Diablo Documentation

Table of Contents

1. Developed Diablo Features and Support

1.1. Supported compiler tool flows

Table 1 shows Diablo's support for recent Linux compiler tool flows.

In addition, support for clang-llvm v3.3 and v3.4, gcc v4.6 and v4.8.1, as well as the `as` and `ld` tools from binutils v.2.23.2 have been developed and tested for `arm-linux-androideabi` as part of the Android NDK API Level 18 (NDK git repository version be9ba71abf2b898fa62a169659ab0b6f8baaa5ca). At the time this NDK version was chosen during our testing, this was the most recent version that supported gcc v4.8 and v4.6. clang-llvm v3.2 was not tested for the arm-linux-androideabi because the used Android NDK (for API Level 18) did not provide standard support for clang-llvm v3.2. So instead of clang-llvm 3.2 and 3.3, versions 3.3 and 3.4 were tested for Android.

Existing open-source compilers need to be patched to provide sufficient relocation and symbol information in the object files handled by Diablo. To that extent, we have updated and extended our existing tool chain patches. Table 2 lists those patches from the diablo toolchains repository.

Table 1: Currently supported Linux compiler tools

Target Architecture	Tool	Supported tool version in Dec 2013	Feb 2015
ARM	binutils (`as` and `ld`)	v2.18a (Aug 2011)	v2.23.2 (Mar 2013)
	eglibc	v2.11 (Oct 2009)	v2.17 (Dec 2012)
	Gcc	v4.3.6 (Jun 2011)	v4.6.4 (April 2013) v4.8.1 (May 2013)
	clang - llvm	none	v3.2 (Dec 2013) v3.3 (Jun 2013) v3.4 (Jan 2014)
	crosstools	v1.15.3 (May 2012)	hg+default-99029fac116b (from crosstools GIT repository - June 2014)
x86	binutils (`as` and `ld`)	2.13.2.1 (Aug 2011)	v2.23.2 (Mar 2013)
	eglibc	glibc v2.3.2 (Feb 2003)	v2.17 (Dec 2012)
	Gcc	v3.2.2 (Feb 2003)	v4.6.4 (April 2013) v4.8.1 (May 2013)
	clang - llvm	none	v3.2 (Dec 2013) v3.3 (Jun 2013) v3.4 (Jan 2014)
	crosstools	none	hg+default-99029fac116b (from crosstools GIT repository - June 2014)

Table 2: Diablo-readying patches to the compiler tool flows

Tool	Patch File Name	Description
clang - llvm v3.x	`clang.patch`	generate $handwritten mapping symbols around inline assembler
	`crosstool-fix.patch`	patch reused from the clang-crosstools fork to integrate clang with crosstools binutils (https://github.com/diorcety/crosstool-ng)

	`diablo_tc_rev.patch` (optional)	patch to generate Diablo-comment that identifies Diablo-ready compiler was used
	`emit_clang_version.patch` (optional)	patch to let older clang generate version comment identifier in object files
	`emit_version.patch` (optional)	patch to let older llvm generate version comment identifier in object files
binutils v2.23.2	`remove_eh_frames.patch`	omit exception handling frames (which are not supported yet in Diablo, support is foreseen for summer 2014)
	`disable-more-merge-eidx.patch`	disable exidx section merging (also not needed yet)
	`mark_code_data_sections.patch`	generate mapping symbols that mark data in code sections
	`mark_code_data_sections_switch.patch`	generate mapping symbols at beginning of resumed sections
	`disable_section_merge.patch`	disable section merging during linking
	`disable_relaxation.patch`	disable symbol relaxation
	`add_relative_symbols.patch`	generate additional mapping symbols that allow Diablo to relocate code more aggressively
	`fix-neon-vshll-qd.patch`	backported patch from later binutils to fix objdump NEON instruction disassembler
	`add_diablo_comment_gas_patch`	patch to let GNU assembler generate comment section with Diablo-ready comment
gcc v4.x.y	`annotate_handwritten_asm.patch`	patch to generate $handwritten mapping symbols around inline assembler
gcc v4.8.1	`disable_tm_clone.patch`	patch to disable the generation of sections related to transactional memory
	`fix_parallel_build.patch`	patch to allow parallel build of the compiler, obtained from crosstools mailing list)
	`enable_dwarf_crtbeginend.patch`	patch to omit debug information in crt object files (such that binutils' link-once support works properly on code generated with all of the above patches)
eglibc v2.17	`add_cfi_calls_to_get_thunk_cx.patch`	add symbols to avoid inconsistency in symbols generated by compiler and included in macros, needed to make binutils detect identical link-once sections when relaxation is disabled
	`remove_i366_stpcopy_data_in_code.patch`	undo a very dirty hack by replacing instruction bytes encoded as data bytes by the corresponding instruction in an x86 assembler code fragment
Android NDK	`ndk-build-gcc-diablo-version.patch` (optional)	adapt GCC version string to generate Diablo-ready comment
	`ndk-gdbserver-try64.patch`	fix NDK script for generating 64-bit version of gdb-server
	`ndk-hardcode-source-hashes.patch`	hardcode versions of tools (gcc, clang-llvm, binutils, ...) to be downloaded by NDK upon

