SEVENTH FRAMEWORK PROGRAMME

Advanced Software Protection:
Integration, Research and Exploitation

D2.10

Anti-Tampering Report

Project no.: 609734
Funding scheme: Collaborative project
Start date of the project: 1st November 2013
Duration: 36 months
Work programme topic: FP7-ICT-2013-10

Deliverable type: Report
Deliverable reference number: ICT-609734 / S2.10/ 1.0
WP and tasks contributing: WP 2 / Task T2.5
Due date: Apr 2016 - M30
Actual submission date: 27 May 2016

Responsible Organization: UGent
Editor: Bart Coppens
Dissemination Level: Public
Revision: 1.0

Abstract:

This Deliverable reports on the final progress and integration of the various static anti-tampering techniques developed in WP2. Furthermore, it reports on the progress of the Diversified Cryptography, which is the replacement for some of the white-box cryptography research foreseen in the original project DoW.

Keywords:
diversified cryptography, anti-debugging, offline code guards, anti-callback checks, control flow tagging

Editor

Bart Coppens (UGent)

Contributors (ordered according to beneficiary numbers)

Bart Coppens (UGent)

Bjorn De Sutter (UGent)

Bert Abrath (UGent)

Fadela Letourneur (GTO)

Jerome d'Annoville (GTO)

The ASPIRE Consortium consists of:

Ghent University (UGent)	Coordinator & Beneficiary	Belgium
Politecnico Di Torino (POLITO)	Beneficiary	Italy
Nagravision SA (NAGRA)	Beneficiary	Switzerland
Fondazione Bruno Kessler (FBK)	Beneficiary	Italy
University of East London (UEL)	Beneficiary	UK
SFNT Germany GmbH (SFNT)	Beneficiary	Germany
Gemalto SA (GTO)	Beneficiary	France

Coordinating person: Prof. Bjorn De Sutter
E-mail: coordinator@aspire-fp7.eu
Tel: +32 9 264 3367
Fax: +32 9 264 3594
Project website: www.aspire-fp7.eu

Disclaimer
The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 609734.
The information in this document is provided as is and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

Executive Summary

This deliverable reports the final progress made Task 2.5 of WP2 Offline Software Protections, whose goal is to improve the tamper-resistance of the protected binary with offline techniques. The design and implementation of the Anti-Debugging technique, which was integrated in the ASPIRE Compiler Tool Chain (ACTC) was modified to account for some requirements from the Use Cases in which it was used. The Offline Code Guards technique was implemented and integrated in the ACTC according to the design of Deliverable D2.08. Anti-Callback checks have been integrated in the ACTC as well. No significant updates were done to the Control Flow Tagging protection. Finally, the design of the Diversified Cryptography is presented in this document.

Contents

Section 1 Introduction **1**
Section 2 Anti-Debugging **2**
2.1 Switching the roles of forked and forking process 2
2.2 Improved support for memory accesses 3
2.3 Improved support for multithreading 3
Section 3 Offline Code Guards **4**
3.1 Attesting and verifying code regions 4
3.2 Offline tamper responses 5
Section 4 Anti-Callback Checks **7**
Section 5 Control Flow Tagging **8**
Section 6 Diversified Cryptography **9**
6.1 Motivation 9
6.2 Requirements 9
6.3 Architecture 9
6.3.1 DCL Bootstrap Module 10
6.3.2 Bridge component 10
6.3.3 Crypto component 11
6.3.4 Credential storage component 11
6.3.5 Glue code 12
Section 7 Conclusions **13**
Section 8 List of Abbreviations **14**

Section 1 Introduction

Section Authors:

Bart Coppens (UGent)

This deliverable reports the final progress made Task 2.5 of WP2 Offline Software Protections. The goal of WP2 is to provide the offline software protections for the ASPIRE project. The specific goal of Task 2.5 is to improve the tamper-resistance of the resulting protected binary with offline techniques. For each of these offline tamper-resistance techniques, the final progress is reported in this deliverable. Furthermore, this report also discussed progress from protection techniques that were not strictly contained in Task T2.5, such as multi-threaded cryptography, which belong to other tasks in WP2. This is because there have been further developments on them in their respective tasks, and because this deliverable is the last deliverable for WP2 in which we can report on these activities.

