Advanced Software Protection:
Integration, Research and Exploitation

D5.03

ASPIRE Offline Compiler 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:	Report
Deliverable reference number:	ICT-609734 / D5.03 / 1.01
WP and tasks contributing:	WP 5 / Task 5.1
Due date:	October 2014
Actual submission date:	10 December 2014
Responsible Organization:	NAGRA
Editor:	Brecht Wyseur
Dissemination Level:	Public
Revision:	1.01

Abstract:
This report describes the activity on implementing the initial ASPIRE Compiler Tool Chain. A tool chain engine has been implemented to be invoked with fixed arguments on toy examples. With this deliverable, MS07 is achieved.

Keywords: ASPIRE Compiler Tool Chain, ACTC, Software Protection Tools

Editor

Brecht Wyseur (NAGRA)

Contributors (ordered according to beneficiary numbers)

Bjorn De Sutter (UGent)

Ronan Le Gallic (EDSI)

Patrice Angelini, 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.

Document Revision History

Version 1.0 21 Nov 2014	Original deliverable submitted to the EC.
Version 1.01 10 Dec 2014	Affiliation of Ronan Le Gallic corrected (from NAGRA to EDSI).

Executive Summary

This deliverable is the first report on the implementation activities of Task 5.1, which concerns the implementation of the ASPIRE Compiler Tool Chain (ACTC) and the integration of protection techniques produced in WP2 and WP3. The actual code of the ACTC as described in this deliverable is tagged as deliverable D5.02, and is made internally available to all partners. As described in the ASPIRE Description of Work, this first release comprises the basis of the ACTC and focuses on offline protection techniques only.

The implementation of the ACTC follows the reference architecture and APIs as presented in deliverable D5.01 ("Framework Architecture, Tool Flow, and APIs of the ASPIRE Compiler Tool Chain and Decision Support System"). The planning and the implementation was done collaboratively by Nagra and GTO. Other consortium partners were involved whenever their tools were to be integrated. To ensure their availability at the right points in time, the other partners were also involved from early on in the planning.

Before starting the actual implementation a number of requirements were specified and agreed upon regarding the definition of components of the tool chain, its file management, the flexible invocation and ordering of tools and steps, release numbering, programming languages used, ASPIRE build VM compatibility, distributed execution, and finally available third-party tools.

To develop the logic that implements the core of the ACTC and that invokes the different tool chain components developed in WP2 and WP3, we opted for Python and DoIt.

A toy example that embodies a license check has been selected to continuously validate the ACTC during the integration phase. This allowed us to validate the integration of the binary level protection techniques. The toy example has been extended to include syntax that the source-to-source tools can parse – at this phase in the project, the tools are only at a preliminary stage (e.g., no complete normalizer is yet integrated) and thus we needed to adapt the toy example to the translation rules that have been presented in deliverable D5.01.

The result of this work is an operational ACTC framework that deploys a set of basic protection techniques implemented as tools, that are integrated and validated. This includes source-level protection techniques (the white-box cryptography tool, an annotation extraction tool, Grammatech's CodeSurfer tool, and the data obfuscation tool from Tasks T2.1 and T2.2) and binary-level protection techniques (the obfuscation techniques implemented in Diablo of Task T2.4 and the SoftVM client-side code splitting of Task T2.3).

With this version of the ACTC, we clearly meet MS07 of the project of which the requirement was that small examples could pass all tools in the tool chain, without actually being protected. In the currently implemented ACTC, we already apply basic forms of the protections that, according to the project plan, only need to be integrated in MS08 at month 18. Moreover, we already apply most of them in an automated way (i.e., based on code annotations rather than manually rewriting source code). As such, we already reached some of the goals for MS11 in M24.

Contents

List of Figures

Section 1 Introduction

1.1 Purpose

This report describes the first release of the ASPIRE Compiler Tool Chain (ACTC) that has been delivered at the end of October 2014 (M12). Through this deliverable, we have accomplished the important milestone MS07 of the project. The motivation of this milestone is to expose the ACTC to all partners early enough in the course of the project to validate the design choices, detect possible blocking points or area of improvement and collect feedback.

The driving factor of this release is to deliver an initial ACTC able to process a selected toy sample. This allows us to test the major processing tools' invocation in the ACTC and their behaviour and co-operation. This initial ACTC implementation will also serve as a basis for adding additional features and allow a smooth, continuous integration of protection techniques in the remainder of the ASPIRE project.

The next major releases that are scheduled are then in M18, the ACTC with offline protections applied, and in M24 the first ACTC with online protections applied.

1.2 Outline

In this section, we present the scope of this ACTC release and how we have organized to achieve this. We had to setup the implementation activities between the different partners involved (NAGRA and GTO), and organize appropriately to implement the ACTC basis and start integrating techniques as described in deliverable D5.01. Section 2 comprises the requirements that we elicited – requirements specific to the ACTC to ensure that it can operate in a future industrial context. Section 3 presents the details of each of the steps that we aimed to integrated, and the final sections capture the installation and the validation of the ACTC respectively.

1.3 Status

As described in the ASPIRE Description of Work, the initial tool chain that is delivered at M12 is referred to as “offline compiler tool chain that will be able to generate working code, but not actually protected code”. As such, it is intended to be a framework that can invoke a standard compiler and linker, into which protection techniques can be integrated in subsequent iterations of the ACTC.

The actual implementation that has been delivered at M12 exceeded this goal. It already has integrated some protection techniques and is able to generate *somewhat* protected code. In particular, the list below presents the tools that have been delivered in WP2 and WP3 and that are actually integrated and validated in the ACTC. Each of them includes a reference to the related step as described in deliverable D5.01, which is also detailed in Section 3.3.

- WBC Annotation Extraction Tool(SLP03.01), WBC Tool(SLP03.02) and WBC Source Rewriting Tool(SLP03.05)
- Annotation Extraction tool (SLP04)
- GrammaTech CodeSurfer (SLP05.01)
- Data Obfuscation (SLP05.02)
- Standard compiler, assembler and linker (patched with UGent's patches, see D5.01)
- Native code Extractor (Diablo BLP01)
- Native code to binary code X-Translator (BLP02)
- SoftVM Integration Tool (Diablo) (BLP03)

We already announced in deliverable D5.01 that we envision being able to integrate these tools this early in the integration phase. Only the integration of the Normalizer has not been initiated (see Section 3.2).

With this version of the ACTC, we clearly meet MS07 of the project of which the requirement was that small examples could pass all tools in the tool chain, without actually being protected. In the currently implemented ACTC, we already apply basic forms of the protections that, according to the project plan, only need to be integrated in MS08 at month 18. Moreover, we already apply most of them in an automated way (i.e., based on code annotations rather than manually rewriting source code). As such, we already reached some of the goals for MS11 in M24.

1.4 Approach and timeline

The architecture and API of the ACTC have already been presented in deliverable D5.01 ("Framework Architecture, Tool Flow, and APIs of the ASPIRE Compiler Tool Chain and Decision Support System"). Based on that design, we have started the implementation phase at M9 of the project, as described in the ASPIRE Description of Work. To facilitate this work, NAGRA and GTO have setup a joint work activity with more detailed tasks and intermediate milestones. The main tasks are depicted in the Gantt chart in Figure 1.

The joint work with the NAGRA and GTO development teams started mid-September, aiming for a delivery of the initial ACTC end of October. After selecting (through a guided process of evaluation) the appropriate tools and a setup phase (e.g., to setup a joint repository and to agree on specifications that need to complement deliverable D5.01), the actual implementation work started.

The implementation work comprises the implementation of the actual basis, as described in Section 3, and the integration of *tools*. With tools, we denote processing steps that embody some software protection technique as developed in WP2 and WP3 of this project. These tools are delivered by the different partners in the project, and thus the integration comprises two steps: (1) agree with the tool owners on the APIs and delivery, and (2) the actual integration into the ACTC. The implementation of the tools itself is in the scope of the corresponding protection technique work package; WP5 comprises the integration activity.

The Validation task enabled each processing tool provider to check that tools behave as expected when integrated into the ACTC. The Delivery task contains last phase actions like final packaging and tagging.

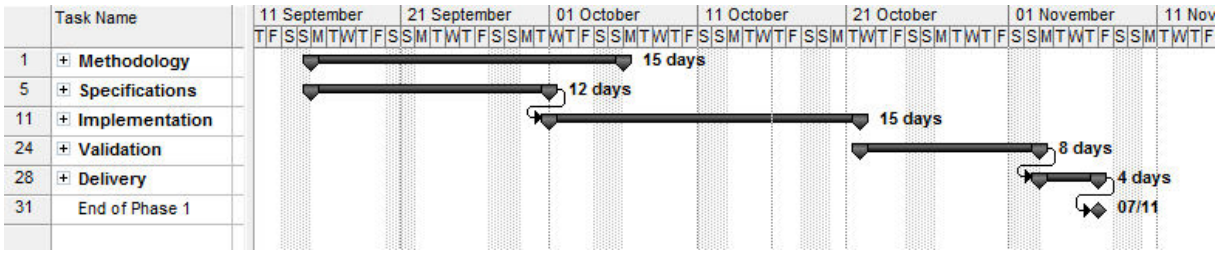

Figure 1: First ACTC iteration main tasks

1.5 References

Reader of this report may refer to deliverable D5.01 Part II that gives a step by step detailed description of the processing done in addition to input and output files at each steps.

