Advanced Software Protection:
Integration, Research and Exploitation

D5.06

Preliminary ASPIRE Online Protection Tool Chain 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:	Deliverable
Deliverable reference number:	ICT-609734 / D5.05 / 1.0
WP and tasks contributing:	WP 5 / Task 5,1
Due date:	November 2015 – M24
Actual submission date:	26 November 2015
Responsible Organization:	NAGRA
Editor:	Brecht Wyseur
Dissemination Level:	Public
Revision:	1.0

Abstract:
This report documents deliverable D5.05, which is the preliminary ASPIRE Compiler Tool Chain (ACTC) that is delivered at M24 and implements support of several ASPIRE online protection techniques.

Keywords:
ACTC, online protections

Editor

Brecht Wyseur (NAGRA)

Contributors (ordered according to beneficiary numbers)

Bart Coppens, Bert Abrath, Jens Van den Broeck, Bjorn De Sutter (UGent)

Rémi Cohen-Scali (NAGRA)

Ronan Le Gallic (EDSI)

Mariano Ceccato (FBK)

Werner Dondl (SFNT)

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.

Executive Summary

This report documents all the work that has been done in Task 5.1 since M18, which is the development on the ACTC and the integration of the ASPIRE protection techniques into the ACTC. This report documents prototype deliverable D5.05, which corresponds to the ACTC tagged as version 1.5.0.

The major achievements during this period include

- The integration of protection techniques such as the ASPIRE Common Communication Logic (ACCL), the code mobility technique, additional data obfuscation techniques, additional white-box tools, the anti-debugging technique, and several more.
- Additional support has been implemented in the ACTC such as for example support for self-profiling
- The ACTC core itself has been improved to include for example shared library support, and
- Addition configuration options have been implemented.

With these improvements, the project consortium has met almost all of the requirements for Milestone 11.

The ACTC has also been deployed on the ASPIRE use-cases. For this, some adaptations to the use-cases were needed. In particular, each of the industrial partners detailed the assets of their use-cases. This was done much more fine-grained than the initial description as in the ASPIRE Deliverable D1.01, and included references to the code sections, concrete requirements on concrete assets in the code, and advice on which techniques may help to meet these requirements. Based on this, annotations were written in the use-case source code such that the ACTC could parse these and as a result, relevant protection techniques can now be deployed by the ACTC on the ASPIRE industrial use-cases.

Contents

Section 1 Introduction

Section Authors: Brecht Wyseur (NAGRA), Ronan Le Gallic (EDSI)

1.1 Content of the document

This report describes the advances made on the ASPIRE Compiler Tool Chain (ACTC) during the M18-M24 period. A first report on the initial progress in the first year has been presented by ASPIRE Deliverable D5.03 “Offline Tool Chain Report” which reflect the ACTC release that has been delivered in M12. That release was tagged as version 0.3.0. Progress made in the M12-M18 period was reported in Deliverable D5.04 with the prototype delivery tagged as version 0.8.0. This report corresponds to the M24 delivery of the prototype deliverable D5.05 which is tagged as version 1.5.0.

We present in this report the advances made on the ACTC itself, details on the tools that have been integrated and how this integration advanced, and the deployment of the ACTC onto the use-cases. This is based on the architecture design and API descriptions of the ACTC as presented in ASPIRE Deliverable D5.01 “Framework Architecture, Tool Flow, and APIs of the ASPIRE Compiler Tool Chain and Decision Support System”, and on ASPIRE Working Document WD5.02 which is an internal working document that is based on D5.01 in which all partners keep up to date the latest definitions of their annotations and specifications of the APIs throughout the project. This WD5.02 document serves as the reference document for all partners in the project and reflects at all times the state of the ACTC. WD5.02 is an essential part of the continuous integration process in the ASPIRE project.

1.2 Overall status

With D5.06, the aim is to meet milestone MS11, which is defined as follows in the DoW:

> *Anti-tampering is integrated from WP2, together with server-side execution and mobile code from WP3.*

This milestone is to be interpreted on top of previous milestones such as MS08 of M18. Below, we present a list of tools that have been delivered in WP2 and WP3 and have been actually integrated and validated in the ACTC. This can be split over

Support tools:

- Annotation Extraction tool (SLP04) [FBK]
- GrammaTech CodeSurfer (SLP05.01)
- Standard compiler, assembler and linker (patched with UGent's patches, see D5.01)

Tools related to offline protection techniques:

- WBC Annotation Extraction Tool (SLP03.01), WBC Tool (SLP03.02) and WBC Source Rewriting Tool(SLP03.05) [FBK,NAGRA]
- Data Obfuscation (SLP05.02) [FBK]
- Native code Extractor (Diablo BLP01) [SFNT, UGent]
- Native code to binary code X-Translator (BLP02) [SFNT]
- SoftVM Integration Tool (Diablo) (BLP03-04) [SFNT, UGent]
- Anti-Debugging (BLP03-04) [UGent]
- Binary code obfuscation (BLP03-04) [UGent]

Tools related to online protection techniques

- Client-Server Code Splitting (SLP06) [FBK]
- Code Mobility (BLP003-04) [UEL, UGent]
- Remote Attestation (SLP07, BLP03-04) [POLITO, UGent]

The references presented for each of these tools are related to the steps as described in ASPIRE WD5.02. Those references are also used in the implementation of the ACTC and in the directories that contain intermediate protection and compilation results as produced by the ACTC. We refer to D5.03 and D5.04 for more details regarding protection techniques that were integrated before M19.

For the delivery of each of the tools, the project partners provided extensive documentation to support the integration process. As an example, the document that supports the integration for BLP03-04 will be made available to the reviewers for the year 2 review.

With respect to the milestone delivery, almost all the tools as presented in the ASPIRE Description of Work have been integrated, except for

- Multi-threaded crypto [GTO]
- Anti-cloning [NAGRA]

As such, the project consortium has met almost all of the requirements for MS11.

Section 2 Evolution of the ACTC framework

Section Authors: Brecht Wyseur (NAGRA), Ronan Le Gallic (EDSI)

2.1 Major Evolutions in M19-M24

The ACTC has evolved significantly in the second half of the second year of the ASPIRE project. Not only have protection technique tools been integrated, but a lot of additional support and features have been implemented to facilitate the deployment of the ACTC on test samples and the ASPIRE use cases, and to increase the usability of the tool chain such that all project partners can use the tool chain efficiently. To further facilitate development, an ASPIRE bug tracking tool was also put in place.

The majority of this work was executed by EDSI.

Below, we present a high level overview the major evolutions. A more complete detailed overview can be found in the change log that is presented in the next Section and in the ASPIRE bug tracking tool.