Some of the updates described in this deliverable have already been used in the experiments with the industrial tiger teams (Task T4.4). In particular, the anti-debugging component, offline code guards, and anti-callback checks that are described here, are all used in both NAGRA's as in SFNT's tiger team experiments, which means that their effectiveness will be evaluated. Furthermore, as some of the updates described here needed to be integrated in the ASPIRE Compiler Tool Chain (ACTC), there have also been interactions with Task 5.1 of WP5 in which the ACTC is developed and maintained.

The remainder of this document is structured as follows. Section 2 describes the final version of the anti-debugging component. The implementation of the offline code guards is described in Section 3. In Section 4, we describe the final integration of the anti-callback checks. Section 5 describes how no significant updates were made to the Control Flow Tagging technique. Finally, we report on ASPIRE's cryptography-related protections other than white-box cryptography in Section 6 with diversified cryptography, as foreseen in the updated DoW as a replacement for some of the white-box cryptography research foreseen in the original project DoW.

Finally, this deliverable D2.10 of type report also documents the tool support that has been developed up until M30 for the project in WP2, and that is delivered as the confidential prototype Deliverable D2.09.

Section 2 Anti-Debugging

Section Authors:

Bart Coppens (UGent), Bert Abrath (UGent)

In M24 we delivered the previous version our anti-debugging component, which we described in detail in D2.08, Section 6.4. Since then, we made several changes and improvements to the design and implementation of the anti-debugging protection: switching of the roles of the forked and forking process, better support for emulating memory accesses, and better support for multi-threaded processes. All of these changes and improvements were guided by the integration testing and debugging of the protection techniques on the NAGRA Use Case. The anti-debugging component has been integrated in the ACTC as part of the Diablo binary rewriter in BLP04. This integration is described in more detail in Section 4.3 of Deliverable D5.08. The version as described in this section was integrated in the ACTC in M29.

2.1 Switching the roles of forked and forking process

First and foremost, we modified the relationship between the protected process and the mini-debugger. Previously, the protected process would fork, with the parent becoming the mini-debugger that protects the child. This is shown in Figure 1, which first shows the parent process calling `fork()`. The forked child process then blocks the execution until a debugger attaches using the PTRACE_TRACEME ptrace-call. In our case this is the parent process that just called fork. However, this scheme has the slight disadvantage that executing such a protected program under a debugger with its default options, will result in the anti-debugger succeeding in attaching to the child process. This is because the debugger will only be attached to the parent process, which serves as a debugger. While this still prevents an attacker from attaching to the protected child process, the fact that attackers will initially be able to run the protected program under a debugger despite anti-debugging measures is somewhat suboptimal. Furthermore, the external environment can try to communicate with the process using its process id; sending signals to this process. These signals now arrive in the debugger, who would then have to forward these signals to the debuggee.

Furthermore, this scheme requires the usage of the PTRACE_TRACEME ptrace-call. This syscall has had additional security constraints imposed in more recent versions of Linux and Android. These constraints would disallow custom applications to execute this call at all, breaking any anti-debugging component implemented in this way. Of course, the PTRACE_TRACEME system call is used for a reason: it allows the child of a debugger to cleanly signal to its parent debugger process that it is ready to be debugged.

To solve these issues, we reversed the role of parent and child process: this is shown in Figure 2. Now the parent process forks off the mini-debugger as a child. Of course, now we can no longer use the PTRACE_TRACEME call. The debugger calls PTRACE_ATTACH on the child, but in without care this child could already be executing arbitrary protected code. Our solution is to have the parent/debuggee process wait in a loop after forking. The loop condition depends on the value of a global variable; the location of which is known by the child/debugger process. Once the debugger is initialized, it attaches to the parent process, and overwrites terminates the loop in the parent process by overwriting this global variable.