The documentation of the well-known GCC compiler is available online at https://gcc.gnu.org/. LLVM documentation is also available at http://llvm.org/docs/

Section 2 Requirements Specification

2.1 Functional requirements

On top of the requirements already specified in deliverable D1.03, this section elicits further requirements regarding the more practical aspects of the ACTC.

Requirement: **REQ-FNC-001**	*Tool Chain* **A list of pre-defined tools can be invoked sequentially by the ACTC.**

This requirement enables the ACTC to call various tools. These can be standard tools, such as a compiler or linker, or tools developed in WP2 and WP3. The ACTC needs to invoke these tools sequentially.

Requirement: **REQ-FNC-002**	*Tool chain file management* **The ACTC needs to manage input and output folders.**

At each step during the tool chain execution, the ACTC needs to create an input and output folder. The input folder is filled with the data required for the step that will be invoked; the output folder is left empty. When control from the tool is returned to the ACTC, it retrieves the result from the output folder.

Requirement: **REQ-FNC-003**	*Tool chain step order* **The step order of the ACTC needs to be pre-defined.**

The step order is either part of the construction of the ACTC itself or maintained in a description that enables ACTC to invoke the processing tools in the expected order. Independently of the way it is described, the step order is fixed.

Requirement: **REQ-FNC-004**	*Flexible invocation* **The ACTC needs to be able to skip one or several steps.**

The ACTC must be flexible enough to skip some steps. When a step is skipped then the content of the input folder is copied into the output folder.

Requirement: **REQ-FNC-005**	*Single tool invocation* **The ACTC should be able to execute a single step.**

By disabling steps the processing can be reduced to a single step to enable testing a dedicated feature in a step.

Requirement: **REQ-FNC-006**	*Logging* **The ACTC must log the step activity.**

A log enables to track the tool invoked at each step. If the invoked processing tool aborts for some reason then some details must be reported in the log.

2.2 Design and implementation requirements

Requirement: **REQ-DES-001**	*Release number* **Each release of the ACTC must have a unique release number associated.**

Requirement: **REQ-DES-002**	*Language* **The ACTC must be implemented in Python 2.7, and use DoIt as build automation tool.**

DoIt (http://pydoit.org/) is a python package that comes from the idea of bringing the power of build-tools to execute any kind of task.

Requirement: **REQ-DES-003**	*ASPIRE VM Compatibility* **The ACTC must be functional on the build environment that has been shared between the ASPIRE consortium members (the 'ASPIRE VM').**

Requirement: **REQ-DES-004**	*Distributed execution* **The ACTC must support task parallelism.**

The ACTC must be designed to execute tasks in parallel automatically. This means that it must be feasible for the ACTC to execute independent tasks in parallel. Steps (which is the deployment of a protection tool) on a single target remains to be executed sequentially.

2.3 System requirements

Requirement: **REQ-SYS-001**	*Python* **The ACTC requires Python 2.7.**

Requirement: **REQ-SYS-002**	*DoIt* **The ACTC requires DoIt version 0.26.0 (+fix GH-#6).**

Section 3 Design Description

3.1 ACTC Description

3.1.1 High level architecture

The ACTC stands for “ASPIRE Compiler Tool Chain”, and as such comprises of combination of tools, such as source-level and binary-level transformation tools (produced in WP2 and WP3) of the ASPIRE project, and a standard-compliant compiler, assembler and linker. Additionally, the ACTC ‘chains’ each of these tools taking into account the build order and dependencies.

We refer to the ASPIRE deliverable D5.01 for more details on these tools and dependencies.

The ACTC comprises two major parts:

- A front-end that interfaces between the tools and the user invoking the ACTC, and
- The build system, which manages and organises the tools. This build system manages the whole process that turns given source code into binaries.

3.1.2 ACTC language: Python

The Python language is often viewed as a “glue” language, which is already an advantage to integrate different tools from multiple contributors. The fact is that Python is clearly powerful enough for industrial-strength software development.

Python can be characterized as object-oriented, interpreted, interactive, modular, dynamic, high-level, portable and extensible in C and C++ and is provided by default with a comprehensive and well documented library.

The alternative choice would have been Java, but we selected Python for its versatility and personal experience.

3.1.3 Build automation tool: DoIt

To improve the operation of the ACTC, we chose to use Python DoIt (http://pydoit.org/) as a building block of the ACTC. DoIt comes from the idea of bringing the power of build-tools to execute any kind of task.

A task is an abstraction for some computation that has to be done. These are actions such as the invocation of external programs that are executed as shell commands or python functions.

Alternatives for DoIt include Waf (https://code.google.com/p/waf/) and Apache Ant (http://ant.apache.org/). We performed a small analysis to select the best tool. The evaluation resulted into a clear preference for DoIt.

DoIt offers many useful features, including:

- flexible and customizable task definition;
- cache task results;
- correct execution order;
- parallel execution.

DoIt also exposes its API to create new applications/tools using *doit* functionality. Additionally, we strongly prefer this combination of Python and DoIt as it provides a more versatile and convenient solution for integrating many tools from many contributors.

3.2 Diagram

The Diagram depicted in Figure 2 presents the steps of the first release of the ACTC

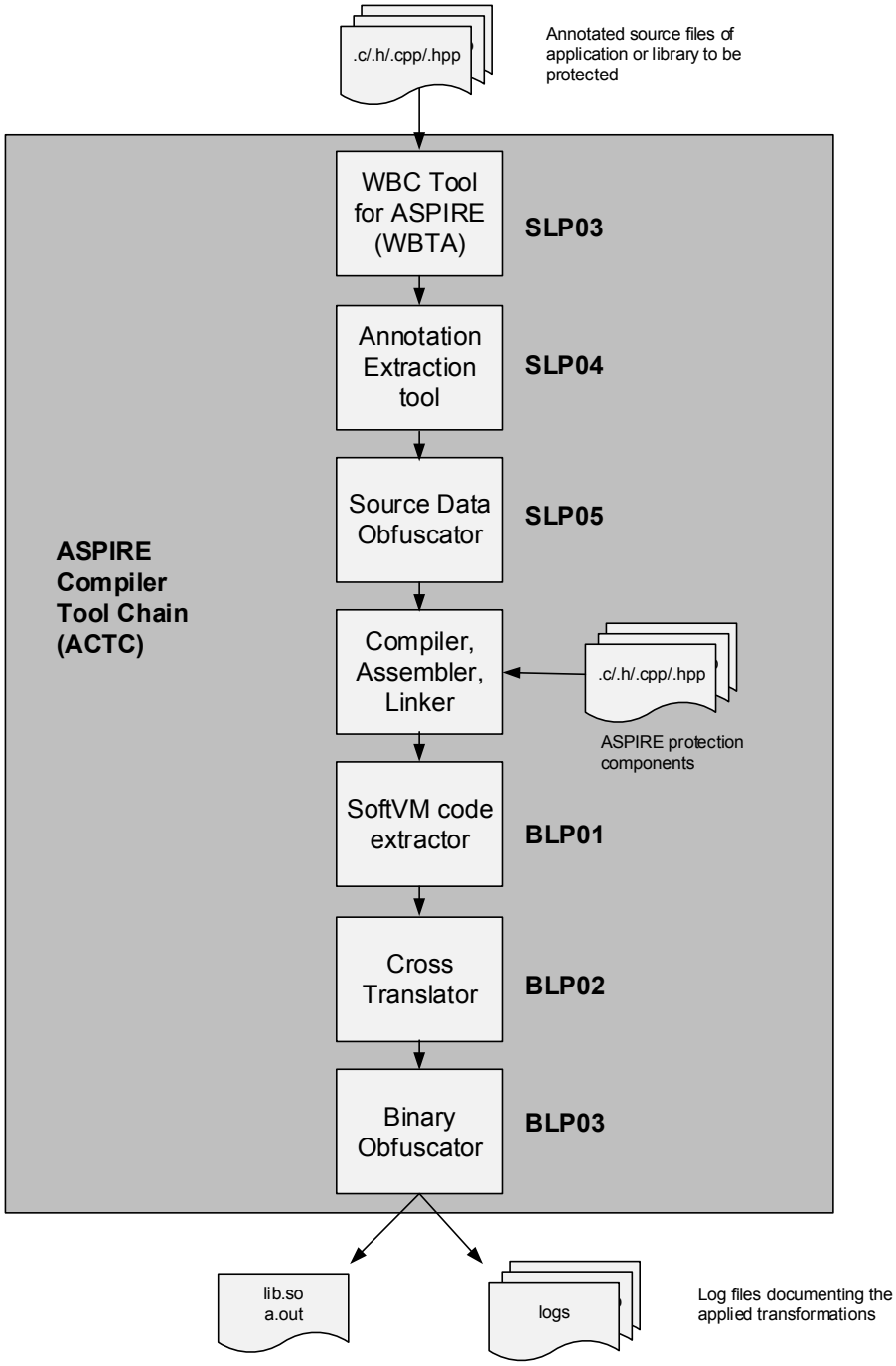

Figure 2: ACTC steps

Purpose of this first release is to validate the ACTC design choices then some shortcuts have been taken to meet the milestone in due time with a tool chain able to generate an execution file/library from annotated source files.

The Normalizer tool (Section 7.2 of deliverable D5.01) is not part of this release. As a consequence, not all patterns in the source code will be handled yet.