- **Integration of tools**. This was the core activity of the work on the ACTC in the M19-M24 period. There were a number of improvements on the tools that were already delivered in the first year of the project that required additional integration work. New tools that were integrated included
  - The ASPIRE Common Communication Logic (ACCL), to allow for communication with the ASPIRE server
  - Code Mobility
  - Additional Data Obfuscation components
  - Client-Server Code Splitting
  - Additional White-Box Tools
  - Remote Attestation
  - Anti-Debugging
  - The SoftVM interpreter as a part of the client-side code splitting protection
  - Additional techniques in the Diablo framework
- **Additional support tools**, such as
  - an FTP upload feature to facilitate file management for integrating server-side generated files
  - Support for self-profiling
  - Adding annotation files merge
- **Improvements to the ACTC core**, such as
  - graph generation based on .dot files
- **Additional configuration options**, such as
  - for invoking features such as the graph generation, or
  - platform configuration

2.2 ACTC Releases

Table 1 contains the change log of the various releases of ACTC delivered since M18. Numbers refer to defaults logged in the ASPIRE bug tracker tool, and as tagged in the releases that have been committed into the ASPIRE SVN.

The last version reported upon was tagged as version 0.8.0 and reported in D5.04. We present the releases since, spanning the M18-M24 period.

Releases	Change Log

1.0.0
May 22, 2015
	- core
	  - added POST processing task (#107)
	  - fixed task SLP03_04 adding include path (#109)
	  - fixed task SLP03_MERGE filtering server files (#113)
	  - fixed task SLP03_02 filtering empty files (#116, #122)
	  config
	  - added POST processing parameters (#107)
	- tools/diablo
	  - updated interface with "dots" folders (#124)

1.1.0
July 10, 2015
	- core
	  - added pragma reverse conversion in task_SLP03_05 (#131)
	  - added AID generation and invocation by compilers (#133)
	  - added AID option for diablo (#133)
	  - added processing graph generation (using dot)
	- config
	  - added codesurfer tool (#132)
	- tools/wbc
	  - added PragmaConverterReverse tool (#131)
	- tools/codesurfer
	  - added Initialiser and Analyser tools (#132)

1.2.0
August 27, 2015
	- core
	  - added client/server code splitting steps (#139)
	- tools/splitter
	  - added client_server_splitter tools (#139)
	
	An additional minor update version 1.2.1. has been delivered on September 16th, which fixes additional compilation issues.

1.3.0
October 16, 2015
	1.3.0a
	- core
	  - added AID.txt creation
	  - updated steps with "platform" paths
	  - reworked client/server code splitting steps
	  - fixed C compilation (#141)
	- config
	  - added "platform" option
	- tools/diablo
	  - added "-CMO" option in DiabloObfuscator tool
	
	1.3.0b
	- core
	  - added offline needed object files in task_BLP03_LINK
	  - fixed order of object files in linker command (#144)
	- config

	- added "accl" & "code_mobility" tools 1.3.0c - core - fixed task_BLP03_LINK parameters - tools/wbc - added "-l logfile.json" option to WbcSourceRewriter tool 1.3.0 - config - added "third_party" option - added "UPLOAD" options - core - added task_UPLOAD - updated task_BLP03_LINK parameters adding OpenSLL libraries - reworked graph generation - tools/splitter - added "–l logfile.json" option to SplitterClientGenerator tool - tools/data - added "-a logfile.json" option to DataObuscator tool - tools/ftpupload - FTPUpload - new
1.4.0 October 21, 2015	- config - added "SERVER" options - core - added task_SERVER_PXX to manage server files - reworked task_SLP04_MERGE & task_BLP03_LINK to link only necessary libraries
1.5.0 (ongoing)	- core - added task_SLP04_COPY - tools/__init__ - fixed folder creation - tools/splitter.py - patched SplitterNormalizer command line - tools/remote attestation

Table 1: ACTC change logs

Section 3 Source-level ACTC components

Section Authors: Mariano Ceccato (FBK), Brecht Wyseur (NAGRA)

3.1 Data obfuscations

During the second year of the ASPIRE project, the activity on task T2.1 consisted in enhancing the Data Obfuscation component by means of the following major extensions:

- *Opaque Obfuscation Parameters*: constant parameters involved into encoding/decoding functions do not appear as literal values in the code, anymore. They are instead encoded as opaque constants, i.e., their values are computed dynamically at runtime. Opaque Obfuscation Parameters are generated by functions whose code is resistant to static analysis. In fact, the analysis to be successful requires an NP-complete problem (the k-clique problem) to be solved.
- *Dynamic XOR masking*: in the previous version of the Data Obfuscation component, the parameter (mask) used in the XOR masking technique was established at obfuscation time. In the new version, a random value for the mask is generated at each execution. Obfuscated data observed in multiple runs are unlikely to have same values hardening the task of the attacker to perform statistical analyses.

Annotations related to data obfuscation were extended to accommodate the set of new configuration parameters required by the extension listed above. Consequently the annotation parser is extended. This extended version of Data Obfuscation has been described in Section 2 of Deliverable D2.08.

3.2 Client-server code splitting

The ACTC integrates client/server code splitting tools as component SLP06 (see Figure 1).

The technique applies barrier slicing to identify the portions of the application to move, and a set of transformations to generate the new client application and its trusted server. The two new components will then execute these portions of code in a synchronous way to preserve the original functionalities of the application (see deliverable D3.04 for further details).

The component (SLP06.01) responsible for the computation of the barrier slice is implemented by means of CodeSurfer. The code in input is analyzed with a custom slicing algorithm that performs the extraction of the barrier slice (indicated as D06.02 in Figure 1). Other data are also extracted at this stage, to apply later code transformations in the correct way. Additional CodeSurfer scripts perform the extraction by generating a set of fact files (D06.01).

Component SLP06.02 and component SLP03.03 generate the protected client application and the server-side code to run the slice, respectively. The two components both apply on the pre-processed code of the original application, while the facts (D06.01) and the barrier slice (D06.02) extracted with CodeSurfer are used as input for the code transformations. Client generation (SLP06.02) and server code generation (SLP06.03) are implemented in TXL. They apply code transformations to produce the protected client application (SC07) and the server-side sliced code (SCS01).

Integration with the ACTC works with the main splitting components described earlier, but also with several other support steps that are required to run the whole client/server code splitting tool in the correct way.