		installation
	`ndk-llvm-binutils.patch`	enables specification of binutils version to be used during NDK llvm build

Known limitations

Float ABI: For all of the supported compilers, we currently use the softfp float-abi. The resulting code exploits hardware floating instructions, but still implements soft-float calling conventions.

We have also tested Diablo in combination with gcc v4.8.1 with the hard float-abi, which allows the generation of floating-point instructions and uses FPU-specific calling conventions. Diablo did not suffer from any problems on that code. Because we were not able to generate correct binaries with clang-llvm-binutils-eglibc with the hard-float API, however, we have not been able to exhaustively evaluate Diablo on such binaries.

Standard C Library: As for Linux libc support, all our development has been performed against the eglibc standard C library 2.17. Versions 2.18 and later are not yet support because they rely on IFUNC-related functionality, which is not yet handled by Diablo. For Android, Diablo has successfully passed our tests on Android's Bionic C library. For these experiments, we used the Bionic C library version that was patched by Boundary Devices for Android 4.3 in support of the SABRE Lite i.mx6 (BD-SL-i.MX6) development boards used for testing at UGent.

Exception handling: Diablo does not fully support C++ exception handling yet, and so the generation of some exception handling data structures needs to be disabled, in particular by passing the option `--no-merge-exidx-entries` to the linker.

Gold linker: Currently, Diablo only supports the `ld` linker from binutils. No linker scripts for the `gold` linker have been developed yet. By default, the Android NDK invokes `gold`. To override this default behavior and link with `ld` instead, the command-line option `-fuse-ld=bfd` needs to be passed to gcc and clang when they are invoked to link the program.

Separate compilation: In some of its data flow analyses, Diablo relies on assumptions about "standard" compiler behavior. This assumed "standard" behavior includes separate compilation. In other words, each source code file needs to be compiled separately to an object file, all of which are then linked by `ld` into a final binary or library, without invoking any whole-program or link-time optimizations of gcc or clang-llvm. Alternatively, all source code files can be compiled at once (i.e., with one invocation of the compiler (gcc or clang) but again without enabling whole-program or link-time optimization), but then the `-save-temps` flag needs to be used to ensure that the individual object files are also produced and stored on disk, because Diablo needs them, together with the produced binary or library.

Name mangling: Diablo's linker map parser cannot handle demangled C++ function names. In order to prevent those from being generated, the `--no-demangle` command-line flag needs to be passed to the linker.

Hash sections: Diablo does not yet support the new style GNU `.gnu.hash` hash tables. It does support the classic ELF `.hash` section however. We noticed that even though the ELF version is supposed to be the default, this default is not always used by `ld`. Therefore one should always provide the `--hash-style=sysv` option to the linker.