Another shortcut is that the SoftVM code Extractor and the Cross Translator are always called with this release of ACTC while this processing should be skipped if there are no

related annotations.

3.3 ACTC steps

The different steps of this first release of the ACTC are described in this section. The symbolic names (SLPxx.yy or BLPzz) refer to the diagrams depicted in deliverable D5.01. The detailed diagrams are copied here to facilitate fluent presentation and discussion on their integration.

3.3.1 Source level: White-Box Cryptography processing

This step involves the White-Box Tool for Aspire (WBTA) and includes the following tools described in D5.01:

- WBC Annotation Extraction Tool (SLP03.01),
- WBC Tool (SLP03.02)
- WBC Source Rewriting Tool (SLP03.05)

WBTA is in charge of generating the white-box source code. This white-box source code is generated to replace a call to a cryptographic function. The call that needs to be replaced and additional metadata are specified using the appropriate annotations as specified in D5.01.

The operations that are integrated into the ACTC are depicted in Figure 3. This replacement process will generate 4 types of files that will be integrated into the application source code:

- The white-box crypto code itself, generated by the WBTA,
- Include files,
- Application code where the call to the cryptographic function has been replaced by a call to the white-box code, and
- Two text files that list all the files generated by the white-box tool.

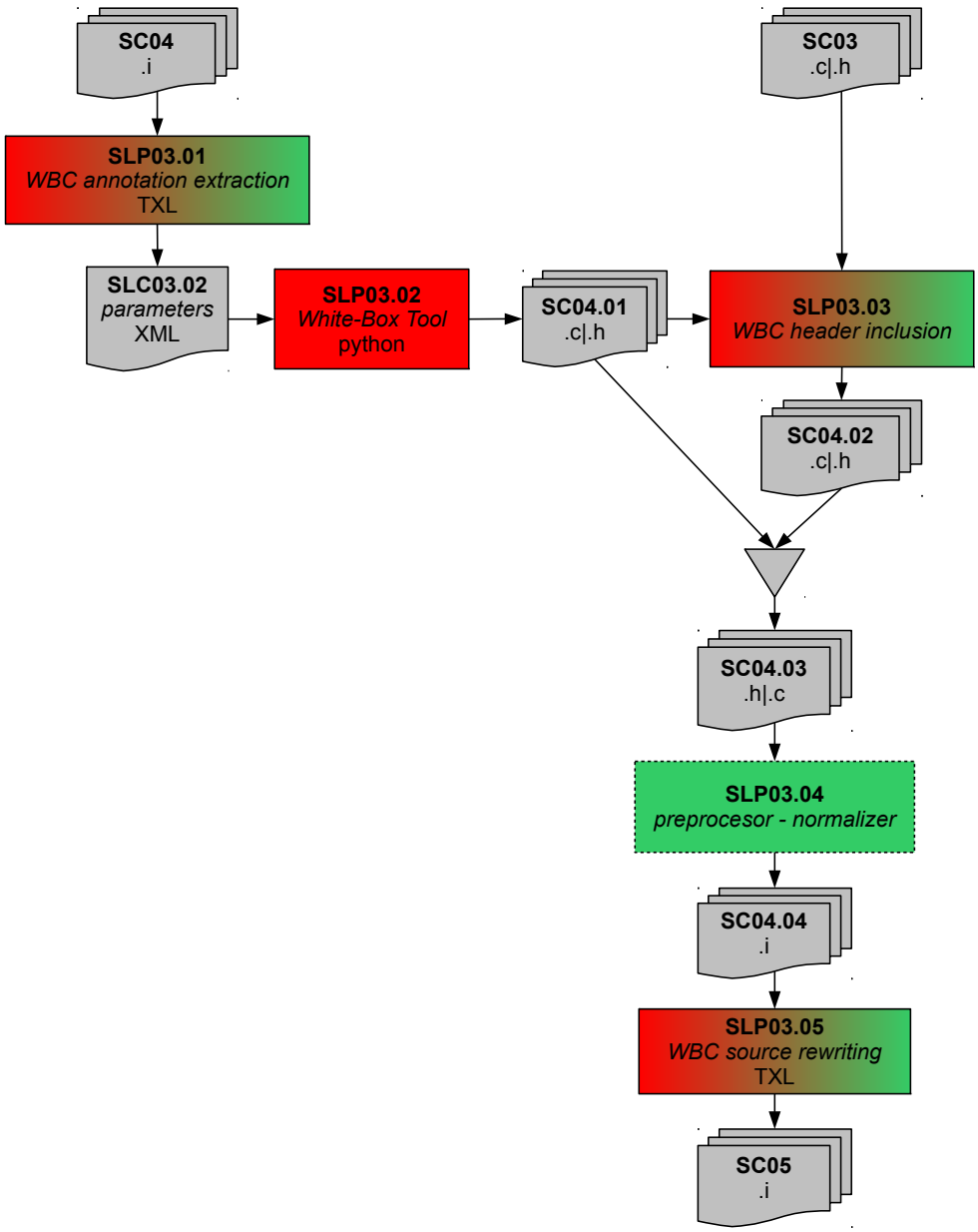

Figure 3: SLP03 detailed description

Remark that in this release of the ACTC, the WBTA generates client-side code only. In subsequent releases, the WBTA will also generate server-side code – to support dynamic white-box crypto implementations – and the ACTC needs to manage these extra files as well.

3.3.2 Source level: Annotation Extraction tool – SLP04

ASPIRE code annotations are provided through the mean of C Language pragma's. Unfortunately, pragma's do not persist during the compilation phase: compilers such as GCC will parse these pragma's and drop them and there is nothing left in the generated object files. As a consequence a specific tool shall save the annotations to enable further protection steps to have access to it.

As depicted in Figure 4, the ACTC integrates an annotation extraction operation during the source-level steps. This step parses the source code files and stores information in the form of facts in a format that is accessible to any other subsequent component of the ACTC.

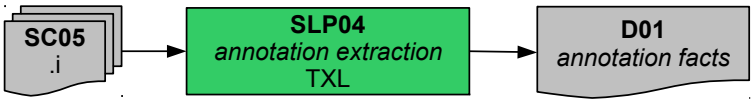

Figure 4: SLP04 detailed description

3.3.3 Source level: CodeSurfer and Source Data Obfuscation – SLP05.01&SLP05.02

The ACTC integrates obfuscation operations both at source level as at binary level. In Part I of deliverable D2.01 (“Early White-Box Cryptography and Data Obfuscation Report”), a large set of data obfuscation techniques is presented. These techniques can be implemented as source-level transformations and can be categorised in three groups: (1) storage and encoding, (2) aggregation, and (3) ordering. Currently implemented are the following transformations:

- Integer encoding (Based on residue number coding)
- Integer masking (xor based)
- Integer variable merging
- Static variable converted to procedural data

Based on annotations specified in the source code this step changes memory allocation and layout. Complex encoding/decoding operations can be applied to the data. Moreover, easy to locate static strings are converted to a procedure that produces a string as a results of a (difficult to guess) computation.

The deployment of these techniques proceeds in two operations, as presented in Figure 5.

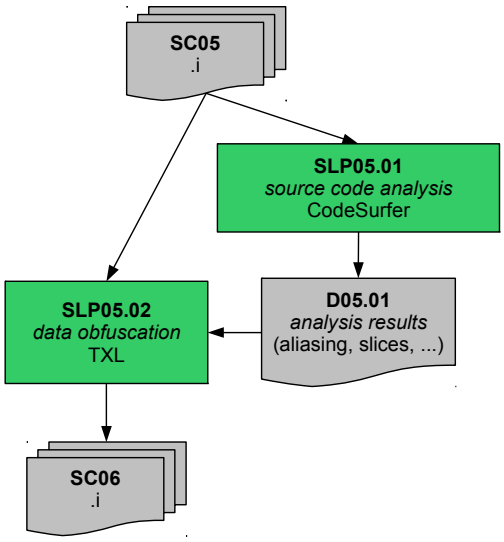

Figure 5: SLP05 detailed description

3.3.4 Compiler & linker

In between the source-level transformations and the binary-level transformations, the source code files need to be compiled into objects and linked. This proceeds with a standard compiler, assembler, and linker that are called by the ACTC. This process is depicted in Figure 6.

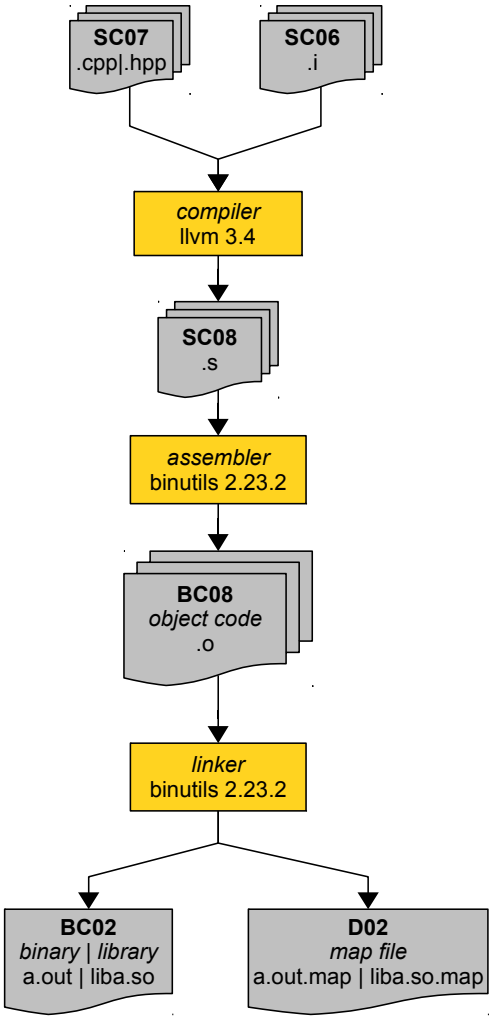

Figure 6: Compiler & linker detailed description

The default behaviour is that the ACTC invokes LLVM and binutils. Nevertheless, the ACTC can be configured to use other tools instead. The sole requirement is that the compiler meets the constraints imposed by the tools that are deployed. In particular, Diablo requires that the compiler and linker are patched. The list of required patches (for LLVM and other compilers) is presented in Deliverable 5.01.

