Advanced Software Protection:
Integration, Research and Exploitation

D3.08
Renewability 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 / D3.08 / 1.0
WP and tasks contributing:	WP 3 / Tasks 3.3
Due date:	July 2016 – M33
Actual submission date:	9 September 2016
Responsible Organization:	UGent
Editor:	Bart Coppens
Dissemination Level:	Public
Revision:	1.0

Abstract:
We present an update to the support for renewability, which includes support for renewable white-box crypto, and improved interactions between the remote attestation protection and mobile code. We describe our research on diversification: published experiments we performed to maximize diversification, and diversified bytecode generation for the SoftVM.

Keywords:
Renewability, white-box crypto, remote attestation, code mobility, diversification

Editor

Bart Coppens (UGent)

Contributors (ordered according to beneficiary numbers)

Bjorn De Sutter (UGent)

Brecht Wyseur (NAGRA)

Alessandro Cabutto, Paolo Falcarin (UEL)

Andreas Weber (SFNT)

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

In this Deliverable, we present an update to the work on renewability. This represents the documentation of Deliverable D3.07 of type prototype. We discuss the improvements for renewable white-box cryptography, which allows us to change the WBC tables dynamically by sending different instances as part of mobile code blocks. Furthermore, we describe how we extended the interaction between remote attestation and mobile code. When code regions in a protected application are now to be protected ‘at start-up’, and this application also contains mobile code, the delivery of the first mobile block will be postponed until the attestations of these regions have been performed. This way, the mobile code can be used as an effective anti-tampering reaction, even for short-lived applications. Previously, all mobile code could be delivered to a tampered application if the requests for mobile code where completed before the (asynchronously executed) attestations had occurred.

We also present our research into code diversification. Concretely, we present our research into maximizing the diversity of a set of applications using machine learning. We do so by comparing diversified applications pair-wise and applying different search heuristics. We also briefly present the new Background contributed to the project by SFNT with regards to the diversification of the bytecode central to the client-side code splitting protection (also called SoftVM) of WP2.

Contents

List of Figures

List of Tables

Section 1 Introduction

Chapter Authors:

Bart Coppens (UGent)

In this deliverable, we report on the progress we made since Deliverable D3.06 in M30 on the research and development related to the renewability aspects of the ASPIRE project. This document also serves as the documentation to deliverable D3.07 of type Prototype. We start with an update of the support for renewable white-box cryptography in Section 2. Next, we discuss the extended server-side support for renewability in Section 3. This includes both support for this renewable white-box cryptography, and improved interactions between remote attestation and code mobility.

The next two sections report on research we performed on diversification. First, Section 4 presents the results of research in applying machine learning to maximize diversity. This resulted in a paper that was accepted at the Symposium on Search Based Software Engineering. Finally, Section 5 briefly reports the work SFNT has contributed as background research regarding the diversification of its SoftVM bytecode.

Section 2 Renewable WBC

Chapter Authors:

Bart Coppens (UGent), Brecht Wyseur (NAGRA)

In Deliverable D3.06 in M30, we already delivered a partial implementation of renewable white-box cryptography (WBC). Since then we have improved the integration of this technique in the ACTC with the following additions.

Firstly, we extended the ACTC to generate and call scripts that compile WBC primitives on an ASPIRE server. Different invocations of these scripts will produce different instantiations of WBC primitives that have been compiled in exactly the same way as each other, but for different random seeds. This will result in binary files that differ in the data sections that store the WBC tables.

Next, we extended the Diablo component of BLP04 to produce additional meta information about the mobile blocks that have been extracted from the binary. This meta information contains information about where which data sections are placed in the mobile blocks. This information is then used to replace data chunks in the mobile code blocks that were extracted from the client-side application by the ACTC, and inject the data blocks from the renewed server-generated diversified WBC instances.

Finally, we extended the mobility server to let it serve the server-generated mobile code blocks instead of the client-extracted ones. This is discussed in more detail in Section 3.