Dynamic relative/absolute relocations: The `ld` linker has some rather obscure optimization decision logic with respect to dynamic absolute ARM relocations. Sometimes they are converted into dynamic relative ones as an optimization, but sometimes they are not. We have currently not been able to reverse-engineer the decision logic completely. For that reason, Diablo will sometimes not be able to emulate the original `ld` behavior, as a result of which Diablo will abort. To prevent ld from behaving unpredictably from Diablo's perspective, one can feed the `-Bsymbolic-functions` option to the linker. This is only necessary for shared libraries.

LLVM assembler & linker: Diablo only supports the GNU binutils assembler and linker, not their LLVM counterparts. In order to instruct clang/llvm to rely on the GNU tools to assemble and link clang-llvm compiled code, the following flags need to be passed to clang-llvm:

```
    -isysroot $(TCPATH)/$(BINUTILSPREFIX)/sysroot -no-integrated-as
    -gcc-toolchain $(TCPATH) -ccc-gcc-name <triplet> -target <triplet>
```

with `<triplet>` either being `arm-diablo-linux-gnueabi`, `i486-diablo-linux-gnu`, or `arm-linux-androideabi`.

Related Diablo command-line options:

Three Diablo command-line options are relevant with regards to the compiler and linker tool chain used:

Compatibility checking: Compiler tools (gcc, clang-llvm, as, ld) built with the crosstools version delivered with this report inject certain comments into the generated object files that identify those object files as being generated with compilers patched with the patches of Table 2. Diablo by default checks for the presence of these comments, and will abort with an error message in case any object files are found in the input library or binary that were not generated with the patched compiler tools. This check can be disabled with the `--no-toolchain-check` command-line option. Disabling it is only advised for expert users, however, who are confident that they generated all their code with appropriately patched tool chains.

ARM Erratum #657417: By default, Diablo generates code that works around this erratum for some Cortex-A8 processors. If this workaround is not needed, it can be disabled by passing the `--no-fix-cortex-a8` option to Diablo.

1.2. Support for architecture extensions

Diablo’s architecture has, during its lifetime, supported many architectures. However, currently we only support the i486 x86 ISA is supported, and ARMv7, including Thumb2, VFP, and Advanced SIMD instructions.

Known limitations

Data flow analyses and optimizations: Diablo's data flow analyses have been extended to deal correctly with the NEON, VFP, and Advanced SIMD instruction set extensions, but no full optimization of those extensions has been implemented. For example, the load-store forwarding analysis was extended to take into account VFP, NEON and Advanced SIMD load and store instructions (like VLDR, VPUSH, ...) to avoid incorrectly applying transformations on load-store combinations involving registers r0-r14. But load-store forwarding optimizations are only applied to standard ARM/Thumb2 load/store (LDR, STR, LDM, STM, LDRB, STRB) instructions accessing only those registers. Similarly, no constant propagation or copy propagation is performed through extended register files (single, double and quad) registers.

By contrast, dead code elimination of dead VFP, NEON, and Advanced SIMD instructions is supported.

As for constant propagation, we conjecture that extending the support to the extended register files will not result in significant additional optimizations, simply because there are very few compile-time constants to be found in those registers at program points where they can be exploited. For that reason, we do not plan to extend the constant propagation to support the extended register files in the future.

Address producers: Absolute 32-bit addresses can be produced in ARM or Thumb2 code (1) using PC-relative loads from data pools in the code section, (2) using PC-relative computations, and, since ARMv7, (3) using the combination of MOVW and MOVT instructions. During control flow graph construction Diablo converts all three kinds of instruction sequences into so-called address producer instructions. When the graph is transformed into one instruction sequence, however, no MOVW/MOVT combinations are inserted yet. As such, the assembling of address producers now always favors code size over performance.

Pure Thumb-1 code: Diablo does not provide pure Thumb-1 code support. Still, a lot of code is present that does support pure Thumb-1 code, and a lot of new code is already written (but not tested) to be capable to generate pure Thumb-1 code, to prepare for the hypothetical by unlikely event that the need would surface in the future to complete that support.