3.3.5 Binary level: Diablo first run – BLP01

Figure 7 presents the operation where a Diablo-based extractor is used to select code fragments to be translated into bytecode that can be interpreted by the SoftVM. No binary code is rewritten yet at this step.

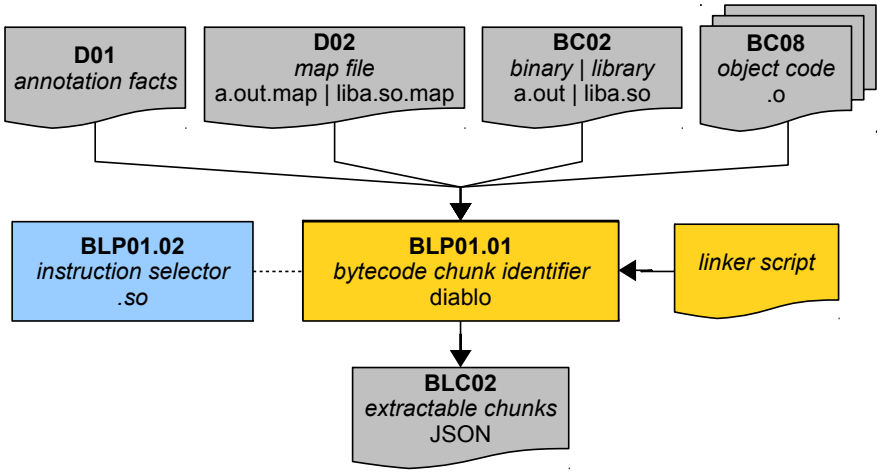

Figure 7: BLP01 detailed description

3.3.6 Binary level: Cross Translator

This step produces the bytecode from the extracted code fragments, and is depicted in Figure 8. Additionally, during this step the extracted application object code fragments are replaced by stubs. A stub is calling a common glue code function to initialize and start the SoftVM. Outputs of this step are the updated application object file, the bytecode under the form of an object file and the SoftVM in another object file.

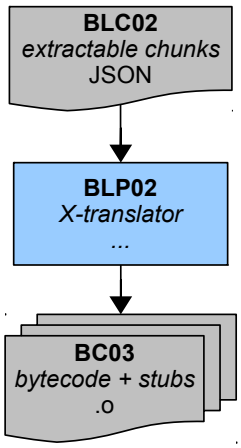

Figure 8: BLP02 detailed description

3.3.7 Binary level: Diablo second run - BLP04

Figure 9 depicts the final step of the ACTC, which is the invocation of Diablo. Based on the inputs files, Diablo generates a new .out/.so file. Processing done during this step are the following:

- Binary obfuscation: the application code is obfuscated based on annotations facts
- SoftVM and bytecode object files are linked together with the application object code

3.4 Requirements and ACTC features matching

Table 1 presents a cross check table to validate that the requirements presented in Section 2 are met with the current ACTC release.

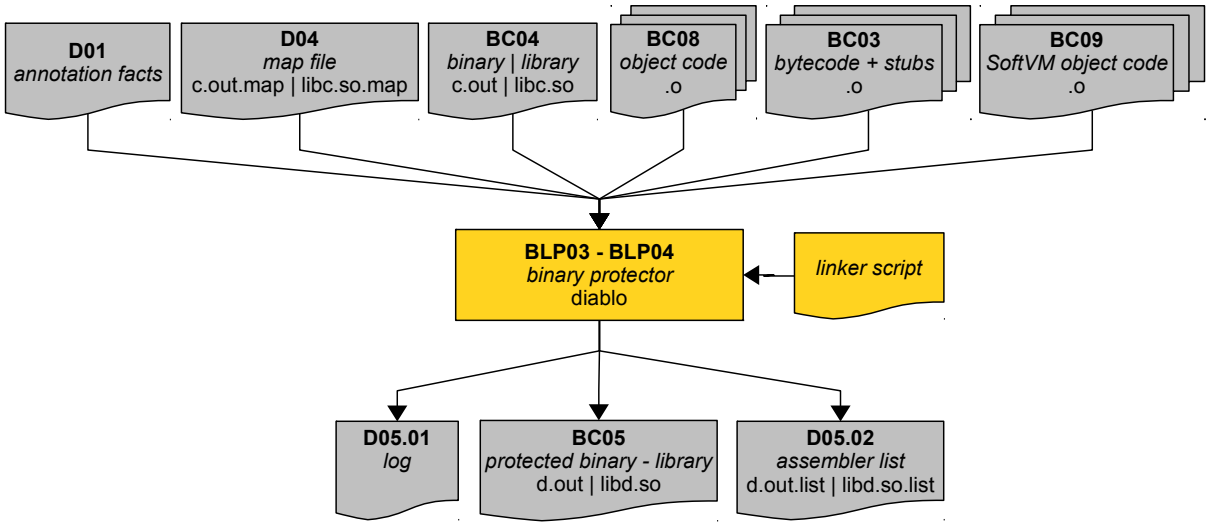

Figure 9: BLP04 detailed description

Requirement	Covered	Deviation
REQ-FNC-001	Yes	
REQ-FNC-002	Yes	
REQ-FNC-003	Yes	
REQ-FNC-004	Yes	
REQ-FNC-005	Yes	
REQ-FNC-006	Partially	Log on stdout/stderr
REQ-DES-001	Yes	
REQ-DES-002	Yes	
REQ-DES-003	Yes	
REQ-DES-004	Yes	
REQ-SYS-001	Yes	
REQ-SYS-002	Yes	

Table 1: Requirement crossref

3.5 ACTC steps integration

To support fluent integration of each of the tools that embodies some protection technique, a set of abstract classes has been defined, which can wrap the interface to each of the tools. These classes are defined in the ACTC in the file `ACTC/tools/__init__.py`.

Section 4 Installation and usage

4.1 Environment

4.1.1 Software environment and configuration

As a Python 2.7 program, ACTC requires a standard Python 2.7 environment. The only one dependency to be installed is the DoIt package (package and install script available on svn).

4.1.2 Hardware environment and configuration

No specific hardware environment.

4.2 Structure of installation

For this first release, ACTC is not package to be installed in the Python environment. Sources are available on svn with the following folder tree:

```
ACTC
├── main                          # Main development branch
│   └── src
│       ├── actc
│       │   ├── *.py
│       │   └── tools
│       │       └── *.py
│       └── actc.py
└── tags                          # Release branches
    └── X.Y.Z
        ├── ReleaseNote
        └── src
            ├── actc
            │   ├── *.py
            │   └── tools
            │       └── *.py
            └── actc.py
```

To install the ACTC tool, one can either:

- Update the PATH environment variable with `/path/to/ACTC/main/src` (for dev branch) or `/path/to/ACTC/tags/x.y.z/src` (for stable branch).
- Create symlink to `/path/to/ACTC/(main|tags/x.y.z)/src/actc.py`

4.3 Usage

4.3.1 Command line invocation

The ACTC can be invoked via a command line interface. The option `-help` can be used to present the different options that are possible, as presented below for the latest release v0.4.0.

```
$ ./actc.py --help
usage: actc.py [-h] [--version] [-f configName] [-g [configName]]
               [-u [configName]] [-j N]
               [{build,clean}]
```

```
ASPIRE Compiler Tool Chain

positional arguments:
  {build,clean}         ACTC commands [build]

optional arguments:
  -h, --help            show this help message and exit
  --version             show program's version number and exit
  -j N, --jobs N        allow 1..N jobs at once [1]

Configuration:
  -f configName, --file configName
                        read configName [aspire.json]
  -g [configName], --generate [configName]
                        generate a template configuration file  aspire.json]
  -u [configName], --update [configName]
                        update old configuration file [aspire.json]

ACTC v 0.4.0
```

4.3.2 Configuration

Upon invocation, a configuration file can be provided. This is a JSON formatted file, which defines which steps the ACTC needs to perform, and allows to provide additional parameters. An example of how such a JSON file is formatted is presented in Figure 10.

```
{"tools"   : { "<program>" : "<path>",
               "<program>" : "<path>"
               },
 "src2src" : { "excluded"  : false,
               "traverse"  : false,

               "<STEP>"    : {"excluded": false,
                              ...}
             },
 "src2bin" : { "excluded"  : false,
               "COMPILE"   : {"excluded": false,
                              "options" : []},
               "LINK"      : {"exclude" : false,
                              "options" : []}
               },
 "bin2bin" : { "excluded"  : false,
               "<STEP>"    : {"excluded": false,
                              ...},
             },
}
```

Figure 10: JSON configuration file structure

Section 5 Validation

5.1 Introduction

The validation of the ACTC comes with several tests

- Unit tests, using the Python standard unittest framework (See https://docs.python.org/2/library/unittest.html)
- Integration tests. These tests are used to validate the integration of tools into the ACTC base on toy examples.
- Quality assurance tests.

