Advanced Software Protection:
Integration, Research and Exploitation

D2.06

Binary Code Obfuscation Report

Project no.:	609734
Funding scheme:	Collaborative project
Start date of the project:	1st November 2013
Duration:	36 months
Work programme topic:	FP7-ICT-2013-10
Deliverable type:	Report
Deliverable reference number:	ICT-609734 / D2.06
WP and tasks contributing:	WP 2 / Tasks 2.4
Due date:	Apr 2015 – M18
Actual submission date:	13 May 2015
Responsible Organization:	UGent
Editor:	Bart Coppens
Dissemination Level:	Public
Revision:	DRAFT

Abstract:
This deliverable presents the support in the ASPIRE compiler tool chain for binary code obfuscation, as implemented and delivered in D2.05 until M18.

Keywords:
binary code, obfuscation

Editor

Bart Coppens (UGent)

Contributors (ordered according to beneficiary numbers)

Bjorn De Sutter (UGent)

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

UGent ported the existing x86 control flow obfuscation support in its Diablo Background software (control flow flattening, branch functions, and opaque predicates) to the ARMv7 architecture targeted in the ASPIRE project. The obfuscation support was restructured into architecture-dependent and architecture-independent software layers, and effort was invested in making the obfuscations more flexible, more generally applicable, and better controllable by the ACTC. To that extent, support for the ASPIRE source code annotations was implemented.

In addition, the control flow obfuscations are combined with code factoring and code layout randomization.

Extensive correctness testing on multiple platforms (Linux + Android, x86 + ARM) has been performed, and the obfuscations have been evaluated and shown to effectively disrupt static reverse engineering tools such as IDA Pro.

Contents

List of Figures

Section 1 Introduction

The goal of this deliverable (see GA Annex II DoW part A) is to document the initial tool support for the binary code obfuscations delivered in ASPIRE's Work Package 2, Task 2.4. This tool support is implemented in the Diablo link-time rewriter, and is itself delivered as a prototype in Deliverable D2.05, on which this deliverable reports.

This binary code obfuscation support is currently integrated into the ASPIRE Compiler Tool Chain (ACTC), which is being implemented for Task T5.1.

The protections implemented for this deliverable are control flow obfuscations applied at the level of assembly instructions. These obfuscations are inserted in the binary to thwart reverse engineering and analyses of the assets protected by the ASPIRE tool flow. These Diablo-based obfuscations are inserted in the protected binary in protection step BLP04 in the ACTC, as described in Deliverable

Even though this task officially only started in M12, to be delivered in M18, work on this deliverable started earlier in order to facilitate the integration and debugging effort.

The remainder of this report first discusses in detail the binary control flow obfuscations implemented and how they were implemented on top of UGent's Diablo Background. Next, we discuss different aspects of the integration of the Diablo-based tools in the ACTC. Finally, we provide some details on how we validated the correctness and robustness of the provided obfuscation transformations.

Section 2 Binary Protections

We implemented several binary obfuscation techniques for ARM binaries: control flow flattening, branch functions, and opaque predicate insertion. Additionally, identical code fragments are factored out as an additional anti-reverse engineering step. Furthermore, we introduced code layout randomization to increase the effectiveness of the obfuscation transformations.

In addition to the obfuscations themselves, we also added support for integrating these obfuscations in the ASPIRE tool chain. In particular, we added extensive logging support for the obfuscation transformations, and we made the insertion of obfuscations configurable through a JSON annotation fact file, which is input D01 to BLP04 of the ACTC.

2.1 Control flow flattening

Control flow flattening is an obfuscation technique that transforms a control flow graph with a clean structure into one in which all control flow is redirected through a single switch block. This technique was originally introduced by Wang et al [Wang00].

Figures 1 and 2 demonstrate the effect of this transformation on a simple example. The control flow of Figure 1 is easy to follow by studying the structure of this graph, even by an untrained reverse engineer. Figure 2 shows the same function after applying control flow flattening. All control flow is redirected through the single switch block at the top. To figure out the exact control flow, studying the structure of the control flow no longer suffices, and reverse engineers have to analyse the code in order to determine the structure of the function.

The existing implementation in Diablo was an x86-only implementation [Mad07]. This implementation used hard-coded registers for all computations, rather than trying to use registers that are dead throughout the flattened function.

