Open Universiteit

FoxDec

Decompilation based on Formal Methods

prof. Binoy Ravindran	PI
dr. Freek Verbeek	co-PI
Joshua Bockenek	PhD
Daniel Spaniol	PhD

freek@vt.edu

Supported by the DARPA project FALCON:
Formal Analysis of Legacy COde domaiNs

User Manual

March 15, 2022

FoxDec User Manual

FoxDec is a tool actively developped at Virginia Tech (US) and the Open University of the Netherlands. Its aim is to lift binaries to a higher level of abstraction, in such a way that formal guarantees can be provided that the lifted representation is sound with respect to the original binary. This document provides a user manual, further information on implementation and limitations, as well as references for further reading.

Remark: *FoxDec is evolving quickly, and new features and capabilities are actively being developped. Do not hesitate to contact us for questions, remarks and suggestions.*

1 User Manual with Example

1.1 Download & Installation

Up-to-date information on where to download FoxDec, and instructions for building and installation, can be found at:

`https://ssrg-vt.github.io/FoxDec/#build`

1.2 Running FoxDec to create `.report` file

Compile. As running example, we will consider the `wc` command. For sake of explanation, we consider a small and simple implementation instead of taking the binary as available in a standard Linux or Mac distribution[1]. First, we compile the example. Go to the directory for the running example `wc_small`. There, we compile the file `wc.c` to an executable `wc`.

Compile the running example

```
cd ./FoxDec/foxdec/examples/wc_small
gcc wc.c -o wc
```

Extract. Subsequently, we extract information from the generated binary. We use standard tools for this: for Linux these are `readelf` and `nm`, and for MacOs these are `otool` and `nm`. Two scripts are provided: `dump_elf.sh` for Linux ELF files, and `dump_macho.sh` MacOs MachO files. Their command-line usage is:

`dump_elf.sh $BINARY $NAME`

[1]The source code of the `wc` example can be found here:
https://www.gnu.org/software/cflow/manual/html_node/Source-of-wc-command.html

`$BINARY` The path to the binary, including its filename.
`$NAME` Any name that clearly identifies the binary, without extensions or dots.

Extract information from binary

```
../../scripts/dump_macho.sh ./wc wc
  ↪ Created wc.dump
    Created wc.data
    ...
```

Run FoxDec. The command-line usage for FoxDec is:

```
foxdec-exe $PDF $DIRNAME $NAME
```

`$PDF` Either `0` or `1`. Iff `1` then Graphviz is used to generate PDFs from .dot files. For larger examples we recommend `0`, as Graphviz may get stuck on large graphs.
`$DIRNAME` Name of directory where the files created above (e.g., `$NAME.dump`) are located.
`$NAME` Use the same name as previously used.

Run FoxDec

```
foxdec-exe 1 ./ wc
```

Observe Output. At this point, FoxDec will have generated output concerning the *control flow* of the program, the *function boundaries*, it will have generated *invariants* and *disassembled instructions*, etc. All of this information is stored in a `.report` file, which can be accessed through a Haskell interface (see Section **??**). For sake of convenience, some of this information is also outputted in humanly readable formats. First, in the file `./$NAME_calls.pdf` an extended call graph is generated. Section 2 contains information on all the results stored in this file. For each function entry `$f`, a subdirectory has been created, and a control flow graph is generated in the file `$f/$NAME.pdf`. An overview of all resolved indirections can be found in the file `$NAME.indirections`. Finally, for each function entry `$f` a log has been maintained providing information on the results per entry (file `$f/$NAME.log` and an overall log has been maintained in `$NAME.log`.

Observe output
```
less wc.log
less wc.indirections
open wc_calls.pdf
less 7c0/wc.log
open 7c0/wc.pdf
```

1.3 Accessing information from `.report` file

All information in the generated `.report` file can be accessed through an interface. Implementation details on that interface, providing the exact list of functions that can be used to access the `.report` file, can be found here:

`https://ssrg-vt.github.io/FoxDec/foxdec/docs/haddock/VerificationReportInterface.html`

We have created several applications that use this interface to extract information from a `.report` file and provide output. The greyed out applications are currently under development.

Application	Functionality
foxdec-disassembler-exe	Basic Instruction Disassembly
foxdec-functions-exe	Function Boundaries
foxdec-controlflow-exe	Control Flow
foxdec-invariants-exe	Invariants
foxdec-isabelle-exe	Isabelle Code Generation
foxdec-symbolizer-exe	Position Independent NASM Generation

Basic Disassembly. Provides an enumeration of all instructions of all functions encountered while running FoxDec.

Basic Disassembly
```
foxdec-disassembler-exe wc.report
  ↪ 870: XOR EBP, EBP 2
    872: MOV R9, RDX 3
    875: POP RSI 1
    876: MOV RDX, RSP 3
    879: AND RSP, 18446744073709551600 4
    ...
```

Function Boundaries. Provides a coarse overview of the function boundaries of all functions encountered while running FoxDec. Splits the address ranges of the instructions belonging to the functions into chunks and shows their boundaries.

Function Boundaries

```
foxdec-functions-exe wc.report
  ↪ Function entry: 9ff
    9ff-->a9b
    Function entry: aa3
    aa3-->b3f
    ...
```

Control Flow. Given an instruction address, provides an overapproximative bound on the set of next instruction addresses. In the example below, address `0xada` may jump to two next addresses.