These tests ensure that the ACTC is functional, and that the individual tools have been integrated successfully. It must be noted though, that this does not validate the correct operation of the tools themselves: if they correctly apply the protection techniques they aim to embody.

5.2 ACTC Unit tests

Functions and methods are unit tested with “assert” methods to check and report failures.

This can be invoked using the `runtests.py` script, as presented below.

```
ACTC/src$ ./runtests.py

test_arg (actc.test.testcli.CliTestCase) ... ok
test_help (actc.test.testcli.CliTestCase) ... ok
test_version (actc.test.testcli.CliTestCase) ... ok
test_xyz (actc.test.testcli.CliTestCase) ... ok
test_basicPythonTool (actc.tools.test.testtools.ToolTestCase) ... ok
test_toList (actc.tools.test.testtools.ToolTestCase) ... ok
test_tool (actc.tools.test.testtools.ToolTestCase) ... ok
test_tool__repr__ (actc.tools.test.testtools.ToolTestCase) ... ok
test_basic (actc.tools.test.testutils.CopierTestCase) ... ok
test_pattern (actc.tools.test.testutils.CopierTestCase) ... ok
----------------------------------------------------------------------
Ran 10 tests in 0.165s

OK
```

5.3 ACTC Quality Assurance

To ensure the quality of the ACTC python code, two additional tests are deployed.

5.3.1 Static Analysis

The python source code is analysed with the tool PyLint, which looks for programming errors and helps to enforce a coding standard onto ACTC.

This can be invoked as follows:

```
ACTC/src$ pylint actc
```

5.3.2 Dynamic Analysis

A tool called Coverage.py is deployed for analysis of code coverage. It uses the code analysis tools and tracing hooks provided in the Python standard library to determine which

lines are executable and which have been executed.

This can be invoked as follows:

```
ACTC/src$ ./runtests.py --coverage
```

As output, an HTML file is produced that allows further analysis.

5.4 Integration tests

5.4.1 Process description

The implemented ACTC reads instructions from its command line interface. During the invocation, a configuration file is provided which defines the project specific items such as input files, the steps to be executed, and the options associated to each step.

For each step, the ACTC creates a dedicated output folder (or several of them) and subsequently calls the tool with the appropriate command line arguments.

The ACTC manages the dependencies between the steps: it manages the content between the input and output folders of subsequent steps, and ensures that steps are called in the correct order.

When an error occurs, the ACTC will stop immediately and report an error message.

5.4.2 Toy examples

The integration test comprises the compilation of toy examples with the ACTC. Two toy examples have been implemented at this phase.

- A basic license checks application.
- An improved license check application. This is the basic version, augmented with code that can be used for the white-box validation. That is, the basic version with a call to a cryptographic function, whose code needs to be replaced by white-box code that has been generated by the WBT.

Both toy examples have been made available to the ASPIRE consortium, such that each partner can execute these tests, and come with appropriate JSON configuration files. These toy examples also come with server-side code that can generate appropriate licenses that can be used as input to the license check application.

Both toy examples compile successfully with Clang + GCC.

5.4.3 Binary-level protection tests

The first integration test comprises the configuration of ACTC to run the standard compiler and Diablo as binary level protection techniques (step BLP04). This is performed on the first toy example.

This test runs successfully and produces a correct binary. The output of the run is presented below.

```
Clang + GCC --> a.out
====================
size: 739028 build/BC02/a.out
test:
  1 year ago:  20 11 2013 --> No 365
  31 days ago: 20 10 2014 --> No 31
  30 days ago: 21 10 2014 --> Yes 30
  today:       20 11 2014 --> Yes 0


Diablo --> d.out
```

```
================
size: 491912 build/BC05/d.out
test:
  1 year ago:  20 11 2013 --> No 365
  31 days ago: 20 10 2014 --> No 31
  30 days ago: 21 10 2014 --> Yes 30
  today:       20 11 2014 --> Yes 0
```

We can demonstrate that the ACTC has indeed deployed protection techniques during its compilation process, in a visual way. We do this by presenting Control Flow Graphs (CFGs) of both the unprotected binary (compiled only with the patched compiler) and the ACTC protected binary (compiled with the patched compiler and the binary-level protection techniques). The CFG pictures are generated using GraphViz, a public tool that can freely downloaded (http://www.graphviz.org/) that is able to turn .dot files into graph figures; the .dot files are generated using Diablo, which comprises a feature to generate .dot files that represent the code in the executables at any state during the code rewriting, including before and after any actual transformations have been applied.

A Control Flow Graph presents a static view on the potential execution flow of a program function, and is the main representation of a program's structure. The more complex the CFGs, usually the more difficult a program is to reverse engineer. We refer to deliverable D1.04 ("Attack model") for more details on such analysis, and to the CFG-related metrics in deliverable D4.02 (Section 5.1.3). To understand this CFG representation generated by Diablo, one has to understand that the addresses that are printed in the CFGs are addresses of the instructions in the original binary as read by Diablo. If the address presented is zero, this means that Diablo injected the instruction itself, during the transformation of the code. All other information that is presented in these graphs should be readable by those who are familiar with static code analysis or analysis tools such as IDA Pro.

In Figure 11 to Figure 13, we present the main CFG of the license toy example that has been compiled with only the patched compiler – without any software protection techniques deployed. The control flow of this binary is very clear: green edges model fall-through paths between basic blocks, black edges model direct (possibly conditional) jumps, red edges model function calls, and blue edges model function returns.

Figure 14 to Figure 18 depict the CFG of the protected binary, which is clearly different from the CFG of the unprotected binary. A closer analysis of the different CFGs proves that different protection techniques have been applied during the ACTC operation. Some parts of the code are no longer present in the CFG of the protected binary, compared to the unprotected binary, because code fragments have been translated to bytecode. Additionally, the control flow itself is all but clear now: at many places there are black edges to empty blocks, which denote jumps from inserted stubs to inserted glue code. Their instructions are not shown here but they invoke the SoftVM. For any of those stubs, it is not clear where the program will continue executing after it returns from the SoftVM as this is not encoded in direct control flow transfers in the code. This is graphically modelled by the red edges coming from the HELL node. This HELL node models an over-approximation of all nodes that might transfer control indirectly; i.e., through procedure pointers as is done to return from the SoftVM to the native code. This demonstrates that replacing a native code fragment by a bytecode fragment to be interpreted, not only protects that native code fragment from reverse engineering, it also hides the control flow between its immediate preceding code and its immediate following code.