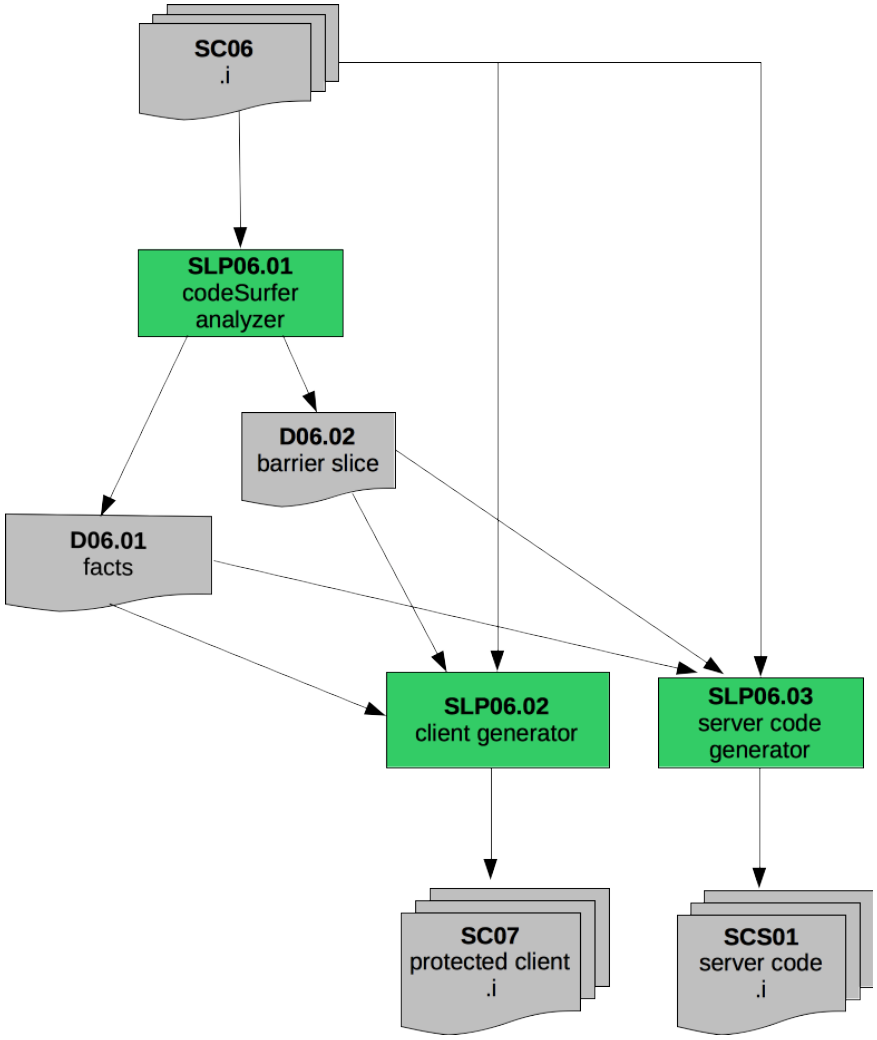

Figure 1 - Client/server code splitting tool (overview)

The integration process has been conducted in incremental steps. We started by integrating CodeSurfer and its analysis scripts, along with few other preparation steps. Preparation steps are necessary to extract and handle information required by CodeSurfer and its scripts for code analysis. To verify the correctness of this early stage of integration, we tested it on small examples, to verify a) the capability of CodeSurfer to run in the integrated environment, and b) the capability of CodeSurfer custom scripts to correctly extract data from pre-processed code.

Integration progressed by integrating the other major components of the client/server splitting tool, SLP06.1 (client generator) and SLP06.2 (server code generator). Preparation steps have been also integrated for components SLP06.1 and SLP06.2. At this stage, we tested the integrated tools on examples to check the correctness of the transformations applied at source-code level. This allowed us to discover and solve few issues (some of them are reported in the ASPIRE bug tracking systems).

At current stage of development, the client/server code splitting tool can be used with the examples provided by the ACTC maintainer. The tools support also annotation logging capabilities. Annotation logging can be activated by using the flag “-l” and it produces a log file to report all the annotations that the tool processed during its application.

3.3 White Box Cryptography Tools

In Year 2 of the ASPIRE project, the White-Box Tool for ASPIRE (WBTA) has been improved to support additional cryptographic primitives and mode of operations, and several white-box implementations have been delivered. These include fixed-key white-box AES implementations (encryption and decryption) and dynamic-key white-box AES implementations (encryption and decryption). Preliminary work on the WBTA has been delivered in M18 with a release tagged as Release 1.2.0 and reported in ASPIRE Deliverable D2.04 “White-Box Crypto Library and Code Generation”, additional progress with report on the delivery of the final WBTA and the white-box AES implementations has been delivered in ASPIRE Deliverable D2.08 “Offline Code Protection Report”.

As reported in Section 3.3.4.3 of Deliverable D2.08, the integration of the additional functionalities in the ACTC did not require any changes to the ACTC itself. That is because (1) all the protection techniques are offline techniques and (2) adding additional modules and features to the WBTA is abstracted by the source-to-source tools. The ACTC invokes these tools. If under the hood the white-box transformation is a white-box AES implementation of a certain type or any other white-box implementation; from the point of view of the ACTC process this is all the same.

3.4 Anti-Cloning

NAGRA envisioned to implement and integrate the Anti-Cloning technique at the end of Year 2 of the ASPIRE project, as was described in the ASPIRE Description of Work. However, due to a priority shift, where NAGRA focused more on other work packages, the work on the anti-cloning technique only concerned the finalization of the design and planning of implementation and integration activities. This is described in ASPIRE Deliverable D3.04 “Intermediate Online Protections Report”, Section 6.

Early in Year 3, NAGRA will implement and integrate the anti-cloning technique. This minor delay does not introduce any issues in the project planning because there are no dependencies on the anti-cloning technique except for the validation and the security evaluation work which is planned to be executed after the updated delivery date of the technique.

3.5 Remote Attestation

In Year 2 of the ASPIRE project, remote attestation techniques have been developed. The tool support to insert the necessary remote attestation code functionality into the application to be protected has been added to the ACTC. Currently, it supports static remote attestation.

Static remote attestation has already been presented at the M18 review as an isolated prototype and as delivered in Section 5 of deliverable D3.02. Since then, static remote attestation has been extended and integrated in the ACTC at M24. Updates to the static remote attestation are reported in Section 5 of deliverable D3.04. During the last 6 months, the isolated prototype has been extensively tested on sample applications to guarantee the correctness of the remote attestation protection. Dynamic remote attestation is still an isolated prototype (under testing) and it is not reported here.

Protection-specific annotations related to remote attestation have been extended to steer the ACTC the application of static remote attestation correctly. Annotations as previously documented in deliverable D5.01 were mainly tailored for dynamic remote attestation, i.e., based on invariants monitoring. Consequently, the annotation parser has been extended to extract the new remote attestation annotations.

The ACTC integrates a first remote attestation tool in the source code processing step SLP07. With the inclusion of that step, the overall source-level tool flow now looks as depicted in Figure 2. SLP07 is invoked after all other source-level protections have been applied.

SLP07 is executed as a Bash script. SLP07 selects the attestator to add into the application as part of the protection. The Attestator to insert is specified in the source code annotations. If an annotation does not specify the Attestator to use, two case are considered. If no annotations specify an Attestator, a default Attestator is inserted. If one or more annotations specify the (same) Attestator to use, this Attestator is also used for other annotated code areas that do not specify Attestators. However, if two or more different Attestators are specified by annotations, the RA tool stops and reports an error, as the current tool does not support multiple attestators yet.