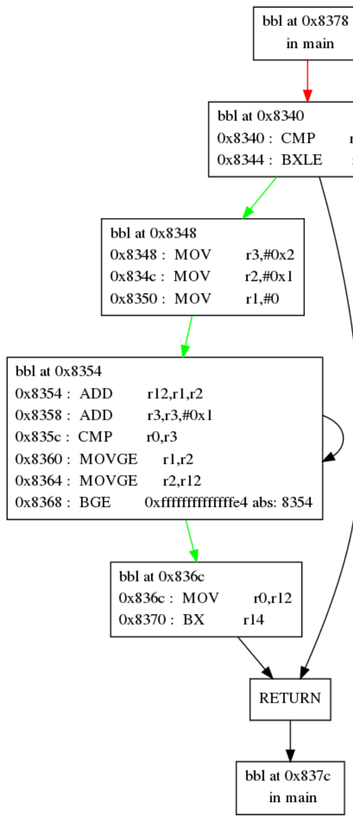
We implemented this functionality starting from this pre-existing implementation for x86 code. The existing implementation was first refactored into a generic part, and an architecture-specific part, so that the code is as generic and architecture-independent as possible. Next, the ARM-specific control flow flattening transformation was implemented on top of this. At this time, Diablo supports flattening the basic blocks of a single function at a time.

The registers used by the switch blocks are now randomly chosen from the set of registers that are dead over all flattened edges. Only when no such registers are found, live registers are chosen at random, and these registers are saved and restored on the stack by the flattening code.

2.2 Branch functions

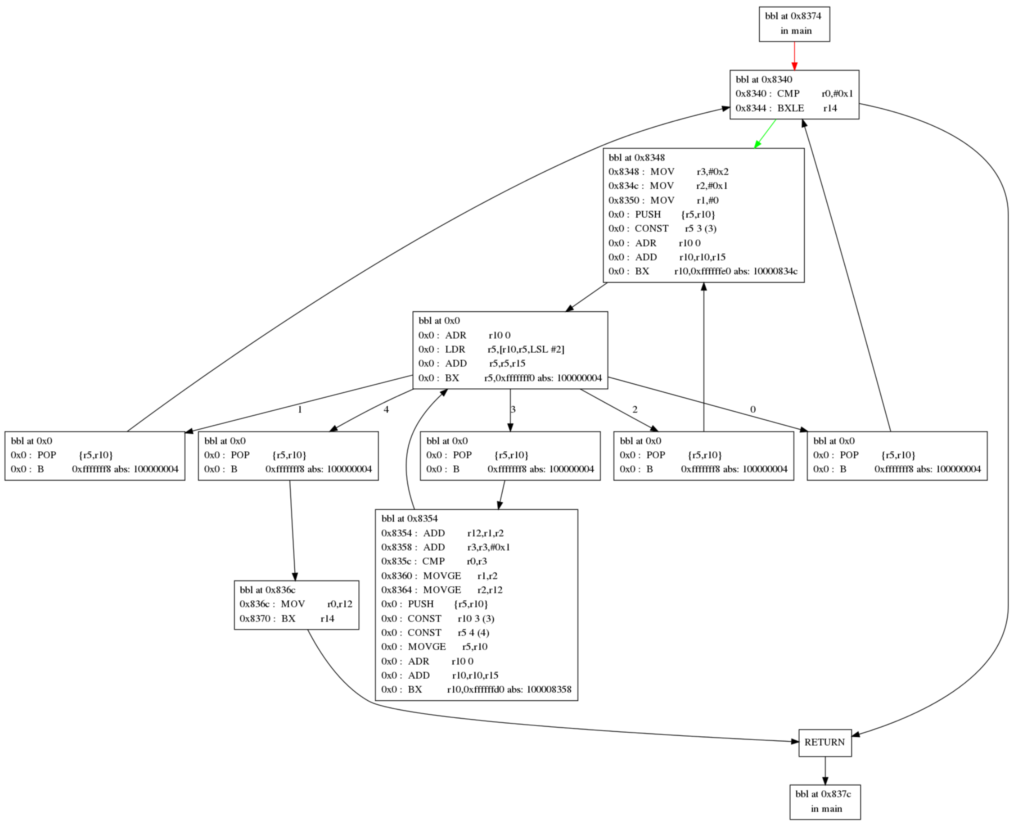
Branch functions transform direct control flow into indirect control flow, using a call to a newly inserted branch function. This implies that jumps are computed at run-time rather than hard-coded into the code. This indirection decreases the precision of the analyses applied by reverse engineers, and increases the overall effort to understand the protected binary. This technique was introduced by Linn and Debray [Linn03].