Section 3 Server Side Support

Chapter Authors:

Alessandro Cabutto, Paolo Falcarin (UEL)

This section reports the work of Task 3.1 on Code Mobility, Task 3.3 on Renewability.

3.1 Code Mobility Server

The Code Mobility Server is the server-side component in charge of mobile blocks delivery. It's design and implementation has been reported before in D3.02 Section 3.3.3 and during the last months it has been enriched to support new features such as Mobile Code and Data Renewability and Remote Attestation.

According to SLOCCount the actual core implementation is composed by ~270 lines of ANSI C code.

3.1.1 Renewability support

D1.04 Section 6 reports different renewability strategies. The strategy we implemented is the Basic Renewability Strategy that foresees diversification in time of mobile blocks. The support to achieve diversification of blocks has been provided by UGent through a bash script that is able to extract new blocks given the object file from which to take data and original blocks path. Those new blocks are finally stored into the Code Mobility's repository.

Since the first Code Mobility Server release the mobile blocks repository access strategy evolved to support new features. Firstly the server was deployed along with blocks (i.e. in the same directory), then during the ACTC integration phase an agreement on a shared location for server-side backends repositories has been defined between online protections owners. This well-known location on the file system is bound to the ASPIRE Application ID so that for each application instance the only requested parameter to access data is the AID itself. To support renewability a further extension to this design has been made: we introduced a new level into the code mobility repository path so that the final version is in the following form

```
/opt/online_backends/AID/code_mobility/REVISION/MOBILE_BLOCK
```

where

- *AID* is the ASPIRE Application ID
- REVISION is a specific renewed version of the application instance in 'rev%08x' format (e.g. rev00000001)
- MOBILE_BLOCK represents the mobile blocks contained into the repository in 'mobile_dump_%08x ' format (e.g. mobile_dump_00000001)

The directory rev00000000 always contains 'original' blocks (i.e. the blocks generated from the first ACTC pass).

3.1.2 Remote Attestation support

The Remote Attestation protection technique owned by POLITO provides the capability of detecting certain tampering actions on a target application. Information about the status of the protected application is stored in a MySQL database shared between protection techniques server-side backends. Supported statuses are:

- *UNKNOWN*
  No information about the status is yet available

- *COMPROMISED*
  According to the Remote Attestation the application has been tampered with
- *NONE*
  No tampering actions has been detected

The initial status is UNKNOWN and as soon as an attestator collects enough information to express a verdict the value stored into the database is updated.

To strengthen the protection of provided by Code Mobility and Remote Attestation for regions marked to be attested at start-up we decided to combine the effect of those protection techniques by extending the Code Mobility Server so that it refuses to serve mobile blocks to applications marked as tampered with.

To further improve the effect of this combination a second update has been done: no mobile block will be served until the application reaches a *NONE* status. The transition from status UNKNOWN to NONE guarantees that Remote Attestation has detected no malicious action. This feature mitigates the exposure of mobile code to possibly attacked applications; in fact Code Mobility protected applications suffer dynamic analysis and memory dump attacks. If an attacker collects enough memory dumps (maybe after several runs) containing already delivered mobile blocks can understand the protection scheme and even try to patch the application to circumvent the blocks' download mechanism by injecting the dumped ones.

The combination of those two protection techniques in the presented way introduces a drawback: deadlocks. When a certain mobile block is requested to the server and it is required to complete the Remote Attestation procedure that writes the NONE status to the database then a deadlock state is met. The Code Mobility Server waits for an input from Remote Attestation and vice versa. To overcome this issue we decided to set specific constraints of composability between protection techniques; of course the first constraint states that crucial code for initial attestation procedures cannot be made mobile.

3.2 Renewability Manager

Renewability Manager is the server-side component in charge of orchestrating delivery to client applications of diversified versions of specific mobile blocks. To achieve this goal it uses renewability support from UGent (see D5.08, Section 3.1) and relies on the Code Mobility Server (Section 3.1).

3.2.1 Renewability policies

The implemented policy is time based.