The remote attestation script takes as input the JSON file containing the extracted annotations (the D01 annotation facts). The script first invokes an interpreter, written in Java and released as a JAR executable archive. The interpreter parses the JSON annotations file and deduces the Attestator to link into the application. Once the Attestator to deploy is identified, the script selects the object files that implement the identified Attestator among a set of precompiled Attestators object files. The set of all the Attestators object files is represented as BC11 in Figure 3. The script copies the selected object file into the output folder (BC12). Together with the attestator object file, the script also copies another object file that implements the functionality common to all the Attestators. Together, these two files form BC12, that will be linked into the program as explained in Section 4.

The SLP07 script is portable and can be executed on any platform where a Bash shell and Java (at least version 1.5) are installed. At the current stage of development, the remote attestation tool can be used with the examples provided by the ACTC maintainers. They are now also being tested (and debugged if necessary) for the ASPIRE use cases as part of the intermediate validation of WP1 Task T1.5.

As explained in Section 4, a second remote attestation processing step is implemented in the binary-level part of the ACTC, in BLP04. In that step, the Diablo-based rewriter determines the exact location of the code areas to attest in the memory space of the protected binary or library.

Further details on the static remote attestation tool are available in deliverable D3.04, Section 5.3.3.

The tools do not yet support annotation logging. That support will be added during Y3. In addition, as agreed during the last ASPIRE workshop, the tool will be updated with support for multiple attestators, better integration with code mobility, and also attestation capabilities for read-only data sections (to meet requirement REQ-NFS-008, see D1.03).

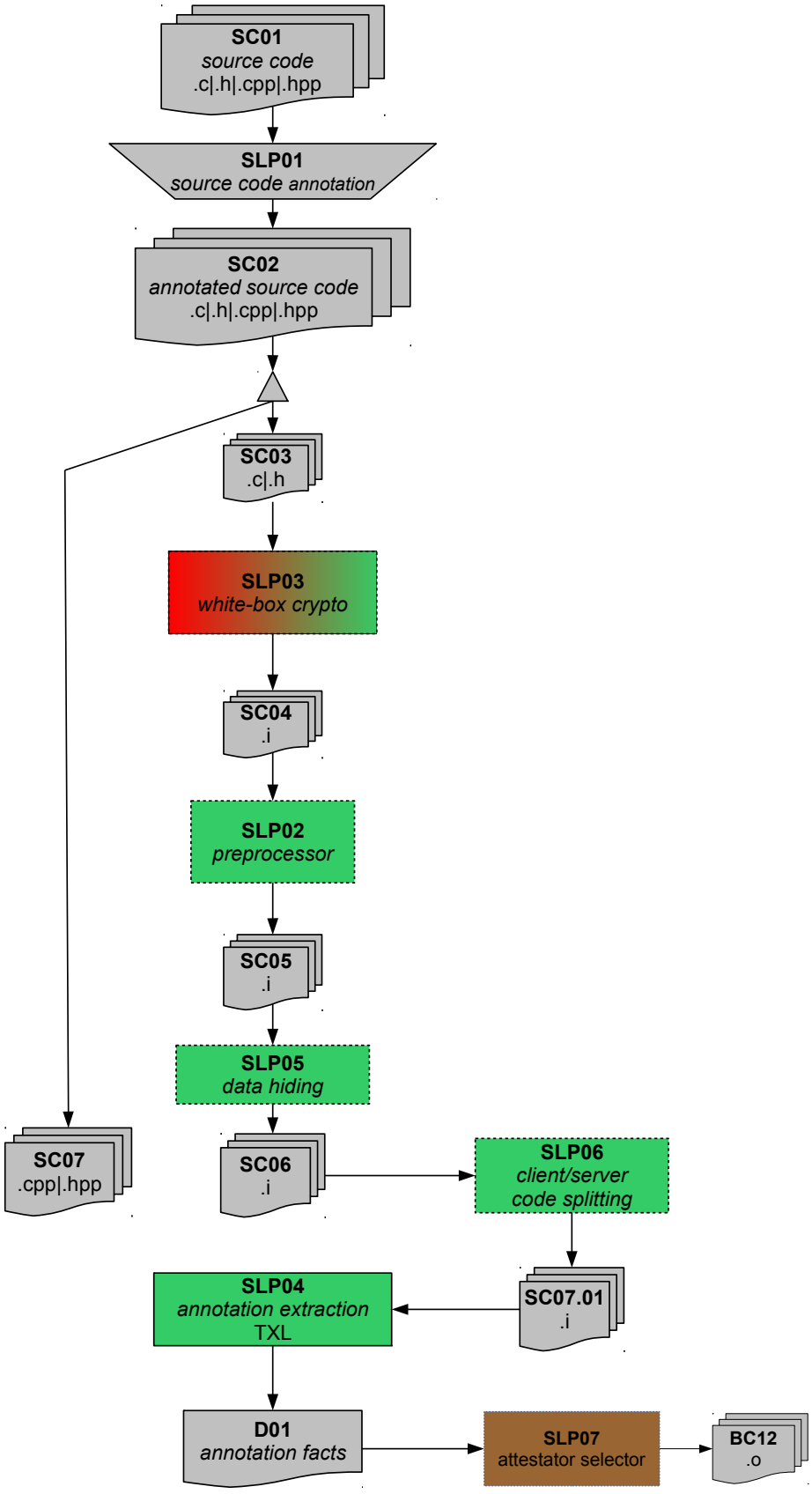

Figure 2 - Overall source-level tool flow of the ACTC after the inclusion of remote attestation

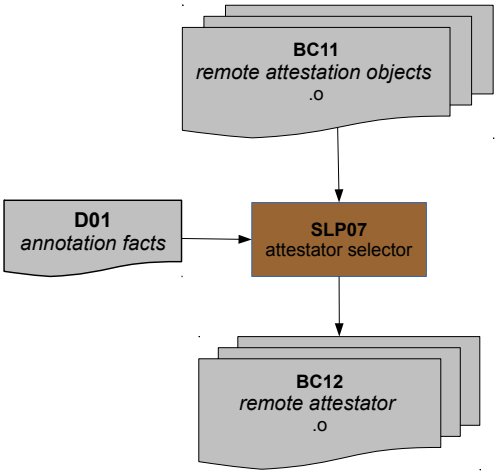

Figure 3 – Remote attestation tool

Section 4 Binary-level ACTC components

Section Authors: Bart Coppens, Bert Abrath, Jens Van den Broeck, Bjorn De Sutter (UGent)

4.1 Communication and Protection Libraries

In the previously reported and developed versions of the ACTC (see deliverables D5.01, D5.03, D5.04), the binary-level component BLP03-04 was responsible for linking libraries and objects that implement precompiled parts of the ASPIRE protections into the protected applications. Those libraries and objects were numbered BC09 before.

This design decision has been revised during year 2. In the updated design, the precompiled libraries and object files now numbered BC10 (everything except remote attestation) and BC12 (precompiled remote attestators and support code, as selected from BC11 during the remote attestation compilation step SLP07, see Section 3.4) are linked into the protected application as part of the standard compilation & linker process. Figure 4 visualizes this. This figure replaces Figure 6 of D5.03 and Figure 10 of D5.01.

