Advanced Software Protection:
Integration, Research and Exploitation

D3.09
ASPIRE Online Protections

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 / D3.09 / 1.0
WP and tasks contributing:	WP 3 / Tasks 3.2 - 3.3
Due date:	October 2016 – M36
Actual submission date:	9 December 2016
Responsible Organization:	UEL
Editor:	Paolo Falcarin
Dissemination Level:	Public
Revision:	1.0

Abstract:
We present an update on the renewability framework, with integrated code diversity, and a section summarizing the updates on remote attestation and its interaction with code splitting. We describe our research on diversification: an update on the metrics used in the experiments to maximize diversification, and an update on making diversification practically useful.

Keywords:
Renewability, remote attestation, code splitting, diversification

Editor

Paolo Falcarin (UEL)

Contributors (ordered according to beneficiary numbers)

Bjorn De Sutter, Bart Coppens, Bert Abrath (UGent)

Aldo Basile, Alessio Viticchiè (POLITO)

Mariano Ceccato (FBK)

Alessandro Cabutto, Paolo Falcarin (UEL)

Philippe Jutel, Paul Gunawan Hariyanto (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 deliverable presents the final status of WP3 activities and the outcomes at M36 on the online protection techniques, grouped in two sections: Remote Attestation and Renewability.

In Task 3.2 (Remote Attestation), POLITO has improved, tested and applied Static Remote Attestation to use cases. Dynamic and Implicit Remote Attestation has also been extended, improved and tested on open source software.

As foreseen in D2.10, GTO reports on the Control Flow Tagging implementation providing details about the process and the Reaction Unit, the integration with ACTC, and its role in the Remote Attestation framework.

In Task 3.3 (Renewability), the Renewability Framework has been finalized, tested and applied to NAGRA's use case: UEL reports on this topic updating information provided in previous deliverables.

UGent reports on the new design and implementation of their work on practically useful software diversity, which allows distributing different versions of software to different users, and still obtain accurate crash reports without too much overhead, bringing the distribution of diversified software one step closer to commercial viability.

Finally, UEL and FBK report progresses made on Software Diversity maximization, in particular on the validation of different similarity metrics.

Contents

List of Figures

List of Tables

Section 1 Introduction

Chapter Authors:

Alessandro Cabutto, Paolo Falcarin (UEL)

The goal of this deliverable is to document the final status and the tool support for the Remote Attestation and Renewability techniques delivered in the ASPIRE Work Package 3.

This document also reports on control flow tagging since reporting of this technique, initially foreseen for D2.10, was not done before. One reason it still fits in this deliverable on online protection techniques, is that the technique, which was foreseen to be an offline technique in the DoW, has actually been designed and implemented in two flavours, being an offline and an online variant.

The remainder of this deliverable reports the updates implemented in the Remote Attestation framework, with more details on the Static, Dynamic and Implicit implementations. Moreover, it presents the integration of Control Flow Tagging with Remote Attestation.

The renewability support integration is reported along with its validation against the NAGRA use case.

We also report here results coming from UGent research on practical useful software diversity, and from the joint effort of UEL and FBK on maximizing software diversity.

This deliverable is structured as follows. Section 2 contains the final report on both Static and Dynamic Remote Attestation and Control Flow Tagging integration with Remote Attestation. Section 3 focuses on Renewability presenting three main topics: ASPIRE Renewability Framework validation against NAGRA use case, updates on diversity in space and crash reporting research conducted by UGent, and the validation of metrics used by UEL and FBK in their research work for maximizing software diversity.

Section 2 Remote Attestation

Chapter Authors:

Cataldo Basile, Alessio Viticchié (POLITO)

This section covers the work performed in Task T3.2 for the remote attestation technique.

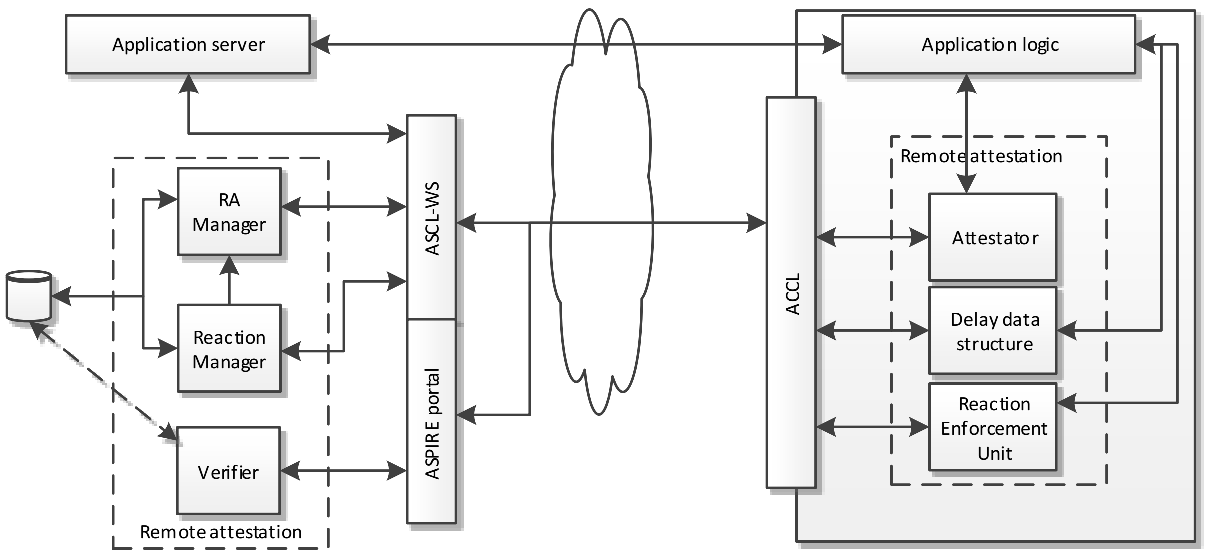
The Static Remote Attestation (Static RA), which was already at reasonable stage of development has been improved, tested and applied to use cases and other open source applications. The updates to the Static RA originated by the need for protecting the use case applications for the tiger team experiments. Together with the details reported in the sections below, effort has been spent to support the integration of the last version of the ASCL that now better supports multiple Attestators and verifiers, and to achieve a better integration into the ACTC. Moreover, Static RA has been integrated with other remote protection techniques developed in ASPIRE to enforce reactions to compromised applications. In particular, we have developed a new technique, named Reactive Attestation, which builds on the integration of Static RA and Client-Server Code Splitting and stops executing server-side code if Static RA detects a compromised application. Moreover, we have integrated Static RA with Code Mobility to stop serving code blocks to compromised applications.

The Dynamic Remote Attestation (DynRA) is the technique that monitors the correct execution of an application based on the (likely and true) invariants. Starting from the design at M24, DynRA has been improved, extended, and implemented. Moreover, DynRA has been tested on open source application, unfortunately, it has not been tested on use cases as, currently, no tools are available for ARM platforms to extract traces that are compatible with likely invariants extractors (Daikon).

Implicit Remote Attestation (IRA) has been built on DynRA. Two versions have been proposed. The Simple IRA, which has been implemented and tested on open source applications, injects verifiers that check invariants into the source code of the application to protect. Then these verifiers are split (with Client-Server Code Splitting) and executed on the server without the need of explicit Attestators. Thus, the attestation is implicit. A more complex version of the IRA has been designed that integrated DynRA with the simple IRA. With this technique, invariants are checked starting from values of variables collected partly with DynRA and partly directly taken from code executed on the server.

2.1 Remote attestation reference architecture

We report the final reference architecture of the remote attestation technique. It has not been changed since the version documented in D3.04 but we repeat it here as it is useful for quick references. Components explanation is available in the deliverable D3.02, D3.04, and D3.06.



2.2 Static Remote attestation

We shortly report here that there are no changes in the architecture and workflow of the static RA, which has been described in-depth in deliverable D3.04 and D3.06.

Only debugging and maintenance has been performed to the static RA in order to (1) apply it on the use cases and on all the other sample applications that have been used for testing and development purposes, and (2) allow a smoother integration with other ASPIRE techniques.

2.2.1 Reactive attestation

Reactive attestation is the technique we have developed to react in case of compromised applications based on the cooperation between remote attestation and client-server code splitting. The design of the reactive attestation has been anticipated in the deliverable D3.06. The reactive attestation first prepares the application and makes it dependent on the server then it applies static remote attestation to detect modification to the areas of the application binaries whose integrity must be ensured. Reaction to compromised applications happens by notifying the server which thus stops serving them.

During the last months, this technique has been implemented and tested based on the D3.06 design. In a first phase, and to avoid to infringe Intellectual properties of industrial partners and being able to publish results on this technique, RA and Code Splitting have been applied in isolation (i.e., as two independent techniques, not through the ACTC and without the ASCL/ACCL infrastructure). Moreover, reactive attestation has been applied on sample open source applications we have download from the Internet. Indeed, the technique has been presented in a paper that disseminates ASPIRE results [raadrst]. Therefore, we avoid the repetition here of the design and the technical details that can be found in the paper that has been attached to this deliverable for convenience.

However, in the last period we have successful tested the application of Reactive Attestation when applied by means of the ACTC and on the use cases.

As an exploitation activity, we have started improving the analysis phase, which aims at determining the best parts to split to have (1) guarantee of enforcement of a disconnect policy in a (2) limited and reasonable overhead. The analysis phase is now very simple and not automated, in the sense that several analyses are performed, mainly statically, then the actual areas to split are identified by human beings (i.e., by ASPIRE experts) by manually applying Code Splitting annotations.

Therefore, we need to analyse dynamic information, that is, traces, and complete the definition of the optimization models that estimate the performance degradation of splitting a specific area, given the constraint that areas are candidate to be split if they are executed at least with a certain threshold frequency. Given the difficulties to have valid traces on ARM infrastructure we have started this work on a Linux x86 infrastructure. The purpose is to publish a journal paper that extends the previously published results.

2.2.2 Integration with Code Mobility

Detective functions of RA have been also used in cooperation with the Code Mobility technique. Since, clients depend on the Code Mobility server to work correctly, integrity violations detected by the RA can be punished by stopping sending code mobility blocks, thus forcing the stop of the application as soon as it needs a new block from the server.

The cooperation between Code Mobility and RA has been applied on the use cases. Actually, it has been applied on all the three use cases for the tiger team experiments.

2.2.3 Improvement of the attest at startup functionality

In D3.06 (Section 3.2.1.2) we introduced the attest at startup functionality. Areas that are annotated to be protected with the RA with the `attest_at_startup` option set to `true`, are attested as soon as the application is launched. Correct support of attestation at startup has required the implementation of a stateful mechanism to declare application valid only after all the attest at startup areas of all the attestators have been successfully attested.

When the application is launched and the ASCL-WS initialization method is invoked by the client application:

1. the RA Manager change the client in the `UNKNOWN` state (these values are in the `name` field of the `ra_reaction_status` table);
2. the RA Manager sends the attestation requests for all the attest at startup areas;
3. the Reaction Logic verifies that all the responses to the sent attestation requests are valid;
4. the currently implemented Reaction Policy marks as `COMPROMISED` the application if any of the attestation responses fails, `CORRECT` if all the attestation responses are valid. The reason for this strong policy is because, since RA was coupled with Code Mobility, we wanted to avoid to provide valid blocks to applications that are known as compromised since the beginning and prevent to supply mobile blocks unless the client is proved to be untampered. However, other policies have been considered, like sending additional attestation requests for the attest at startup areas. However, they have not been implemented.

2.3 Dynamic Remote Attestation

The purpose of the Dynamic Remote Attestation (DynRA) techniques is to detect violations of the integrity of the actually executed application code. They represent a big step forward compared to the static techniques, which only base their verification on data that do not depend on the executed code (like hashing the binaries as done by our static remote attestation).

In ASPIRE, after a careful analysis of the available techniques, we have decided to implement the DynRA that detects violations by exploiting invariants monitoring.

Invariants are predicates built on variable values. They can be explicitly defined by the application developers, these are the true invariants, or deduced after analyzing the application to protect, in this case they are named likely invariants.

```
for( int i = 0; i < 100; i++ ){
      for( int j = 0; j < i; j++ ){
           ...
```

```
        }
}
Invariants:
i<100, j<i, j<100
```

Since the number of true invariants is usually limited, that is, insufficient for the purpose of integrity monitoring, our technique requires the identification of the likely invariants. While techniques have been proposed to extract invariants from static analysis whose effectiveness is debatable, we have focused on dynamic techniques and selected Daikon as the likely invariants extractor candidate tool. Daikon has a major limitation (at least for the C language), it is not able to infer invariants about inner scope variables that is, it does not consider the variables used within functions or any other local scope (e.g., conditional or loop statements) that are not passed to another function. In addition, Daikon is not able to detect inter-function invariants, as it is only able to extract invariants regarding the same function scope.

Verifying that invariants are satisfied consists in taking the values of the variables that are part of the invariants, and finally evaluating the invariants' expression.

Therefore, it is important to be able to extract the value of the variables from the target application depending on the reached execution point. Consequently, being able to analyse the function call stack is also crucial, as it allows understanding whether a variable value is available or not. The last information that is fundamental to extract variables' values is the functions lifecycle, in terms of instruction pointer intervals. In fact, starting from the call stack, the stack frame of each called function can be extracted together with the instruction pointer value of each called function so that it is possible to evaluate if a variable is available in memory. With all these data, we are able to:

1. stop the execution of the target application;
2. unwind the call stack and, for each stack frame, read the instruction pointer and deduce the function that is associated to it;
3. for each function stack frame, deduce the extractable variables;
4. collect variables values.

Variables need to be univocally identified in order to allow the Verifier to recognise the variables received from the Attestator.

DWARF information is used to predict the runtime information, that is, variables locations and lifecycle, functions location and lifecycle.

Finally, there is another important aspect of the invariants to consider when designing a RA based on invariants: invariants are related to values of variables extracted in the same execution, that is, theoretically, it is not possible to evaluate an invariant by using the values of the involved variables that are collected in different moments.

2.3.1 DynRA formats

2.3.1.1 Attestation request

An attestation request is a message that contains just a random nonce. No other data are sent to the client. To avoid repetitions and replay attacks, nonces are 256 bit long.

We have considered alternative approaches, like asking for an explicit set of variables but we have discarded them. Indeed, apart from global variables, all the other types of variables are available in memory only when a specific part of the code is executed (e.g., variables allocated into the stack). This may create two types of problem. First, the probability to find the requested variables in memory was estimated as very low. Therefore, since the client (thus an attacker) is legitimate to answer that a variable is not in memory, it may be allowed to answer that even none of requested variables were available in memory during the attestation response preparation to all the attestation requests.

2.3.1.2 Attestation response

The attestation response contains the values of all the variables that are available in memory at the moment of the attestation request. These data are sent as a sequence of *n* pairs (where *n* is the total number of pairs):

$P_i = (\mathrm{ID}(var_i), \mathrm{Value}(var_i))$

With these pairs, the attestator computes (|| is the concatenation operator):

$data = n \,||\, P_1 \,||\, \ldots \,||\, P_n$

$\mathrm{attestationResponse} = data \,||\, H(data||nonce)$

where H is a hash function of choice. For diversification purposes, we can use different hash functions including SHA1, SHA256, and Blake2.

Again, for diversification purposes, we have also implemented Attestators that send the keyed digest and the HMAC of *data* that use the nonce as the key.

2.3.2 Components

The RA Manager, the Attestator and the Verifier are independent of the target application. As depicted before, the RA Manager is in charge of generating a random nonce and send it to the Attestator, whenever it deems. The Attestator collects all the available variables' values and sends the attestation response to the verifier. The only application dependent part of the DynRA technique is the data structure that describes the variables and functions lifecycles.

2.3.2.1 Attestator

The Attestator collects the value of all the variables and prepares the attestation response.

The current implementation of the DynRA simply sends all the available variables. This simplified approach introduces a risk: attestation responses may too big in case an application has too many global variables (which are always accessible thus always sent by the Attestator). The risk is limited, however, we planned the implementation of a smarter version of the Attestator that only sends the significant variables or a subset of the global variables (and also knows when invariants depend on the time when values are collected). Since this risk was not manifesting itself on the use cases, we have marked this issue at low priority.

2.3.2.2 Verifier

The Verifier receives an attestation response. It reads the pairs (*VarID, value*), verifies the hash then it looks for the invariants that can be evaluated. In this first simple implementation of the Attestator, the Verifier queries the DB in order to determine all the invariants that involve at least one of the variables found in the response. The only invariants that are actually checked are those that involve only variables from the last attestation response and for which all the involved variables are available. Optimized data structures to perform quick invariants verifications have been considered outside the scope of this prototype.

2.3.2.3 Security Analysis

The proposed attestation request/response protocol is vulnerable to spoofing attacks, given that it includes values of variables stored at the client-side, thus not directly verifiable, it is formally impossible to create a communication protocol that is not vulnerable to forged attestation response. Solutions that involve secure hardware could be considered but they are outside the scope of the ASPIRE project. It is possible to block Man-in-the-middle attacks if attestation requests and replies are transmitted within a secure communication channel (for instance, the ASCL-WS with TLS enabled). However, this technique is still vulnerable from Man-at-the-End attacks. Indeed, if an attacker is able to spot the location where the extracted variables' data are stored before the hash is computed, he can replace the data arbitrarily thus going unnoticed. Therefore, the Attestator should be reinforced with the use of protections that mitigate the risks associated to static and dynamic analysis.

2.3.3 DynRA annotations

There are two types of DynRA annotations.