By default Diablo for ARM assumes it may generate Thumb-2 code. To indicate otherwise (to test the pure Thumb-1 code support), the command-line option `--fullthumb2` can be specified.

By default, Diablo will only generate Thumb-2 instructions when transforming other Thumb-2 code. If no such code is present in the binary or library being rewritten, Diablo will not insert it either.

i486 code: Diablo does not support any x86 instruction set extensions beyond the basic i486 ISA. Furthermore, no 64-bit Intel or AMD ISAs are supported. Therefore all x86 code currently needs to be compiled with the `-march=i486 -m32` options if later rewriting with Diablo is needed.

1.3. Support for dynamically linked binaries and libraries

Today, Diablo also supports dynamically linked ELF binaries: for the supported compiler and standard library versions mentioned in Table 1, all necessary relocations, symbols, section types, etc. are now supported. The existing analyses, optimizations, obfuscations and profiling all work on the statically linked as well as on the dynamically linked binaries. This includes standard dynamically linked binaries, as well as so-called PIE binaries.

Diablo supports statically and dynamically linked binaries and libraries, this includes PIE binaries.

Known limitations

PIE binary OS support: On Linux and Android, the ASLR-capable loader is broken for PIE binaries. As a result, when ASLR is enabled, the PIE binaries sometimes get loaded at addresses that for some reason (unknown to us) cause memory allocation problems once the applications are running. We have observed that in all cases in which this issue occurs with Diablo-rewritten binaries, it also occurs with the original binaries. As this is clearly the result of a kernel bug unrelated to Diablo, we have not investigated this further.

1.4. Support for profiling

The main diablo frontends `diablo-arm` and `diablo-i386` can generate instrumented self-profiling versions of the rewritten binaries and libraries. More precisely, the then generated versions perform basic block profiling, of which the produced profiles can be read by Diablo to be used in profile-guided optimizations.

Known limitations

Instrumentation has only been tested on otherwise unoptimized or unobfuscated binaries, and libraries, i.e., when the aforementioned tools are invoked with the `-Z` flag. On optimized or obfuscated binaries, the instrumentation does not work completely correctly.

1.5. Support for the Android source tree (AOSP)

For now, Diablo has only been tested on Android 4.3 Jelly Bean, patched for an i.MX6 Sabre Lite development board. Other AOSP versions may or may not be supported. In order to be able to generate Diablo-compatible Android binaries and libraries, the vanilla AOSP sources have been patched.
The available patches are listed in the table below, in which the required patches are marked in **bold**. These patches are available in a separate tarball, next to the one containing the Diablo source code. They can be applied to an AOSP source tree by issuing the following command, assuming the current directory is the root of the AOSP tree:

```
        patch -p1 < <path to patch file>
```

Details on how to build the AOSP source tree with a Diablo-compatible tool chain can be found in section 2.3.

Table 3: Diablo-readying patches to the AOSP source tree

Patch file name	Description
`aosp-4.3_toolchain.patch`	This patch is always required if a Diablo-compatible toolchain is used to build the AOSP source tree. This patch removes -werror to avoid that the compiler aborts on unused parameter warnings that are introduced with our other patches.
`aosp-4.3_diablo.patch`	This patch is always required if the Android binaries or libraries are to be rewritten by Diablo. It ensures that: • map files are generated; • the BFD linker is used (and not the Android-default gold linker); • the correct compiler/linker flags are used, so the resulting binary is compatible with Diablo; • no gold-specific linker flags are used.
`aosp-4.3_libvideoeditor.patch`	This patch is required only if the libvideoeditor library is to be rewritten by Diablo.
`aosp-4.3_openssl.patch`	This patch is required only if the openssl library is to be rewritten by Diablo.
`aosp-4.3_libstagefright.patch`	This patch is required only if the libstagefright library is to be rewritten by Diablo.

The latter three patches update some inline assembly code in which instructions are encoded as data (e.g., the assembly code contained .word 0xe3a01000 instead of the mnemonic mov r1, #0). On top, some PIC code patterns are adapted that Diablo cannot handle yet.