Renewability policies are stored in a dedicated MySQL database called *RN_development.* This database contains information about managed applications, i.e. policy parameters and issued revisions (renewed releases). That information is persisted in the following tables:

- **rn_application** Stores information about existing applications.

Column name	**MySQL type**	**C type**	**Description**
`application_id`	VARCHAR(32)	`char[32]`	Application id and primary key
`description`	TEXT	`char[50]`	Optional application description
`inserted_at`	TIMESTMAP	`char[19]`	Automatic application insertion time reference.

- **rn_revision** Stores information about issued revisions.

Column name	MySQL type	C type	Description
id	INT(11)	long	Record id and primary key
application_id	VARCHAR(32)	char[32]	Foreign key that references the application.
number	VARCHAR(10)	char[10]	Revision number.
issued_at	TIMESTMAP	char[19]	Automatic revision insertion time reference.
apply_from	TIMESTMAP	char[19]	Start of revision's validity range.
apply_to	TIMESTMAP	char[19]	End of revision's validity range.

- **rn_application_policy** Stores information about applications' policy definition.

Column name	MySQL type	C type	Description
application_id	VARCHAR(32)	char[32]	Application Identifier.
revisions_duration	INT(11)	long	Default revisions duration is seconds.
timeout_mandatory	TINYINT(1)	bool	Defines whether revisions' timeout has to be strictly respected.

3.2.2 Renewability support tools

Some tools have been developed as API access to the renewability framework. Though those tools is possible to integrate with ACTC and easily take advantage of renewability features. Those tools are mainly implemented as bash scripts and can be used to perform basic renewability operations. Renewability support tools are intended to be used inside the ASPIRE VM but can be easily adapted to run in a standard Linux box.

3.2.2.1 Framework setup

Firstly, the renewability server-side backend has to be setup (once per VM instance): the database support needs to be initialized and some support files have to be created. An automated script (*database_setup.sh*) able to do so is provided; the same tool can be used to reset an existing installation.

3.2.2.2 Create a new application

In order to start using renewability, an entry for a new application has to be inserted into the *rn_application* table. An optional description can be provided and creation time is logged automatically. To create such an entry a *'create_new_application.sh'* script is provided; it takes the ASPIRE Application ID and a description as arguments. Application ID is unique and the insertion of an existing one will result in an error.

3.2.2.3 Create a policy

A policy defines how the Renewability Manager should behave with respect to a given application. The implemented policy is time based (i.e. a renewed version is released when a timeout expires) and for each application is possible to define some parameters:

- **Revisions Duration**
  Specifies the number of seconds after which a new diversified version should be issued and served to a client application.

- **Timeout Mandatory**
  Specifies if server-side components have to stop serving a client that does not acknowledge a revision change within a given timeout.

To define a policy, a support tool is provided (*create_new_policy.sh*). It takes the ASPIRE Application ID, the revision duration and the timeout mandatory as arguments.

3.2.2.4 Create new revision

A new revision consists of a set of diversified mobile blocks obtained by means of the *update_mobile_blocks.sh* script.

To create a new revision for a certain application a support script (*create_new_revision.sh*) is provided. It takes the ASPIRE Application ID, a revision number, a diversified source object file and the validity range as arguments.

When a new revision is requested, the original mobile blocks set is cloned into a new directory named after the revision number. This set is used to create a new diversified one and a reference to that revision is stored in the database. Depending on revisions' validity range, the Renewability Manager, will request the client to unbind its already downloaded mobile blocks. New requests for mobile blocks will result in new revision delivery.

Section 4 Experiments to maximize diversity

Chapter Authors:

Alessandro Cabutto, Paolo Falcarin (UEL), Mariano Ceccato (FBK)

This section reports the work of Task 3.3 on Software Diversity: its improvement and extension since last report (D3.04, M24), and experimental results.

The results of this work has been reported in a paper titled "Search Based Clustering for Protecting Software with Diversified Updates", that will be published in October 2016 in the 8th edition of the annual symposium dedicated to Search Based Software Engineering (SBSE2016), co-located with the 32nd Conference on Software Maintenance and Evolution (ICSME2016).