The main reason for this design change is that it simplifies the overall tool flow design, making it easier to compose protections and to compute complexity metrics.

Android and Linux versions of those libraries are available in the project's shared build VM, and the ACTC feeds the linker the appropriate files based on the presence of annotations in the annotation facts file D01 corresponding to the ASPIRE protections that come with libraries. The ACTC chooses Android or Linux versions based on settings in the JSON ACTC configuration file.

Currently, libraries and objects are available for

- **SoftVM**: the interpreter to be embedded as part of the client-side code splitting protection (see D2.08 Section 4 and D1.04 v2.0 Section 3.3).
- **Code Mobility**: the binder and downloader (see D3.04 Section 3 and D1.04 v2.0 Section 3.4)
- **Remote attestation**: communication with the server, attestators, reaction logic, etc. (see D3.04 Section 5 and D1.04 v2.0 Section 4)
- the client-side **ASPIRE Common Communication Logic** library (see D1.04 v2.0 Section 2) and third party libraries this library relies on (openssl, curl)
- embedded self-debugger of the **anti-debugging** protection (see D2.08 Section 6.1 and D1.04 v2.0 Section 3.2)

The precise list of all files (libraries and individual object files is as follows:

```
./xtranslator/obj/android/vmExecute.o
./xtranslator/obj/android/vm.a
./xtranslator/obj/linux/vmExecute.o
./xtranslator/obj/linux/vm.a
./xtranslator/tags/2.4.5/obj/vmExecute.o
./xtranslator/tags/2.4.5/obj/vm.a
./code_mobility/downloader/obj/android/downloader.o
./code_mobility/downloader/obj/linux/downloader.o
./code_mobility/binder/obj/android/downloader.o
./code_mobility/binder/obj/android/binder.o
./code_mobility/binder/obj/linux/downloader.o
./code_mobility/binder/obj/linux/binder.o
./3rd_party/openssl/android/lib/libssl.a
./3rd_party/openssl/android/lib/libcrypto.a
./3rd_party/openssl/linux/lib/libssl.a
```

```
./3rd_party/openssl/linux/lib/libcrypto.a
./3rd_party/curl/android/lib/libcurl.a
./3rd_party/curl/linux/lib/libcurl.a
./ACCL/obj/android/accl.a
./ACCL/obj/linux/accl.a
./ACCL/curl/android/libcurl.a
./ACCL/curl/linux/libcurl.a
./RA/obj/android/libattestator_1.o ...   ./RA/obj/android/libattestator_40.o
./RA/obj/android/libracommon.o
./RA/obj/linux/libattestator_1.o ... ./RA/obj/linux/libattestator_40.o
./RA/obj/linux/libracommon.o
```

The BLP03-04 component (based on Diablo) has been updated to ensure that the linked-in libraries and objects are not removed from the program during Diablo's initial construction of the program's control flow graphs, despite the fact that some of the code in those libraries and objects is not yet reachable in the program during that construction.

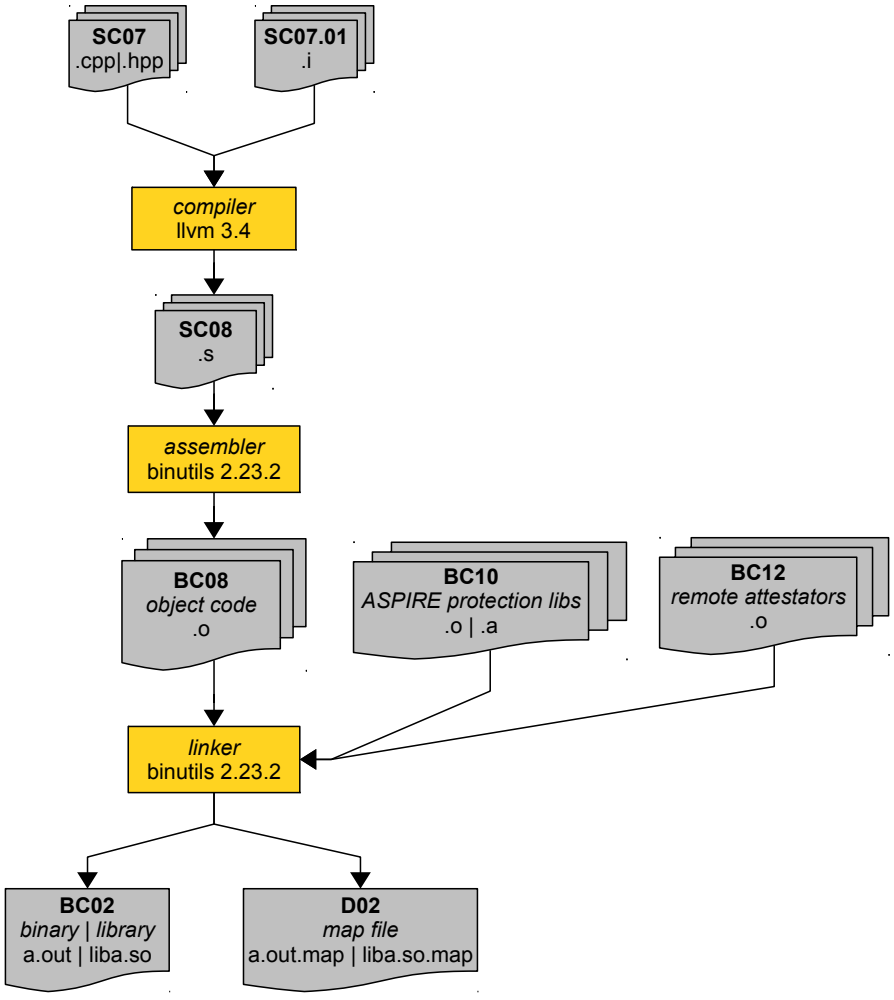

Figure 4 - Updated compiler & linker phase of the ACTC.

Of course the binary part of the ACTC is updated accordingly. The overall picture now looks as shown in Figure 5. This figure replaces Figure 11 of D5.01. BLP01 still consists of the selection of native code fragments to be X-translated to bytecode, BLP02 still performs the (initial) X-translation, BLP03 is responsible for integrating the bytecode and the stubs that replace the translated native code, and BLP04 is responsible for performing the remaining transformations. BLP03 and BLP04 are in fact executed in one run of a Diablo-based binary-rewriting tool, which is hence called BLP03-04.

BC09 is still present because BLP03-04 requires some external object files to produce self-profiling versions of the binaries/libraries as needed for performance estimation, profile-based protection optimization, and for computing dynamic security metrics.