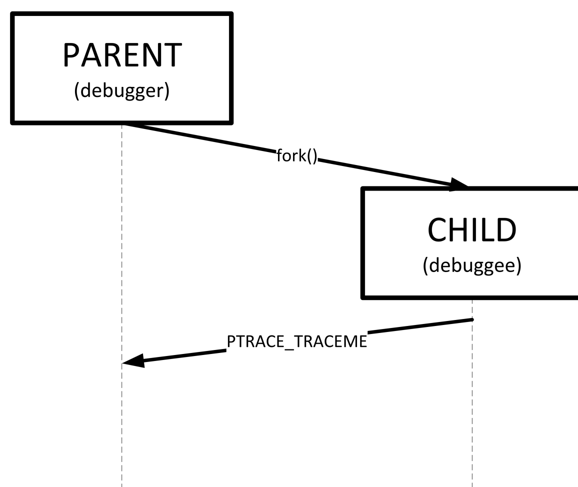

Figure 1 Old scheme: parent debugs child

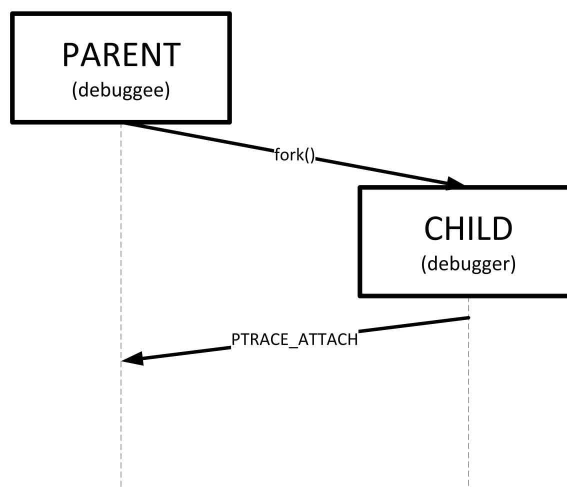

Figure 2 New scheme: child debugs parent

2.2 Improved support for memory accesses

We had to improve the support for handling memory accesses as the code written for this by our master thesis student did not suffice for our purposes. The mini-debugger used to do all read/write memory accesses to the protected process using `/proc/<pid>/mem` interface. However, write access is not supported on all kernels through this interface. In particular, Linux kernels such as those that are run Android 4.3 do not support this.

To solve this issue, these accesses now happen using the ptrace interface. More specifically, we now use PTRACE_POKEDATA). Read accesses however still go through the `/proc/<pid>/mem` interface.

2.3 Improved support for multithreading

As both of the libraries that are protected for the NAGRA use case are loaded into multithreaded programs, this use case provided a good stress-test for the multi-threading support available in our mini-debugger. However, we noticed that we required much more robust support for multithreading than was provided in the master's thesis student's implementation. In effect, we had to write all threading-related code related code from the ground up to provide robust multi-threading support for the anti-debugging technique.

In particular, we now support the following correctly:

- The mini-debugger now attaches to all threads of the protected process after the fork, and now correctly attaches to any threads created after attaching the debugger, and also detaches from threads that terminate.
- Furthermore, we now also support that the protected module is unloaded from the process even though the process itself is not terminating. This can happen because in the NAGRA use case, the protected library is not only loaded, but is sometimes unloaded from the application (using the `dlclose` function call). The mini-debugger now detaches from all running threads, shuts itself down, and lets the process continue on its own (without the protected module). The case where the library is then re-loaded in the application is also handled well now.
- Finally, we had to fix several problems involving subtle race conditions that occur due to the combination of multi-threading and signal handling.

Section 3 Offline Code Guards

Section Authors:

Bert Abrath (UGent), Bart Coppens (UGent)

For this Deliverable, we took the design from M24, which is described in detail in Section 6.4 of Deliverable D2.08, and implemented it. However, while we previously wrote in Deliverable D2.08 that we would re-use significant portions of the Remote Attestation code base to implement offline code guards, we were not able to re-use as much code as indicated. In particular, we re-use both the code that generates and parses the Area Data Structure (ADS) from Remote Attestation (RA). However, we did not re-use the functionality to attest and verify code regions. The code we implemented to attest and verify code regions for code guards is significantly less complex than the code for RA.