The previous implementation of branch functions in Diablo consisted of an x86-only implementation [Mad07]. In this previous implementation, the argument to the branch function is an offset relative to the continuation point of the call to the branch function. The branch function itself is not randomized and not optimized: it uses a fixed register, and always spills and restores this register and the flags register to the stack.



```
bbl at 0x8378
  in main

bbl at 0x8340
0x8340 :  CMP       r0,#0x1
0x8344 :  BXLE      r14

bbl at 0x8348
0x8348 :  MOV       r3,#0x2
0x834c :  MOV       r2,#0x1
0x8350 :  MOV       r1,#0

bbl at 0x8354
0x8354 :  ADD       r12,r1,r2
0x8358 :  ADD       r3,r3,#0x1
0x835c :  CMP       r0,r3
0x8360 :  MOVGE     r1,r2
0x8364 :  MOVGE     r2,r12
0x8368 :  BGE       0xffffffffffffffe4 abs: 8354

bbl at 0x836c
0x836c :  MOV       r0,r12
0x8370 :  BX        r14

RETURN

bbl at 0x837c
  in main
```

Figure 1: Toy code example of a function before control flow flattening



```
bbl at 0x8374
  in main

bbl at 0x8340
0x8340 :  CMP       r0,#0x1
0x8344 :  BXLE      r14

bbl at 0x8348
0x8348 :  MOV       r3,#0x2
0x834c :  MOV       r2,#0x1
0x8350 :  MOV       r1,#0
0x0 :  PUSH       {r5,r10}
0x0 :  CONST      r5 3 (3)
0x0 :  ADR        r10 0
0x0 :  ADD        r10,r10,r15
0x0 :  BX         r10,0xffffffe0 abs: 10000834c

bbl at 0x0
0x0 :  ADR        r10 0
0x0 :  LDR        r5,[r10,r5,LSL #2]
0x0 :  ADD        r5,r5,r15
0x0 :  BX         r5,0xfffffff0 abs: 100000004

bbl at 0x0
0x0 :  POP        {r5,r10}
0x0 :  B          0xfffffff8 abs: 100000004

bbl at 0x0
0x0 :  POP        {r5,r10}
0x0 :  B          0xfffffff8 abs: 100000004

bbl at 0x0
0x0 :  POP        {r5,r10}
0x0 :  B          0xfffffff8 abs: 100000004

bbl at 0x0
0x0 :  POP        {r5,r10}
0x0 :  B          0xfffffff8 abs: 100000004

bbl at 0x0
0x0 :  POP        {r5,r10}
0x0 :  B          0xfffffff8 abs: 100000004

bbl at 0x836c
0x836c :  MOV       r0,r12
0x8370 :  BX        r14

bbl at 0x8354
0x8354 :  ADD       r12,r1,r2
0x8358 :  ADD       r3,r3,#0x1
0x835c :  CMP       r0,r3
0x8360 :  MOVGE     r1,r2
0x8364 :  MOVGE     r2,r12
0x0 :  PUSH       {r5,r10}
0x0 :  CONST      r10 3 (3)
0x0 :  CONST      r5 4 (4)
0x0 :  MOVGE      r5,r10
0x0 :  ADR        r10 0
0x0 :  ADD        r10,r10,r15
0x0 :  BX         r10,0xffffffd0 abs: 100008358

RETURN

bbl at 0x837c
  in main
```

Figure 2: Toy code example of a function after control flow flattening

For ASPIRE, we added support for a similar form of branch functions for the ARMv7 back-end. This branch function is similar to the pre-existing implementation in the x86 backend. It consists of a PC-relative indirect jump. The PC-relative target address is passed in a randomized register, which is chosen from the dead registers when dead registers are available. If necessary, a dead register is created by spilling a live register onto the stack.

A toy code example is shown in Figure 3. Compared to the code in Figure 1, a branch to the branch function has been inserted in the first basic block of the function (with start address `0x8340`). No spill code for the target stored in `r3` was inserted, but for the return address register `r14`, spill code needed to be inserted. The (unreachable) continuation point of the call to the branch function is left unspecified. Thanks to layout randomization, this continuation point is randomly chosen from all possible continuation points across the binary.

