodComponentRef();
getMethodComponentLength();
getExportComponentRef();
getExportComponentLength();
getDescriptorComponentRef();
getDescriptorComponentLength();
getContractComponentRef(); //function to retrieve the Contract Custom component
getContractComponentLength();
```

Table 5: The functions of the CAPlibrary

6.1 The PolicyChecker and the Policy Applet Implementation

We do not report about the details of the PolicyChecker implementation in the current paper. 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 all the contracts of the currently loaded applications. A contract in the form supplied on the card (see Section 4 for details) 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 our on-card security policy 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. 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, the entries in the MayCall list are created 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 AID), thus they are space-consuming objects. The WishList set 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 AID of the service provider and the service as a tuple $\langle I, t\rangle$. 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 so 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 ClaimChecker Implementation Memory statistics

In this section we present the overview of the memory consumption by the prototype. We first review each of the suggested metrics and provide the data for the prototype, and then we provide an aggregated Table 6 with all the measurements.

The most important characteristics for an on-card component are RAM and EEPROM consumption [16]. 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 [20]. The chip is entitled for 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 real card and measure its footprint in the linked state, we have proposed two metrics for the EEPROM/ROM usage measurements: 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 lines of code, and the .h file contains 7 lines.

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.

Feature	Metrics	Quantity
EEPROM consumption	LOCs (.c + .h)	155 + 7
EEPROM consumption	Object files size (bytes)	6.36 KB
RAM usage	Size of temporary buffer (bytes)	255 bytes

Table 6: Aggregated Details for the ClaimChecker Prototype

7 Related Work

The desire for on-card Java Card bytecode verification emerged together with appearance of the Java Card technology itself, because the off-card bytecode verification creates a single point of failure for the Java language safety. Leroy was one of the pioneers that investigated the on-board verifiers [15]. He proposed an optimized algorithm for the bytecode verification, that improved the lacks of the Sun's bytecode verification algorithm by reducing the dictionary size, thus allowing the dictionary to be stored in the EEPROM memory. Deville and Grimaud continued this work by enforcing the card to use the RAM memory to store the stack maps thus reducing the usage of the EEPROM, and proposing an efficient types encoding [7]. Casset et al [5] investigated a bytecode verifier for Java Card with a goal to ensure formally its compliance with the specifications. The authors explored the B method and the proof-carrying-code techniques, that allowed them to demonstrate feasibility of the embedded bytecode verifier. Bernardeschi et al [2] proposed an efficient approach for the Java Card byte code verification that performed verification for each type separately, reducing the size of the dictionary. The authors built a formal semantics of the Java Card bytecode, thus demonstrating security guarantees of the bytecode verifier.

Rose explored an approach [18] based on the proof-carrying-code technique. She proposed to split the bytecode verification process into an off-card part, when an auxiliary verification information is constructed in a form of a certificate to be sent together with a CAP file, and an on-card part, where the verification is performed as a check of the code against the certificate. The approach, called lightweight verification, has the advantage of running in an almost constant space and almost linear time. The technique was proven to be sound and complete with respect to the standard bytecode verification.

Though the on-card bytecode verification approaches exist, we did not find in the literature an explicit algorithm that processes the CAP files. The unique structure of the CAP files required us to investigate thoroughly the specifications and to operate carefully with the memory contents. Our current work is close to the one by Bernardeschi et al, because we defined an application execution model that is based on the call graphs, but our focus was limited to the scope of application interactions.

A plethora of works exist for verification of application interactions security on Java Card. Ghindici et al [12] 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 [14], [13], [4], [19]. Dynamic scenarios were considered in [1] and [10]. 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.

The work of Fontaine et al [10] is the closest to our paper, as the authors considered the same dynamic scenario and on-card loading time verification approach. With respect to [10] our work enforces less stronger policies, as we do not consider transitive calls and application collusions. However, the S×C approach offers greater flexibility than the transitive control flow policies 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, has to store a significant amount of additional data related to the invoked methods, what might be a prohibitive feature for the on-card implementation. Thus it is still questionable if that approach can be implemented on a smart card at all.

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 [17], and we had developed the CAP modifier tool for contract generation and addition to the CAP files.

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 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] C. Bernardeschi, G. Lettieri, L. Martini, and P. Masci. A space-aware bytecode verifier for Java Cards. *Electronic Notes in Theoretical Computer Science*, 141(1):237 – 254, 2005.

[3] C. Bernardeschi, G. Lettieri, L. Martini, and P. Masci. Using control dependencies for space-aware bytecode verification. *Comput. J.*, 49:234–248, March 2006.

[4] 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.

[5] L. Casset, L. Burdy, and A. Requet. Formal development of an embedded verifier for Java Card byte code. In *In the IEEE International Conference on Dependable Systems & Networks*, pages 51–56, 2002.

[6] Z. Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide.* Addison-Wesley Longman Publishing Co., Inc., 2000.

[7] D. Deville and G. Grimaud. Building an "impossible" verifier on a Java Card. In *Proceedings of the 2nd conference on Industrial Experiences with Systems Software - Volume 2*, pages 2–2, Berkeley, CA, USA, 2002. USENIX Association.

[8] 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.*

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] P. Girard. Which security policy for multiplication smart cards? In *USENIX Workshop on Smartcard Technology.* USENIX Association, 1999.

[14] 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.

[15] X. Leroy. On-card bytecode verification for Java Card. In *Smart Card Programming and Security*, volume 2140 of *LNCS*, pages 150–164. SV, 2001.

[16] K. Mayes and K. Markantonakis (Eds.). *Smart Cards, Tokens, Security and Applications*. SV, 2008.

[17] Sun Microsystems. Virtual Machine and Runtime Environment. Java Card^{TM} platform. Specification 2.2.2, Sun Microsystems, 2006.

[18] E. Rose. Lightweight bytecode verification. *J. Autom. Reason.*, 31:303–334, 2004.

[19] 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.

[20] Philips Semiconductors. P5CT072 Secure Dual Interface PKI Smart Card Controller. On the web at http://www.usmartcards.com/images/pdfs/pdf-199.pdf.