One type of annotation is used to manually specify (true) invariants. These annotations include a `Pragma` directive that is used to specify the invariant formula that uses variables labels, `__attribute__`annotations used to label the actual variables used in the invariants.

```
int f(int max) {
      int i,sum = 0;
      int y
      __attribute__((ASPIRE("protection(remote_attestation,dynamic_ra
      _variable(y))")));

_Pragma("ASPIRE begin protection(remote_attestation,
                                 dynamic_ra_invariant(x+y<100))");
      for(i=0; i<max; i++){
            y=2*max;
            sum+=y;
      }
      _Pragma("ASPIRE end");
      return sum;
}

int main(){
      int x _attribute__((ASPIRE("protection(remote_attestation,
      dynamic_ra_variable(x))")));
      x = 33;
      printf("Sum=%d",f(x));
      return 0;
}
```

The second type of annotations is used to declare the code areas were the likely invariant discovery tool must work to automatically discover likely invariants. These code areas are the ones whose integrity must be protected.

```
int f(int max) {
      int i,sum = 0;
      int y;
      _Pragma("ASPIRE begin protection(remote_attestation,
                                       dynamic_ra_autodiscovery)");
      for(i=0; i<max; i++){
            y=2*max;
            sum+=y;
      }
      _Pragma("ASPIRE end");
      return sum;
}
```

2.3.4 DynRA application: workflow

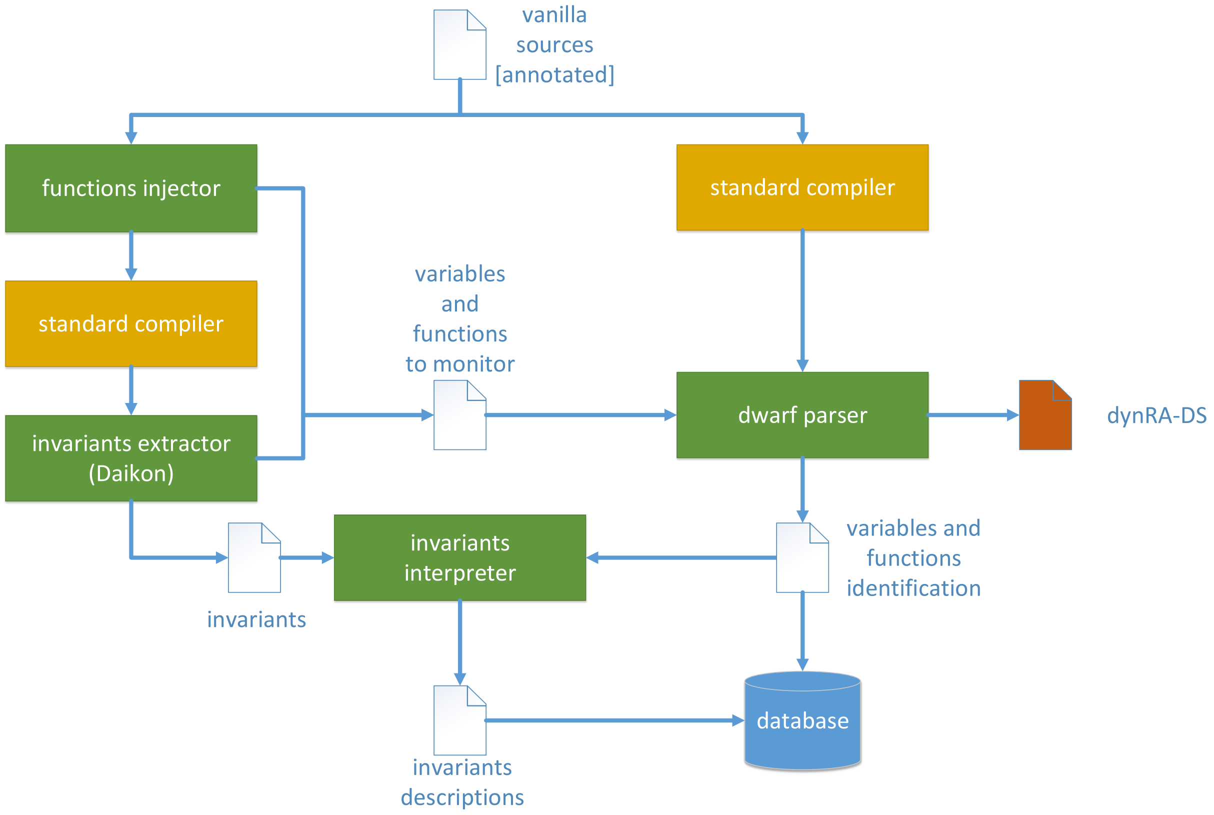


Figure 1: The workflow

The application of DynRA (see Figure 1) starts, as usual in ASPIRE, with an unprotected application whose assets have been explicitly annotated with annotations compatible with the ones in Section 2.3.3).

The application source code is taken as input by two processes:

- discovery of likely invariants;
- extraction of DWARF information.

In order to identify likely invariants, in a first step we tried to overcome the Daikon limitations by adding, for each inner local scope, a call to an ad hoc generated function with an empty body that takes as input all the variables declared in the scope and returns without doing anything. In this way, Daikon can access all the variables and determine more invariants. The injected source files are processed with the standard compiler with the debugging options enabled (i.e., `-g -gdwarf-2`) and optimisation options disabled (i.e., `-O0`) in order to generate a binary that can be analysed by an instrumenter (to extract traces) and then by Daikon.

This first process produces two outcomes:

- the list of the invariants detected by Daikon, and
- the list of the variables whose values need to be retrieved at runtime.

The second process also starts from the unprotected application sources. The application is compiled with the standard compiler with the options `-g -gdwarf-2`. Starting from the obtained binary, a parser analyses the DWARF symbols contained in it and:

1. collects all the data about interesting functions and variables lifecycles;
2. assigns a unique ID to all the monitored variables (a progressive integer);

3. produces a binary file, namely DynRA-DS, which contains the information about functions and variables that are useful for the Attestator to know how to retrieve variables values at runtime. The DynRA-DS will be injected in the final protected binary.

Finally, an invariant interpreter processes the output of Daikon and stores invariants in a format that is suitable for verification. These invariants are stored in the ASPIRE DB and are formatted so that the verification phase is facilitated. All the variable names in each invariant are replaced with their unique ID. In this way, the Verifier only has to replace each variable's ID with the received value and to evaluate the resulting expression.

A closer look at the workflow and components actually used by DynRA (presented in Figure 2) can help understanding the actual state of development of the DynRA tool.

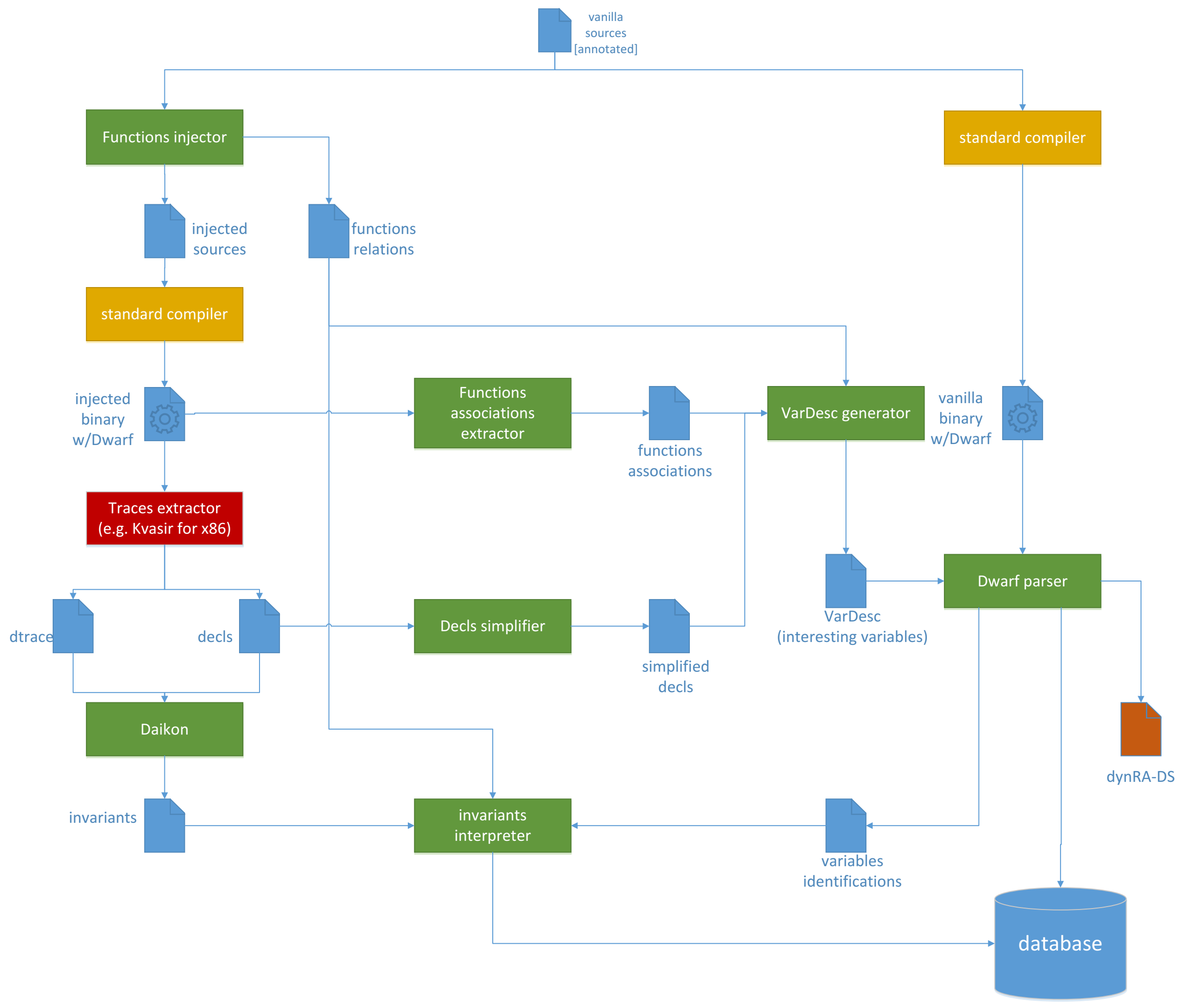


Figure 2: Detailed workflow.

The components that implement the workflow are:

Compiler. It is the standard compiler used to generate executable binaries.

Function injector. This component analyses the source code of the application to protect. For each inner scope found it lists all the declared variables and generates ad hoc functions that takes the variables as parameters.

```
double a(int a, int b){
```

```
                  int c; double d;
                  c=a+b;
                  d=(double)a/c;
                  return d;
            }
```

The injector recognizes `c` and `d` as declared variables and generates the function:

```
            void _____injectedFunction_rand (int c, double d){}
```

It injects a call to the generated function at the end of the scope where `c` and `d` are declared. At the end of the process the resulting code is:

```
            void _____injectedFunction_rand(int c, double d){}

            double f(int a, int b){
                  int c; double d;
                  c=a+b;
                  d=(double)a/c;
                  _____injectedFunction_rand (c,d);
                  return d;
            }
```

where rand is actually a random string that makes the function unique.

In addition, this component outputs a description file in which it reports, for each function in the original program, all the injected functions. This output allows further steps in the workflow to keep track of the injected functions, i.e., it allows going back to the original function knowing the injected function.

This component is fully implemented and tested.

Traces extractor. This component executes the application with injected functions, which has been previously compiled with debugging information, in order to collect execution traces. We recall that the traces must be compatible with the Invariants extractor, in this case Daikon, it is not just needed to have a tool that extracts traces. For x86 architectures, the Daikon instrumenter, namely Kvasir, is works on the majority of the applications. In the ARM environment, neither we were able to find a tool similar to Kvasir nor we had enough resources to implement such a tool from scratch. In addition to traces, this component will output also a declarations file (as Kvasir is able to do). The declarations file is useful because it contains only the declaration for those variables that are actually interesting for invariants evaluation; all other variables in the program are not reported.

Invariants extractor. This component properly calls Daikon that guesses the invariants from the input traces. It outputs the list of the invariants.

This component is fully implemented and tested.

Function associations extractor. This component lists all the functions found in the binary build after the functions injection phase. It associates each function name to its compilation unit path name. This component is necessary to avoid problems tied to duplicated filenames inside the folder structure of the original sources.

This component is fully implemented and tested.

Decls simplifier. This component processes the declarations file generated by the Traces Extractor and simply selects the name of the variables that the technique must be able to extract from the protected application to evaluate the deduced invariants.

This component is fully implemented and tested.

VarDesc generator. This component elaborates the data from the function injector, the function associations extractor and the *decls* simplifier in order to produce a file that specifies which are the data to extract from the binary built from the unprotected vanilla sources. In particular, it will output a formatted file that specifies for each compilation unit (i.e. original source file) the function and the variables for which the locations and lifecycles must be extracted for runtime evaluations.

This component is fully implemented and tested.

Dwarf parser. This component processes the DWARF symbols inside the binary built from built from the unprotected vanilla sources and extracts the data that are actually needed to collect variables' values at runtime. That is, it actually produces the DynRA-DS that will be injected in the protected application. In addition, the dwarf parser inserts all the variables in the database and uniquely identifies them. Each variable is stored in the database along with the data about its owner function and compilation unit. The variables identifications are also output as file for next steps.

This component is fully implemented and tested.

Invariants interpreter. This component elaborates the invariants extracted in previous steps and rewrite them according to the variables identifications. In practice, it replaces the variables names with the respective unique ID so that each invariant expression is directly associated to the involved variables. The obtained expressions are stored in the database of the verification phase.

This component is fully implemented and tested.

2.3.5 Overcoming limitations of the trace extractor on ARM

Currently, the DynRA has been tested on sample applications taken from open source repositories. Together with the bzip2 example already used also for

However, DynRA has not been yet tested on use cases. Indeed, the limitations of the available traces extractors working on ARM platforms have blocked the testing on the use cases applications.

To overcome this limitation, we tried to compute the invariants by collecting traces on a Linux x86 architecture. Theoretically, the invariants are independent on the platform on which the application is executed. However, invariants are extracted from the traces collected on a compiled application. Therefore, there may be differences.

However, we had problems with the NAGRA and GTO use cases, which are strongly dependent on the Android framework and cannot be compiled for a Linux x86 platform. Indeed, they rely on Android specific functions and libraries (the DRM framework and UI classes), thus, they cannot be support due to lack of traces extractor limitations. We have thus focused on the SFNT, which is also working perfectly in x86 environments. However, for some reason, kvasir fails to extract traces. We have asked support to Kvasir developers but we were not able to give them enough information to reproduce the bug[1]. Indeed, the use case is confidential thus we did not provide them the source code.

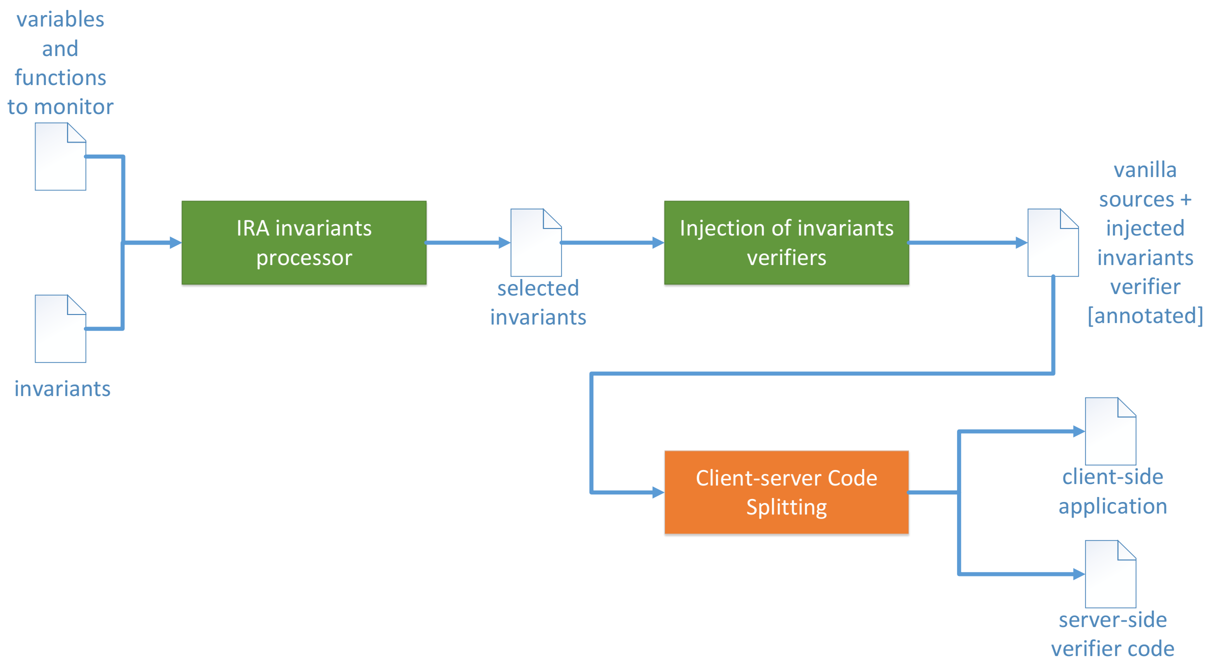
2.4 Implicit Remote Attestation

The Implicit Remote Attestation (IRA) we have implemented builds on the DynRA. The idea behind this Implicit Remote is that, if the Verifier knows the part that the client is actually executing, it has much more possibility to avoid fake attestation replies and get the actual variable value. Indeed, with this additional information the Verifier can guess with a reasonable precision what are the variables that the client should be able to collect at the moment of an attestation request.