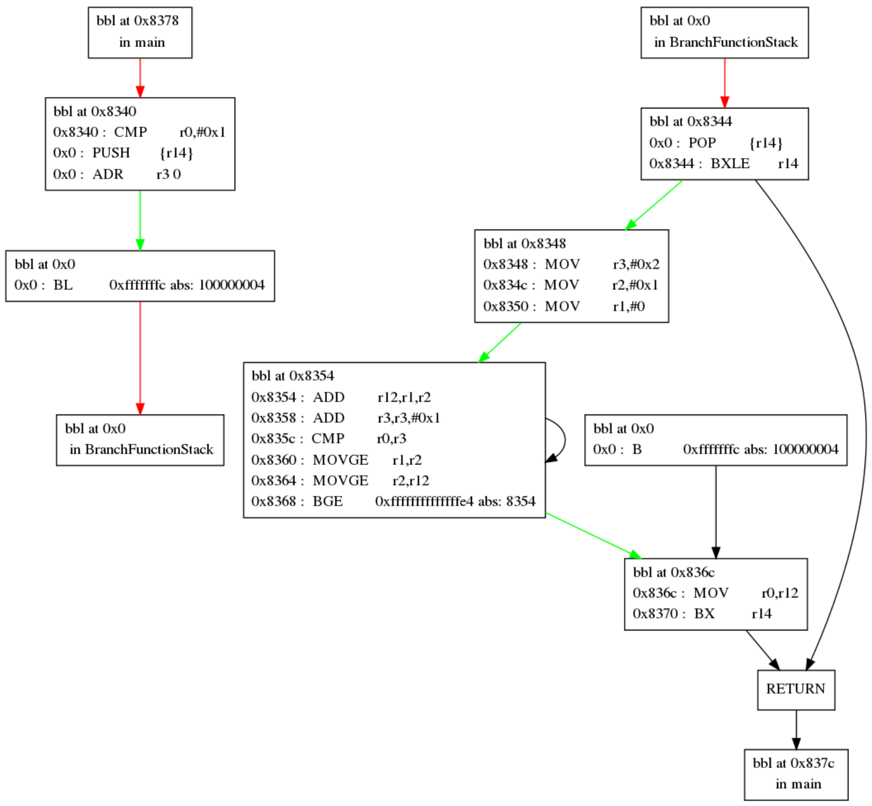


Figure 3: Toy code example of a function after branch function insertion

2.3 Opaque predicates

Opaque predicate insertion injects predicates and associated control flow edges into the binary, whose computation always evaluates to the same value. The control flow edges associated with the value that never occurs in a program execution can be directed to unrelated code locations in the binary. This obfuscation technique was introduced by Collberg et al [Col97]. This again significantly increases the effort for a reverse engineer.

The prior implementation of opaque predicates in Diablo is an x86-only implementation [Mad07]. It consists of a database of 9 hard-coded opaque predicates. Similar to the implementation of branch functions, this implementation used hardcoded registers, hard-coded integer constants (that are used as masks on the variable inputs of the opaque predicate computations in order to ensure that those computations do not introduce integer overflows or other exceptions), and unnecessarily spilled registers to the stack.

Support for multiple opaque predicates on ARM has been implemented and tested, based on this x86 implementation.

We introduced two different sources of randomness in the generated instruction sequences. Firstly, the registers used for the opaque predicate are randomly chosen from the set of dead registers; only when there is an insufficient amount of dead registers, live registers are randomly chosen and temporarily spilled to the stack. Secondly, when opaque predicates require hard-coded integer constants, these constants are randomly chosen.

By default, the current implementation chooses an existing basic block entry in the same function to redirect the not-taken path.

Opaque predicates and the corresponding branches can be inserted at any program point between any pair of instructions within a basic block where the condition flags are dead. If Diablo decides to insert an opaque predicate in a basic block, it chooses a random opaque predicate from its collection, and inserts it between a randomly selected pair of adjacent instructions where the flags are dead.

An example is shown in Figure 4, where an opaque predicate was inserted in the middle of an existing basic block, i.e., between the instructions at addresses `0x834c` and `0x8350`. The target block of the edge that is never taken, is taken to be a random block from the function.

The following set of opaque predicates is implemented:

- 2 | x+x^2
- 7*y^2-1 != x^2
- 2|x+x
- x^2 >= 0
- 2 | x^2/2
- 2 | x v 8 | (x^2-1)
- 3 | x^3 –x
- Sum of all odd i in the range (1, 2*x-1) == x^2.

2.4 Factoring

In addition to the aforementioned obfuscation transformations, Diablo also has support for factoring code. This transformation identifies identical code fragments across the binary, extracts these different fragments into a single function that is then called from all locations that previously contained the original, duplicated code fragments. The original motivation for this transformation was for code compaction [Deb00].

In the context of ASPIRE, we factor code to thwart reverse engineering, rather than to compact code. Because factoring combines syntactically identical code fragments, the combined fragments need not be semantically related. Thus, the factored code will be called from semantically different contexts, which breaks the relationship between code fragments and their semantics. This makes it harder for an attacker to analyse the factored code and their surrounding context. Furthermore, it is our eventual goal that original program code is factored together with code from the protection techniques themselves.

This is part of Diablo's background IP, i.e., we currently re-use the pre-existing ARM implementation.