The anti-debugging component has been integrated in the ACTC as part of the Diablo binary rewriter in BLP04, and as a separate source-rewriting tool SLP08. Their integration in the ACTC is described in more detail in Sections 3.2 and 4.4 of Deliverable D5.08. The initial integration of the Offline Code Guards in the ACTC was performed in M28; however significant fixes were still made after this integration. The implementation as described in this section was shared with all partners in M29.

Our work on implementing code guards is split in two parts: the code required for performing and verifying the local attestations, and the code required for performing local reactions based on the results of these local attestations.

3.1 Attesting and verifying code regions

The flow to rewrite the binary to insert the code guards is as follows:

1. Invocations of the code guard attestation and verification functionality itself are inserted through a small source code rewriting tool. The current implementation is limited to one verifier per attestator; however, this is not a fundamental limitation.
2. We have separate source code that computes hashes over a region, which is compiled and linked with the protected program by the ACTC. As the interface that was provided by the hashing algorithms written for RA proved to be too cumbersome to integrate in code guards, we wrote these ourselves.
   For each of the attestators, we inject a separate instance of this source code file, each with unique names for both the function calls as well as the global variables used. These variables will contain a flag indicating if tampering was detected, the correct hash values, etc.
3. Our binary rewriter tool Diablo extracts the annotations that describe which code regions need to be attested in the same way as we do for RA, re-using that code.
4. Throughout the program, Diablo inserts calls to the tamper response mechanisms, the design and implementation of which is explained in more detail in the next section. These calls depend on the variable that stores whether or not tampering was detected.
5. After the final layout of the binary has been determined, Diablo computes the correct hash values and inserts these in the corresponding variables.

While our current implementation only supports a single non-randomized correct hash value per attestation region, our design is flexible enough so that we can easily extend the implementation with nonces. Rather than storing a single value per attestator, we store an array of nonces to be used in each attestation, and also store the corresponding hash

values. The code to compute and verify the hashes can then iterate over consecutive values of nonces.

3.2 Offline tamper responses

As the tamper responses or reaction mechanisms for call stack checks had not been implemented yet, we could not reuse these. Therefore a small framework and a number of reaction mechanisms were designed during the implementation of code guards. The verifiers are also more complex as described in Deliverable D2.08. Instead of immediately triggering a reaction mechanism when tampering is detected, we just set a flag, which is checked by the reaction mechanism invocations that have been inserted throughout the protected program by Diablo. This ensures that the reaction to a tampering will be triggered separately from detecting the tampering. This way, it will be harder for a potential attacker to connect his tampering and the reaction to it.

The reaction mechanisms are implemented in C, in separate reaction mechanisms source files. Each of these files contains a number of reaction mechanisms, shared data structures, and a degradation function that - once called - causes the reaction mechanisms to trigger later on in the future. Figure 3 shows an example of such a mechanisms file that uses mutexes in its reaction mechanism. It defines an initialization function that initializes 2 mutexes, and lets both variable x and variable y point to the same mutex.

Each mechanisms file defines one or multiple reaction mechanisms that can be called safely as long as tampering was not detected. With this in mind, Diablo inserts invocations to the reaction mechanisms all throughout the program, on suitable locations. In the case of Figure 3, the reaction function locks mutex pointer x, does a computation, and then unlocks mutex pointer y. As both pointers were initialized to point to the same mutex, this function can be called safely.

Once tampering is detected, the degradation function is invoked, which will cause subsequent calls to the reaction functions to fail. In our example, it lets mutex pointer x point to the second mutex; which will cause a deadlock.

```
pthread_mutex_t mutex_1, mutex_2, *x, *y;
INIT_REACTION() {
  pthread_mutex_init(&mutex_1, NULL);
  pthread_mutex_init(&mutex_2, NULL);
  x = y = &mutex_1;
}
START_DEGRADATION(original) {
  x = &mutex_2;
}
void DIABLO_REACTION_G(int x)
{
  pthread_mutex_lock(x);
   …
  pthread_mutex_unlock(y);
}
```

Figure 3 Example reaction mechanisms file

Reaction mechanisms will thus be invoked continuously during the execution of the program, but only after the verifier detects tampering and sets in motion the degradation will the mechanisms corrupt the program during their invocation.