4.1 Introduction

IN D3.04 Section 7.2 we already presented the problem formulation, the case studies, the algorithms and the metrics that we to applied during our experimentation. We closed our last report with an open problem: the search space reduction (see D3.04 Section 7.2.5). We solved that problem by formulating and applying a reduction algorithm (see Filtering Twin Obfuscation section for more details). During our work we also realised that we were applying NCD metrics calculation outside its validity domain; we also found a solution to address this issue (see Validity of the Normalized Compression Distance section).

4.2 The experimental procedure

The empirical investigation is conducted according to the following experimental procedure:

- The original version of an app (as it is distributed by the apps market) is subject to all the atomic obfuscation transformations available in Zelix KlassMaster (no combinations of obfuscations);
- Twin obfuscations (i.e. obfuscations that do not introduce much similarity) are then detected and one of them is excluded for this particular app;
- The remaining atomic obfuscation transformations are applied to the app, in all the possible combinations, resulting in the versions candidate for diversified updates;
- Pairwise similarity is computed among all the pairs of these versions;
- The search heuristics (agglomerative clustering, hill climbing and genetic algorithm) are applied to compute optimal clustering based on similarity.

Agglomerative clustering is a deterministic algorithm and it requires a fixed number of fitness function evaluations that is equal to the number of versions to group into the clusters. Conversely, hill climbing and genetic algorithm are non-deterministic, so we set a search budget: in particular, they are stopped after 100.000 fitness function evaluations or when a plateau (a local optimum) is detected.

4.2.1 Filtering Twin Obfuscations

Many versions can be generated by blindly combining all the available code obfuscation transformations. However, some of these distinct transformations in the catalogue could generate programs that are not so different, so they should be detected and excluded.

Since transformations can be combined, let's call the transformations in the catalogue the atomic obfuscations. If we consider m atomic obfuscations, we can elaborate $n = 2m$ distinct combinations of atomic obfuscations to deliver n candidate versions for updates. Since the number of versions n is exponential in the number of atomic obfuscations m, we need to carefully select the m atomic obfuscation to keep, i.e. only the relevant ones.

When two atomic obfuscations are just small variations of the same transformation algorithm, or when they are two different algorithms that emit very similar obfuscated code (for example an atomic obfuscation only targeting and rewriting exception handling code may have little effect on an original application with few exception code blocks), it does not make sense to consider both of them for diversity. Including one of the two similar variants is enough, and the other can be considered redundant: we propose to apply a preliminary filtering to drop some of the m atomic obfuscations from the search space, when they are not promising as a diversifier component for the application. When two atomic obfuscations a and b are very similar to each other, we call a and b twin obfuscations.

Our approach to detect twin obfuscations and filter them out is as follows:

- We consider only the atomic obfuscations, i.e. each version is obtained by applying only an atomic obfuscation from the catalogue: in this way, we only obtain m versions;
- We compute the pairwise similarity of these m versions. Similarity values are stored in a similarity matrix of size $m \times m$. A value in the similarity matrix in the i-th row and j-th column represents the similarity between version i and version j; For each atomic obfuscation a, the a-th row in the similarity matrix represents the *signature vector* X_a. The signature vector contains the similarity values between a and all the other $m-1$ obfuscated versions. The b-th element of this vector, namely $X_a(b)$, represents the similarity between code obfuscated with a and code obfuscated with b.
- Two atomic obfuscations are twins when their signature vectors are very similar, i.e. the two transformations generate code with the same values of similarity when compared with the same alternative versions. We compute the twin value $t_{a,b}$ be- tween atomic obfuscation a and b as the square of the distance between their signa- ture vectors X_a and X_b with the sum of squared residuals:

$$t_{a,b} = \sum_{i=1..n, i\neq a, i\neq b} (X_a(i) - X_b(i))^2$$

- When all the pairwise twin values $t_{x,y}$ are available (one for each obfuscation pair (x, y)), we sort them in ascending order to detect the most likely twins;
- We exclude the twins by excluding the atomic obfuscations with lowest twin values. Let us say that $t_{a,b}$ is the smallest value among all the twin values (first value in the sorted set). At this stage, we can exclude either a or b. To decide which one to exclude, we consider the next twin value $t_{x,y}$ (in the sorted twin values in ascending order). There could be three cases:
    - $(x = a) \vee (y = a)$: we make the decision to exclude a;
    - $(x = b) \vee (y = b)$: we make the decision to exclude b;
    - $(x \neq a) \wedge (y \neq a) \wedge (x \neq b) \wedge (y \neq b)$: we make no decision at this point and we iterate. We consider the next twin value $t_{w,z}$ in the sorted list, and we compare a and b with w and z.

There are multiple strategies to decide when to stop excluding twin obfuscations. A possible strategy is to set a threshold and exclude atomic obfuscations whose twin values are below the threshold. Alternatively, we can set a target size m_{max} for the number of atomic obfuscations and stop filtering when this target is met, i.e. when $m \leq m_{max}$.

In this work, we opted for the second strategy. We set the upper limit to the number of versions mn_{max} to 500. Therefore, the number of atomic obfuscations m is approximately 9 (2^9 =

512). Eventually, the number of pairwise similarity values k to measure is 130,816, in fact the distinct pairs of n versions are $k = \frac{n(n-1)}{2}$. The number of atomic obfuscations m can be actually larger, because some combinations cause errors in the obfuscation tool or simply do not work. Thus more atomic obfuscations are required to meet the target number of versions n.

The filtering strategy is required to keep the number of versions to generate and the number similarity values to measure limited to a tractable size.

4.2.2 Validity of the Normalized Compression Distance

We adopted NCD calculation as distance metrics as described in [Cil05] and firstly we chose *gzip* as compressor but then we realized that we were using NCD outside its validity domain. This particular issue was introduced by our particular compressor choice, in fact for historical reasons (mainly for 16-bits systems compatibility) the gzip's deflate algorithm uses a very small "history buffer" (i.e. the portion of data considered to detect redundancy and apply compression) of 32kB. Therefor, according to [Ceb05], the maximum input file size using NCD with gzip as compressor is 32kB; applying it to bigger files results in distance values close to 1 (maximum distance) even when comparing very similar files. This and other common pitfalls using the NCD distance metrics are explained and benchmarked in details in [Ceb05].

Since our candidate's size is generally between 5 and 20MB we analysed the source code of gzip investigating the effort required to modify its standard implementation to increase the history buffer. This goal seemed to hard to obtain in the short term and so we decided to look for a compressor replacement; we firstly evaluated *bzip2* but its history buffer is 900kB, still not enough for our purposes. We eventually found a compression program called *rzip* [rzip] that was specifically designed by its author to overcome the history buffer issue; its actual limit is about 900MB.