2.5 Generic obfuscation infrastructure

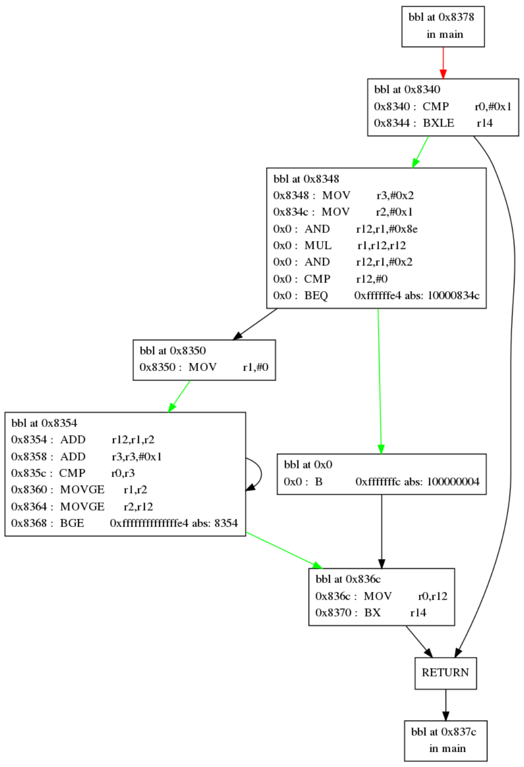
The previous x86-only implementation of obfuscations in Diablo was not generic. To obfuscate a binary, Diablo's obfuscation back-end contained a hard-coded sequence of calls to different obfuscation transformations. That framework was rigid and not suitable for the flexibility and externally configurable behavior demanded by the ACTC from the obfuscation transformations.

In addition to the aforementioned refactoring of the x86-specific code into an architecture consisting of a platform-independent part and platform-specific implementations, we also genericized the entire obfuscation infrastructure. All obfuscations have been modeled in a class hierarchy, that can dynamically be queried for random obfuscations from a (sub)set of the class hierarchy. For example, a user can now ask to insert a specific opaque predicate by querying its name, but s/he can also insert just as easily *any* random opaque predicate by specifying "`opaque_predicate`" as the requested obfuscation transformation. Similarly, a user can now even request for *any* random obfuscation transformation to be inserted. Diablo simply queries the class hierarchy for the set of transformations that match the description, and picks a random transformation from that set.

2.6 Layout randomization

We added support for code layout randomization to Diablo's ARM backend. Rather than optimizing the code layout for minimizing the code size of the final binary, this transformation randomizes the order in which chains of basic blocks (i.e., sequences of basic blocks through which control flow can fall through) are placed in the final binary [Cop13].

With this randomization, related chains (such as the chains from one function) are no longer placed next to each other, but spread all over the code section of the binary or library. This is particularly useful in combination with control flow obfuscations that insert indirect control flow, because it is then no longer obvious from the code layout which code fragments are related to each other, e.g., how chains are partitioned in functions.



```
bbl at 0x8378
in main
```

```
bbl at 0x8340
0x8340 :  CMP       r0,#0x1
0x8344 :  BXLE      r14
```

```
bbl at 0x8348
0x8348 :  MOV       r3,#0x2
0x834c :  MOV       r2,#0x1
0x0 :  AND       r12,r1,#0x8e
0x0 :  MUL       r1,r12,r12
0x0 :  AND       r12,r1,#0x2
0x0 :  CMP       r12,#0
0x0 :  BEQ       0xffffffe4 abs: 10000834c
```

```
bbl at 0x8350
0x8350 :  MOV       r1,#0
```

```
bbl at 0x8354
0x8354 :  ADD       r12,r1,r2
0x8358 :  ADD       r3,r3,#0x1
0x835c :  CMP       r0,r3
0x8360 :  MOVGE     r1,r2
0x8364 :  MOVGE     r2,r12
0x8368 :  BGE       0xffffffffffffffe4 abs: 8354
```

```
bbl at 0x0
0x0 :  B         0xfffffffc abs: 100000004
```

```
bbl at 0x836c
0x836c :  MOV       r0,r12
0x8370 :  BX        r14
```

```
bbl at 0x837c
in main
```

Figure 4: Toy code example of a function after opaque predicate insertion

Section 3 Integration in the ACTC

An initial version of the binary code obfuscation support was already integrated into the ACTC by M12 for milestone MS07. However, this initial integration was kept relatively shallow. Even though Diablo could already apply the obfuscations, and Diablo could be invoked by the ACTC, the process of selecting and applying obfuscation transformations in Diablo was not really controllable from the ACTC. To resolve this, we added annotation support to Diablo, so that the process of selecting obfuscation transformations can be guided through user-specified JSON annotation files. In addition, after integrating the obfuscation transformations with other protection techniques such as the Client-Side Code Splitting as implemented in BLP03 of Deliverable D5.01. Finally, we also extended some of the existing obfuscation transformations, for example by making the selection of possible continuation points for the opaque predicates and branch function insertion a global process, rather than a function-local one.

3.1 Annotations

To make the application of the obfuscations externally configurable, we added support for applying the transformations as they are specified in the annotations in the source code.

First, we added support to Diablo to read debug information containing source file and line number information from the object files. This line number information is then associated with the assembly instructions in Diablo.