Control Flow

```
foxdec-controlflow-exe wc.report 0xada
  ↪ ada --> [adc,afc]
```

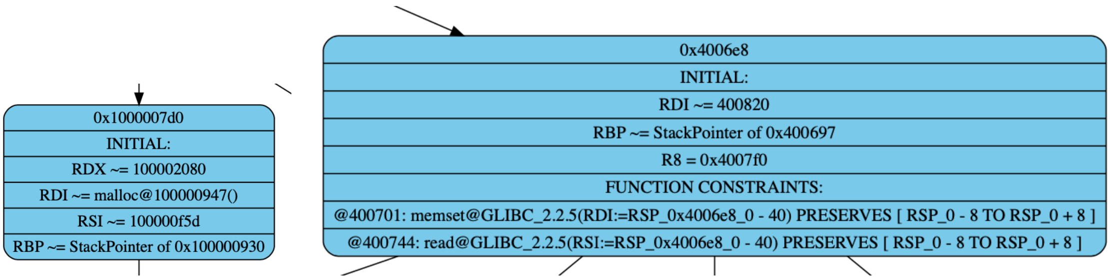
Invariants. Given an instruction address, produce the invariant. In the example below, some registers have not been modified wrt. their original value (e.g., `rcx` and `rdx`). The stack frame below the stack pointer stores certain values, e.g., the original value of register `rbp` and of the lower 32 bits of register `rdi`. The return address at the top of the stack frame has not been modified. Register `rax` holds an unknown value, returned by function `vfprintf`.

Invariants

```
foxdec-invariants-exe wc.report 0x9d1
  ↪ Invariant at address 9d1
    RIP := 0x9d1
    RAX := Bot[c|vfprintf@GLIBC_2.2.5|]
    RCX := RSI_0
    RDX := RDX_0
    RDI := Bot[m|[0x202080, 8]_0|]
    RSI := RSI_0
    RSP := (RSP_0 - 40)
    RBP := (RSP_0 - 8)
    R9 := R9_0
    R8 := R8_0
    [RSP_0, 8] := [RSP_0, 8]_0
    [(RSP_0 - 8), 8] := RBP_0
    [(RSP_0 - 12), 4] := b32(RDI_0)
    [(RSP_0 - 24), 8] := RSI_0
    [(RSP_0 - 32), 8] := RDX_0
    ...
    flags set by CMP(DWORD PTR [RBP - 4],0)
```

2 Annotated Call Graph

FoxDec produces a call graph with as vertices function entries, and an edge between two function entries if one function calls the other. The graph is *annotated* with information on assumptions and derived invariants made during verification. We maintain following four categories of information.



(a) Initialization (b) Function Constraints

Figure 1: Examples of vertices in annotated call graph.

INITIAL. Each function is verified with a certain initialization. An initialization is an initial predicate such that for any path in the binary, for any state in which the given function is called, the initial predicate holds. The initialization typically assigns pointer-relevant information to stateparts.

> **Example: INITIAL**
> Figure 1 contains a snippet of the annotated call graph produced using the running example. This initialization shows that at all times, when function with entry `1000007d0` is called, register `RDX` contains a pointer that roughly points to the global data section that contains address `100002080`. Register `RDI` contains a pointer produced by `malloc`. Register `RBP` contains a pointer to the stack frame of function entry `100000930`. The initialization does not provide exact information here, but sufficient to know that, e.g., the pointers in registers `RDX` and `RDI` point to separate regions.

FUNCTION CONSTRAINTS. When a function is called, it may be necessary ot make assumptions over it that cannot be proven. For external functions, we must make basic assumptions such as calling convention adherence. But even an internal function may require additional assumptions: it may write above its own stack frame, and in such case assumptions must be made that the return address of the caller is not overwritten. All these assumptions are summarized as function constraints.

Example: FUNCTION CONSTRAINTS

Figure 1 contains a snippet of the annotated call graph produced using the example `rop_emporium_ret2win/ret2win`. Two external functions are called (`memset` and `read`). Both functions are asumed to preserve the top of the stackframe of the caller (`4006e8`). In this example, function `read` may actually *violate* the assumption: it has been provided a pointer to the stackframe of the caller in register `RSI`, and needs to write more than 32 bytes to violate the assumption.

PRECONDITIONS AND ASSERTIONS. Preconditions and assertions formulate assumptions over separation of memory writes. A *precondition* states that two regions in memory are separate whose addresses can be defined in terms of the initial state of the function. An *assertion* states that two regions are separate at runtime, i.e., specifically during execution of a certain instruction.

Example: PRECONDITIONS AND ASSERTIONS

The following precondition:

StackPointer of `10000298c` SEP [`10000388c`, 8]_0

states that regions based on the stackpointer of the function with entry `10000298c` are separate from regions based on the pointer *initially* stored in the global variable with address `10000388c`.

The following assertion:

$$@\texttt{100000925}\text{: } (\texttt{RDX\_}0 + \bot) \text{ SEP } \bot_{\texttt{100000f5d}}$$

states that when the instruction at address `100000925` is executed, the adress of a memory write resolves to the initial value of register `RDX` plus some unknown value. The region pointed to by that resolved address is assumed to be separate from regions based in the global data section of address `100000f5d`.