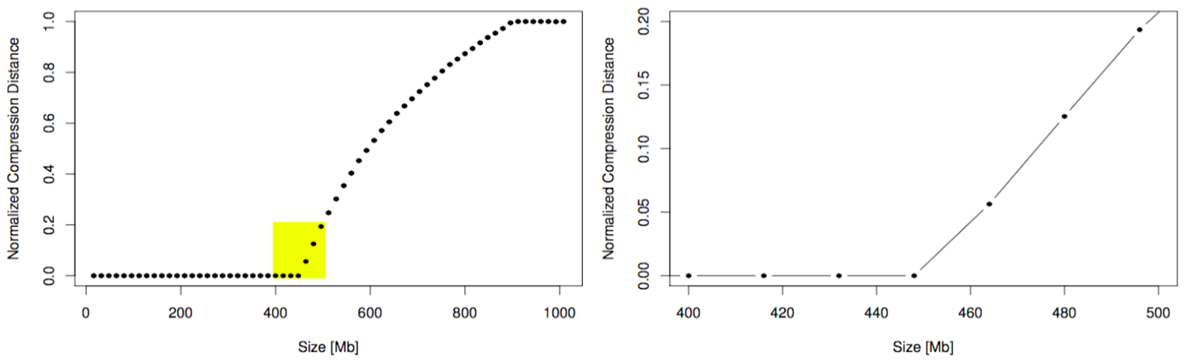

Figure 1 Interval of validity of the Normalized Compression Distance

After this we proved its validity when used as NCD compression function and so we ran a sanity check using the technique proposed by [Ceb05]: we studied the *idempotency* property of NCD based on rzip that requires $NCD(x, x) = 0$. We took a large text file and we truncated it to have a shorter file x. Then we plot $NCD(x, x)$ for increasing size of x, from 0 to 1GB with steps of 16MB. This experiment eventually showed that the idempotency property (0 distance between x and x) is satisfied when the size of files is lower than 448MB and NCD values are not reliable for larger files (as shown in Figure 1).

4.3 Experimental results

4.3.1 *Distribution of similarity*

First of all, we examine the distribution of the values of similarity. Figure 2 shows the histogram of *Similarity* for *Skype*. The histogram contains all the versions, after filtering twin obfuscations, for approximately 130,000 pairs.

As we can see, values of similarity are clustered in two groups. A first group that contains quite dissimilar pairs is centered in 0.4, ranging mostly in the interval [0.1, 0.5]. The second group contains quite similar pairs and it is centered in 0.8. Diversified updates would be selected among versions whose similarity falls in the first group.

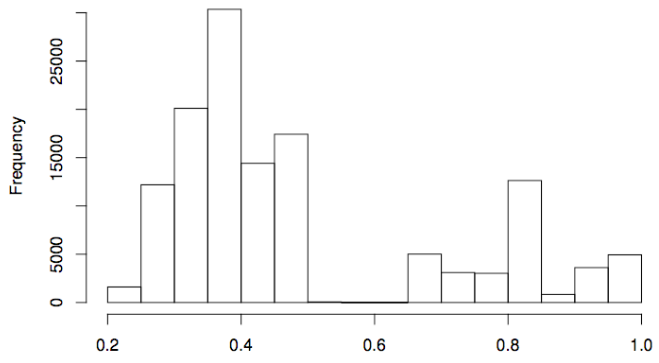

Figure 2 Histogram of Similarity for Skype

4.3.2 *Effectiveness of Filtering*

Table 1 shows which atomic obfuscations remain after applying filtering, more precisely, which atomic obfuscations are combined to diversify the code. A check mark shows when an atomic obfuscation (column) passes filtering and so it is used to generate candidate diversified versions for a case study (row). The last row summarizes on how many apps each obfuscation has been applied. As we can see, the set of obfuscations that passes filtering is quite different among different apps. Some obfuscations are applied to most of the case studies (two obfuscations are applied to all 10 apps, an obfuscation to 9 apps and four obfuscations are applied to 8 apps), while others are used less frequently (one obfuscation is applied on 2 apps and two obfuscations are applied to 3 apps).

This suggests that the filtering step is quite app dependent, because the effectiveness of atomic obfuscation transformations in diversifying the code indeed depends on the code to transform. Thus, there is no universal rule on what atomic obfuscations to adopt in general when diversifying the code. The filtering step shall be repeated for each app that we want to diversify.

It should be noted that this filtering step is fully automatic, based on the algorithm presented in Section 4.2.1.

Due to the fact that the obfuscation tool Zelix KlassMaster (that we do not control) fails to generate certain configurations, the number N of the atomic obfuscations required to reach n_{max} combinations is different for different case study apps.

Table 1 Obfuscation transformations that pass filtering

App	**Atomic obfuscations**

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*
airdroid	x		x		x	x	x		x	x	x	x		x	x	x
chrome	x	x	x		x		x				x	x	x		x	x
contacts	x	x	x	x			x	x			x	x	x		x	x
esx-explorer	x		x	x	x					x	x	x	x		x	x
facebook	x		x		x		x		x		x	x	x		x	x
gotetris	x	x	x	x	x		x	x		x			x	x		x
opera	x	x	x	x	x					x		x	x	x		x
skype	x	x	x	x	x					x	x	x	x		x	x
twitter	x	x	x	x		x	x	x	x	x	x	x	x	x	x	x
wordfriends	x			x			x			x	x				x	x
Total	10	6	9	7	7	2	7	3	3	7	8	8	8	4	8	10