The JSON file containing the source annotations (that is, the D01 input to BLP04) refers to file names and line numbers. When this file is parsed by Diablo, for each annotation, we create an annotation region that consists of the instructions that correspond to the line range specified by that annotation.

To apply the obfuscation transformations, we iterate over all annotation regions, and apply the obfuscation transformation specified for that region. These annotations can specify additional information for the decision logic, such as the percentage of basic blocks that has to be transformed in a protected region. The exact syntax and semantics of these annotations is described in detail in Appendix B of Deliverable D5.01.

Furthermore, we added additional types of annotation not specified in Deliverable D5.01 to test the obfuscation transformation and annotation functionality independently from the progress made in WP5 on the ACTC. In particular, we added support for wild-card matching of function names, so that we can easily instruct Diablo to transform entire regions of the program with a single annotation. This allows us, for example, to transform all crypto-related functions with the function name "`*crypt*`", without having to annotate all crypto-functions in the program's source code manually and having to rely on the ACTC to produce a correct JSON annotation file from these source-code annotations.

The support for additional annotations is not document in formal ACTC documentation, since it is meant for internal UGent development purposes only, and hence no dependencies on this support should be created in the ACTC.

3.2 Logging

We added extensive logging support to Diablo in order to satisfy REQ-ASR-005 of Deliverable D1.03, which states that '*It must be feasible to have an overview of which protection techniques have been deployed by the ASPIRE tool chain, in what order, and on which code or part of the binary they have been deployed.*'

All transformations applied by Diablo are assigned consecutive IDs. A log file is produced that maps each such ID to high-level information of the transformation, such as the type transformation. Furthermore, we have made it possible to optionally dump significantly more information for each transformation applied. This information contains a list of all source lines that are affected by the transformation applied, and dumps of all control flow graphs of the functions affected, both before and after applying the transformation, similar to the ones shown in figures 1 to 4. Because of the significant overhead in producing all this information, this level of detail in logging is disabled by default.

As an example, the high-level log of a run of BLP04 could start as follows:

```
0,OpaquePredicate,0x8914,spec_init,'arm_opaque_predicate_2|x_v_8|(x^2-1)'

1,BranchFunction,0x10cac,BZ2_bzCompress,
```

which indicates that the first transformation was applied in the `spec_init` function, to a basic blocks that starts at addresses 0x8914 in the original binary, and that as a second transformation, Diablo inserted a branch function in the `BZ2_bzCompress` function. For the first transformation, the more detailed logging information then provides line number information that starts as follows:

```
OpaquePredicate,spec_init,0x8914,spec.c:91

OpaquePredicate,spec_init,0x8918,spec.c:91

OpaquePredicate,spec_init,0x8920,spec.c:91
```

This indicates that the instructions at addresses `0x8914` to `0x8920`, which originate from the source file spec.c at line 91, are transformed. Additionally, a pair of "before"and "after" control flow graphs of the affected functions would be generated, similar to Figures 1 and 2.

Section 4 Validation

We verified both the correctness of the obfuscation transformations as well as their effectiveness against reverse engineering tools. We mostly relied on the C, C++, Fortran, and combined C/Fortran benchmarks from the SPEC2006 benchmark suites for our evaluation. The specific benchmarks used are shown in Table 1. Note that some benchmarks of SPEC2006 do not compile or run on the Android platform, even without trying to transform them with Diablo.

4.1 Correctness testing

Because we made our obfuscation implementation architecture-independent, we did not only verify the correctness on the ARM platform, but also on the x86 platform. This ensured that the refactoring we performed, and extensions we implemented for ASPIRE did not introduce any regressions in the existing x86 code.

We evaluated the transformations on ARM for Linux and Android, and on Linux for x86. The benchmarks were both statically and dynamically linked binaries, including PIE and non- PIE binaries, compiled with both gcc 4.8.1 and llvm 3.4 with the –O2 flag. Furthermore, we tested these transformations on a C library and on a C++ library. We flattened all suitable functions in all those binaries compiled, and separately applied branch function insertion and opaque predicate insertions to all suitable basic blocks with a 50% percent chance. At each point we inserted an opaque predicate, we selected one at random from the collection of opaque predicates. All transformations were tested with code layout randomization enabled. All obfuscated binaries behave correctly.