We also support more low-level reaction mechanisms, where the degradation function can corrupt register values that are used by the program.

Section 4 Anti-Callback Checks

Section: Authors:

Bart Coppens (UGent)

Already for Deliverable D2.08 in M24 we started the implementation of lightweight anti-callback checks. These are small stack checks that verify that internal functions are not invoked from outside the library. These are injected by the binary rewriter component BLP04 when protecting dynamically linked libraries. These checks inspect the return address of the last call and check whether or not it comes from inside the code segment of the protected library itself, or from the outside.

Since M24, we have debugged the implementation of the anti-callback checks that was described in D2.08, and have integrated this protection technique into the ACTC by M30 as part of the Diablo binary rewriter tool in BLP04. The integration itself is discussed in Section 4.1 of Deliverable D5.08.

One of the issues we had to address was the incompatibility of anti-callback checks and some of the other protection techniques. In particular, we had to resolve an incompatibility with the code mobility protection technique. In this technique, executable code is downloaded from a remote server; a new page is allocated to store this code in, and finally this code is executed. However, this allocated page is not located in the code segment of the protected library, but is located in the heap. Thus, mobile code calling into internal functions protected with anti-callback checks would trip these checks.

It would be possible to extend the anti-callback checks and integrate them with the code mobility's binder component to allow calls from mobile blocks as well. However, this would significantly increase the overhead of these checks, which were explicitly designed to be very low-overhead checks. In particular, the authors of the ASPIRE Use Case annotations used the anti-callback check annotations extensively as they are supposed to be very low-overhead.

Thus, we did not opt for adding explicit support for mobile code to the anti-callback checks. Rather, we chose to disable the anti-callback checks on regions with mobile code annotations. The reason that we disable the anti-callback checks rather than disable the mobile code on regions with a conflict is similar to the reason above for not integrating the mobile code's binder with anti-callback checks: users of the ASPIRE annotations extensively added anti-callback check annotations everywhere. In the NAGRA use case in particular, anti-callback checks were added to a generic protection profile, meaning that all functions were automatically annotated with this protection without the annotator having to give more thought to this. This is in stark contrast with the other annotations, such as code mobility: these annotations were added explicitly on very specific code regions, most likely because they contain assets. Thus, we decided that the more specific, heavy-weight code mobility annotation should override the generic anti-callback checks.

However, we do let the rewriter produce explicit warnings when it disables the anti-callback checks, so that users are aware of it and can verify this if necessary.

Section 5 Control Flow Tagging

Section Authors:

Jerome D'Annoville (GTO)

No significant work done on this protection for the considered period. Most of the effort has been spent on the Reaction mechanism described in the Deliverable D3.06 – Remote Attestation and Server Mobile Code Report. The Reaction mechanism was on the critical path since it is required to enable an effective protection done by the Remote Attestation provided by POLITO.

A first iteration of Control Flow Tagging (CFT) protection will be implemented before the GTO Tiger team experiment in M32. If time is too short then another iteration will be released in M33 and will be reported in another deliverable.

Section 6 Diversified Cryptography

Section Authors: Fadela Letourneur, Jerome d'Annoville (GTO)

6.1 Motivation

A Gemalto internal research project is currently conducted to provide a software component in case a mobile device does not have a hardware secure environment. In that case, this software component should be provided to ensure the continuation of service even if less secure than the hardware support. The current output of this project is a shared library that can be integrated in an Android application. This library will enable the distribution of the application on devices independently of the device characteristics at download time.

As a replacement of a theoretical research task on cryptography in the project it appeared that the library implemented in the aforementioned internal project could be customized with a reasonable development effort to provide a Cryptographic Library to any Android mobile application. Then we proposed a new protection in the project which we call Diversified Crypto Library (DCL) that enables to protect credentials of an application with a library customized at compile time.