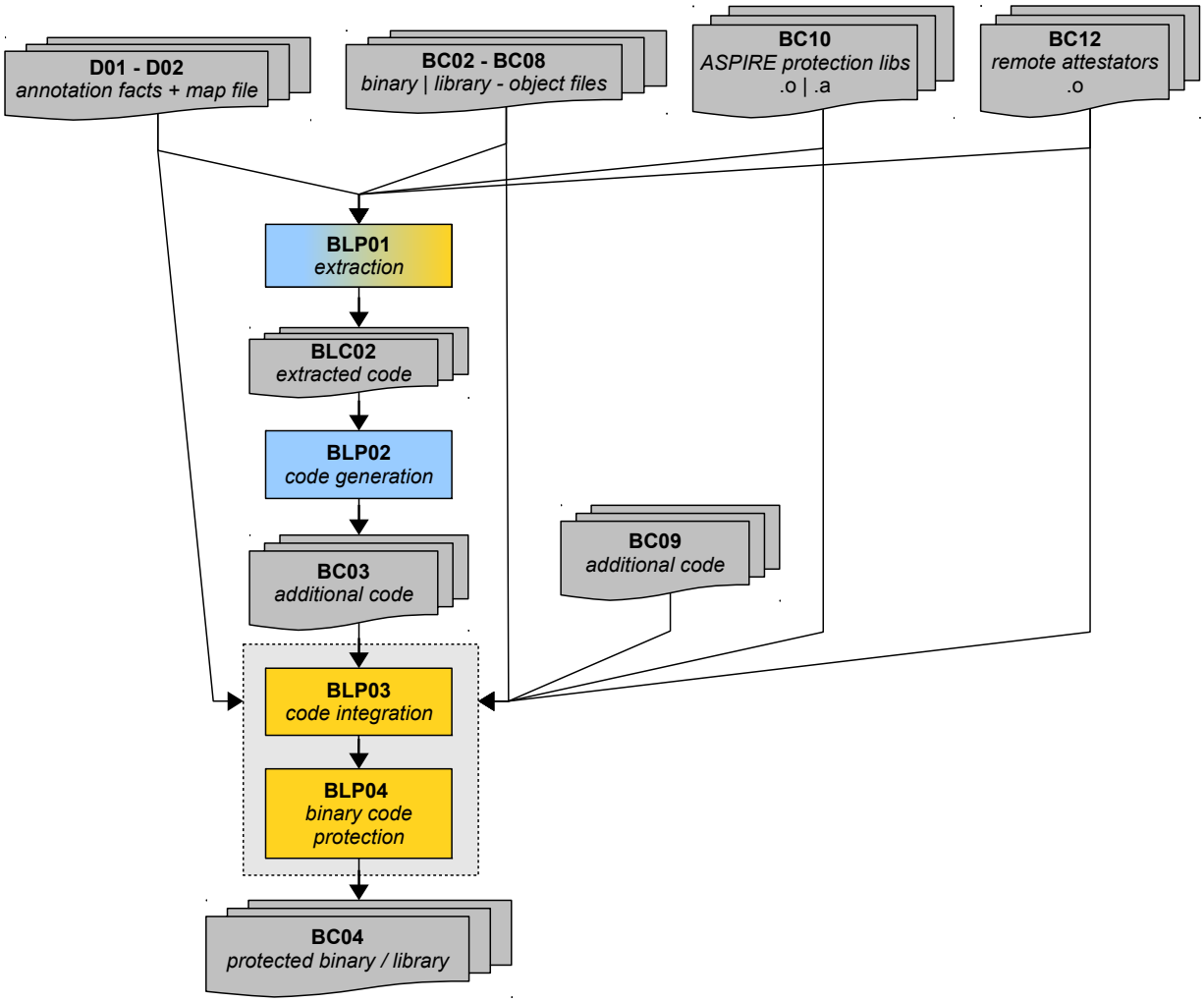

Figure 5 - Four steps of the binary part of the ACTC.

4.2 Updates for client-side code splitting (BLP01,02,03,04,05)

The SoftVM (previously in BC09, now in BC10) and cross-translator (BLP02) have been upgraded from a custom Java-based VM to an LLVM-based lli provided by SFNT (see also D2.08 Section 4).

A number of modifications were necessary to the Diablo-based tools BLP01 and BLP03-04 to support a number of API changes for interacting with this new SoftVM implementation. The most important one is the invocation of second X-translator BLP05, as shown on the left of Figure 6.

The LLVM-based lli VM supports bytecode fragments with multiple exit points, of which the (position-independent) continuation addresses in the native code are hidden in the bytecode. Initially, as BLP02, the X-Translator generates bytecode BC03 that only holds placeholders for those continuation addresses: As BLP03 and BLP04 have not been executed yet, the addresses in the final binary/library BC05 are simply not known yet.

One option would have been to let Diablo, as part of the BLP03 integration of the generated bytecode, fill in the addresses in those placeholders. However, that would make BLP03-04's

internal operation dependent on internal aspects of the VM and of the X-translator, and therefore violate the separation of concerns that we strive for in the plugin-based ACTC design.

We therefore decided to implement an alternative approach, in which BLP03-04 simply invokes the X-translator again, now as BLP05 and via a slightly different API. This invocation is performed when the final layout of the binary has been determined as part of BLP04, and the final addresses of the continuation points are passed to the X-translator, such that it can fill in the placeholders itself, using whatever (diversified) encoding it chooses.

This invocation of the X-translator as BLP05 is visualized in Figure 6. This figure replaces Figure 16 in D5.01 and Figure 9 in D5.03.