Table 1: SPEC CPU2006 benchmarks used to test Diablo

Benchmark suite	Programming Language	Benchmark	Android
SPEC CINT2006	C	400.perlbench	-
SPEC CINT2006	C	401.bzip2	+
SPEC CINT2006	C	403.gcc	+
SPEC CINT2006	C	429.mcf	+
SPEC CINT2006	C	445.gobmk	+
SPEC CINT2006	C	456.hmmer	+
SPEC CINT2006	C	458.sjeng	+
SPEC CINT2006	C	462.libquantum	-
SPEC CINT2006	C	464.h264ref	+
SPEC CINT2006	C++	473.astar	-
SPEC CINT2006	C++	483.xalancbmk	-

SPEC CFP2006	Fortran	410.bwaves	-
SPEC CFP2006	Fortran	416.games	-
SPEC CFP2006	C	433.milc	+
SPEC CFP2006	Fortran	434.zeusmp	-
SPEC CFP2006	C, Fortran	435.gromacs	-
SPEC CFP2006	C, Fortran	436.cactusADM	-
SPEC CFP2006	Fortran	437.leslie3d	-
SPEC CFP2006	C++	444.namd	+
SPEC CFP2006	C++	447.dealII	+
SPEC CFP2006	C++	450.soplex	+
SPEC CFP2006	C, Fortran	454.calculix	-
SPEC CFP2006	Fortran	459.GemsFDTD	-
SPEC CFP2006	Fortran	465.tonto	-
SPEC CFP2006	C	470.lbm	+
SPEC CFP2006	C, Fortran	481.wrf	-
SPEC CFP2006	C	482.sphinx3	+

4.2 Impact of the protections against attack tools

We also evaluated the impact of the obfuscations on reverse-engineering tools, i.e., to what extent the obfuscations hinder IDA Pro (version 6.5, released early 2014) in constructing valid control flow graphs and call graphs. To assess the performance of IDA Pro quantitatively, we measured for a number of dynamically linked benchmarks, compiled for the ARMv7 platform with gcc 4.8.1 with `-O2`, how well IDA Pro reconstructs functions from the disassembled code.

First, the charts in Figure 5 present the number of functions into which IDA Pro partitions the disassembled instructions. The benchmark versions studied are (1) original binaries with symbol information, (2) original binaries stripped from symbol information, (3) binaries rewritten by Diablo (i.e., with the whole code layout reorganized by Diablo on the basis of a control flow graph from which unreachable code is removed, and stripped from symbol information), and (4-6) binaries obfuscated by Diablo at three levels of obfuscation:

- **Light**: Each basic block is split with a probability of 10%. If so, it is split for inserting an opaque predicate or a branch function, each of which with an equal probability of 50%. Thereafter, each function is flattened with a probability of 10%.
- **Medium**: Same strategy, but with probabilities of 30% instead of 10%.
- **Heavy**: Same strategy, but with probabilities of 50% instead of 30%.

For all binaries, the heavy obfuscations result in IDA Pro partitioning the code into an order of magnitude more functions. Clearly, the function reconstruction is heavily distorted. Moreover, it can be seen that the amount of distortion correlates well with the amount of obfuscation.

Next, Figure 6 presents how well instructions are partitioned into functions that actually match the original functions of the programs. More precisely, we measured how well IDA Pro succeeds in putting together in the same function the instructions that actually belong in the same function using the following metric: Instructions A and B are related when they originate from the same function in the original binary. This relation is the ground-truth as determined by Diablo. In the functions reconstructed by IDA Pro, we measure false negatives, i.e., how many of those relations are missing. In other words, we count the number of instruction pairs that should be in the same function, but are not because IDA Pro is thwarted. Moreover, we use weighting factors to ensure that the weight of each function is equal to its number of instructions, instead of the square of that number: A function A with 10 instructions has 10x9 instruction pairs that we can count, whereas a function B with 100 instructions has 100x99 instruction pairs. Giving all of them the same weight would let the function B contribute a 100 times more to the metric than function A. So we weight the contribution of each pair by dividing it through the number of instructions in its function.

These results confirm the previous results and again indicate that as the obfuscations are applied more heavily, they more and more obstruct IDA Pro in grouping the instructions correctly into functions. The result for the statically linked 445.gobmk compiled with GCC shows that *when the obfuscator gets lucky*, light obfuscation can thwart IDA Pro as well as medium obfuscation.