Warning: it can happen that some library needs (additional) patching if a modified version of the AOSP is used, which is especially the case for parts written in assembly language. Also not all libraries get through Diablo yet, so the above list of patches may be expanded in the future.

2. Manual of Diablo-Ready Tool Chains and Diablo Tools

2.1. Installation of Diablo compatible tool chains

All of the Diablo-compatible tool chain components mentioned in Section 1.1 can be downloaded, patched and installed automatically on a Linux host using the scripts, configuration and other files in the `diablo-toolchains` repository. The main script to be used is `build.sh`.

It is important to note that when building a LLVM tool chain the result will only use LLVM/clang to compile. Assembling and linking will still require a GCC/binutils tool chain built with this script. However, the script does make the necessary fixes in order for the resulting clang compiler to invoke the binutils tools without issues (and with some required arguments).

When invoking the build.sh script without options, all options are shown on screen.

In the unpacked directory structure, none of the files in the following directories should be changed in order to build a tool chain. Only modify them if you know what you are doing.
The directories hold the following data:

- `diablo-patches`: This directory contains all our patches for the different versions of GCC, LLVM, EGLIBC and Binutils. The patches found in here are applied when building a tool chain that uses the corresponding versions of the tools.
- `config`: A collection of all our crosstool-ng configuration files. One such file can be found in here for each supported tool chain version.

- `example`: This folder contains an example 'Hello World'-project with makefiles for i486 and arm. The makefiles are set up with all the necessary options to compile for and rewrite with Diablo.

- `package`: This directory contains some templates for building Debian packages of the toolchains.

2.2. Compilation of software to rewrite with Diablo

When using the built toolchains, some arguments are required to produce a rewriteable binary.

In addition, the linker needs to be instructed to generate a so-called linker map with the same name as the produced binary, extended with the extension .map. This can be achieved by passing the option `-Map,<name>.map` to the linker., with `<name>` the name of the linked binary or library.

The `example` directory of the diablo toolchains tarball contains two makefiles showing what extra arguments to pass when using a tool chain.

Warning: if you modify existing Makefiles that compile & link multiple binaries, simply hard-coding the `–Map,<name>.map` may break the resulting files, since linking the second binary will overwrite the first binary's map file. A separate map file should be generated (with the appropriate name) for each binary.

Warning: some compiler and linker flags should not be used, as they remove required information from the produced library or binary. These flags are:

- `-s`: This linker argument removes all symbol table and relocation information from the executable.

2.3. Compilation of Android binaries and libraries to rewrite with Diablo

An existing AOSP source tree needs to be patched according section 1.5 in order for it to generate Diablo-compatible output. One of the available Diablo-compatible Android tool chains should be built prior to doing anything described in this section.

Before using the Diablo-compatible Android tool chain, some library archives need to be generated from the AOSP source tree. More specifically, the versions of `libc`, `libm` and `libstdc++` meant for static linking need to be generated.

Building Diablo-compatible versions of libc, libm and libstdc++

1. Navigate to the root of the AOSP source tree.
2. Apply the patches marked in **bold** in Table 3; other patches may be applied if desired.
3. By default, the AOSP build system will put its output binaries in the `./out` directory. Another directory can be specified by setting the `OUT_DIR_COMMON_BASE` environment variable: `export OUT_DIR_COMMON_BASE=<destination directory>`.
4. Initialise the build system by executing the command "`. buid/envsetup.sh`", followed by the "`lunch`" command (both without quotes). The latter command requires you to choose the target platform for which the AOSP should be built.
5. Build the AOSP by executing the command:
   ```
   m –j<parallel job count> TARGET_TOOLS_PREFIX=<path to Diablo-
   compatible Android tool chain>/bin/arm-linux-androideabi-
   TARGET_SYSROOT=<path to Diablo-compatible tool chain>/sysroot libc
   libm libstdc++
   ```
6. Copy over the generated libraries to `<path to Diablo-compatible Android tool chain>/sysroot/usr/lib` (paths are relative to `<destination directory>/target/product/<target>/obj/STATIC_LIBRARIES`):
   - `libc_intermediates/libc.a`
   - `libm_intermediates/libm.a`
   - `libstdc++_intermediates/libstdc++.a`