Therefore, we use the Client-Server code splitting to interleave the execution between client and server.

[1] https://github.com/codespecs/daikon/issues/79

We have considered two forms of IRA. In the simplest form, which we have named Simple IRA, entire pieces of code are split so that they are executed on the server and the values of the variables composing invariants are available to be verified at server side. A more advanced form, of IRA has been designed, named hybrid dynamic IRA, where parts of the application functions are split and executed on the server so that invariants are evaluated based on variables' values sent by the Attestator and, at the same time, values that are available at the server.


variables and functions to monitor
invariants
IRA invariants processor
selected invariants
Injection of invariants verifiers
vanilla sources + injected invariants verifier [annotated]
Client-server Code Splitting
client-side application
server-side verifier code

2.4.1 Simple IRA

The instrumentation of the code is performed as follows

IRA invariants processor. This component reads the invariants and the actual functions to monitor and selects the invariants to be monitored. Since Client-server Code Splitting introduces an overhead (network latency, computation at the server, risk of server overload) that is not negligible, the process that selects the invariants must decide based on dynamic information. While an optimization process would be suggested based on traces information, during the project we have focused on selecting a subset of invariants with process driven by humans.

```
void f(int a){
    int i, j;

    for(i = 0; i < a; i++ ){
      for( int j = 0; j < i; j++ ){
          ...
      }
  }
  return;
}
```

Invariants examples:

0<i<a, 0<j<a, j<i.

Injection of invariants verifiers. This component creates ad hoc functions that are inserted in the application source code for all the invariants to be protected. The functions include the code to verify the invariants (indeed it is a custom invariant-specific micro verifier) and the reaction logic that interrupts the execution of the application whenever an invariant is not satisfied. Other reaction policies are possible, but since the reaction status must be maintained at server side to be effective, the actual integration would have required too much engineering.

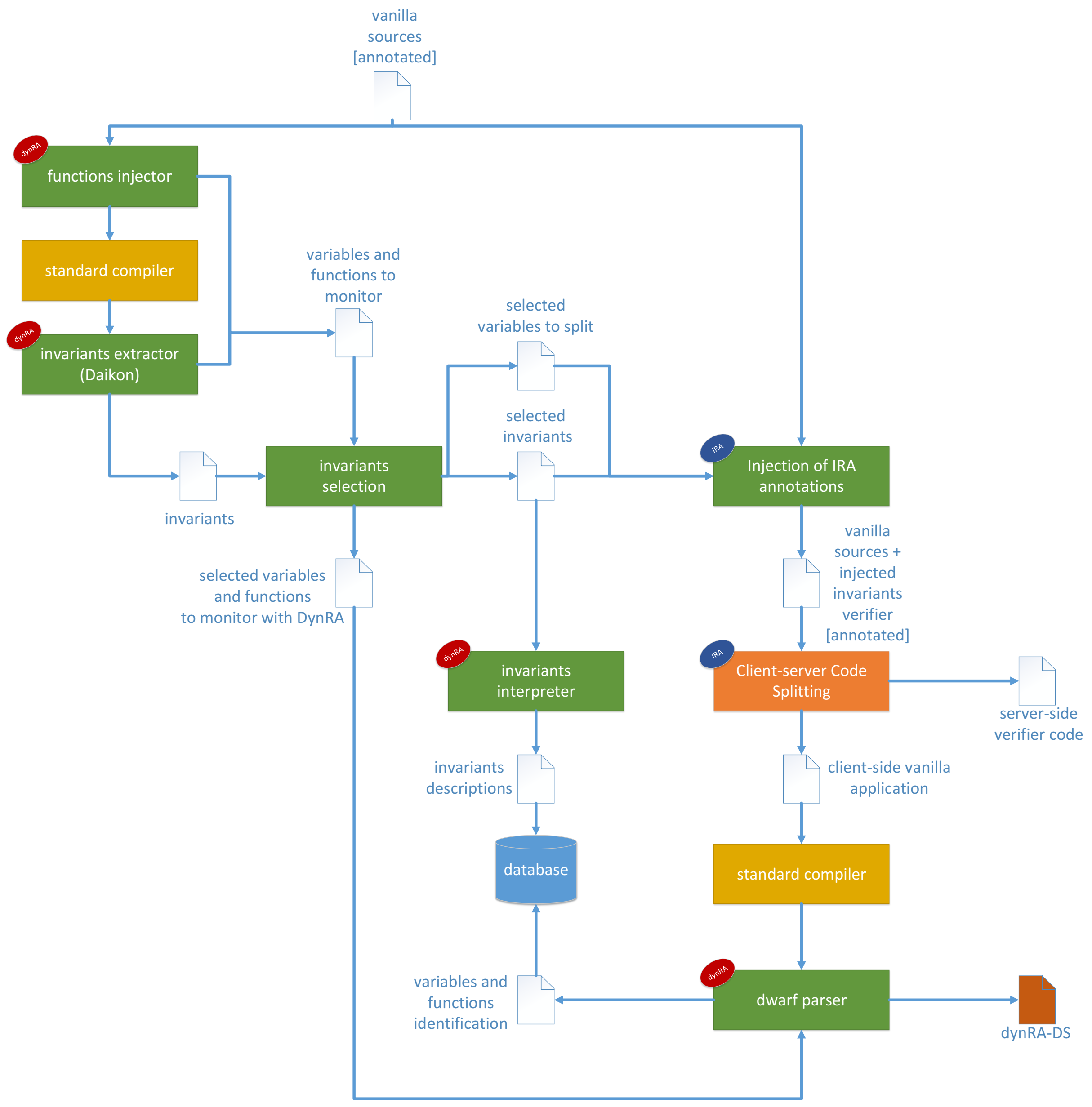
```
void _____injectedVerifier_rand(int i, int j, int a){
      _Pragma("ASPIRE begin protection (
                  barrier_slicing,
                  criterion(i,j),
                  label(slicing1))")

      if(i>0 && i<a && j>0 && j<a && j<i)

            return;

      else

            interrupt_execution();

      _Pragma("ASPIRE end")

}


void f(int a){
       _Pragma("ASPIRE begin protection (
                  barrier_slicing,
                  barrier(i, j),
                  label(slicing1))")
      int i, j;
      _Pragma("ASPIRE end")

        for(i = 0; i < a; i++ ){
        for(j = 0; j < i; j++ ){
            ...
            _____injectedVerifier_rand(i,j,a);
        }
        _____injectedVerifier_rand(i,j,a);
        }
     _____injectedVerifier_rand(i,j,a);
  return;
}
```

Code Splitting. This component executes the ASPIRE Client-Server Code Splitting protection that coherently removes the previously inserted functions based on the annotations and automatically adds all the network communication logic.

Note that this architecture of the implicit remote attestation does not require neither an explicit Attestator at client side nor a Verifier at server side.

The following approach presents the following limitations. First, it includes in the code the reaction, thus it is not allowed to change the reaction type after the protection has been deployed without a rebuilding of the server. In the other RA techniques, the reaction is decoupled from the detection to achieve this flexibility. Then, another limitation is on the number and types of invariants to monitor. That is, it is not only important to consider how effective may be monitoring an invariant, it is also important to estimate the overhead generated by the splitting.

2.4.2 Hybrid Dynamic IRA


vanilla sources [annotated]
dynRA
functions injector
standard compiler
dynRA
invariants extractor (Daikon)
variables and functions to monitor
selected variables to split
selected invariants
invariants
invariants selection
IRA
Injection of IRA annotations
vanilla sources + injected invariants verifier [annotated]
selected variables and functions to monitor with DynRA
dynRA
invariants interpreter
IRA
Client-server Code Splitting
server-side verifier code
invariants descriptions
client-side vanilla application
database
standard compiler
variables and functions identification
dynRA
dwarf parser
dynRA-DS

When the Hybrid Dynamic IRA needs to be deployed, in the first phase we borrow steps and components from the Dynamic RA to have the list of invariants and the related functions and variables to monitor.

Functions Injector. As presented in Section 2.3.4, in this phase ad hoc functions are inserted in the vanilla application to identify inner scope invariants thus overcoming likely invariants limitations.

Invariants extractor (Daikon). As presented in Section 2.3.4, Daikon is used as tool to infer likely invariants.

Then another phase is performed to select the invariants that will be monitored and to decide if variables' values will be collected with a DynRA-like attestation of directly read with server-side execution.

Invariants selection. This component reads the invariants and the related functions and selects the invariants to be monitored. Once the invariants have been selected, it is also needed the selection of the variables that will be observed on the server and the ones that will be acquired by injecting the dynamic RA Attestator. There are two important aspects to consider. As anticipated before, Client-server Code Splitting introduces an overhead that is not negligible, therefore, invariants to monitor and variables to move on the server must be carefully decided based on dynamic information. Then, the Attestator of the DynRA is not able to retrieve all the variables that are needed at a given point as the computation can be moved to another function. However, this risk can be reduced as the interleaving of server-side and client-side computation allows a precise understanding of the code currently running. Indeed, this interleaving strengthens the DynRA as it renders much more difficult to replay variables values.

Then a task processes the variable selected to be monitored with a server-side computation, as for the Simple IRA approach.

Injection of IRA annotations. As explained in Section 2.4.1, this component inserts annotations to drive the client-server code splitting and move the execution of some variables on the server. Annotations are places so that the code to move on the server is not trivially understandable thus making hard to build a fake server. No additional code for the verifier is injected in this phase.

Code Splitting. As explained in Section 2.4.1, the ASPIRE Client-Server Code Splitting protection that removes the variables and related code and automatically adds all the network communication logic.

The identification of variables must be performed, not on the vanilla application but on the after splitting client-side application, which is compiled with the standard compiler to produce DWARF information.

Dwarf parser. As presented in Section 2.3.4, this component processes the DWARF symbols and extracts the data that are actually needed to collect variables' values at runtime. Finally, it produces the DynRA-DS that will be injected in the protected application and stores the needed information in the ASPIRE DB.

Invariants interpreter. As presented in Section 2.3.4, this component elaborates the selected invariants and stores them in the ASPIRE DB in a format prone to fast verification.

2.5 Remote Attestation and Control Flow Tagging

Chapter Authors:

Philippe Jutel, Paul Gunawan Hariyanto (GTO)

2.5.1 Control Flow Tagging protection

This sub-section details the work done to implement the Control Flow Tagging (CFT) protection. This anti-tampering protection aims to check that some assertions are verified during the execution of the application. An assertion in CFT is a logical expression that refers variables that capture the way the application is executed.

Besides minor adaptations, the protection mechanism described in this document conforms to what has been described in the subsection 4.5 of the Reference Architecture- D1.04 and in the sub-section 6.3 of ASPIRE Offline Code Protection Report - D2.08. The reader should not be confused by the fact that the CFT protection is alternatively described in WP2 and WP3 documents because according to the place where the logical expressions are checked the protection can be offline or online.

The granularity of the CFT protection is the Basic Block of the generated code. In a compiler, a Basic Block is a section of intermediate or machine code that has a single entry and a single exit. CFT applies on the Basic blocks of the generated code.

As any others ASPIRE protections CFT relies on annotations placed in the source code of the application. There are two types of annotations with CFT. The Gate annotation and the Verifier annotation. A Gate is a region of the code that is associated with a counter. Each time the activation of the application a Gate then the associated counter is incremented. The Verifier is a region of the application code where it is possible to verify the value of the Gate counters. Each Verifier annotation specifies a logical expression that refers to one or several Gate counter variables. The verification of the expressions can be done locally on the device or remotely on the server. If a logical expression is not verified, a reaction is triggered.

The Reaction Unit (RU) can be invoked in case the logical expression returns False when it is evaluated. A reaction level is passed to the RU in order to trigger the adequate reaction behaviour according to severity of the non-conformance controlled by the logical expression. When RU is invoked it is necessary to put the initialization RU annotation before any Verifier annotation.

The protection is implemented at both source and binary level. Both steps are required to implement the protection.

2.5.1.1 Source level processing

The source level step prepares the code that evaluates the logical expressions. A logical expression specified in the Verifier annotation can be any valid C language Boolean expressions. The design is to prepare the source code that contains the code of the annotation and to use the compiler to analyse and generate the binary code.

To implement this source level step, ACTC is adapted with an extra processing step. In the SLP12 step ACTC calls a Python script that generates the C files that contain the verifiers code embed in functions. These functions are either compiled with the application or placed on the server for remote verification according to the 'location' option set in the annotation. In the latter case, the script also generates code inserted in the application to send the counter values to the server. The script scans the application code looking for CFT annotations. The script uses some template C files, with pre-written functions to be filled with the code taken from all CFT Verifier annotations.

Advantage of this design that relies on the compiler is that the code extracted from verifier annotation can always be successfully compiled because it must be valid C code that only refers to counter values with a known scope and data type.

The restriction comes from the specification of the expression where there is no way to mix counter values with application code or application variables. More powerful verifier could be envisioned with such expressions but possible side effects could lead to complex bugs.

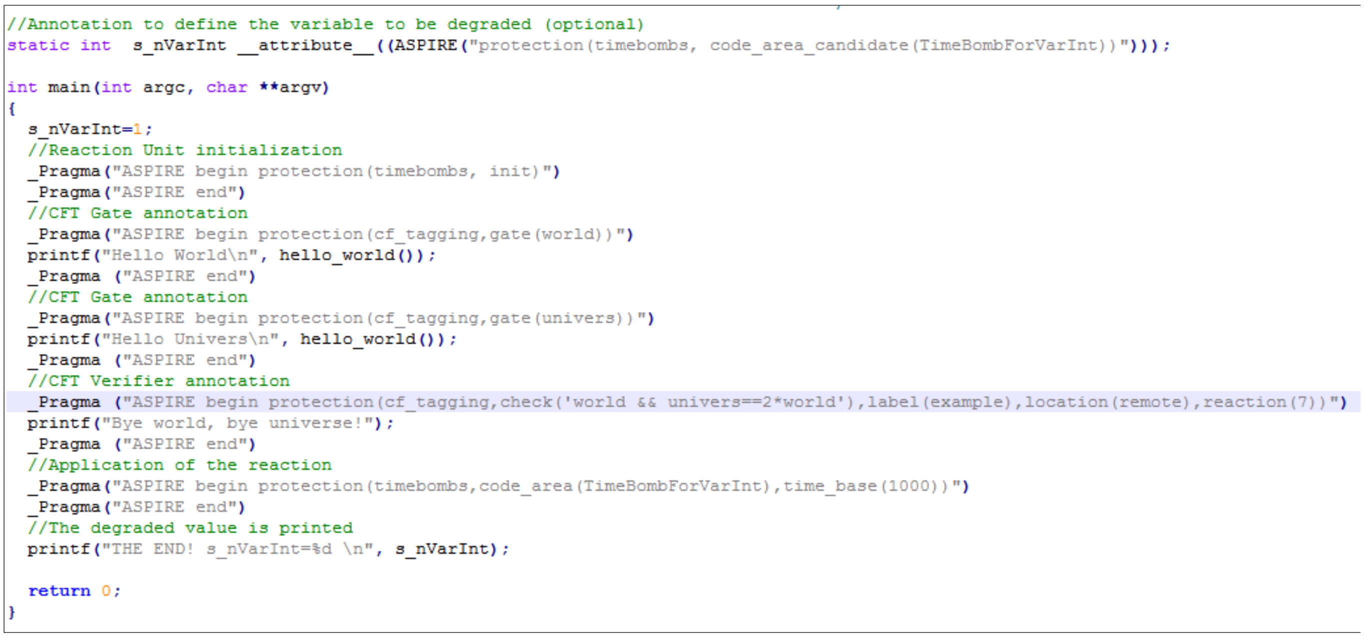
2.5.1.2 Binary level processing

The binary step of CFT is implemented using the Diablo framework. The diablo-obfuscator frontend is adapted to be able to inject the CFT binary code.

Only the code needed to identify and process the CFT annotations is included into the frontend itself. The majority of the CFT code is compiled in a separate library that is accessed by the frontend if required. This design was chosen to dissociate the CFT protection from the other protections implemented in the frontend mainly for Intellectual Property and maintenance reasons. That way, the frontend can recognize CFT annotations, but the CFT implementation depends on the presence of the extern CFT library.

The frontend creates regions for each CFT annotation, as it does for the other protections. A region is a group of basic blocks (BBLs) in the Control Flow Graph that is part of an annotated section of the source code. Annotated section means that this code section is marked out by an ASPIRE annotation. An example of CFT annotated code is shown in Figure 3.

A new command line argument to activate the protection is added in diablo-obfuscator as well. When the ACTC calls the diablo-obfuscator frontend, it now passes an additional command '-CFT on/off'.



```
//Annotation to define the variable to be degraded (optional)
static int  s_nVarInt __attribute__((ASPIRE("protection(timebombs, code_area_candidate(TimeBombForVarInt))")));

int main(int argc, char **argv)
{
  s_nVarInt=1;
  //Reaction Unit initialization
  _Pragma("ASPIRE begin protection(timebombs, init)")
  _Pragma("ASPIRE end")
  //CFT Gate annotation
  _Pragma("ASPIRE begin protection(cf_tagging,gate(world))")
  printf("Hello World\n", hello_world());
  _Pragma ("ASPIRE end")
  //CFT Gate annotation
  _Pragma("ASPIRE begin protection(cf_tagging,gate(univers))")
  printf("Hello Univers\n", hello_world());
  _Pragma ("ASPIRE end")
  //CFT Verifier annotation
  _Pragma ("ASPIRE begin protection(cf_tagging,check('world && univers==2*world'),label(example),location(remote),reaction(7))")
  printf("Bye world, bye universe!");
  _Pragma ("ASPIRE end")
  //Application of the reaction
  _Pragma("ASPIRE begin protection(timebombs,code_area(TimeBombForVarInt),time_base(1000))")
  _Pragma("ASPIRE end")
  //The degraded value is printed
  printf("THE END! s_nVarInt=%d \n", s_nVarInt);

  return 0;
}
```

Figure 3: Annotated code example

The main CFT binary level code is located in a shared library. If CFT is activated, diablo-obfuscator access this library and call two functions: a first one to prevent Diablo from deleting the verification functions that are injected at source level and a second one that does most of the binary code handling that will be described next.