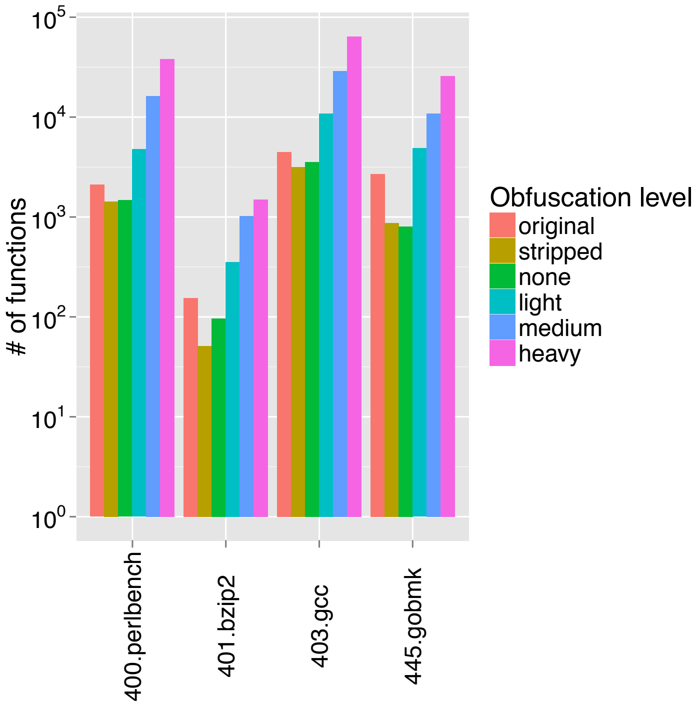


Figure 5: Number of functions reported in the binaries according to IDA Pro on dynamically linked gcc 4.8.1 ARM binaries

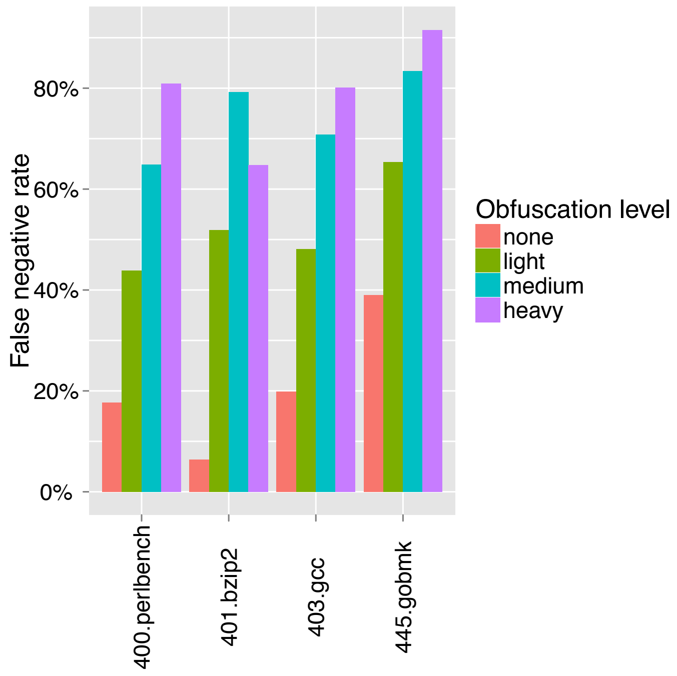


Figure 6: Partitioning of instructions in correct functions on dynamically linked gcc 4.8.1 ARM binaries

Section 5 Future Work

We are planning to add extend the binary obfuscation techniques in several ways in the future. In particular, we would like to:

- Improve the selection of continuation points for branch functions and opaque predicates. Currently, this selection is made globally over the entire application, but the process is not yet tweakable. We would like to make this selection process steerable as well, in order to ensure that semantically different code fragments seem linked together for an attacker.
- Flattening larger code regions. Currently, code flattening is limited to function boundaries. We are planning to extend this functionality so that a single switch block can redirect the control flow to basic blocks from multiple functions. This will seemingly link semantically unrelated code fragments through the shared switch block.
- Improve factoring. Currently, factoring is applied to all suitable code fragments. However, we will also make the decision process of which code fragments to factor more controllable. This includes trying to merge code fragments from the protected code with code fragments from the protections.
- Implement decision support logic to trade-off security and performance overhead.

Section 6 List of Abbreviations

ASPIRE	Advanced Software Protection: Integration, Research and Exploitation
ACTC	ASPIRE Compiler Tool Chain
BLP	Binary-level Processing
PC	Program Counter

Bibliography

[Cop13] Feedback-driven binary code diversification
Bart Coppens, Bjorn De Sutter, Jonas Maebe
ACM Transactions on Architecture and Code Optimization, 2013, pp 24:1--24:26

[Col97] A Taxonomy of Obfuscating Transformations
Christian Collberg, Clark Thomborson and Douglas Low
Technical Report 148, University of Auckland, 1997, 36 pages

[Deb00] Compiler techniques for code compaction
Saumya K. Debray, William Evans, Robert Muth, Bjorn De Sutter
ACM Transactions on Programming Languages and Systems, 2000, Volume 22 Issue 2, pp 378-415

[Linn03] Obfuscation of Executable Code to Improve Resistance to Static Disassembly
Cullen Linn, Saumya Debray
ACM Conference on Computer and Communications Security, 2003, pp 290-299

[Mad07] Application Security through Program Obfuscation
Matias Madou
PhD thesis, Ghent University, 2007, 145 pages

[Wang00] Software Tamper Resistance: Obstructing Static Analysis of Programs
Chenxi Wang, Jonathan Hill, John Knight, Jack Davidson
Technical Report, University of Virginia, 2000, 18 pages