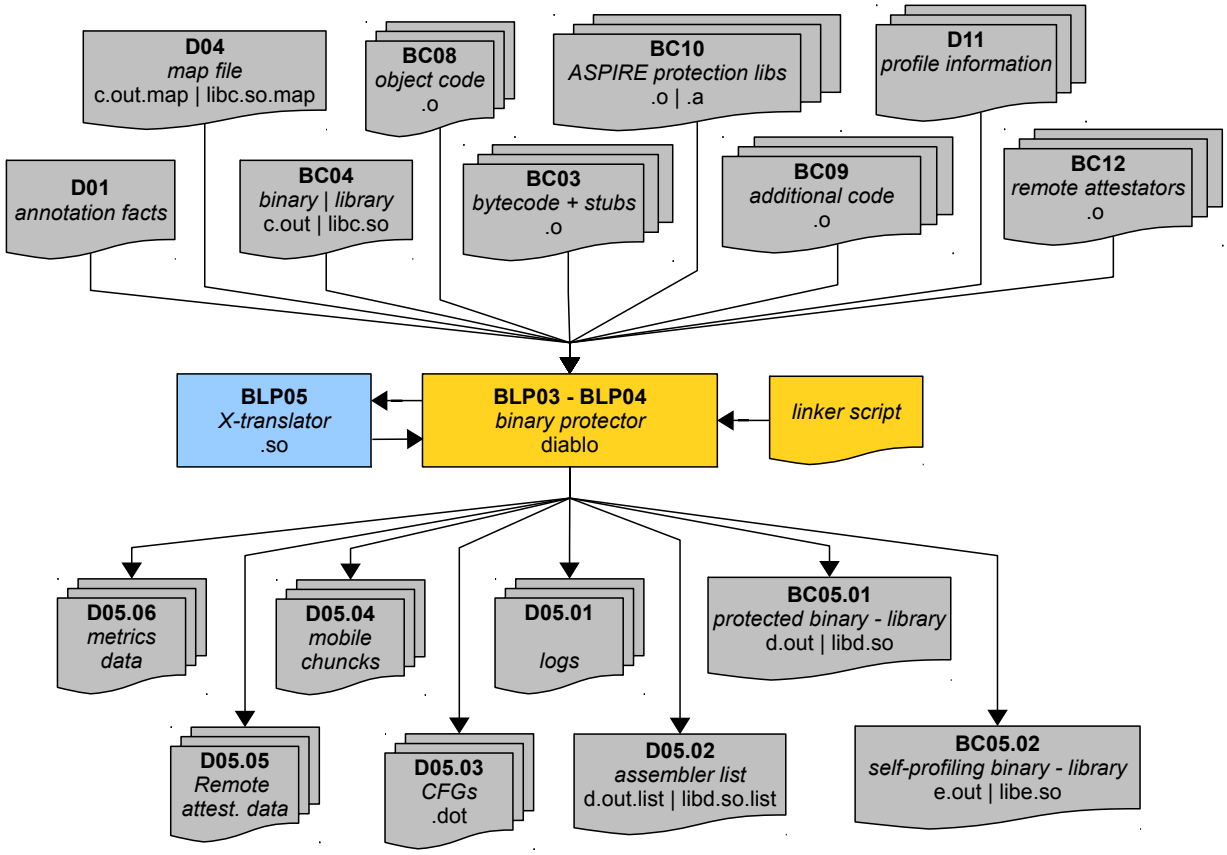

Figure 6 - Final tool BLP03-04 invoked in the ACTC to deploy several protections.

4.3 Updates for other protections (BLP04)

Compared to Figure 16 in D5.01 and Figure 9 in D5.03 that visualized BLP03-04 in the previous iterations of the ACTC, Figure 3 is much more complex. This of course results from the many protections that have been integrated into the ACTC from M19 to M24 of the project.

We have chosen the following order to apply the many transformations needed to deploy the protections

1. **Code factoring** (to hide and blur the boundaries between application code (originating from BC08) and code of the different protections (originating from BC10). See D2.0X Section X for more information about this form of obfuscation contributed in WP2 Task T2.4 by UGent.
2. **Control flow obfuscations** (opaque predicates, branch functions, control flow flattening). See D2.06 and D2.08 Section 5.1.1 for more information about these forms of obfuscation contributed in WP2 Task T2.4 by UGent.

3. **Anti-call back stack checks**. See D1.04 v2.0 Section 4.3 and D2.08 Section 6.2 for more information about this protection contributed in WP2 Task T2.5 by UGent.
4. **Anti-debugging**. See D1.04 v2.0 Section 3.2 and D2.08 Section 6.1 for more information about this protection contributed in WP2 Task T2.5 by UGent. A first version of anti-debugging has been implemented and is integrated in the ACTC. This version is capable of moving regions that do not make any function calls to the debugger context.
5. **Code Mobility**, including the extraction of mobile blocks into files D05.04. See D3.04 Section 3 and D1.04 v2.0 Section 3.4 for more information about this protection contributed in WP3 Task T3.1 by UEL. This protection is integrated completely.
6. **Layout randomization**. See D2.06 for more information about this form of code obfuscation contributed in WP2 Task T2.4 by UGent.
7. **Remote attestation data.** For this protection, the Area Data Structures are produced in data files D5.05. These require the final code layout. It is for that reason that this protection is applied after layout randomization. See D3.04 Section 5 and D1.04 v2.0 Section 4 for more information on this protection contributed in WP3 Task T3.2 by POLITO with some help from UGent. Remote attestation has been integrated in the ACTC. A version where multiple attesters can be present in the protected binary is being worked on.
8. **Bytecode re-X-translation** (BLP05), see Section 4.2 where it was already noted that this translation also requires the final code layout.

To enable these protections not only on the application code (from BC08) but also on the code implementing the other protections (from BC10), the ACTC combines the annotation fact file D01 received from the source-level part of the ACTC (and hence limited to code from BC08) with an externally defined fact file that describes which protections to apply onto the pre-compiled libraries.

All support for the protections in BLP03-BLP04 based on Diablo has been implemented and integrated by UGent.

During the integration of the different protection techniques, we discovered some issues in the implementation of the binary control flow obfuscations. While these caused no problems when applying only control flow obfuscations, it was impossible to combine control flow obfuscations with some other binary protection techniques. We fixed these issues; now we can combine binary control flow obfuscations and other protection techniques in the ACTC.

Throughout all protections, detailed logging information is produced in separate log files D05.01, one per protection. Moreover, textual, machine readable versions of the control flow graphs of the code transformed as part of the protections are produced for every transformation in the form of the D05.03 .dot files.

4.4 Support for metrics and self-profiling

On top of the support for the individual protections, UGent integrated the necessary support for metrics and self-profiling as described in D4.03 Section 4.2 into BLP03-04. The instrumentation mentioned for item 2. in Section 4.2.2 of D4.03 is performed in between the above steps 4 and 5. The reason is that the basic block structure of the binary code (i.e., the granularity at which instrumentation operations are inserted) is finalized in step 4. From step 5 on, native code is converted and moved around, but the structure remains fixed.

4.5 Support for profile-guided protections

BLP03-04 can also read in profile information (obtained from an application version to which no binary-level protections had been applied yet). That information is represented by D11 in Figure 6.

This profile information is already used for two different purposes. First, it is used to compute dynamic complexity metrics as described in the first of two mechanisms in Section 4.2.2 of D4.03. Furthermore, support for profile-guided annotation region transformations has been implemented in the Diablo tools. This means that when the binary control flow obfuscations get an execution profile as an additional, optional input, the obfuscations will be applied on code regions that are executed the least. This reduces the overhead of the obfuscations on the protected application. Results obtained with this capability have been reported in D2.08 Section 5.1.1.

Section 5 Application of ACTC on Use cases

5.1 NAGRA use case

Section Authors: Brecht Wyseur (NAGRA), Rémi Cohen-Scali (NAGRA)

With respect to having the NAGRA use case compiled by the ACTC, the activities are related to adding annotations to the use case source code, and modifications to make sure that the use case can be used by all partners.

The correct application is being assessed as a part of the intermediate validation that will be reported in D1.05.

5.1.1 Annotations

Annotations have been added based on an asset description document that was produced. This document is an internal working document, referred to as ASPIRE Working Document WD6.02 “NAGRA Use Case Assets”, which is attached as an annex to this deliverable.

We refer to Section 6.1 in D1.05 for a detailed description of the protection selection process.

5.1.2 Adaptations to meet requirements of source-level ACTC components

Several additional adaptations on the use case were done to make sure that the academic project partners could easily use the NAGRA use case and test protection techniques thereon, and to meet demand for other activities in the project, such as for example performance tests for WP4.