6.2 Requirements

The DCL shall be able to hide a key that is embedded at compile time in the library and then distributed with the application.

A derivation function shall be present in the library in addition to encrypting and decrypting functions. This extra derivation function enables getting a derived key within the library and avoid to expose the master key to the application in the clear.

Encryption or decryption cryptographic functions shall be available with a key that is kept within the library. It must be implemented in such way that an attacker cannot easily retrieve the key value.

6.3 Architecture

Figure 4 shows all components involved in the protection. The protection is provided by the DCL library. The DCL library can be called either by the Native part of the application or by the DCL Bootstrap Module. The DCL library contains the Bridge component, the Bootstrap component, the Crypto component and the Credential storage component. The Bridge component exposes the interface of the Crypto component to the application. Cryptographic functions are implemented in the Crypto component and the Credential storage component provides internal storage as suggested by its name.

What has been done for ASPIRE is to implement the Bridge component in the DCL library, to provide a new DCL Bootstrap Module and a clean design of the glue code that is put in the native part of the application that calls the DCL library. This glue code is inserted in the application by the ACTC. The DCL Bootstrap Module has been updated to enable offline initialization of the DCL library.

Inside the DCL library, the Bootstrap component, the Crypto component and the Credential storage component have been integrated with minor updates.

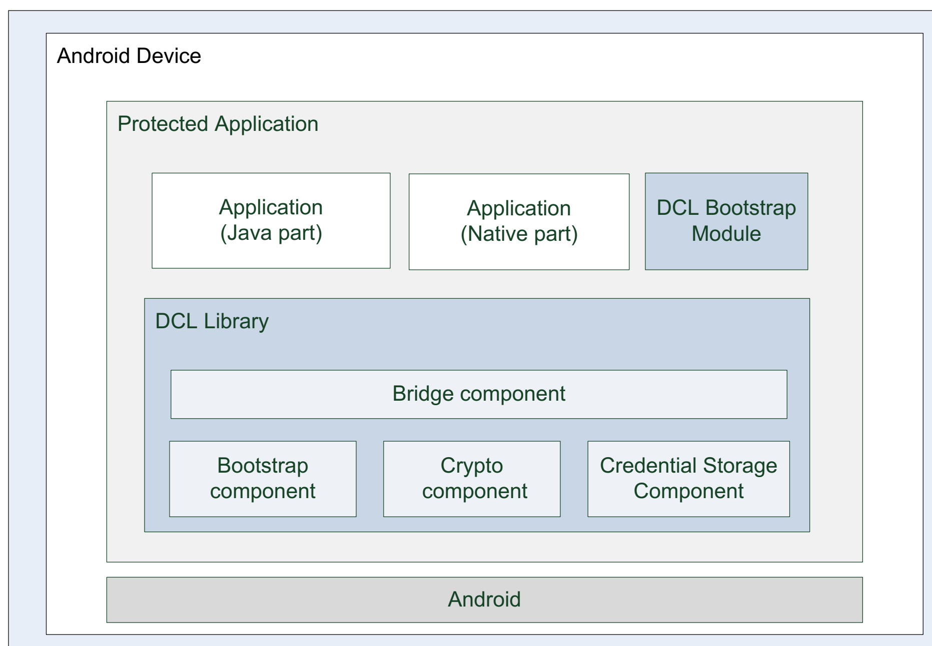

Figure 4 – DCL Protection Architecture

6.3.1 DCL Bootstrap Module

The role of this component is to initialize the DCL library when the application is launched for the first time. Initialization means setting the keys that are used by the DCL library.

A key is used in Credential Storage Component of the DCL Library to protect the data when they are stored. This key has to be set as part of the initialization of the library. A static key is bundled in the DCL library that is diversified with data given by the DCL Bootstrap Module. The device fingerprint function uses the following data to make the fingerprint:

- Android ID (Settings.Secure.ANDROID_ID)
- RO Serial (ro.serialno)
- IMEI if it exists

The provisioning of the DCL library is done after the internal key setting described above. The DCL Bootstrap Module passes the credentials to the Bootstrap component of the DCL library. These credentials have been previously prepared during the build of the application. A symmetric static key is used to encrypt the credentials and the same static key is used by the Bootstrap component to retrieve the credentials.