The first step is to create the Gates and the respective counters. Thanks to the Diablo framework, it is possible to analyse the CFT Gate regions and to define the entries of these regions. An entry is any edge that originates from a basic block from outside the region and ends in an inside basic block. The only exceptions are the return edges from functions called from inside the region. The entries are the places where the additional ARM assembly code of the protection are added. This way we guarantee that each time the execution of the application enters a region, it will first execute the Gate instructions. Figure 4: Adding a Basic Block with binary code at the entry of a region pictures the insertions of a basic block in a CFT region.

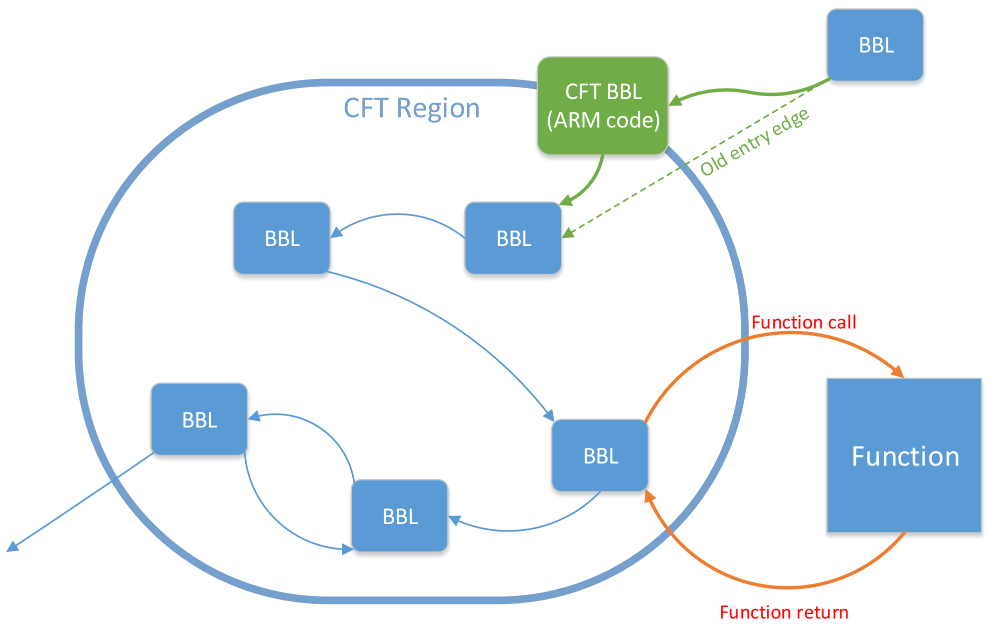


Figure 4: Adding a Basic Block with binary code at the entry of a region

Once the entries are defined, a data subsection in the Static Data segment is created. This subsection is reserved to store the counters of each Gate. The counters are stored as unsigned integers. Then the ARM instructions are injected. They are responsible for loading the right counter value, incrementing the counter, verifying if it reached the maximum value of an unsigned integer (in this case, the counter is not incremented) and, finally, storing the new value.

Then, a similar procedure is implemented for the Verifier regions. After analysing and finding the entries, the tool adds the proper ARM instructions. The goal of these instructions is to load all the counter values referenced in the logical expression of the Verifier and to create a branch to the proper Verifier function. The counter values are placed in the first four registers to be passed to the function. For that reason, each logical expression can use four counters at most. As explain at the beginning of this subsection, the functions are injected at source level and compiled with the application.

Injecting all these assembly codes required a register management. Since the processing needs to handle registers, each time a required register is being used by the normal execution of the application, its value is stored in the stack and retrieved after all CFT transformation. This is not very good because it helps the attacker during the static analysis.

2.5.2 *Attestation*

2.5.2.1 Local attestation

When the 'location' option of the Verifier annotation is set as 'local', the verification of the logical expression is offline. The Python script creates the functions that verify if the counters have reached their maximum value, in which case the verification cannot be done, checks if the logical expression is valid and call a reaction otherwise.

In case of offline processing, the 'reaction' field in the annotation can specify three various behaviours:

• The reaction level of the Reaction Unit: A number from 1 to 8 to define the level of the reaction to be applied. The function 'reactionUnitSyncNotification' is called.

• 'exit': In this case, the application will simply stop if the verification is not successful

• A call to a custom function: If a custom reaction function is defined in the application, it can be called in case the verification fails.

2.5.2.2 Remote attestation

The remote attestation for CFT is set when the annotation presents ‘remote’ in the ‘location’ field. It can only use the Reaction Unit to respond to attacks, so the ‘reaction’ option in the annotation can only be a number from 1 to 8. The ASPIRE Portal is used to manage the communication between the device and a server side component, as described in the Figure 5.

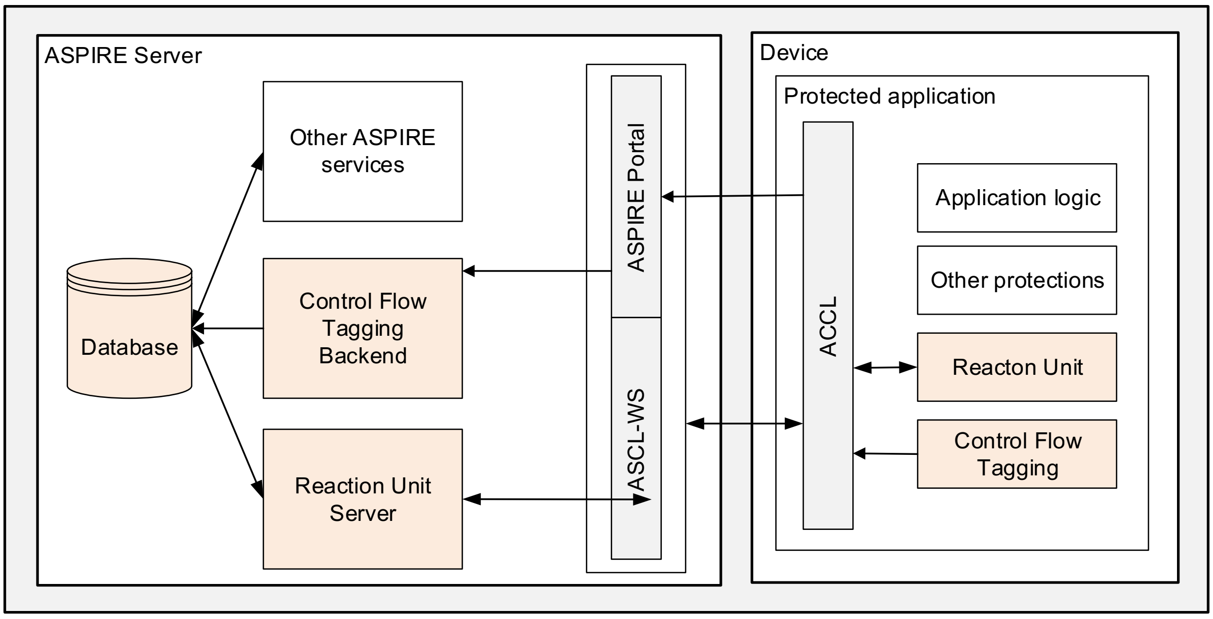


Figure 5: CFT Remote Attestation architecture

Instead of owning the verification function, in this case the device has a function whose role is to receive the counter values, create a payload and send it to the server using the Simple Request ASPIRE protocol. It is up to the binary manipulations performed by Diablo to retrieve the counter values and invoke this function.

On the server side, the ASPIRE Portal receives the payload and executes the CFT Backend, the server-side component of the CFT protection. The role of this backend is to receive and interpret the payload forwarded by the ASPIRE portal, which includes the ASPIRE application ID. Then, it accesses a shared library that contains the actual verification functions for the given application ID. This library is built at compilation time by ACTC from the verification functions created by the Python script. The file name must be the application ID with the .so extension. The backend and the library must be placed in the same directory.

Finally, the CFT backend executes the verifications and connects to the ASPIRE database dedicated to the Reaction Unit. It upgrades two tables: a specific CFT table that stores the results of each Verifier, and a general Reaction Unit table, that can be updated by any protection using the Reaction Unit, that contains the notifications to be sent to the devices.

The succeeding steps are managed by the Reaction Unit, responsible to send notifications back to the device and apply the reactions.

2.5.3 Reaction Unit

The Reaction Unit (RU) is a reaction component embedded in the application that can be used by any protection techniques. It is the component used by CFT to alter the application in case an abnormal behaviour is detected. To be applied, its code must be compiled together with the application by ACTC and a server side component is also required.

In the application code, an annotation is used to initialize the mechanism. The initialization tries to connect the device to the ASCL component on the server side. If the connection is not established, the offline mode is set and the status of the application is only updated through the 'reactionUnitSyncNotification' function. If the device connects to the server, a new thread is created to listen to the WebSocket channel. The 'reactionUnitAsyncNotification' is set as a callback function, called to update the device status every time the device receives a payload from the server.

A second annotation defines where the reactions must take place. The reaction level defines if the application will be slowed down only, if sensitive variables are to be altered or if the application shall be crashed.

On the server-side, the Reaction Manager had already been developed to be a server responsible to send notifications payloads to the device, triggering reactions in the client-side. A more generic server side component, called Reaction Unit Server, has been implemented based on the Reaction Manager.

2.5.3.1 The Reaction Manager and the Reaction Unit

The Reaction Manager (RM) developed earlier was designed to work with the Remote Attestation (RA) described in the section 3 of D3.06. The Remote Attestation Manager sends requests to the device asking for a code integrity verification. The results are sent back to the server and stored in an ASPIRE database dedicated to this remote attestation. Then, the RM queries this database and process the results according to a policy provided by the developer of the application. At the end, it sends a notification payload to the Reaction Unit components in the device.

This architecture correctly links this particular Remote Attestation technique to the Reaction Unit, but it cannot be used for managing the communication between other protections and the Reaction Unit. All the database accesses, the data processing and the policy application are tailor made to work with the RA Manager.

2.5.3.2 Reaction Unit Server

The Reaction Unit Server, whose architecture is represented in Figure 6: CFT Reaction Unit Server architecture, is designed to be a more generic mechanism than the Reaction Manager. Most of its design derives from the Remote Manager developed earlier.

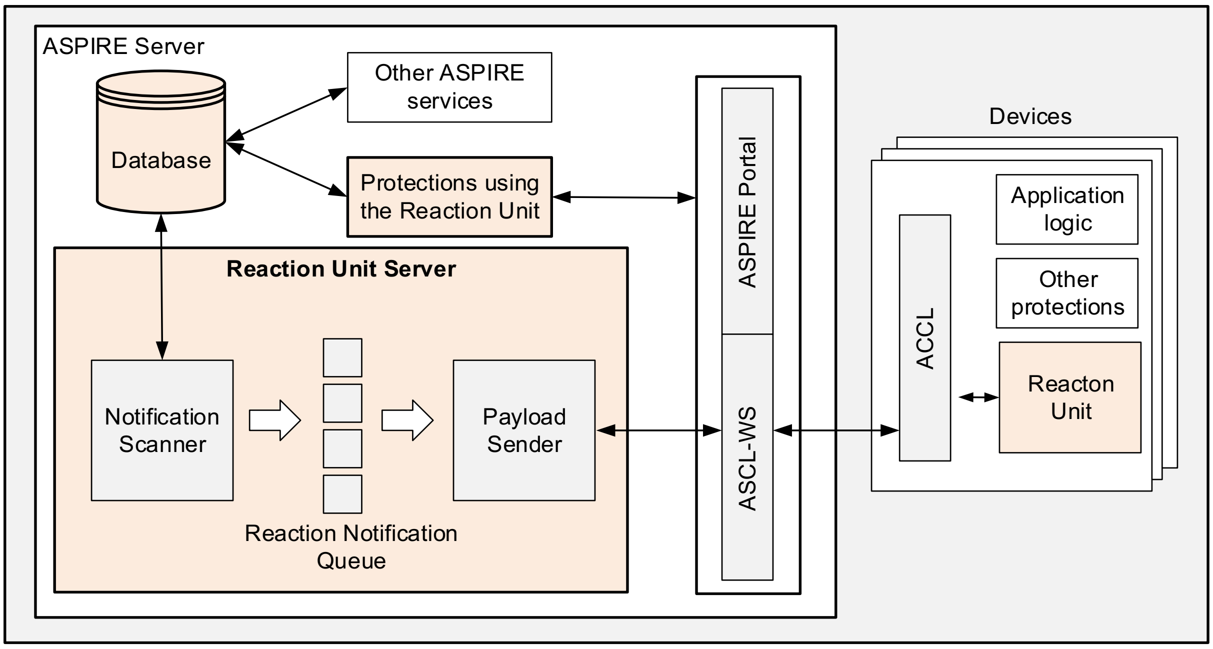


Figure 6: CFT Reaction Unit Server architecture

The Reaction Unit Server can be used by any protection technique using the Reaction Unit. The only requirement is that the protection technique must update the Reaction Unit database. Each protection has to insert the result of its analysis in the database.

There is a main notification table where each entry represents an attestation result. This table stores the ID of the technique that performed the attestation, the ASPIRE application ID, the status of the application (tampered or not) and the reaction level to be applied.

Additional tables can be included in the database to store important information about particular protections and maintaining a history of the technique. Nevertheless, it is mandatory that the technique interacts with the main notification table to be able to communicate with the devices.

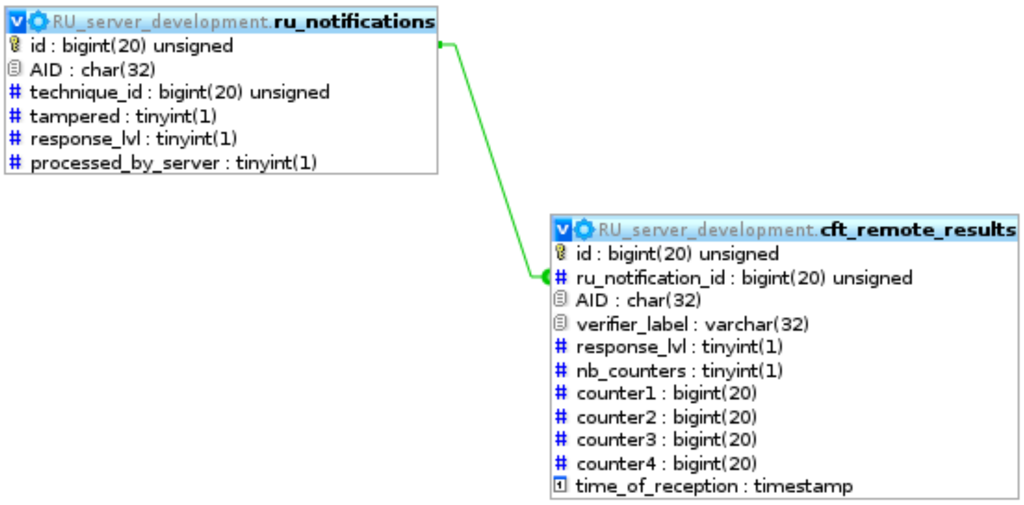

RU_server_development.ru_notifications
id : bigint(20) unsigned
AID : char(32)
technique_id : bigint(20) unsigned
tampered : tinyint(1)
response_lvl : tinyint(1)
processed_by_server : tinyint(1)
RU_server_development.cft_remote_results
id : bigint(20) unsigned
ru_notification_id : bigint(20) unsigned
AID : char(32)
verifier_label : varchar(32)
response_lvl : tinyint(1)
nb_counters : tinyint(1)
counter1 : bigint(20)
counter2 : bigint(20)
counter3 : bigint(20)
counter4 : bigint(20)
time_of_reception : timestamp

Figure 7: CFT Reaction Unit Server database with CFT table

The Notification Scanner is the component from the Reaction Unit Server responsible for constantly scanning the database looking for unprocessed notifications. If an unprocessed notification indicates that the application is tampered, then Notification Scanner creates a Reaction Notification to be put in a queue.

It is the role of the Payload Sender to push and to process every element of the queue. A pushed Reaction Notification originates a Reaction Unit Payload to be sent to the device using the ASCL WebSocket protocol.

When the device receives a payload from the Reaction Unit Server, the 'reactionUnitAsyncNotification' callback mentioned before is invoked to reconstruct the payload and update the device status, allowing the Reaction Unit in the client to apply the required measures.

The Reaction Unit Server is a multi-threaded component. A main thread creates the other elements and manages the communication with the WebSocket channel. In the meantime, a pre-defined number of Notification Scanners and Payload Senders threads can run separately.

2.5.4 Protection analysis

The online CFT, combined with the Reaction Unit, has the advantage that the logical expression is not contained in the device because it has been placed on the server. That way, an attacker cannot know the logical expression through reverse engineering. With offline CFT, the Verifier codes that checks the logical expressions are a part of the application code and are exposed to reverse engineering.