Next, the AOSP source tree can be built using the Diablo-compatible Android tool chain. If an entire build is desired, a 64-bit tool chain is required. This is due to the fact that the link phase of the chromium libraries needs a lot of memory, which exhausts the available memory for a 32-bit linker. Such a 64-bit tool chain can be generated with the build.sh script by passing the optional parameter "`-w 64`".

<u>Building Diablo-compatible versions of the AOSP libraries</u>

1. Execute steps 1-4 of the build instructions for the statically linked library archives.
2. The AOSP consists of many different modules, which all together provide a full Android run time environment. A list of these modules can be generated by executing "make modules", without quotes. In the next step, either a space-separated list of these module names can be appended
3. Build the AOSP by executing the following command (specific modules can be built by appending a space-separated list of names generated in the previous step):
   ```
   m –j<parallel job count> TARGET_TOOLS_PREFIX=<path to Diablo-
   compatible Android tool chain>/bin/arm-linux-androideabi-
   TARGET_SYSROOT=<path to Diablo-compatible tool chain>/sysroot
   ```
4. The generated Diablo-compatible dynamically linked libraries can be found in `<destination directory>/target/product/<target>/obj/SHARED_LIBRARIES`.

Warning: as the build system automatically uses the `--gc-sections` linker option, this option should also be passed to Diablo, as described in section 2.5.
Warning: Diablo may fail parsing some AOSP-generated map-files. One recurring issue is that these map-files contain "vtable for ", a string that Diablo can't parse. This issue can be fixed by fixing the map file as follows:

```
        sed –i ‘s/vtable for /vtable_for_/g’ <map file>
```

2.4. Building and installing Diablo

Cmake/ccmake is the preferred build tool for Diablo. See the website http://www.cmake.org/ for a general manual on how to use these tools. Specifically for Diablo, the following additional information is needed.

The only option interesting to be tweaked by users is `CMAKE_BUILD_TYPE`, for which `Debug` or `Release` should be chosen. The `Release` version is significantly faster, and providing no option (i.e., the default setting) results in Diablo being compiled with -O0, and hence in very slow tools.

2.5. Standard Diablo Optimization/Compaction frontends

To invoke the standard frontends diablo-arm and diablo-i386, the following command can be used:

`[diablo-arm|diablo-i386]`

`-L <librarypath>`	path where libraries linked into the binary are stored
`-O <objectpath>`	path where object files linked into the program, the program itself, and the linker map file can be found (for binaries or libraries).
`[-o <output file]`	name of the output file, defaults to `b.out`
`[-g]`	read line information when present in the input files
`[-Z]`	disables optimizations beyond unreachable code elimination
`[-S]`	generate symbol information in the output binary/library
`[-D]`	generate .dot files of the functions' control flow graphs in the `dots` (before optimization) and `dots-final` (after optimization) directories

Additionally, the following non-standard options might prove interesting/useful for ARM binaries:

- Even though we do not fully support rewriting C++ exception handling code, limited rewriting of the exception-related data structures is provided by Diablo, given the

options: `-Z -kco`. This means that the code will not be optimized, and the final program's layout will be the same as the original. These flags can be combined with:
  - `-se` which shuffles the order of the exidx section contents
  - `-esf` which creates one exidx entry for every function in the binary
- `--gc-sections` which needs to be passed to Diablo if, and only if, this option was already passed to the original linker, meaning that the original linker will run garbage collection on the sections.

Finally, to get an up-to-date overview of additional options available to those frontends, invoke them with the `--help` option.

2.6. Instrumenting Binaries for Profiling

To enable `diablo-arm` and `diablo-i386` to instrument code for some target architecture, some IO routines needs to be pre-compiled for the target architecture, such that Diablo will be able to inject them into the instrumented application.