4.3.3 Diversified versions

After filtering twin obfuscations, we applied the three search heuristics to the subject apps, to see how many diversified versions they are able to identify. Table 2 compares the results of the three search heuristics on the 10 apps, relevant values are highlighted in boldface. We observe negative values of similarity quality SQ when, according to the equation

$$SQ = \frac{1}{n_c}\sum_{i=1}^{n_c} A_i - \frac{1}{\frac{n_c(n_c-1)}{2}}\sum_{i,j=1}^{k} E_{i,j}$$

the inter-similarity term $E_{i,j}$ prevails on the intra-similarity term A_i.

Agglomerative Clustering was able to elaborate the most diversified versions for the majority of the cases (for 6 out of 10 apps), because the corresponding clustering configurations score the highest values of Similarity Quality. Hill climbing elaborated configurations that were always more diversified in the other four cases.

Considering the number of clusters, the Genetic Algorithm was able to identify the largest set of diversified versions in almost all the apps (9 out 10 apps). In two of them, the number of diversified versions was quite impressive (107 versions for esx-filexplorer and 96 versions for skype) however the corresponding Similarity Quality was low, but still comparable with the values obtained with the other two approaches. Hill Climbing elaborated optimal configurations with many clusters for the remaining app (i.e., opera). Eventually, the greedy algorithm elaborated large sets of diversified versions for no app.

Table 2 Results of clustering

App	**Agglomerative Clustering**		**Hill Climbing**		**Genetic Algorithm**	
	SQ	**N**	**SQ**	**N**	**SQ**	**N**

airdroid	**0.3533**	13	0.3377	24	0.2093	**35**
chrome	**0.4547**	10	0.4148	28	0.2332	**35**
contacts	**0.5431**	15	0.4786	23	0.2447	**34**
esx-explorer	0.1637	11	**0.3193**	27	0.2068	**107**
facebook	-0.5674	14	**0.0017**	17	-0.1105	**27**
gotetris	**0.3927**	12	0.3711	32	-0.0346	**34**
opera	0.2934	16	**0.3854**	26	0.2360	**41**
skype	**0.4351**	10	0.4287	32	0.2502	**96**
twitter	**0.4337**	13	0.4255	24	0.2562	**41**
wordfriends	-0.5792	12	**0.0011**	10	-0.1991	**15**
Average	0.1923	13	0.3164	24	0.1292	46

4.4 Conclusion

In this work, we tackle the problem of maximizing software diversity by searching the best subset of diversified code versions to be deployed in parallel or within an update plan. Many candidate diversified versions are generated using combinations of off-the-shelf obfuscation transformations, which can generate a huge number of possible versions; we proposed an algorithm to reduce the number of versions to generate, by discarding redundant obfuscations for the particular application code, and then we use clustering to identify the most different versions to deploy. The empirical assessment shows that our approach works in diversifying 10 popular Android apps.

4.5 Future work

As future work, we intend to investigate alternative metrics to compute similarity in a way that approximate more appropriately program difference from an attacker point of view. In particular it would be interesting to prove that a strong correlation does occur between NCD calculation over javap code (as we did) and NCD calculation over java code and even directly on bytecode. We are actually trying to produce enough experimental material to submit a paper on this topic to the 2nd International Workshop on Software Protection (SPRO2016).

Section 5 SoftVM Bytecode Diversification

Chapter Authors:

Andreas Weber (SFNT)

5.1 Introduction

The implemented VM-diversity integrates into the ACTC as of D5.08 and extends the work contributed for WP2, Task 2.3, client-side code splitting (as presented in D2.03 and D2.08) by turning its LLVM-based VM into a diversifiable design.