However, using the online CFT has constraints. The developer that wants to protect his application must consider that when the execution reaches a Verifier region, there is a delay between the dispatch of the payload containing the counter values and the application status update that allows the reactions to be applied by the Reaction Unit. This delay exists because the technique depends on the quality of the connection with the server. There is also a delay due to the processing done by the CFT backend and the Reaction Unit Server.

Another aspect to be taken into account is the fact that the Reaction Unit needs a separate thread to listen to the WebSocket channel. If an attacker identifies this thread and stops its execution, the application will not receive the attestations results, disabling the protection. A solution would be to insert a vital portion of the application in the same thread. That way, killing the thread would cause the application to stop.

Finally, the CFT either offline or online can be combined with code integrity checking techniques brought by code guards.

Section 3 Renewability

Chapter Authors:

Alessandro Cabutto, Paolo Falcarin (UEL)

3.1 Finalized implementation

Since last report on Renewability (D3.08) the design and implementation did not change and only minor improvements have been contributed. Among those improvements the most impacting concerns the deployment layout of mobile blocks on the server side and their subsequent renewing process. The repository structure has been updated as follows:

```
/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 '%08x' format (e.g. 00000001)
- MOBILE_BLOCK represents the mobile blocks contained into the repository in 'mobile_dump_%08x' format (e.g. mobile_dump_00000001)

The revision count starts from zero and is incremented by one unit each time the diversification script is invoked. A pointer to the diversification script, produced during the ACTC pass into directory BC05, is saved in the database backend along with other policy information so that the Manager can invoke it when necessary.

The final implementation of the policy table (*rn_application_policy*) looks like the following

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.
`start_from_revision`	VARCHAR(10)	`char[10]`	Use this revision when a client starts.
`diversification_script`	TEXT	`char[1024]`	Path to the diversification script.
`disable_renewability`	TINYINT(1)	`boolean`	Disable diversification at all.

Newly introduced columns are highlighted. By default, new instances of the same application start using revision 00000000, but a *start_from_revision* field has been introduced for future improvement. The *disable_renewability* flag has been introduced to allow disabling of the whole renewability architecture for a certain application at a given time.

3.1.1 Unbinder

The Unbinder is a client side component that is used to restore calls to the Binder in GMRT table. When invoked it loops through all GMRT entries by resetting *downloaded* flags and setting up a call to the Binder instead of previously downloaded mobile blocks.

For sake of simplicity it is implemented into the same object file containing Binder capabilities.

3.2 Renewability on use cases

The Renewability Framework is composed by several components both on the server-side and the client-side. Those components are described in previous deliverable such as D3.06 and D3.08. To reliably verify the correct functionality of all of them a full deployment of the architecture is needed.

Server Side. The Renewability Manager has to be running to ensure that clients are queried for mobile blocks unbinding and code blocks are renewed. The ASPIRE Portal and Code Mobility Server must be ready.

Client Side. The protected application must be linked to the Binder, Unbinder and Renewability components in order to work correctly.

ACTC takes care of all the setup process both on the server and client side. A first round of tests has been carried out on a toy example developed on purpose and then on NAGRA's use case.

3.2.1 Renewability on a toy example

A small toy example has been provided along with Renewability's source code. It is very simple but particularly useful in our test; it composed by a main function containing an infinite loop with some delay between each operation:

```
while (1) {
    fn1();
    sleep(2);

    fn2();
    sleep(2);
}
```

Functions fn1 and fn2 are invoked continuously from the loop. They execute some computation, emit some text on standard output and return to main. Output generated by each function is different so that is easy to understand where it comes from. Both functions are made mobile and renewable via annotations. While configuring ACTC's input file, *aspire. json*, we set a small timeout for diversified code blocks (see D3.08 Section 1.2 for further details), 10 seconds in our case. In a non-renewable context mobile blocks containing functions fn1 and fn2 would be downloaded only once. In a renewable scenario the Renewability Manager detects client start-up and runs a process to apply pre-defined renewability policies requesting the client to discard already downloaded mobile blocks when necessary. By monitoring log files generated by Code Mobility Server and Renewability Manager is easy to see that mobile blocks containing fn1 and fn2 are downloaded again and again after timeouts expiration.

3.2.2 NAGRA use case

A full description of NAGRA's use case can be found in D6.01 Section 2.

To test renewability features on NAGRA use case we used its 1.2.1.1 version and ACTC 2.8.0. Since our renewability implementation is built on top of Code Mobility framework, only mobile functions can be renewed, so we firstly selected some candidate functions to be made mobile. So one of the criteria we used to select such functions is that candidates must be repeatedly called by the client application over time. As first approach function *DrmKernel_getRightKey* contained in *DrmKernel.c* of DRM plugin has been selected and accordingly annotated in the source code. It is used for right verification purposes and so it is periodically invoked (i.e. each time a content is purchased) by the application.

This scenario is similar to the one created for the toy example and can be easily validated by using the same approach.

3.2.3 Composability with other protection techniques

Renewability has been tested successfully in combination with other techniques. First of all, Code Mobility, but this is obvious since Renewability relies on Code Mobility Framework. Some Binary Obfuscations (Flattening Functions and Opaque predicates) have been applied in combination with Renewability. Finally, WBC and SoftVM have been also applied in combination with Renewability.

To test these combinations of protections we applied annotations coming from patch files produced to instrument source code of the use case for Tiger Team experiments. This ensured us that suitable protections were applied in a realistic way.

3.2.4 Conclusions

The Renewability Framework provides a fully automated diversity in time deployment system. Starting from the source code of an application is possible to obtain a first release of binary, which can be delivered to users. Such release can then be updated on the client side according to a pre-defined time based renewal policy. In our current implementation the software producer defines the policy at compile time but, with minor engineering effort, is possible to extend this feature making policies updatable at run time.

Thanks to its design Renewability can be used in combination with most of the other protection techniques similarly to Code Mobility, used as basic component.

3.3 Practically useful software diversity

Section Authors: Bjorn De Sutter, Bart Coppens, Bert Abrath

In deliverable D3.04 – Intermediate Online Protections Report, UGent already reported its initial work on making software diversity in space, where different users get different binaries of their software to prevent attacks on one user's copy from attacking another user's copy. Since D3.04 (Oct 2015), UGent has redesigned their approach and prototype implementation, and has obtained good results. This section presents the current state, in the form of the content of a draft paper. UGent intends to submit a full paper in early 2017, when more measurement data will be available.

3.3.1 Introduction and Motivation

The monoculture in software, in which identical copies of programs are distributed to all users, has long been blamed as easing the profitable exploitation of malware [For97, Coh93]. As a mitigation, software diversity has been proposed [Bau15]. This is the practice of distributing and executing many structurally different versions of software components without altering their observable behaviour from a benign user's perspective. The main goal is to prevent that an identified attack path can automatically be scaled up to many systems, thus lowering the expected profit of attacks. Many aspects of a program can be diversified

[Lar14] and diversity can be introduced at many stages of the software development life cycle (SDLC) [Lar15]. As software diversification can protect against many types of attacks, its use is becoming mandated for more and more systems. Examples include the requirement in many settings to use Address Space Layout Randomization (ASLR) and MovieLabs' Specification for Enhanced Content Protection [Mov13]. The latter mandates software diversity and so-called copy & title diversity, albeit without prescribing specific diversification schemes.

In practice, however, we observe that very few, and only very simple diversification schemes gain widespread traction. With ASLR, for example, only absolute addresses are randomized. But offsets within the code and static data segments of executable binaries remain constant, and so do the offsets on the processes' stacks. These limitations open the door to information leak attacks. When academics present new, more advanced diversification schemes, industrial developers typically appreciate their usefulness in terms of protection strength, but they also express reluctance. The costs and limitations the proposed schemes impose on the SDLC severely restrict their practical usability. Amongst others, industrial developers and vendors are afraid that the diversification might trigger bugs and that it will it harder to support customers.

One of the customer support issues relates to crash collectors. Google Breakpad, for example, is a small software component that can be embedded in applications to facilitate the collection of useful crash reports, even when the application binaries are distributed to end users without debug information. Its operation involving three parties is visualized in Figure 8. When the application crashes on a user's system, the embedded Breakpad component sends a stack dump (called minidump) to the crash collector server. On that server, a tool then combines the minidump information with the debug information stored in a so-called symbol file on the server. The tool then generates a stack trace, which most often is first analyzed and classified automatically. If no equivalent traces are found in a database of previously received traces, the vendor's developers are notified that a previously unknown bug or previously unknown trigger has been identified, at which point they can start to study the trace manually. For obvious reasons, crash collector tools like Breakpad have become quite popular.

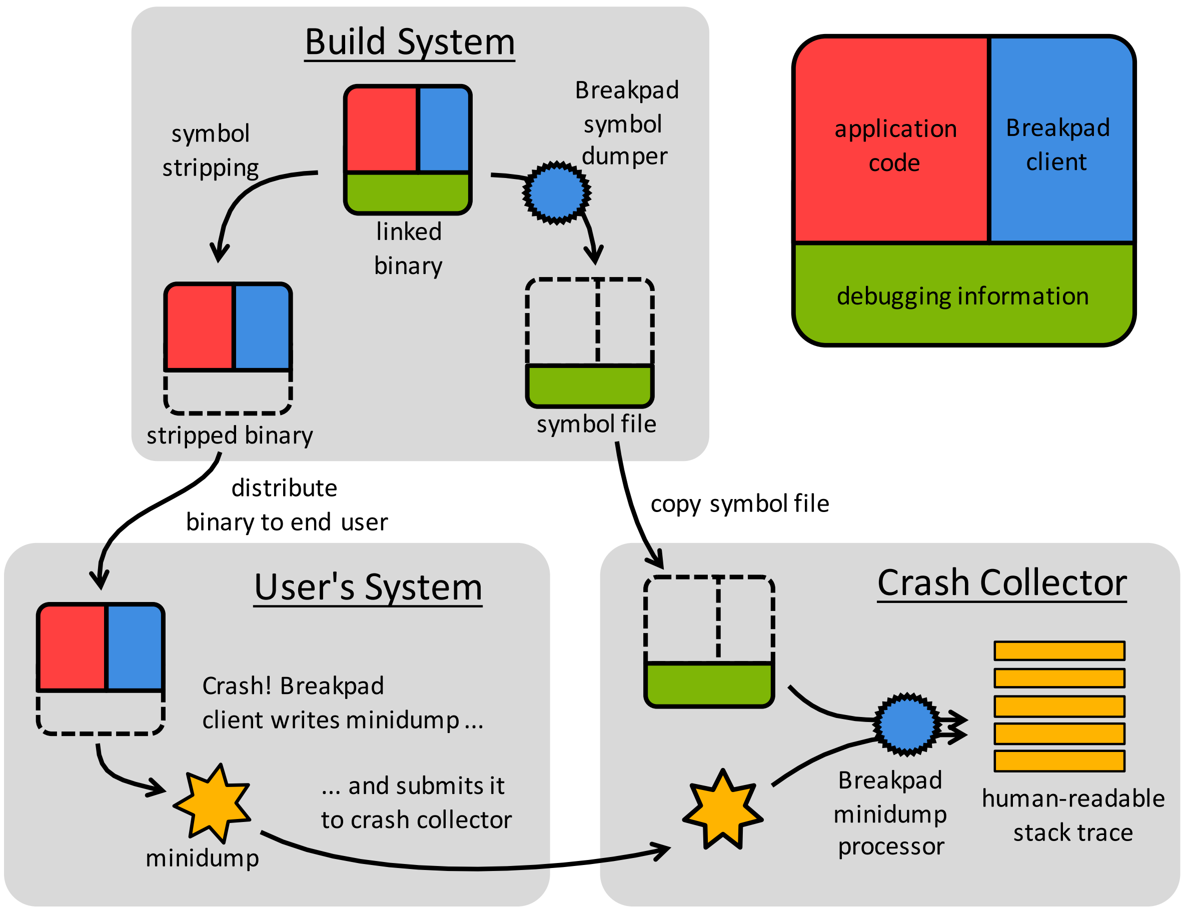


Figure 8: Overview of Google's Breakpad tools for crash collection.

With spatial diversification schemes in which different users of an application execute different code versions, the described crash collector system no longer works out of the box. Unless the crash collector stores symbol files for all of the different versions, it will lack the necessary information to identify and interpret the diversified stack frames in the received minidumps. Simplistic solutions to overcome the mismatch between diversified minidumps and a single symbol file, such as permanently storing debug information for all diversified versions, are infeasible because symbol files are quite big. The alternative solution of rebuilding a software version and its debug information on the server when a crash report comes in is impractical as well: For larger programs, recompilation of every crashed version would be compute-intensive, and it requires the precise reproduction of the developer's build environment in the crash collection environment, which might reside on a third party's infrastructure. Even if the precise reproduction would be considered technically feasible, it will often be unacceptable because of security requirements, such as confidentiality.

As an alternative solution framework, we propose to extend both the diversified stack dumps and the debug information stored on the crash collector server with a minimal amount of *delta data*. Its purpose is to let the crash collection tool overcome the mismatch between the diversified stack dumps and a single instance of debug information, without requiring large amounts of persistent storage or communication bandwidth. Within this solution framework, the research question then becomes the following: To which extent can we adopt diversification techniques to provide more protection than simple ASLR without bloating the delta data needed to support crash collection?

This is a non-trivial question because compilers are complex tools, in which seemingly trivial diversifications of local code fragments can have global effects. On RISC architectures in particular, we have observed that even minimal changes to the offsets between pairs of instructions or to the sizes of stack frames can impact decisions made during instruction selection, register allocation, and instruction scheduling. As a result, even when it is possible to predict the direct impact of a diversification scheme on the individual code fragments being diversified, it is hard to predict the indirect effects on the surrounding code. Consequently, it is quite impossible to replicate the diversification process on the crash collector server on the basis of only the original binary, its debug information, and the parameters (such as random seeds) that were used as input to the original diversification process. Instead, a mechanism is needed that combines good but imperfect replication of the diversification process with patching that can make up for the imperfection. To minimize the resource requirements (bandwidth, storage, computational power), the mechanism should exploit the fact that the replication and patching process only needs to produce sufficient debug information, not a fully working binary.

In this paper, we provide a first answer to the above research question by presenting ∆Breakpad. This approach and prototype tool is the first practical solution to the problem of crash reporting for applications with fine-grained layout diversification as a defence mechanism against code injection and code reuse exploits. The tool and the presented techniques comprise minimal adaptations to compilers to perform code and stack layout diversification to significantly raise the bar for attackers, and to generate the necessary, minimal delta data. It also comprises a method to bridge the gap between a diversified stack dump and the debug information of an undiversified copy of the software on the crash collector.

3.3.2 Background & Problem Statement

3.3.2.1 Offset Diversification

For our initial experiments with crash reporting for diversified binaries, we focus on diversification schemes that alter offsets between instructions in a program binary and offsets between elements in stack frames. We focus on compiled languages such as C and C++ that provide no memory safety. The studied types of diversification have proven to be useful on top of basic ASLR, because they raise the bar for information leak attacks: when offsets within memory segments are diversified on top of the start addresses of the segments, one leaked address no longer directly informs the attackers about the locations of all code fragments in a binary or of all data on the stack.

We deploy the following offset diversification schemes:

- **Function Shuffling** The order of all the functions in a whole binary is randomized. This randomizes inter-procedural code offsets with high entropy [Kil06].
- **Randomized NOP Insertion** at random locations, for some average frequency, NOPs (no-operations) are inserted into the code bodies of all the functions in a binary. This randomizes intra-procedural code offsets [Hom13].
- **Randomized Stack Padding** A random number of bytes is inserted in between the stack locations of buffers and the stack locations of the return addresses [For97]. The impact on the stack frames is visualized in Figure 9. This randomizes the distance from buffers to the location where the return addresses are stored, as well as the distances between return addresses in different frames on the stack.

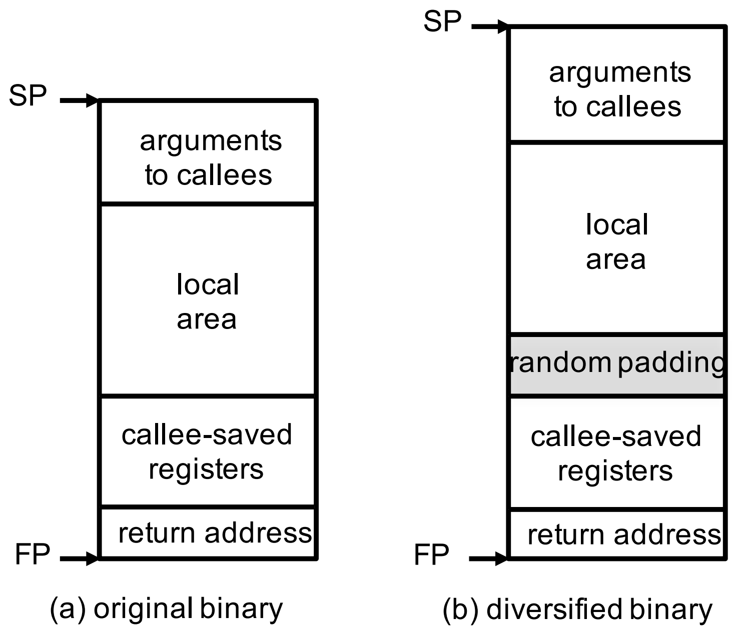


Figure 9: Stack frames in original and diversified binaries.

To implement these forms of diversification, stochastic decision processes typically decide on the function ordering, on the locations to insert NOPs, and on the amounts of stack padding to insert. The stochastic decision processes are deterministic and based on a pseudo-random number generator (PRNG). To generate diversified binaries, it then suffices to feed the PRNG different random seeds.

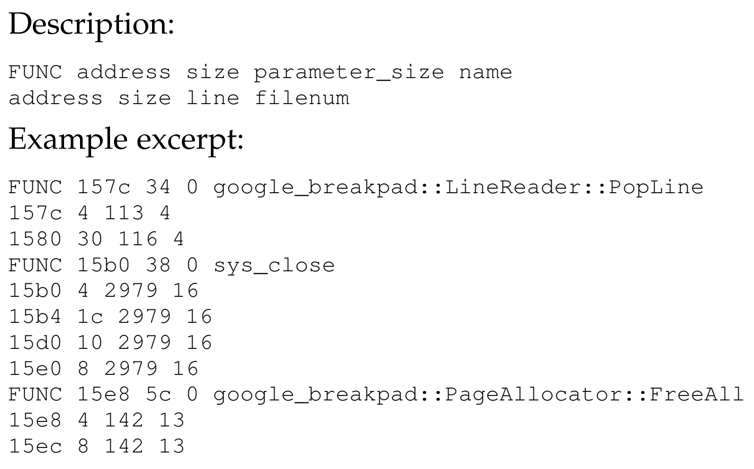
Importantly, as these diversification schemes are rather simple, the stochastic decision processes do not involve checks of complex pre-conditions on the code fragments to be diversified. In other words, no complex compiler technology is needed to replicate the decision process of a diversifying compiler (or other tool), outside that compiler: only the number and order of the functions plus a minimal amount of information on the locations of their function bodies and stack sizes need to be known. All of that information is readily available in standard debug information.

A direct effect of all three the diversification schemes is that offsets encoded in the code section of a binary change. With the first two schemes, the displacements between instructions change, as does the offset of all instructions relative to the start of the code segment of the binary. In the code section, this implies that the PC-relative offsets encoded in, e.g., direct control flow transfers change. With the second scheme, the direct changes occur in the displacements between the base pointer on the one hand, and the data items in a stack frame on the other hand. So offsets encoded in stack memory operations change, and so might the immediate operands of instructions that produce pointers to stack-allocated data.

In all three schemes, the diversification hence results in changes to offsets encoded in instructions as immediate operands. The indirect effect of those changes on the debug information depends significantly on the type of processor architecture, as we discuss below.

3.3.2.2 Necessary Debug Information

Conceptually, the debug information of interest, which is embedded in the symbol files used by Breakpad, consists of source line information on the one hand and stack unwinding information on the other hand. For both forms of information, the code is partitioned in regions, i.e., in short sequences of consecutive instructions. The line information then consists of a single list of regions. For each region, the start address, the size, and the corresponding source file and source line number are stored. In the symbol files that Breakpad uses, this information is stored in human-readable form, as shown in Figure 10. Each line containing only (hex) numbers corresponds to one region.



Description:

```
FUNC address size parameter_size name
address size line filenum
```

Example excerpt:

```
FUNC 157c 34 0 google_breakpad::LineReader::PopLine
157c 4 113 4
1580 30 116 4
FUNC 15b0 38 0 sys_close
15b0 4 2979 16
15b4 1c 2979 16
15d0 10 2979 16
15e0 8 2979 16
FUNC 15e8 5c 0 google_breakpad::PageAllocator::FreeAll
15e8 4 142 13
15ec 8 142 13
```

Figure 10: Source line mapping in the symbol file.

The stack unwinding information also consists of a list of regions. For each region the start address is stored. In addition, for each region information is stored that pinpoints where in the program state the stack unwinder can find the information necessary to unwind a stack. Figure 11 shows an example of that information in the symbol file.

The post-fix expressions on registers (sp, r11, lr, ...) express how to compute the necessary properties of the frames on the stack when execution has reached a program point in a given region. These properties are the canonical frame address (.cfa), the return address (.ra), and the values of callee-saved registers in a function's caller. The first three entries in the symbol file excerpt relate to function1, which has a FP, as can be seen in the assembly code of its prologue. Note that the ARM EABI reserves register r11 for the FP. The expression for .cfa on the first line encodes that on entry to function1, the stack pointer (SP) still points to the start of the function's stack frame. The second line clarifies, amongst others, that after the push instruction, two callee-saved registers can be found on the stack, and that the SP now points 8 bytes beyond the start of the frame.

Description:

```
STACK CFI INIT address size reg1: expr1 reg2: expr2 ...
STACK CFI address reg1: expr1 reg2: expr2 ...
```

Example symbol file excerpts:

```
STACK CFI INIT 1bdc f0 .cfa: sp 0 + .ra: lr
STACK CFI 1be0 .cfa: sp 8 + .ra: .cfa -4 + ^ r11: .cfa -8 + ^
STACK CFI 1be4 .cfa: r11 4 +

...

STACK CFI INIT 28a4 f8 .cfa: sp 0 + .ra: lr
STACK CFI 28ac .cfa: sp 20 + .ra: .cfa -4 + ^ r4: .cfa -20 + ^
             r5: .cfa -16 + ^ r6: .cfa -12 + ^ r7: .cfa -8 + ^
STACK CFI 28b4 .cfa: sp 904 +
```

Corresponding assembler code excerpts:

```
<function1>:
     push    {fp, lr}
     add     fp, sp, #4
     sub     sp, sp, #16
     ...
<function2>:
     push    {r4, r5, r6, r7, lr}
     cmp     r3, #0
     sub     sp, sp, #884    ; 0x374
     ...
```

Figure 11: Stack walking information in the symbol file.

So in essence, when replicating the diversification of a binary to support the construction of a stack trace from undiversified debug information, we need to be able to replicate changes to the number and ordering of regions, changes to their start addresses and sizes, and changes to the locations where relevant pieces of program state are stored.

3.3.2.3 Indirect effects of in x86 binaries

On variable-width CISC architectures such as Intel's x86, the indirect effects are mostly limited to additional changes in the displacements between instructions. When, as a result of a changed offset, less or more bytes are required to encode the changed offset as an immediate operand to some instruction, the x86 compiler will simply generate another form of the same instruction that uses less or more bytes, as needed. In addition, as the compiler might decide to put certain instructions on specific alignments, e.g., to optimize instruction fetching or instruction caching, the compiler might insert different amounts of padding when the location of code fragments is altered because of the diversification. Most often, these changes only alter the addresses and sizes of regions in the symbol files.

More or less the same happens as a result of the additional, randomized stack padding that is inserted. In many functions, no instructions are present in the function prologues/epilogues that only increment/decrement the stack pointer. To allocated/deallocate the additional randomized padding in such functions, additional instructions have to be inserted in the prologue/epilogue. In the symbol file, this comes mostly down to splitting regions in the stack unwinding information.

So replicating the effect of diversification on the debug information stored on a crash collector requires updating the number, addresses and sizes of regions, as well as the offsets where relevant program state is stored in stack frames. To replicate these changes when a crash report comes in from a diversified copy, it suffices for the crash collector to have (i) the original, undiversified binary including its debug information; (ii) a script that replays the deterministic decision processes of the randomizing diversification schemes; (iii) and the random seeds that were used for generating the diversified binary.

So on architectures like the x86, for extending the flow of Figure 8 to support binaries diversified with the three studied schemes, it more or less suffices to embed the random seeds in the diversified binary, to patch the Breakpad client to let it send the seeds along with the minidump to the crash collector, and to extend the Breakpad minidump processor to let it replicate the impact of the diversification process on the symbol file. For that replication, not the whole original compiler is needed, only a simple script that replays the stochastic diversification decision process.

3.3.2.4 Indirect effects in ARMv7 binaries

On architectures like the ARMv7 RISC architecture, the situation is quite different. The same effect plays, e.g., with respect to the function prologues and epilogues, but in addition, there are many more indirect changes as a result of offset diversification. There are three underlying reasons.

Fixed-width instruction encoding. ARMv7 instructions are 16-bit or 32-bit wide. The immediate operands of ALU and LD/ST instructions can therefore only be quite narrow, so when offsets grow bigger because of diversification, it can become impossible to encode them as immediate operands. Instead, the offsets then have to be stored in registers instead. This requires additional instructions and puts extra pressure on the register allocator, as a result of which instructions can become scheduled in different orders. In fact, we have observed that if the same offset has to be generated multiple times, the compiler sometimes applies common-subexpression-elimination, which can have a global impact on register allocation and instruction scheduling. Furthermore, we have observed that the compiler sometimes changes the base register used in LD/ST instructions, e.g., when the offsets of a location in the stack frame relative to the SP and/or the FP change.

Rotating immediate operands. The ARMv7 architecture has a peculiar way of encoding offsets as 8 consecutive bits that can be rotated over a 5-bit amount. It therefore also happens that offsets that could not be encoded as immediate operands in the original binary, become perfectly fine ones after they have become bigger in the diversified binary. For example, an original offset 0x3ff0 cannot be encoded in an immediate operand, but the bigger offset 0x4000 that might result from stack frame padding can.

The visible program counter. ARMv7 code is full of PC-relative computations, both in position-independent code and in position-dependent code. The reason is the visible program counter. To produce constants that cannot be encoded in immediate operands, (or constants unknown at compile time, such as absolute addresses or inter-modular offsets) constants are often loaded from so-called literal pools: short data chunks dispersed in between the code. These pools are accessed through PC-relative load operations. As our diversification schemes can result in changes in the size of code, and as only narrow offsets can be encoded, the diversification affects the location where the compiler injects the literal pools into the stream of instructions.

In conclusion, when targeting an architecture like the ARMv7, we have to expect much further reaching changes to the code section contents, even if we only apply our three relatively simple offset diversification schemes.

Moreover, in this case it is impossible to replicate the changes to the corresponding symbol file completely without replicating a lot of the compiler infrastructure that was used during register allocation, instruction selection, and instruction scheduling. In other words, it will not suffice to put a simple script on the crash collector server to replicate the impact of the diversification on the symbol file.

3.3.3 The ∆Breakpad Approach

To overcome the discussed problem, our ∆Breakpad approach combines three main concepts. The first concept is *imperfect replication* of the diversification process' impact on the symbol file. The second is *patching* of the imperfect replication result to make it perfect.

The crash collector will not only receive the necessary random seeds to replicate the diversification decision process, but also a patch that will allow it to fix any imperfection of the performed replication. So the ∆Breakpad client has to send both the minidump, the random seeds, and the patch to the crash collector.

The third concept is ∆ *minimization*, with which we denote the adaptation of the diversification process to minimize the sizes of the patches that the client has to send to the crash collector.

Figure 12 presents an overview of our ∆Breakpad approach. It looks much more complicated than Breakpad in Figure 8, but the main Breakpad components are still present, and are in fact reused as is.

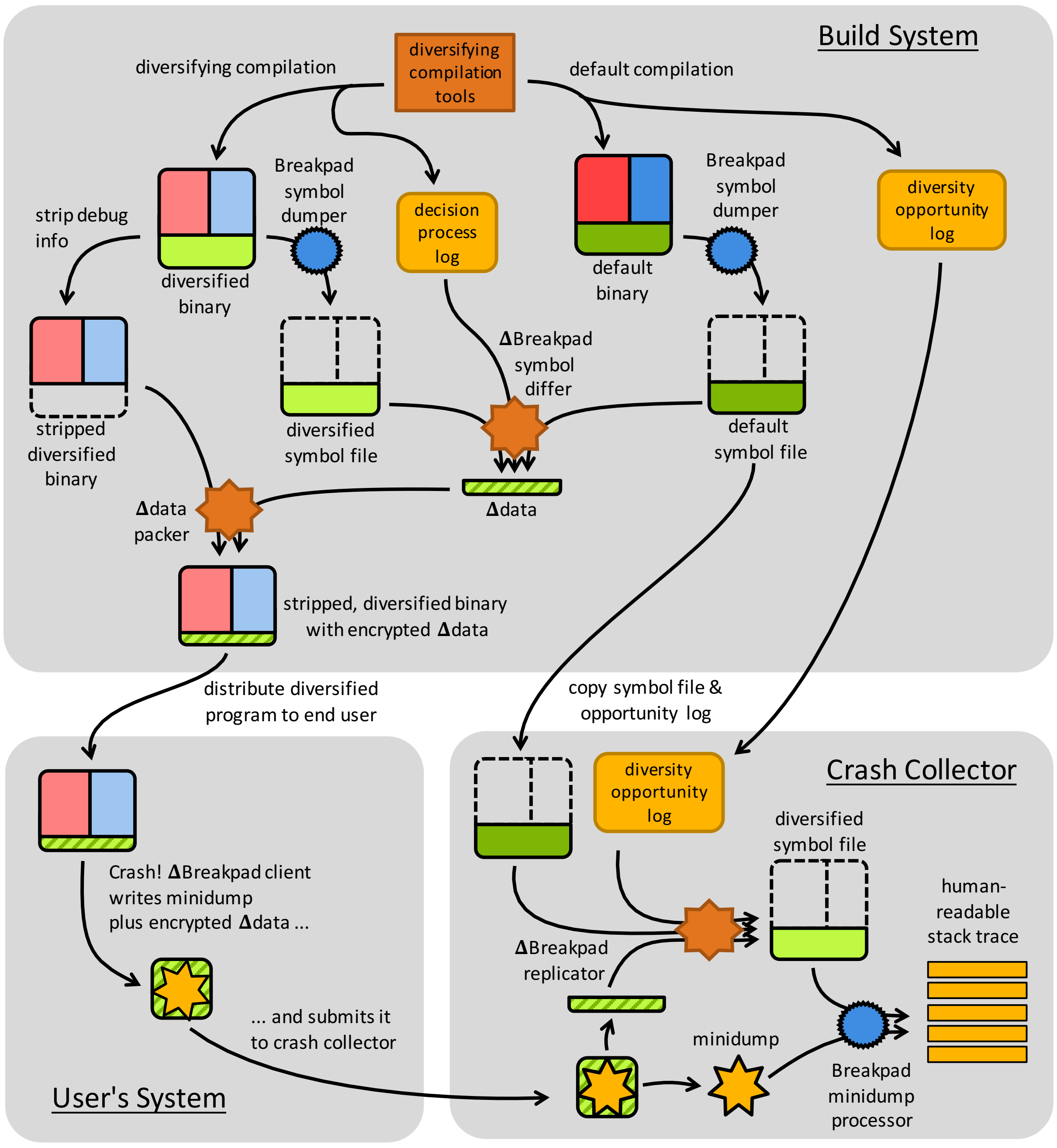

Build System
diversifying compilation tools
diversifying compilation
default compilation
Breakpad symbol dumper
strip debug info
decision process log
Breakpad symbol dumper
diversity opportunity log
diversified binary
default binary
ΔBreakpad symbol differ
stripped diversified binary
diversified symbol file
default symbol file
Δdata packer
Δdata
stripped, diversified binary with encrypted Δdata
distribute diversified program to end user
copy symbol file & opportunity log
Crash! ΔBreakpad client writes minidump plus encrypted Δdata ...
... and submits it to crash collector
User's System
Crash Collector
diversity opportunity log
diversified symbol file
human-readable stack trace
ΔBreakpad replicator
minidump
Breakpad minidump processor

Figure 12: Overview of ∆Breakpad as an extension of Google Breakpad for reporting crashes of diversified binaries

3.3.3.1 Crash Handling & Stack Trace Generation

Importantly, our ∆Breakpad approach does not require any change to the minidump that is sent by the client to the server. The minidump file format as developed by Microsoft is similar to core dump files, but much smaller, better documented, and less OS-specific. A minidump contains

- A list of the executable and all shared libraries loaded into the process when the dump was created.
- A list of the process threads, with their stacks and processor register contents. Complete stacks are included because the applications typically do not contain debug information to analyse the stack.
- Some more system information, incl. the processor and OS versions, as well as the reason for the crash.

We only adapt the Breakpad client such that it sends the server a small chunk of ∆*data* along with the minidump (bottom right of Figure 12). The ∆data contains the random seeds and other parameters that the server needs to perform the imperfect replication, as well as the aforementioned patch. The ∆data can be encrypted with the crash collector's public keys to guarantee integrity and confidentiality.

The crash collector server still persistently stores debug information of the *default binary* in Breakpad's existing symbol file format. Our approach requires no changes to that format, so all Breakpad symbol dumper utilities for the major OSs, which simply extract the necessary information from the DWARF or STABS debug sections in ELF object files or from stand-alone PDB (Microsoft's Program Database format) files, still operate out of the box.

In addition, the server persistently stores a *diversity opportunity log*. This log is generated during the *default compilation*, i.e., when the diversifying tool chain is invoked without applying any actual diversification to generate the default binary. It lists all the opportunities for diversification that occurred during the generation of that binary, but that were, per definition, not exploited. For example, it lists all the program points where the diversification process considered (but skipped) inserting NOPs. The essential feature of the diversity opportunity log file is that it lists all decision points where, during an actual diversifying run of the tools, random numbers are drawn from the PRNG.

When a crash report arrives on the server, the ∆*Breakpad replicator* replicates the impact of the complete diversification process on the symbol file in a couple of steps. First, the replicator extracts, decompresses and decrypts the ∆data.

Next, the replicator extracts the random seeds and parameters from the ∆data, and uses them to replicate the impact of the diversification decision process on the *default symbol file* by means of the opportunity log. To that extent, the replicator initializes a PRNG with the same parameters and random seeds that were already used on the build system for the actual diversification of the binary from which the crash report was achieved. The replicator then draws random numbers from that PRNG at each point where the original diversification process had already drawn numbers. For each drawn number, the replicator then adapts the content of the symbol file to reflect approximately what change the diversification step had caused on that file. The overall result of this step is an approximation of the *diversified symbol fil*e, the latter being the symbol file that the original Breakpad symbol dumper tool had produced on the build system for the *diversified binary*.

The resulting symbol file is only an approximation because the replicator only models direct effects of the diversification, such as increased region sizes resulting from inserted NOPs, but none of the secondary effects like the ones we discussed in Section 3.3.2.4. So finally, the replicator extracts the patch from the ∆data and applies it to the approximation, thus reproducing an exact copy of the diversified symbol file.

As the contents of that diversified symbol file matches the contents of the received minidump, the existing Breakpad minidump processor can then be used to produce the human-readable stack trace, which can then be processed as needed. Notice that this stack trace only contains information at the abstraction level of the source code. Crashes occurring in corresponding regions in differently diversified versions of the binaries will hence produce exactly the same stack trace. As such, all existing manual or automatic tools and techniques to analyse and classify the stack traces still work out of the box.

3.3.3.2 Generating the ∆data

The top part of Figure 12 shows the adapted build system. On the right, the standard Breakpad symbol dumper flow is shown to generate the default symbol file to be stored persistently on the crash collector server. This symbol file is extracted from the default binary.

On the left of the build system in Figure 12, the diversified binary is generated, along with the diversification *decision process lo*g that consists of the same info as the opportunity log plus a description of the actual result from the applied diversification, and a *diversified symbol fil*e. Based on this log and symbol file, and on the default symbol file, the ∆*Breakpad symbol differ* then generates the ∆data. Finally, the ∆*data packer* compresses and encrypts the data and injects it as an additional section into the stripped diversified executable. The resulting binary is then distributed to the end user, ready to crash.

3.3.3.3 Combining Multiple Diversification Processes

In order to make the described approach work, we need to ensure that the replication of the decision processes on the crash collector on the basis of the opportunity log generated for the default binary stays synchronized with the decision process as it was executed during the generation of the diversified binary. This is non-trivial when one wants to apply multiple forms of diversification one after the other: As the replication process does not know the exact outcome of an earlier diversification applied to some code fragment, it does not know the exact form of the code fragment onto which the later applied diversification is applied.

For example, suppose randomized padding is injected into a function's stack frame first, and random NOPs are inserted in its code body afterwards, after instruction scheduling has been performed. Given the ordering of compilation phases in a compiler, this is not an unreasonable assumption. As discussed in Section 3.3.2.4, the injected padding can cause changes in the number of instructions of the function body. If this actually happens, and if the later NOP insertion process draws a random number for each instruction in the code to decide whether or not to insert a certain number of NOPs after that instruction, the replicator will draw more or less random numbers from the PRNG than were counted during the generation of the default binary.

So in that case, the replication of the decision process on the crash collector will at some point become desynchronized with how the actual diversification was decided. Unless special care is taken, this will result in completely diverging replication from that point on, which can only be compensated by including a huge patch in the ∆data.

To avoid this, two approaches can be combined. First, the decision processes of the combined diversification schemes need to be carefully designed to become mostly, if not completely independent. We achieve this by applying the later decision processes at a granularity of code fragments that is not likely impacted by earlier decision processes.

Trivially, the order in which functions are shuffled is completely independent on the number of NOPs inserted in them or on their stack padding size.

We also observed that although random stack padding and function shuffling often result in changes in the number of instructions in the function bodies, in particular when the ARMv7 architecture is targeted, they only rarely impact the structure of the functions' control flow graphs. The only occasions in which we saw this happening was when trampolines had to be inserted or could be removed as a result of changed displacements in the code, or when basic blocks became so big or small that they (no longer) had to be split, e.g., to provide space for a literal pool.

We build on this observation by performing the stack padding insertion first, then the function shuffling, and finally the NOP insertion, of which the decision process is performed basic block per basic block, and with a fixed number of random numbers drawn per block. So however the number of instructions in the basic blocks are impacted by the former two diversification steps, as long as the CFG of a function is not impacted, the replicator's decision process will remain synchronized.

To ensure that the few cases in which a function's CFG is actually impacted by the former two diversifications do not result in a desynchronization that spills over into other functions, i.e., to contain the desynchronization, our ∆data format offers a way to resynchronize the decision process, i.e., the number of drawn random numbers, upon entry to a function. The ∆Breakpad symbol differ can easily determine when and where such resynchronization is needed.

3.3.3.4 ∆ Minimization

Our main research goal is to demonstrate that crash reporting for diversified software is feasible with minimal overhead. So we aim for small ∆data. We have opted not to achieve that small ∆data at all cost, however. In particular, we want to make as many of the additional processing steps of our approach as generic as possible. So we opted to design the ∆Breakpad symbol differ, the ∆Breakpad replicator, and the ∆data format to be architecture-independent and compiler-independent.

Furthermore, apart from the restrictions discussed in 3.3.3.3, we do not want to impose strict limitations on the freedom with which to apply the diversification schemes. For example, when we let a compiler select a randomized amount of stack padding for some function, we do not want to restrict its selection to values that preserve the code schedules in the function body. Besides helping us to keep the diversification process decision logic (in the compiler as well as in the replicator) independent of compiler internals, this ensures that the entropy generated by means of the diversification does not depend more than strictly necessary on artefacts of the code being diversified. From the perspective of security, this is obviously an advantage.

Finally, we also want to limit the changes we need to make to existing compilers used for generating and/or diversifying the binaries.

What remains then, to minimize the size of the ∆data, is the selection of the default compilation strategy, and a minimal set of adaptations to the compilation tools to enforce the default compilation.

For the three forms of offset diversification that we evaluated, we only found one generically useful adaptation that can be implemented with minimal changes to the compiler: To generate the default, non-diversified binary, ensure that all functions to which random stack padding will be added during the diversification, get 8 bytes of such padding. During the diversification itself, add 8 bytes or more of padding to all those functions.

There are two reasons why this adaptation is useful. First, because it enforces the insertion of padding in all functions, it limits the number of cases where the code regions in the function prologues and epilogues need to be split as discussed in Section 3.3.2.3. Furthermore, we observed quite some functions where the local area of a stack frame only holds relatively large arrays whose sizes are powers of two. In those functions, the prologues/epilogues contain instructions that increment/decrement the SP with large values of which the least significant bits are all zeroes. Those values can hence be encoded as immediate operands in the ARMv7 and similar architectures. By adding another 8 bytes of padding, a lower bit becomes set as well. So in those functions, the value can no longer be encoded as an immediate operand in the default binary, just like it won't be encoded as an immediate operand in the diversified binary. The average difference between the default binary and the diversified binaries, and hence the average amount of information to be stored in the ∆data, is hence reduced. For other functions, such as those with small local areas, the added 8 bytes typically don't change anything. But there the insertion of a (limited) amount of random padding does not change anything either.

An additional advantage is that this adaptation can be implemented very easily in a (diversifying) compiler, as it is completely architecture-independent. In, e.g., a diversifying version of the LLVM 3.6.2 we only needed a 3-line patch.

In addition, we also adapted the LLVM ARM back-end, with a one-line patch that disables the optimization in which accesses via the FP are selected over accesses via the SP depending on the offsets of a stack location to those two base registers. This patch also reduces the number of changes introduced in the diversified code as a side-effect as discussed in Section 3.3.2.4.

3.3.3.5 Profile-Guided Diversification

Some forms of diversification can benefit from profile information to reduce the performance overhead. For example, as it is typically not necessary to insert a random number of NOPs in between every pair of instructions, the performance overhead of NOP-insertion can be reduced by concentrating NOPs on infrequently executed program points.

Our approach supports such profile-guided diversification without any problem: as long as both the default compilation and the diversifying compilation runs are served the same profile information, the decision process logs and the diversity opportunity log will be consistent with each other, so the ∆Breakpad replicator will work just fine.

3.3.4 Experimental Evaluation

3.3.4.1 Experimental Setup

As we want to demonstrate that our approach can work with acceptable ∆data sizes, we evaluated it on the more challenging ARMv7 architecture. In particular, our proof-of-concept implementation supports the 32-bit subset of the ARMv7-A architecture. So in our experiments, we do not target 16-bit Thumb or Thumb2 code.

To ease our research, the uncompressed ∆data we generate consists of simple human-readable ASCII text files. With more but relatively simple engineering, smaller patch sizes can be obtained. So any ∆data sizes we report, i.e., sizes of the ASCII text files, are upper bounds on what could be achieved with a more mature implementation. Moreover, there size would still be reduced if they are compressed, e.g., with bzip.

As noted in literature, diversification processes can be applied at many stages during the SDCL. To demonstrate that we can actually support combinations applied at different stages, we opted for the following tool flow.

First, we adapted LLVM 3.6.2 to apply randomized stack padding. In our prototype implementation, all functions get a random stack padding between 8 and 256 bytes, but always a multiple of 8 bytes. This ensures that the generated code keeps respecting the ARM EABI, which requires stack frames of non-leaf functions to be aligned on 8-byte boundaries.

Next, we use the standard GNU linker to perform function shuffling. In preparation, we use the -ffunction-section compiler flag to ensure that each function is put into a separate code section in the generated object files. To perform the actual shuffling, we simply generate a linker script that enforces a shuffled order of all the code sections, and hence of all functions. This way, we only use existing functionality, and we did not need to apply any patch to the GNU code base.

Finally, we use the post-link-time binary code rewriter Diablo (http://diablo.elis.ugent.be) to perform randomized NOP insertion, implementing a decision process as discussed in Section 3.3.3.3.

As a result of using three separate tools, our prototype implementation functions slightly differently from the approach presented in Section 3.3.3.3. In particular, the Δdata consists of two parts, and the ΔBreakpad replicator correspondingly applies not just one decision process replication followed by one patching phase, but two iterations consisting of a round of replication and a round of patching.

3.3.4.2 Benchmarks and Correctness

For evaluating and testing the ΔBreakpad approach, we relied on C and C++ benchmarks from the SPECint 2006 benchmark suite. We compiled the benchmarks with a patched LLVM 3.6.2 to deploy stack padding (with flags -g -O2 -ffunction-sections -fomit-framepointer) used the binutils ld linker version 2.23.2 with customized, randomized linker scripts to perform the function shuffling, and used Diablo to apply the NOP insertion. The produced, diversified benchmarks were executed on the test inputs to verify their correctness.