This pre-compilation has to be done manually.

First, a `Makefile` has to be generated by running the `generate.sh` script in the `self-profiling` directory of Diablo. As options, the path of the target (cross-)compiler to be used needs to be provided, a name for the object to be generated, and the target architecture. For example, we used the command

```
./generate.sh ~/Work/toolchains/armv7-gcc-481-binutils-223-eglibc-217-arm-
objects/bin/arm-diablo-linux-gnueabi-cc IOarm.o arm
```

to compile the routines for ARMv7. So far, we have always used gcc as a target compiler.

Next, `make` has to be executed in the `self-profiling` directory, without arguments, to generate the `IOarm.o` file in the `self-profiling` directory.

This procedure allows one to generate multiple versions of the IO routines for different target systems. In order to generate multiple such versions with different compiler options, one can edit the generated `Makefile` before running `make`.

To create an instrumented version of a library or binary, use Diablo's option `-SP`, and pass the path to the compiled IO object file as argument to the `-SP` option. The name of the generated, instrumented binary can still be chosen with Diablo's `-o` option.

When the produced binary or library with name `<filename>` is executed on some inputs, a file `profiling_section.<filename>` is generated (or overwritten if it already existed) that contains the basic block execution counts.

This profile file can be converted to another format with the `./binary_profile_to_plaintext.py` script that can be found in the Diablo `self-profiling` subdirectory, as follows:

```
./binary_profile_to_plaintext.py profiling_section.<filename> > <profile-
name>
```

To use the profile information in a subsequent run of Diablo in which the binary or library is optimized, invoke Diablo with the options `--rawprofiles off -pb <profile-name>.`

To merge multiple profile files (after they have been converted to plaintext), we have foreseen the `merge_profiles.py` script. Invoke it as

```
./merge_profiles.py <profile-name-1> <profile-name-2> > <combined-profile-
name>.
```

2.7. Diablo Obfuscation

The obfuscating rewriters can be used like any other Diablo front-end, i.e., by using the -o flag to name the output file, and by passing it the file to transform. In addition, it has 3 obfuscation-only flags:

`-off` to enable the function flattening transformation

`-obf` to enable the branch function transformation

-oop to enable the opaque predicate transformation

--orderseed <seed> enables code layout randomization, whose randomness is seeded with <seed>.

The randomized decision process of where to apply the obfuscations can easily be modified in the source code. All transformations are applied in the ObjectObfuscate function of the obfuscation/diablo_obfuscation_main.cc C++ file. First, the framework loops over all functions, and applies the flattening transformation. Next, the framework loops over all basic blocks of two or more instructions, and randomly selects one of the available basic block transformations (currently the branch function insertion and the opaque predicate insertion) to apply those basic blocks. If the preconditions are met, the obfuscation is applied in the basic block.

The structure of the transformations is based on inheritance. The base classes for function obfuscations and basic block obfuscations are defined in obfuscation_transformation.h. The generic, architecture-independent code can be found in the generic/ subdirectory. The architecture-specific transformations that extend these generic transformations are located in the architecture_backends/ARM and architecture_backends/i386 subdirectories. For example, the ARM flatten function transformation inherits from the generic code, implementing architecture-specific virtual functions that the generic code calls when flattening.

To get an up-to-date overview of the options available to the obfuscation frontends, invoke them with the --help option.

2.8. Diablo Iterative Diversification

To run iterative diversification, you are required to have access to:

- A license for IDA Pro (for Linux)
- BinDiff (>= 4.0)

Because of the interaction Diablo requires with other software (such as BinDiff), some set-up is required before being able to use this software (which mostly consists of entering the correct path names in a script). The integrated tool flow only runs Linux. All files related to the iterative diversification framework reside in the directory obfuscation/diversity_engine/iterative_diablo.

Warning: the diablo-diversity-* binaries should not be called directly. They require specific options and input files that are difficult to set up manually; the scripts discussed here automatically generate those files and will invoke the underlying Diablo components with the correct options.