To facilitate performance tests, it was required that the core libraries from the NAGRA use case could be invoked from the command line. The original use case implementation launched as a graphical user interface and the libraries in the Android Media Framework were invoked from this interface. Additional tests were implemented to allow for direct command line invocation.

Bug fixes were also implemented and a new SDK was delivered to the consortium as part of release 1.2.0.2 of the use case.

5.2 SFNT use case

Section Authors: Werner Dondl (SFNT)

5.2.1 Annotations

Annotations have been added, following the process described in D1.05 section 6.1 and for using the protections marked for the SFNT use case in D1.05 section 6.2.

A more detailed description of assets and protections can be found in WD6.03.

5.2.2 Adaptations to meet requirements of source-level ACTC components

In order to protect a cryptographic key using white-box cryptography, changes to the source code had to be made. We have split out encryption routines with the so called “root key” which are implementing the AES XTS mode, replacing the AES single block encryption functionality with calls to the WBC functions.

Small changes also have been made to support data obfuscations. Here, sometimes strings had been directly passed to subroutines and not be declared as explicit variables. The explicit declaration is necessary to add the according annotations.

5.3 GTO use case

Section Authors: Jerome d'Annoville (GTO)

5.3.1 Application description

This is a quick description of the GTO use case to understand this section. More details are available in ASPIRE Deliverable D1.01, Section 4 Software-based security for credentials. The purpose of application is to generate a One Time Password (OTP) that user will enter to authenticate on a web site. User is provided with a registration number and a Personal Identification Number (PIN) offline.

When the application is installed the user has to enter is registration number that is sent to the Application Server. The server returns a Device Key that is stored. This key enables to generate the OTP together with a counter that is initialized to zero. When the Device Key and the counter value are stored on the device the provisioning phase is finished. The user may now ask for an OTP. He will be asked to authenticate with his PIN and an OTP is generated if authentication succeeds.

The application is deployed with a Master Key. It is derived during the provisioning phase to retrieve a Session Key. The Device Key used for the OTP calculation is sent encrypted with this Session Key by the server.

5.3.2 Annotations

The same portion of code can be protected by different protections. This composability of the protections is studied in the reference architecture; it may have a cumulative effect or neutral to negative effect. In the table below, each line describes a protection to be applied on a portion of the code. Each line has an ID to document possible conflicts.

Protection	Asset	ID	Purpose	Location
Code splitting	PIN	cs1	Prevent attacker to tamper with comparison result	PINchecker
	Master Key	cs2	Whole Master Key value retrieval on server to avoid attacker to understand the logic. FBK proposes to put several portions of code from the same function on the server: • Part four of the Master key assignment to Master key • Final Master key processing	MasterKeyBuilder
	Master Key	cs3	Hide the retrieval of part1,2 and 3 of the Master Key	ProvisioningWrapper
	Device Key	cs4	Hide the key derivation logic	Key Derivation Function
	Master Key	cs5	Concatenation function used to prepare the argument of the derivation function	Utils

Data obfuscation	Master Key	do1	Protect a part of the Master key stored in a static variable initialized with an immediate value	MasterKeyBuilder
		do2	Protect immediate value on the call key derivation function Conflicts with cs2 protection	MasterKeyBuilder
Code Mobility	Storage Key	cm1	Hide the code that handles the data saving/restoring processing of the application	prepare_data_after_provisioning, update_stored_data, extract_stored_data, derive_storage_key in Storage
Binary Obfuscation	Session Key and code logic	bo1	All function used during the communication with the application server can take advantage of the obfuscation. Still, coding style with small functions doing atomic processing would advantage the attacker.	DeriveKeyBuilder, PayloadBuilder, CheckIntegrity, ReponseBuilder, Communication in Provisioning
Anti-debugging	Device Key Counter value	ad1	Protect the OTP calculation logic. The asset here is more the logic than the key and the counter values	Generate_otp in OtpGenerator
Call stack check	Application code isolation	csc1	The application has to be responsive then this protection will be used in the provisioning part, not in the OTP generation part.	Provisioning
Code Guards	Application code integrity	cg1	Issue that this protection should be associated with Remote Attestation to be that is not very adequate for this use-case	Provisioning
Multi-threaded crypto	Device Key	mt1	Hide the Device key used to generate the OTP	AES_cbc_encrypt in AES
Remote Attestation	Hash values	ra1	Not considered in a first step because of OTP generation part is offline while this protection needs a connection with the server. Now reconsidered for the provisioning step, to be combined with Code Guards. Asset to be protected are the hash values of the Guards, not an asset of the application	Provisioning

Client-side code splitting	Master Key	vm1	Not considered in a first step because of competition between Safenet and Gemalto. Master key retrieval process is a good candidate. Conflicts with do2, cs2.	MasterKeyBuilder
White-box cryptography			Not considered because of potential Intellectual Property issue between Nagra and Gemalto companies	

Table 2: Protections on assets

5.3.3 Adaptations to meet requirements of source-level ACTC components

The application is made of Java and C files. The Java part is the Android activity and the graphical user interface management. Three C files implement the logic of the application that is the provisioning, the PIN control and the OTP generation. These files are called by the Java part. These three files call other C files gathered in a library that provides services. To implement the cryptography and the HTTP communication this library calls existing external libraries that are curl and openssl.

In a first phase, all C files have been compiled together with ACTC into a single library. It was not a good idea because curl and openssl are large libraries with some perl code used here and there to configure the source files according to the settings. Some strange syntactic constructions generated by the pre-processor were not well handled by TXL and need to be circumvented. Conclusion of this phase is that the purpose of ACTC is to chain the various tools but it hasn't the flexibility to be easily inserted in an existing build.

A drawback of ACTC is that there are many file copies between the different steps and this clean design leads to average performance when a significant number of files have to be processed. ACTC is like a new compiler where the main objective is to generate files able to be executed and performance would be considered in an industrialisation phase.

In a second phase, the curl and openssl libraries are produced separately and only the application C code and its library are compiled by ACTC.

5.3.4 Adaptations to meet requirements of the binary-level ACTC components

Only configuration constraints required because Diablo needs the linkage map as an input together with the compiled library. Other requirement is that files have to be compiled with the debug option because these debugging data are exploited by Diablo.

5.3.5 Adaptations to the target platform

The cross compilation of the curl library is not completely supported and the size of long has to be adapted, so two distinct files curbuild-32.h and curlbuild-64 have to be provided. This is not an ASPIRE issue.

Section 6 List of Abbreviations

ACTC	ASPIRE Compiler Tool Chain
ADSS	ASPIRE Decision Support System
AES	Advanced Encryption Standard
APK	Android Application Package
ASPIRE	Advanced Software Protection: Integration, Research and Exploitation
CBC	Cipher Block Chaining
DRM	Digital Rights Management
DoW	Description of Work
ELF	Executable and Linker Format
GUI	Graphical User Interface
OTP	One Time Password
PIN	Personal Identification Number
SQL	Structured Query Language
WBC	White Box Cryptography