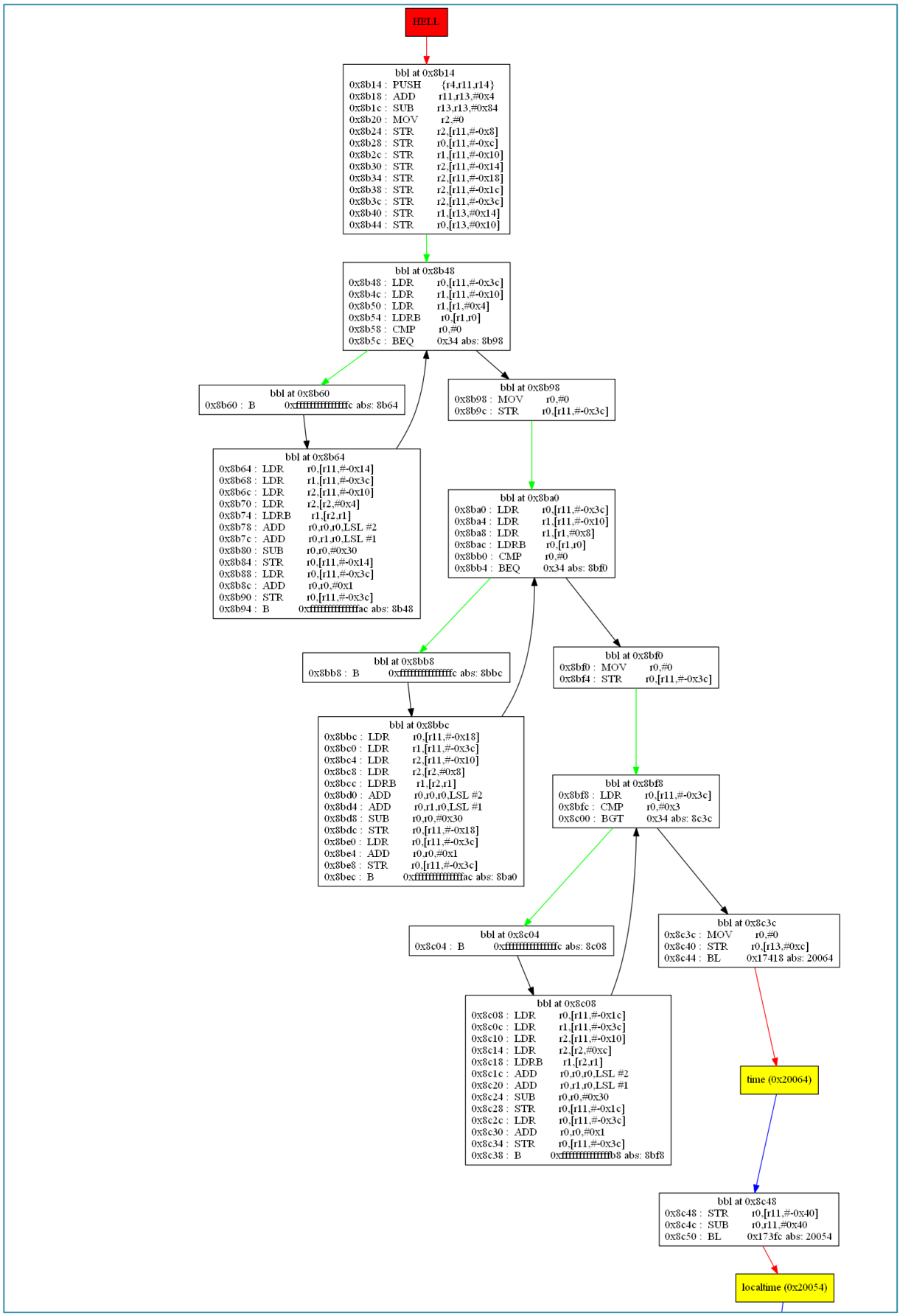

Figure 11: CFG of unprotected binary (part 1)

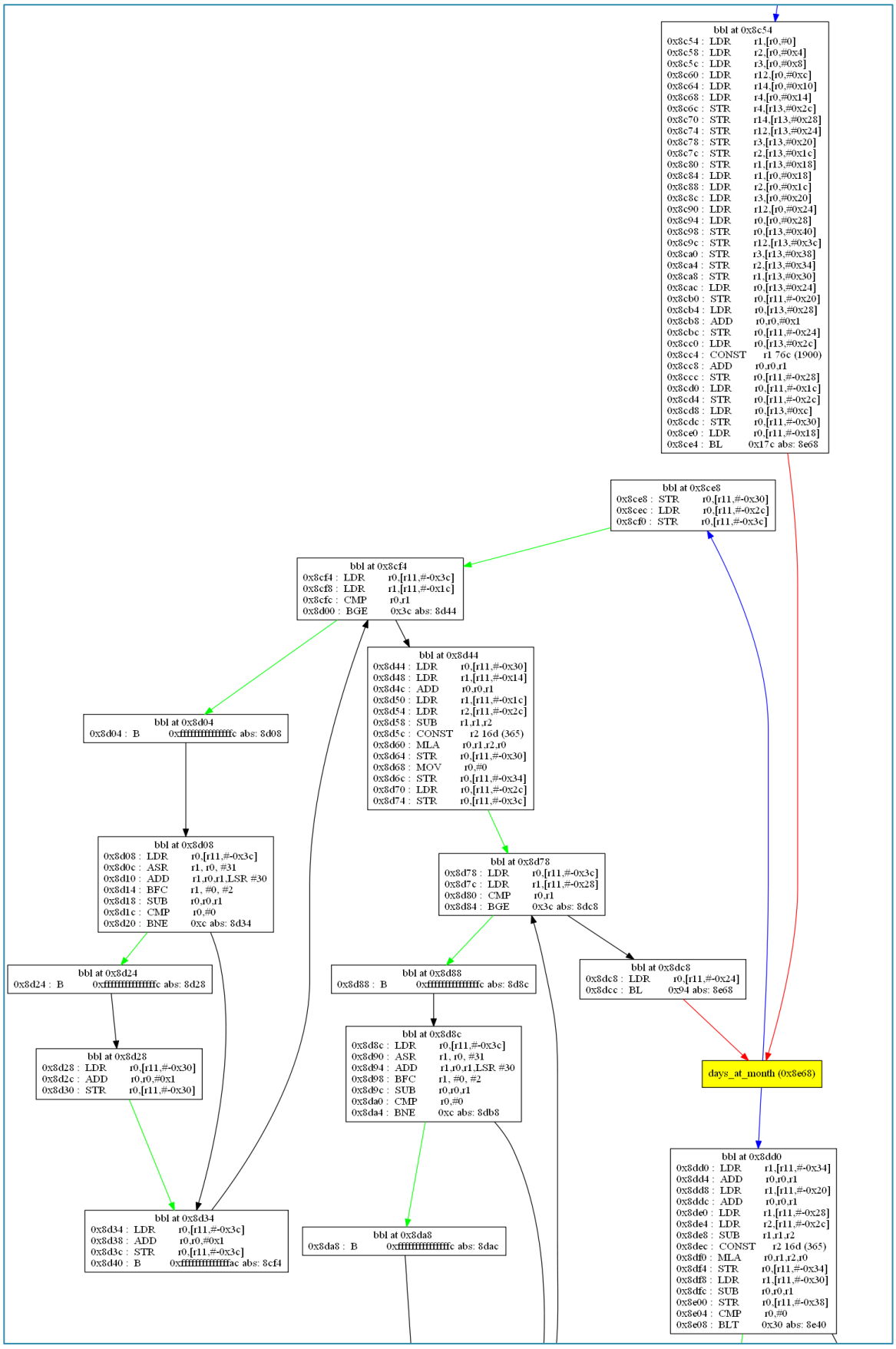

Figure 12: CFG of unprotected binary (part 2)

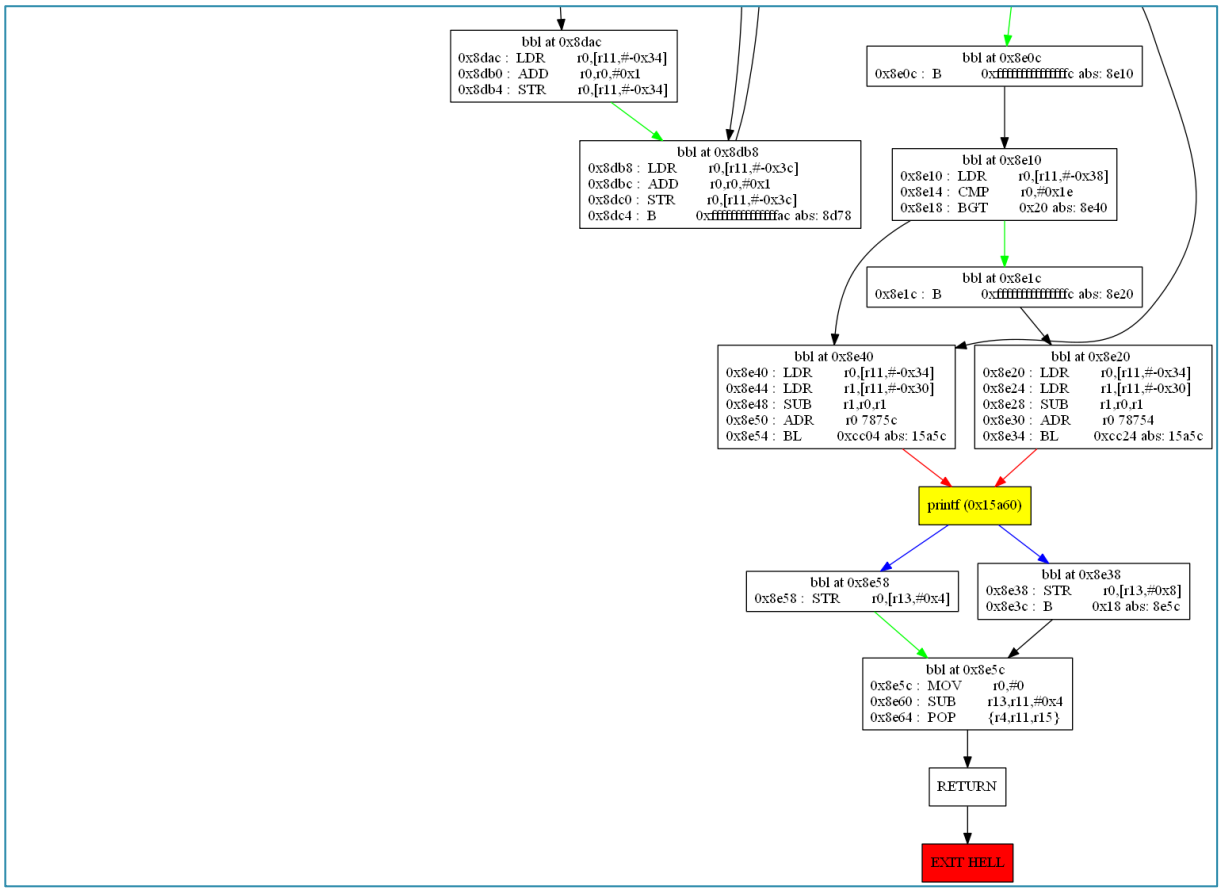

Figure 13: CFG of unprotected binary (part 3)