Initial setup of the diversification framework

1. Build Diablo with the obfuscation backend enabled in the CMake configuration (this is enabled by default, this option also builds diablo-diversity binaries).
2. Edit settings.py in the aforementioned directory:
    - modify 'linux_diablo_regular' to point to your diablo-diversity-ARM binary that you built.
    - modify 'python_path' point to the path where these Python files are located.
    - modify 'ida_linux' to point to the Linux IDA binary (if you point it to the 'idal' binary rather than the 'idaq' binary, you can run this set of scripts without an X server)
    - modify 'bindiff_linux' to point to the 'differ' binary in your BinDiff install
    - modify 'experiments_binary_path_base' to the base path where you want your output files to be stored
    - modify 'homedrive_scripts' to point to the 'startscripts' subdirectory of this directory

Create a benchmark

For real usage, where the goal is to hide the semantic difference a patch introduced, a benchmark consists of 2 binaries. However, to demonstrate the working of the tool, we apply iterative diversity to two identical copies of the same program, with one copy standing in for the '`unpatched`' program, and one copy standing in for the '`patched`' program:

1. Create a directory for your benchmark, for example, `~/demo_iterative/`. Create two subdirectories: `~/demo_iterative/unpatched` and `~/demo_iterative/patched`
2. Create the '`unpatched`' version of your benchmark as you would usually create a binary for Diablo (i.e., using a patched tool chain) in the `~/demo_iterative/unpatched` directory.
3. Create the profile information for the unpatched version, as described in the manual for the self-profiling code.
4. Ensure the final (plain-text) `output.prof` is in the directory containing the unpatched binary and its object files (`~/demo_iterative/unpatched`).
5. For testing purposes, copy the contents of the '`unpatched`' subdirectory to the '`patched`' subdirectory. Alternatively, repeat steps 1 and 2 for the '`patched`' version of your benchmark in the '`patched`' subdirectory. (Or, you can compile a version of your source containing an actual patch in the patched subdirectory.)

Set up iterative diversity for this benchmark

We can now point the iterative diversity framework to these binaries, and then let the iterative diversity framework run.

Edit the iterative_diversity_demo.py, modify the binary name, base paths and version information in the `iterative_arm_demo = benchmarks.Benchmark` assignment. What happens is that the base path gets expanded, and the '`%s`' is replaced using the contents of the version map. So for version 1 (the unpatched version), the map contains `{ 1: "unpatched" }`, so the string "`unpatched`" is expanded in the base path.

Create a target directory for the iterative results, for example `~/iterative_results`, and edit `settings.py` to let `experiments_binary_path_base` point to this directory.

The final files will be put in a subdirectory of this directory, determined as follows: `base_directory/<seed>_<benchmarkexecutable>_<extra>`, where <seed> is the random seed argument given to the `iterative_diversity` function call in `iterative_diversity_demo.py`, and <extra> is a custom string to distinguish different experiment directories. Finally, let the `rules_file` argument point to the `diablo-source/obfuscation/diversity_engine/transformation_rules` file.

(If you would want to edit the `transformation_rules` file: the format is a rule per line: <numeric BinDiff matching strategy>, <maximal percentual weight of a BBL to apply this rule, 0.0 being unexecuted code, 1.0 the maximum BBL execution count>, <first iteration to consider applying this rule>, <transformation>. The mapping of numeric BinDiff rule to human-readable string is stored in any BinDiff file, and can be accessed by executing '`sqlite <BinDiff file>`', and typing '`.dump functionalgorithm`' on the sqlite command prompt.)

Running the iterative framework

Execute `iterative_diversity_demo.py`. This script should then start producing files in the target directory. After each iteration, the transformed binary's name is of the form `<executablename>_iterated_binary_<iteration number>_final`. After a while, the process should finish (after the number of iterations given in the script, which is in the example set to 20). It is possible that IDA Pro or BinDiff crashes (for example due to out of memory errors, 'bad allocation' errors, etc). This script cannot recover from such errors, and will instead fail.