Protecting native code by translating it into bytecode for an embedded VM is considered an effective obfuscation technique because it confronts an attacker with code for an unknown machine. This means an attacker trying to understand the protected code is forced to first analyze the implementation of the VM and use that knowledge to build an appropriate disassembler before being able to study the semantics of the bytecode. Unfortunately this protection scheme is prone to significant learning effects on the attacker's site, as once a disassembler is available the "unknown machine" property does not apply anymore, which means analyzing the bytecode is of similar difficulty as analyzing the native code of a maybe obscure but known CPU architecture.

VM-diversity aims at mitigating this problem by not embedding a fixed VM into the protected application, but instead generates a different VM for every unique invocation of the protection tools and embeds this VM together with its specific bytecode into the protected application. Thereby a "unique invocation" is characterized by the specific ARM code that will get translated into bytecode (the specific chunks defined in the JSON) and a seed value that initializes a pseudo random number generator steering all "random" decisions in the diversity process. Mutating either parameter results in a different VM whose bytecode is not compatible with other VMs. This means different applications have different VMs, but also a single application has different VMs when protected multiple times using different seed values. The second property is especially useful because it does not only slow down an attacker analyzing the bytecode but also prevents code-lifting between differently protected copies of the same application.

[Note of the editor: As this report is public, and the specifics of the diversification scheme itself are to remain confidential, no details of that scheme are discussed here. Instead, this report only discusses how the delivered implementation was integrated with the rest of the ASPIRE protections and tools. That integration of SFNT's Background constitutes SFNT's (and UGent's) Foreground work on this topic.]

5.2 Implementation and integration

SFNT contributes its solution for VM-diversity as background. This solution had been fully integrated into the XTranslator to be easily usable by the ACTC. From the perspective of the ACTC protecting a binary with a diversified VM is not much different from protecting the binary with a pre-written VM.

In both cases Diablo passes the JSON defining the to-be-translated native code chunks to the XTranslator and receives the necessary assembly gluecode and the final size of the bytecode images. With this information Diablo finalizes the layout of the shared object's memory image (linking all additional components and applying its binary transformations) and determines the addresses of the memory objects the chunks reference. After obtaining this information Diablo triggers the XTranslator a second time to receive the final bytecode images which now embed the necessary memory offsets, so that the bytecode can interact with its enclosed process memory.

Protecting with a diversified VM slightly changes this picture as in this case XTranslator's first phase does not only generate the gluecode and the image sizes, but also the source code of a diversified VM in C. The VM's instruction set is pseudo-randomly generated from a seed value and the specific input chunks it should support. This results in a VM that is specific to its intended code (different JSONs produce different, incompatible VMs) and to the particular seed value used for VM generation (identical JSONs with different seeds produce different, incompatible VMs). The generation of the diversified VM's instructions set is completely deterministic, so that XTranslator's second phase is in fact able to create bytecode images that match the already generated VM from phase one (assuming an identical seed and identical code is passed into phase one and phase two).

Integrating VM-diversity into the ACTC boils down to executing an additional build step between XTranslator's phase one and phase two. This build step takes the generated C source code of the VM and creates the vm.a archive which is then statically linked into the protected binary by Diablo.

An XTranslator supporting VM diversification had been delivered to Ghent Aug 19th. Aside from a small integration hiccup that was addressed by an additional point release, integration went well and Ghent confirmed successful integration into the ACTC August 24th.

Section 6 List of Abbreviations

ACTC	ASPIRE Compiler Tool Chain
AID	ASPIRE Application ID
ASPIRE	Advanced Software Protection: Integration, Research and Exploitation
NCD	Normalized Compression Distance
RA	Remote Attestation
SQ	Similarity Quality
WBC	White-Box Cryptography

Bibliography

[Cil05] R. Cilibrasi, P.M.B. Vitanyi, Clustering by compression, IEEE Trans. Inform. Theory, 51:12(2005), 1523–1545.

[Ceb05] M. Cebrian, M. Alfonseca, A. Ortega, "Common pitfalls using the normalized compression distance: what to watch out for in a compressor", Communications in Information & Systems Vol. 5, No. 4, pp. 367-384, 2005

[rzip] https://en.wikipedia.org/wiki/Rzip