In a previous release of this component implemented outside of the project it was designed as a proxy between the DCL library and a remote initializing server. It is a nice feature when the application is finished and validated but is very difficult to use during the debugging and validation phases of a project. The offline release of the DCL Bootstrap Module implemented for the project enables to protect credentials involved in a cryptographic function in a simple way without complex server configuration synchronized with library configuration.

Note that using the offline implementation still leaves the possibility to use one day an online DCL Bootstrap Module. The existing online DCL Bootstrap Module would just need to be upgraded to support the interface of the DCL Library. It is not proposed in ASPIRE because of the administration complexity of the server that is not adequate for a research project.

6.3.2 Bridge component

The Bridge component is the external interface of the DCL library. Functions of this components fall into two categories. Functions in the first category are implemented partly in

the Bridge component itself. Functions of the second category are fully implemented in the Crypto component and the processing done in the Bridge is just to call the adequate function in the Crypto component with the required arguments. The functions in the first category are listed in the Table 1 – Functions implemented in the Bridge component.

Table 1 – Functions implemented in the Bridge component.

Function name	Description
computeHmac	keyed-Hash Message Authentication Code
Pbkdf2	Key derivation, convenient function that calls functions in Crypto component
ComputeHOPT	HMAC-Based One-Time Password Algorithm
CryptDataWithAESKey	AES encryption,
CryptDataWithRSAKey	RSA encryption

The algorithms implemented in the Crypto component that are called by the Bridge component are in the Table 2 – Algorithms of the Crypto component called by the Bridge component.

Table 2 – Algorithms of the Crypto component called by the Bridge component.

Algorithm	Comment
AES (CBC and ECB modes)	ENCRYPT & DECRYPT
RSA	ENCRYPT & DECRYPT
Key Generation	AES, RSA
PBKDF2_SHA1	Key derivation
PBKDF2_SHA256	Key derivation

6.3.3 Crypto component

Most of the Bridge component functions ultimately calls functions of the Crypto component. This component contains many cryptographic functions and only a part of the Crypto component functions have been made accessible by the application through the Bridge component. Giving access to more functions is only a matter of engineering and validation. This component has not been implemented for the project.

6.3.4 Credential storage component

The Credential storage component stores data under an encrypted format. It uses a name/value pair to store/retrieve data to/from the storage area. Each data is encrypted with AES-CBC. Authenticity and integrity are achieved with a keyed-hash message authentication code (HMAC). This component has not been implemented for the project.

There can be multiple storages each data protected with this mechanism has their own individual keys. The storage key is derived from three parameters:

- The inode number that is the current node number of the storage file itself.
- The Device fingerprint as introduced in 6.3.1

- The diversified key also introduced in 6.3.1; don't be confused by the name, the storage key used for a data is derived from this common diversified key.

6.3.5 Glue code

Some glue code is inserted in the native code of the application based on annotations set in the source code. This insertion is done at source level by ACTC. Annotations and ACTC support is described in the deliverable D5.08 in the Section 3.6.

Section 7 Conclusions

This deliverable is the final progress report of WP2 on Offline Software Protections. Almost all offline protection techniques have been designed, implemented and tested. The only remaining protection techniques that have not yet been fully implemented and tested are Control Flow Tagging and Diversified Cryptography, which both still require some work. However, the plan is to have these tested and integrated in time for the GTO tiger team experiments, and their final status will be discussed in future non-WP2 deliverables.

Section 8 List of Abbreviations

ACTC	ASPIRE Compiler Tool Chain (ACTC)
ADS	Area Data Structure
ASPIRE	Advanced Software Protection: Integration, Research and Exploitation
CFT	Control Flow Tagging
IMEI	International Mobile Station Equipment Identity
DCL	Diversified Crypto Library
RA	Remote Attestation