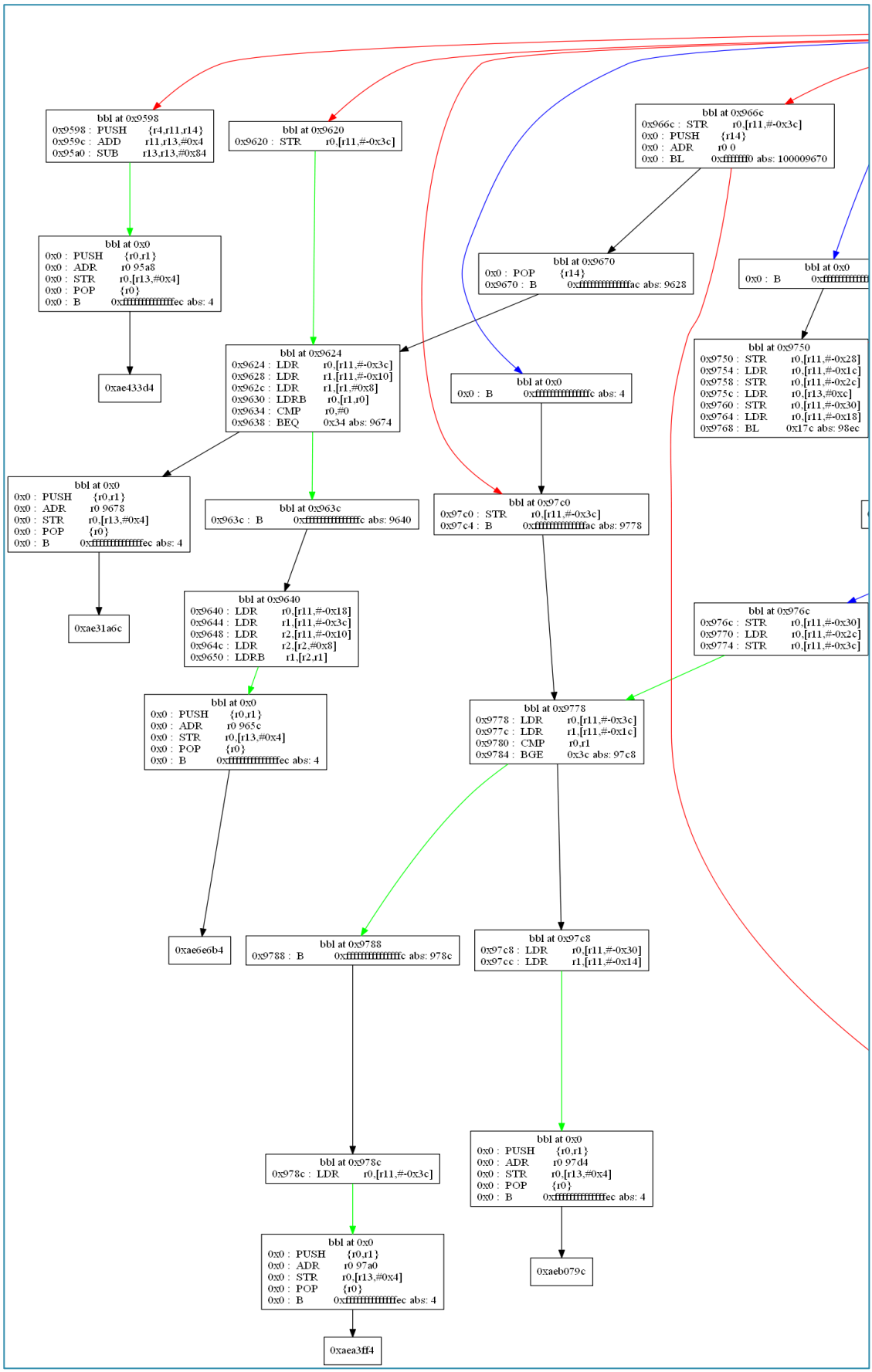

Figure 14: CFG of protected binary (part 1)

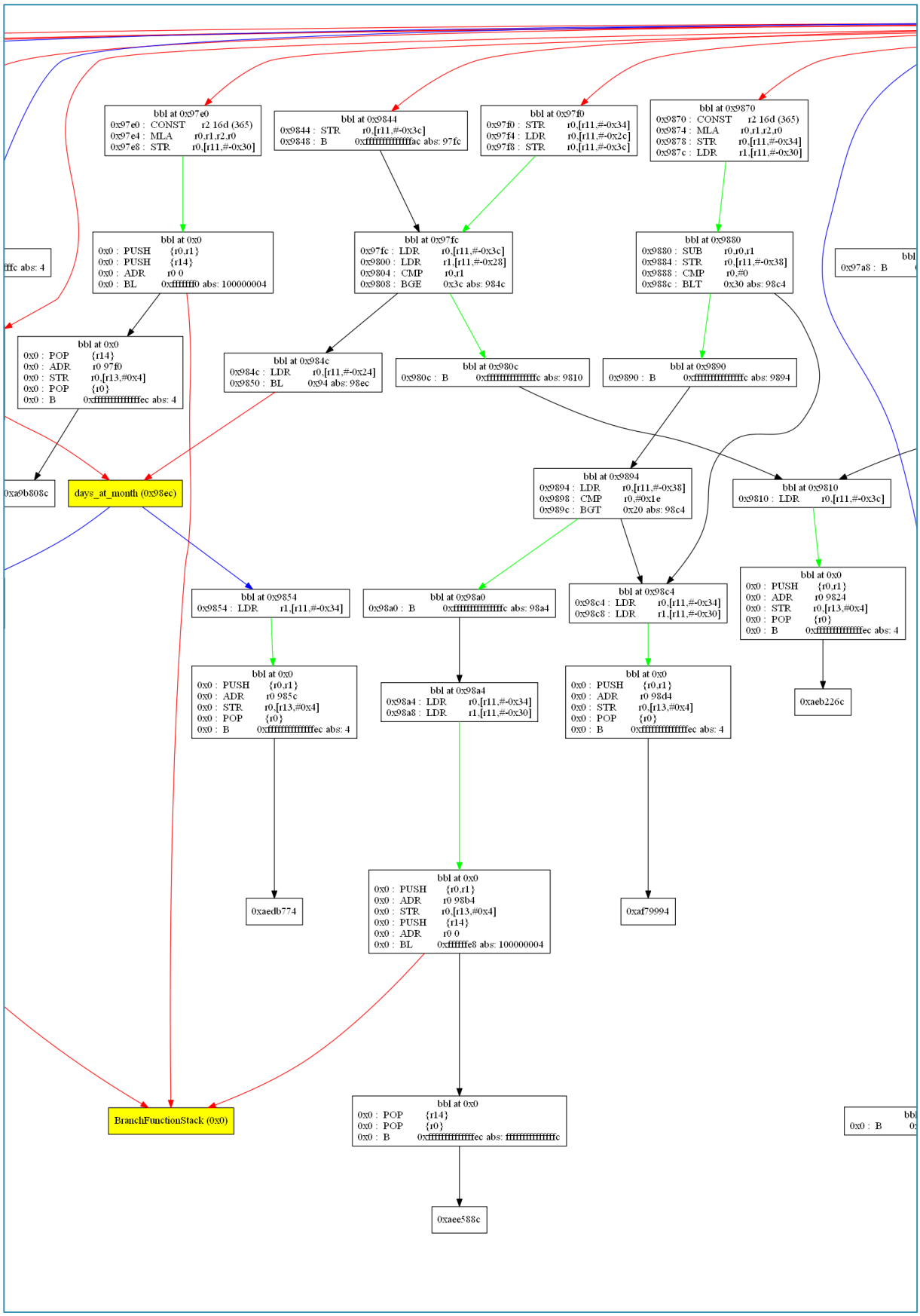

Figure 15: CFG of protected binary (part 2)