3.3.4.3 Overhead

Table 1 presents the measurements results on the SPECint benchmarks we used. It is clear that the Δdata is in the order of kilobytes, even for the largest benchmarks of which the binaries are multiple MB large. It can also be seen that the diversification itself adds little size overhead, and that the sizes of the extra files to be stored in the crash reporting server are acceptable. In particular, the opportunity log adds in between 5% and 20% to the server-side data (i.e., the data to be stored there on top of the default symbol file).

Table 1: Size overheads for crash reporting

	file sizes (bytes)												
	stack padding		function shuffling		NOP insertion		three diversifications combined						
Benchmark	Δdata (average)	Δdata (max)	Δdata (average)	Δdata (max)	Δdata (average)	Δdata (max)	Δdata (average)	Δdata (max)	opportunity log	default symbol file	diversified symbol file (average)	stripped default binary	average stripped diversified binary
perlbench	225	285	142	180	510	1394	877	1859	365618	1831701	1831377	1083072	1083087
bzip2	274	480	40	55	347	451	661	986	15299	93199	93205	77080	77080
gcc	1495	2994	378	410	2235	3582	4109	6986	1164354	5403175	5409708	3153972	3154145
mcf	80	83	15	18	0	0	95	101	2355	20086	20084	17032	17032
milc	127	130	44	61	0	0	170	191	29588	226073	225544	133976	133977
namd	111	207	285	312	16	48	412	567	48542	349884	350079	255356	255387
gobmk	1125	1448	169	190	2245	2474	3539	4112	214270	1557066	1556648	3298912	3298912
soplex	889	979	258	315	157	470	1304	1764	95159	816099	815740	349300	349358
povray	1479	2633	794	876	1861	2031	4134	5540	234612	1853193	1854007	922332	922344
hmmer	254	413	266	303	40	119	559	835	86187	509540	509726	284796	284817
sjeng	493	913	29	38	313	509	835	1460	33796	207938	208293	148684	148660
libquantum	58	59	50	67	0	0	108	126	8300	64305	64333	43224	43013
h264ref	434	475	239	276	55	58	728	809	116478	1078908	1079615	623380	623380
lbm	66	67	22	32	0	0	88	99	1011	21048	21037	14444	14444
omnetpp	163	189	263	308	260	780	687	1277	114649	1007846	1008165	648160	648160
astar	94	126	49	60	0	0	143	186	8155	77388	77307	39220	39219
sphinx3	339	540	93	118	318	381	750	1039	48310	284746	284306	181572	181572

Extensive performance measurements are still being conducted. We can already report, however, that the performance overhead of using diversification is small on the user system.

3.3.5 Related Work

In the past, both spatial and temporal software diversity has been proposed as a solution to a wide range of problems: Instruction set randomization can prevent, or at least delay, reverse-engineering and tampering [Wil09]. Multi-variant execution can be used to detect malware intrusions [Vol15]. Limited, rather coarse-grained forms of run-time randomization, such as address space layout randomization (ASLR), are widely used and significantly raise the bar for memory corruption attacks [PAX04]. In the academic literature, more fine-grained forms of diversification have been proposed to raise the bar even further [Kil06,Giu12], including for code dynamically generated with JIT compilers [Hom13]. Dynamic temporal diversity has been proposed to mitigate timing side channel attacks [Cra15]. Advanced software fingerprinting schemes can help in identifying the source of illegitimate software copies [Col07]. Diversification can prevent collusion attacks to identify software vulnerabilities based on patches [Cop13]. Some software vendors diversify the code of their applications when major new versions are released, to hide the location of the new, valuable functionality in the new versions. Obfuscation tools and other software protection tools inherently rely on diversification to minimize the learning capabilities of attackers and to achieve stealthiness [Col09]. Microsoft diversifies the Window's system call numbering over time to prevent (malicious and benign) software targeting APIs they do not want to keep backwards compatible [Jur].

With the exception of the latter form of diversification, the other forms can only provide strong protection if code is diversified, i.e., if the diversification is not limited to changes in the embedded data.

3.3.6 Conclusions

With ∆Breakpad, we have demonstrated that it is possible to diversify software in space, i.e., distribute different versions of software to different users, and still obtain accurate crash reports without too much overhead. It suffices to include a couple if kilobytes of extra data in the binaries, and to send that data to the crash server along with the stack trace to restore a debug report on the crash server. As such, we have brought the distribution of diversified software one step closer to commercial viability.

3.4 Validation of Metrics for maximizing software diversity

Chapter Authors: Alessandro Cabutto, Paolo Falcarin (UEL), Mariano Ceccato (FBK)

D3.08 Section 4 reports our experimental results about maximizing software diversity. After that we improved the analysis with a validation of metrics used in our work. Following sub-sections contain considerations about NCD, its applicability to the experimentation context and some usage guidelines.

3.4.1 Effect of Different Code Representation

As a black-box metric, NCD requires no knowledge about the content of the files to compare. In fact, the semantics of the compared files is not considered. The compared files are never parsed, they are just read and compressed, possibly after concatenation (see Equation 1).

$$NCD(v_1, v_2) = \frac{C(v_1 v_2) - \min\left(C(v_1), C(v_2)\right)}{\max\left(C(v_1), C(v_2)\right)}$$

Equation 1: Normalized Compression Distance formula

As such, several different code representations could be potentially considered when computing NCD, and they might be not equivalent.

We consider several representation of the code. They are:

- **Jar**: This is the Java bytecode. We obtain Java bytecode from Android apps by converting their Dalvik bytecode with the dex2jar tool [dex2jax];
- **Javap**: The Disassembled textual representation of the Java bytecode. We obtain this representation by executing the javap disassembler, which is part of the official Java Development Kit, distributed by Oracle. Disassembled code contains the instructions that comprise the Java bytecode, for each of the methods in a class;

- **Java**: Decompiled Java source code, obtained by running the jd Java Decompiler [jd] using jd-cmd [jd-cmd] command line tool.

Alternative representation should be considered, because an attacker might be working on any of them to elaborate, reuse or adapt an attack. In fact, Android apps are distributed as Dalvik bytecode, a high level language that is quite easy to convert with popular tools (as those used in our experimental setting). After conversion, an attacker might work on the representation that is more appropriate to her/his experience.

Name	Category	Downloads	Score	Size (kB)	Classes
Airdroid	Utility	10-50M	4.5	2.000	2.165
Chrome	Internet Browser	500-1.000M	4.3	3.484	3.069
ESX-File Explorer	Utility	500-100M	4.6	4.720	4.107
Go Tetris	Game	10-50M	4.3	163	190
Opera	Internet Browser	100-500M	4.4	267	185
Twitter	Social Network	100-500M	4.1	747	634

Table 2: Subject apps considered in the experimental study.

As case studies, we consider real world Android apps. We select 6 from the most popular apps as ranked in the official Android store, namely Google Play (data collected in 2013). They spread on different categories (utility, social network, games, internet browser) and their popularity goes from 10 to 500 millions of downloads. Their size is between 100kB to almost 5MB. The smallest apps contain about 200 classes, while the largest apps contain about 4,000 classes. Table 2 lists the apps considered in this study, with their category, popularity (number of time they have been downloaded), the score given by users (i.e. number of stars in the range [0-5]) and their size and the number of classes in the app.

We generate many different versions of each app using Zelix KlassMaster8 a commercial obfuscation tool for Java and Android. Zelix KlassMaster supports 15 distinct configuration parameters to control which transformations are activated and how they are configured. For each app, we obtain 15 variants by simply running this tool on the original app 15 times, each time with a distinct parameter activated.

3.4.1.1 Analysis of NCD Distribution

To study the impact of the representation used to compute NCD, we compute NCD among pairs of the same app obfuscated in different versions.

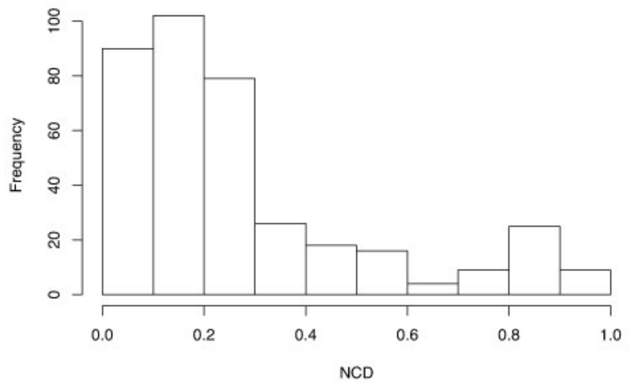


Figure 13: Histogram of NCD based on *Jar* representation.

Figure 13 shows the histogram of the distribution of NCD values when computed on the Java bytecode. NCD values are concentrated on the left-hand size of the graph, with the majority of them around 0.2. Generally low distance might be due to a number of reasons. Maybe the adopted obfuscation is ineffective in generating quite different versions, or the Java bytecode format might include common recurring structures that (independently from the classes content) make versions all quite similar.

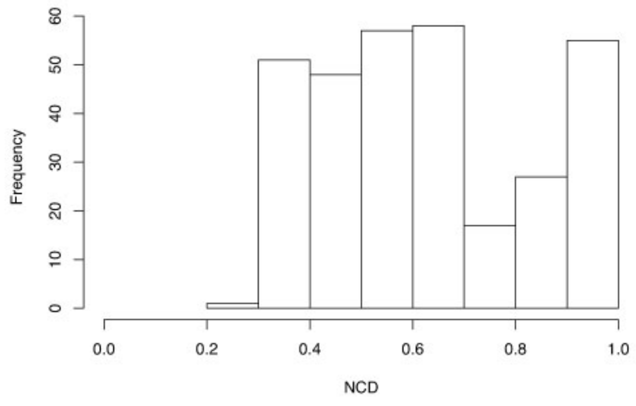


Figure 14: Histogram of NCD based on *Javap* representation.

Figure 14 shows the histogram of the distribution of NCD value computed on the disassembled Java bytecode. Differently from the previous case, now most of the NCD values populate the right-hand side of the graph, suggesting that higher values of NCD can be reached on average. Thus, we can exclude that the adopted obfuscating tool is ineffective.

Eventually Figure 15 shows the histogram of the distribution of NCD values computed on the decompiled Java source code. Decompiled code seems to have a trend similar to dis-assembled code, with most of the values in the right-hand side of the graph. In the following, we will Investigate these NCD distributions are not just similar but also correlated.

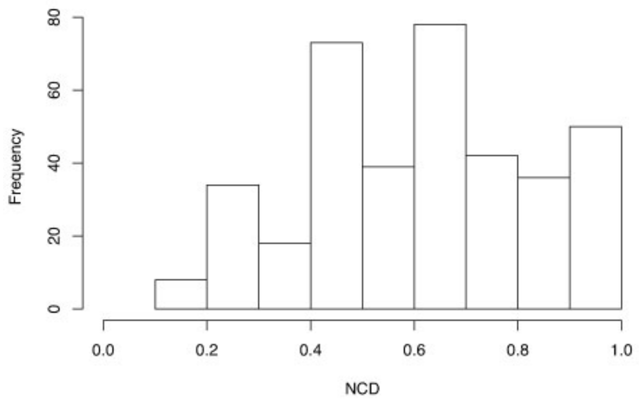


Figure 15: Histogram of NCD based on *Java* representation.

Table 3 reports the corresponding descriptive statistics (median and standard deviation). Values in this table confirm that NCD computed on bytecode (Jar) tends to assume low values, while NCD computed on the textual bytecode representation (Javap) and on the Java source code (Java) on average assumes higher values. The comparison of this trend is also evident in the boxplot of NCD reported in Figure 16.

Representation	**Mean**	**SD**
jar	0.28	0.24
javap	0.63	0.21
java	0.61	0.22

Table 3: Descriptive statistics

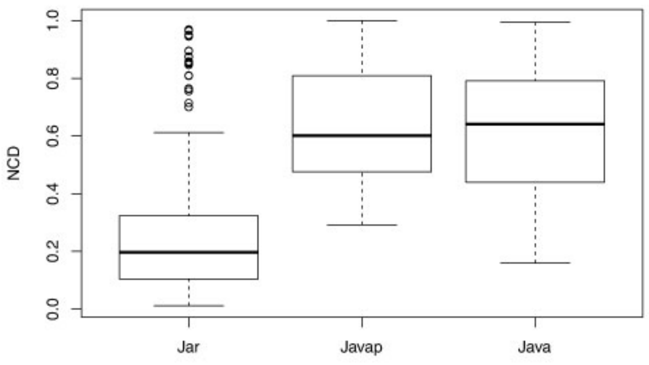


Figure 16: Boxplot of NCD based on different program representation.

3.4.1.2 Analysis of NCD Correlation

Then, we study if there is any correlation between NCD computed on different code representations. To this aim, we use the Pearson correlation test, available from the R statistical package [r-stat]. This test computes the correlation coefficient r, a symmetric, scale-invariant measure of association between two random variables. It ranges from −1 to +1, where the extremes indicate perfect (positive or negative) correlation and 0 means no correlation. The correlation is considered significant when the p-value is <0.05 (assuming a confidence of 95%). Significant cases will be reported in boldface.

Table 4 shows the results of the analysis of correlation of NCD among all the pairs of representations. As we can see, the only case of statistically significant correlation is between the bytecode representation (Jar) and the Java source code representation (Java) with a correlation coefficient r = 0.75.

	Javap		Java	
	Correlation	P-value	Correlation	P-value
Jar	0.07	0.19	**0.75**	**<0.01**
Javap	-	-	0.07	0.24

Table 4: Analysis of correlation (Pearson correlation test).

This result suggests that using Jar or Java to compute NCD would lead to overall similar research considerations, because the two distance metrics are correlated.

Based on these results, we can formulate the following:

NCD computed on the disassembled Java bytecode (Javap) and on Java code (Java) assumes higher values than NCD computed the Java bytecode (jar). NCD computed on the Java bytecode (Jar) and computed on the decompiled Java source code (Java) are highly correlated, however the former one tends to contains lower values of distance than the later one. NCD computed on the disassembled Java bytecode (Javap) are not correlated with NCD computed on the other representations.

3.4.1.3 Guidelines

Based on the experimental results and observations, we can formulate the following guidelines for NCD users:

The size of the files to compare matters. When using NCD, researchers and practitioners should pay attention to the size to the files that they are measuring. In fact, the most common compression algorithms, namely gzip and bzip2, involve serious limitations to the file size, respectively 32Kbytes and 470Kbytes (see D3.08 Section 4.2.2). Depending on the adoption domain, the most common compression algorithms could be still usable. However, in our case, we had to measure distance among Android apps, whose code size clearly exceeds these limits. Thus, the only option was to compute NCD with rzip as compression algorithm, because its validity interval was compatible with our domain.

Validate NCD before using it. Many more compression algorithms are available to researchers and practitioners, and all of them can be potentially used to compute NCD. We recommend adopting the approach described in D3.08 Section 4.2.2 to validate new compression algorithms before using them for NCD. Validity intervals should be identified and respected in the particular adoption context.

NCD on compiled code is lower than NCD on textual code. The difference in code representations involves a difference in the distribution of distance values. In particular, NCD tends to assume lower values when computed on compiled Java bytecode, and higher values on its corresponding textual representation, disassembled or decompiled code (see analysis of NCD distribution, Section 3.4.1.1). Researchers and practitioners should be careful in evaluating the performance of their diversifying tools, when they tools generate compiled programs with low distance values. In fact, when considering compiled code, NCD values are, on average, quite low. This trend is probably connected to the particular structure of compiled code, that is subject to distance measurement.

NCD on Java bytecode is correlated with NCD on Java source code. Decompiled code represents an alternative to compute distance among programs that is highly correlated with distance on compiled bytecode (see analysis of NCD correlation, Section 3.4.1.2). When source code is not available or not easy to recover, using the compiled bytecode represents a good proxy (high correlation). Even if they are highly correlated, NCD based on compiled bytecode assumes on average low values, while NCD based on decompiled source code assumes quite high values. This difference in distributions of values could be useful when NCD is used in analysis algorithms that are sensitive or problematic when distance values are near to 0 or near to 1.

3.4.1.4 Conclusions

NCD has been used to measure difference between programs. Despite this metrics looks intuitive and easy to implement, it involves dangerous pitfalls. We showed that the interval of validity of NCD depends on the underlying compression algorithm, which could limit this metrics to files whose size is of few kilobytes or of many megabytes. Moreover, NCD computed on Java source code is highly correlated to the NCD computed on Java bytecode, but little correlated with NCD computed on disassembled Java bytecode.

Section 4 Conclusions

Chapter Authors:

Alessandro Cabutto, Paolo Falcarin (UEL), Bjorn DeSutter (Ugent), Cataldo Basile (POLITO)

The work of the ASPIRE consortium in the online protections work-package (WP3) has achieved more than the foreseen objectives, by delivering through the project many innovative results: the design, development, implementation and integration of code & data mobility; client-server code splitting; static, dynamic, and implicit remote attestation; anti-cloning; and several forms of renewability (incl. native code diversity in space and time, WBC renewability and bytecode diversification).

The Renewability Framework has been fully integrated with ACTC and successfully tested against use cases, relying on the code mobility framework to provide diversity in time deployment features to the ASPIRE Framework; it has been applied to some of the ASPIRE online protection techniques:

- bytecode and SoftVM diversification, such that each software instance can have a custom bytecode and interpreter to limit the learnability of the client-side code splitting protection;
- mobile code diversity, such that mobile code can vary from one program execution to another, again to limit the learnability of the code and the ease to collect and combine multiple execution traces of the same program;
- white-box crypto renewability, such that by delivering re-randomized keys by combining code and data mobility, attackers can only reuse broken keys for very short time frames.

In parallel with the aforementioned renewable forms of protections, code diversity in space was further researched: one on error reporting for diversified code, and another on diversity maximization (whose results has been published in a conference paper [ssbse16]).

Static Remote Attestation and anti-cloning protections had been integrated into the ACTC.

Implicit Remote Attestation (IRA) has been fully designed based on the Dynamic Remote Attestation (DynRA), as a technique that monitors the correct execution of an application based on its invariants. DynRA has been improved, extended, and tested on an open source application; unfortunately, it has not been tested on use cases as no tools are available for the ARM platforms to extract traces compatible with the tool for invariants extractors (Daikon).

Control Flow Tagging (CFT) has been implemented and integrated with ACTC and used in combination with other online and offline protection techniques.

Reaction Mechanisms are used to degrade the application when tampering has been detected. They are typically called by the CFT protection when a CFT attestation is not verified successfully, and they have been applied on the OTP use case.

Reactive attestation is another form of reaction, based on the cooperation between remote attestation and client-server code splitting, which transforms applications to make them de-pendent on the server, applies remote attestation to detect modification, and thus reacts by stopping serving compromised applications: a peer-reviewed paper on this combination of protections has been published [raadrst].

Support for occasionally connected scenarios was supposed to rely on the integration between the SFNT SoftVM, code mobility, and the Implicit Remote Attestation (IRA) technique. The composability analysis, at the end of year 2, has showed that the techniques are not composable: the main reason is that IRA and code mobility would need a VM that emulates an entire physical machine to support occasionally connected scenarios and temporarily substitute the real server. However, the SFNT SoftVM cannot be used to this purpose, being a protection technique to be injected into existing applications.

However, we exceeded the expectations by providing additional solutions not foreseen in the DoW: fully automated static and dynamic remote attestation, reactive attestation, integration of code mobility and remote attestation, IRA was implemented with invariants monitoring and even applied to the use-cases.

Section 5 List of Abbreviations

ACTC	ASPIRE Compiler Tool Chain
AID	ASPIRE Application ID
ALU	Arithmetic- Logic Unit
ARM	Not an acronym, only a company name and its architecture
ASPIRE	Advanced Software Protection: Integration, Research and Exploitation
CFG	Control Flow Graph
CFT	Control Flow Tagging
DynRA	Dynamic Remote Attestation
DynRA-DS	Dynamic Remote Attestation Data Structure
EABI	Embedded Application Binary Interface
FP	Frame Pointer
IRA	Implicit Remote Attestation
LD/ST	Load/Store
LLVM	Low-Level Virtual Machine (now just the name of the compiler project)
NCD	Normalized Compression Distance
NOP	No-Operation
OS	Operating System
PC	Program Counter
PRNG	Pseudo Random Number Generator
RA	Remote Attestation
SP	Stack Pointer
SQ	Similarity Quality
WBC	White-Box Cryptography

Bibliography

[Bau15] B. Baudry and M. Monperrus, “The multiple facets of software diversity: Recent developments in year 2000 and beyond,” ACM Comput. Surv., vol. 48, no. 1, pp. 16:1–16:26, Sept. 2015. [Online]. Available: http://doi.acm.org/10.1145/2807593.

[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

[Coh93] F. B. Cohen, “Operating system protection through program evolution,” Computers & Security, 1993.

[Col07] C. Collberg, C. Thomborson, and G. M. Townsend, “Dynamic graph-based software fingerprinting,” ACM Trans. Program. Lang. Syst., vol. 29, no. 6, Oct. 2007. [Online]. Available: http://doi.acm.org/10.1145/1286821.1286826

[Col09] C. Collberg and J. Nagra, Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection, 1st ed. Addison-Wesley Professional, 2009.

[Cop13] B. Coppens, B. De Sutter, and K. De Bosschere, “Protecting your software updates,” Security Privacy, IEEE, vol. 11, no. 2, pp. 47–54, March 2013.

[Cra15] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, “Thwarting cache side-channel attacks through dynamic software diversity,” in Proceedings of the Network and Distributed System Security Symposium, Feb 2015.

[For97] S. Forrest, A. Somayaji, and D. Ackley, “Building diverse computer systems,” in Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI), ser. HOTOS ’97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 67–72. [Online].

[Hom13] A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, “Librando: transparent code randomization for just-in-time compilers,” in Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, ser. CCS ’13. New York, NY, USA: ACM, 2013, pp. 993–1004. [Online]. Available: http: //doi.acm.org/10.1145/2508859.2516675

[dex2jax] Dex2jar tool. On-line at https://sourceforge.net/projects/dex2jar/

[Giu12] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, “Enhanced operating system security through efficient and fine-grained address space randomization,” in Proceedings of the 21st USENIX Conference on Security Symposium, ser. Security’12. Berkeley, CA, USA: USENIX Association, 2012, pp. 40–40. [Online]. Available: http://dl.acm.org/citation.cfm?id=2362793.2362833

[jd] Java Decompiler project. On-line at http://jd.benow.ca/

[jd-cmd] Command line Java Decompiler. On-line at https://github.com/kwart/jd-cmd

[Jur] M. Jurczyk, “Windows X86 system call table (NT/2000/XP/2003/Vista/2008/7/8),” [Online] Available: http://j00ru.vexillium.org/ntapi/.

[Kil 06] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning, “Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software,” in Proceedings of the 22Nd Annual Computer Security Applications Conference, ser. ACSAC ’06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 339–348. [Online]. Available: http://dx.doi.org/10.1109/ACSAC.2006.9

[Lar14] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, “SoK: Automated software diversity,” in Proceedings of the 2014 IEEE Symposium on Security and Privacy, ser. SP ’14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 276–291. [Online].

[Lar15] P. Larsen, S. Brunthaler, and M. Franz, “Automatic software diver- sity,” Security Privacy, IEEE, vol. 13, no. 2, pp. 30–37, Mar 2015

[Mov13] MovieLabs Specification for Enhanced Content Protection – Version 1.0, Motion Picture Laboratories, Inc., 2013.

[PAX04] PaX Team, “Address space layout randomization,” http://pax.grsecurity.net/docs/aslr.txt, 2004.

[r-stat] R Core Team. R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria, 2015.

[raadrst] Alessio Viticchié, Cataldo Basile, Andrea Avancini, Mariano Ceccato, Bert Abrath, and Bart Coppens. 2016. Reactive Attestation: Automatic Detection and Reaction to Software Tampering Attacks. In *Proceedings of the 2016 ACM Workshop on Software PROtection* (SPRO '16). ACM, New York, NY, USA, 73-84. DOI: http://dx.doi.org/10.1145/2995306.299531

[rzip] RZip algorithm. On-line at https://en.wikipedia.org/wiki/Rzip

[ssbse] M Ceccato, P Falcarin, A Cabutto, YW Frezghi, CA Staicu, “Search Based Clustering for Protecting Software with Diversified Updates”. In International Symposium on Search Based Software Engineering (SSBSE-2016), pp 159-175, Springer.

[Vol15] S.Volckaert, B.Coppens,andB.DeSutter,“Cloning your gadgets: Complete ROP attack immunity with multi-variant execution,” Dependable and Secure Computing, IEEE Transactions on, vol. PP, no. 99, pp. 1–1, 2015.

[Wil09] D. Williams, W. Hu, J. W. Davidson, J. D. Hiser, J. C. Knight, and A. Nguyen-Tuong, “Security through diversity: Leveraging virtual machine technology,” IEEE Security and Privacy, vol. 7, no. 1, pp. 26–33, Jan. 2009. [Online]. Available: http://dx.doi.org/10.1109/MSP.2009.18