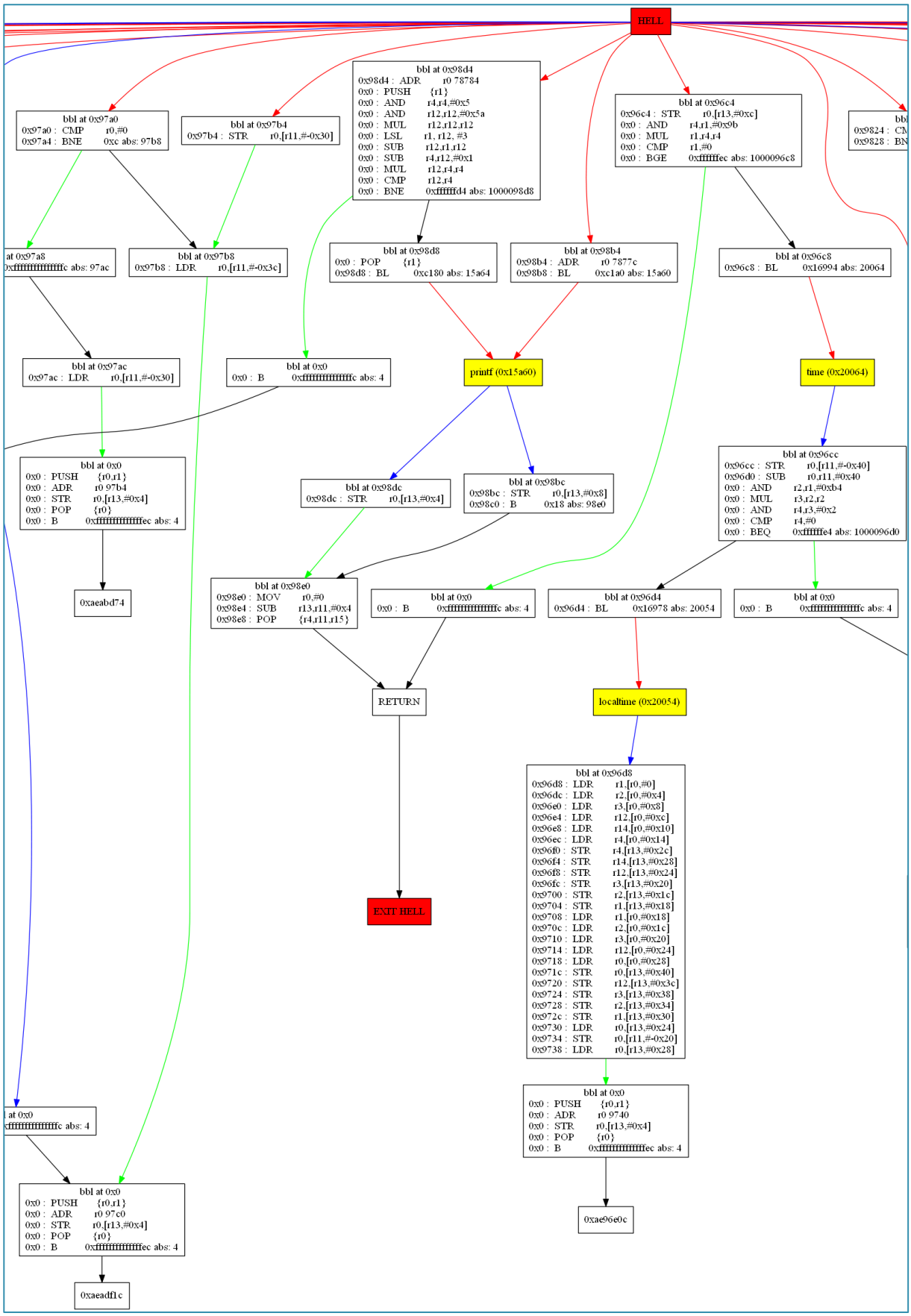

Figure 16: CFG of protected binary (part 3)

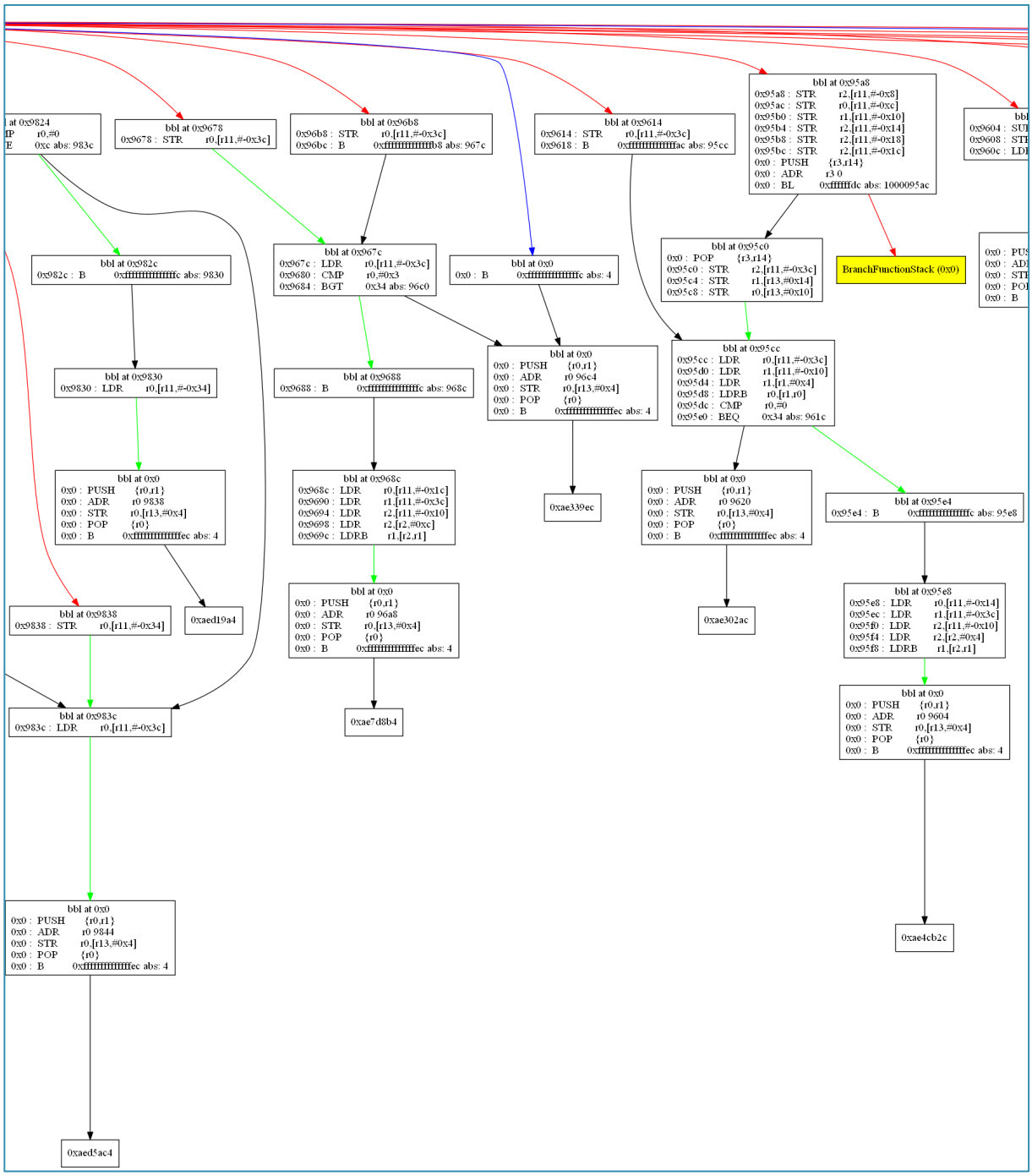

Figure 17: CFG of protected binary (part 4)

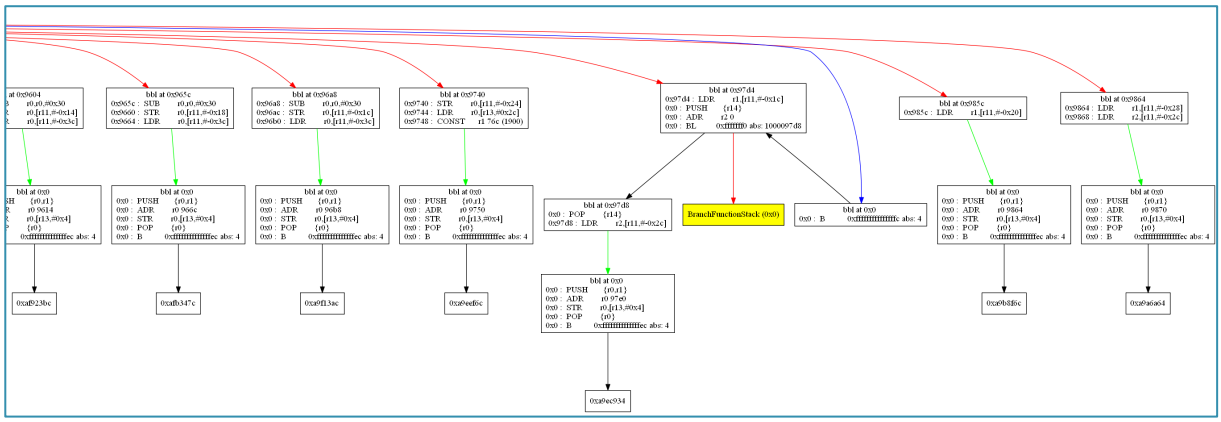

Figure 18: CFG of protected binary (part 5)

5.4.4 Source-level protection test

A second integration test comprises the configuration of the ACTC that includes source-level protection techniques.

This test runs successfully and produces a correct binary. The output of the test is presented below.

```
Server: generate licenses
=========================
test:
  1 year ago:  20 11 2013 --> 795554411244595A4110425451425447
  31 days ago: 20 10 2014 --> 795554401244595D4110425551425440
  30 days ago: 21 10 2014 --> 795454401244595D4111425551425440
  today:       20 11 2014 --> 795554411244595D4110425451425440

Client: check licenses
======================
size: 740345 build/client/BC02/a.out
test:
  1 year ago:  795554411244595A4110425451425447 --> No 365
  31 days ago: 795554401244595D4110425551425440 --> No 31
  30 days ago: 795454401244595D4111425551425440 --> Yes 30
  today:       795554411244595D4110425451425440 --> Yes 0

Client + WBC: check licenses
============================
size: 746153 build/client_wbc/BC02/a.out
test:
  1 year ago:  795554411244595A4110425451425447 --> No 365
  31 days ago: 795554401244595D4110425551425440 --> No 31
  30 days ago: 795454401244595D4111425551425440 --> Yes 30
  today:       795554411244595D4110425451425440 --> Yes 0
```

Section 6 List of Abbreviations

ASPIRE	Advanced Software Protection: Integration, Research and Exploitation
ACTC	ASPIRE Compiler Tool Chain
CFG	Control Flow Graph
GCC	GNU Compiler Collection
JSON	JavaScript Object Notation
LLVM	Low Level Virtual Machine
WBTA	White-Box Tool for ASPIRE
WBC	White-Box Cryptography
WP	Work Package