UMEÅ UNIVERSITY

INCREASING COMPLETENESS OF ANDROID BINARY CFGS IN ANGR

Joel Gerdin

Bachelor Thesis, 15 hp/credits
Education program

2023

Abstract

Control Flow Graphs are widely used for static binary analysis today but their completeness is often lacking. The main problem when recovering Control Flow Graphs from binaries is indirect jump recovery where target addresses are stored in a register or memory. This problem is prevalent in the Python framework Angr when trying to generate a CFGFast for an Android binary compiled with the Bionic function `__libc_init`. At the moment an error is thrown, stating that the function `static_exits`, which resolves control flow transfers for Angr SimProcedures, is not implemented for the SimProcedure `__libc_init`.

In this paper, the missing function `static_exits` is implemented and verified. An answer to how the implementation of `static_exits` affects the CFGFast completeness in Angr is also presented. To answer the research question an experiment was performed on 11 binaries compiled with the Bionic function `__libc_init`, where CFGFasts were recovered for each binary file using the implemented `static_exits` function. The results gathered show that the expected indirect jump performed in `__libc_init` was resolved for each of the 11 binaries. The conclusion from this paper is that the completeness of CFGFasts in Angr recovered from binaries compiled with the Bionic library did increase after implementing the `static_exits` function.

Acknowledgements

I want to thank my supervisors Alexandre Bartel and Sabine Houy for helping me during my work on this thesis.

Contents

1 Introduction **1**
1.1 Research question 1

2 Related work **3**

3 Background **5**
3.1 Control Flow Graphs 5
3.2 Indirect jump resolution 6
3.3 Angr framework 6
3.4 Angr SimProcedures 7
3.5 Android Bionic library 7

4 Contribution **9**

5 Methodology **11**
5.1 Identify affected binaries 11
5.2 Implementing static_exits 11
5.3 Verification 12

6 Results **13**
6.1 Visually obtained results 13
6.2 Results from static analysis 14

7 Discussion **15**
7.1 Future work 15

8 Conclusion **17**

References **19**

A Appendix **21**
A.1 Control flow graphs from Android fauxware binary 21

1 Introduction

Control flow graphs (CFG) are used for static analysis of the control flow of a program and are recovered from executable binary files. CFGs can also be used for more advanced analysis of a program in pursuit of detecting vulnerabilities. A CFG consists of nodes and edges where the nodes are basic blocks and the edges are control flow transfers between blocks. The main problem during CFG recovery is the occurrence of indirect jumps in a program. An indirect jump is a control flow transfer where the target does not have a constant address, but an address stored in a register or memory and accessed only at run-time. Recovery of indirect jumps is important for increasing the completeness of a CFG, i.e. increasing the percentage of control flow transfers resolved for the program. If a CFG contains all possible control flow transfers of a program, it is considered complete and enables more precise program analysis.

Angr is a Python framework used for binary analysis and is capable of doing dynamic symbolic execution and other kinds of static analysis on binaries [1]. It supports analysis of binaries from multiple system architectures. One form of analysis that Angr offers is the construction of CFGs. There are two kinds of CFG:s in Angr, CFGFast and CFGEmulated. CFGFast is a static CFG that is generated by using static analysis while CFGEmulated is a dynamic CFG that is generated with symbolic execution [2].

This paper focuses on increasing the completeness of CFGFasts in Angr and in particular for CFGFasts recovered from Android binaries using the Bionic library. The Bionic function `__libc_init` which is replaced by a SimProcedure, a class used for replacing functions in Angr, is performing an indirect jump that is not resolved during recovery of a CFGFast for that binary. At the moment an exception is thrown when trying to recover a CFGFast for such binaries in Angr. It is known that there is a lack of research on static analysis of native Android libraries such as Bionic [3]. Resolving the indirect jump in `__libc_init` will increase the completeness of CFGFasts in Angr for Android binaries using the Bionic library.

The outline of this paper is as follows: Section 2 presents related work for this thesis; Section 3 introduces the reader to relevant concepts used for this paper; Section 4 describes the contribution of this thesis; Section 5 goes through the methodology behind verification of the `static_exits` function; Section 6 presents results from the verification process; Section 7 contains a discussion about the results, an answer to the research question and possible future work; finally, Section 8 is a conclusion of this paper.

1.1 Research question

How implementing the `static_exits` function in Angr affects CFGFast completeness for Android binaries.

2 Related work

This chapter presents two papers that focus on CFG construction and mainly the complications indirect control flow transfers introduce when it comes to CFG completeness. The first paper describes a method of reconstruction of interprocedural control flow graphs, and the second paper presents the tool Jakstab that can reconstruct CFGs for binaries.

In a paper from 2000, De Stutter et al. [4] describe a method used for handling indirect control flow transfers when constructing CFGs from binary programs. Because binary modification had at the time become increasingly more used in for example binary translation, debugging and whole program optimization at run time, the demand for binary analysis techniques had increased. The authors focus only on the reconstruction of interprocedural control flow graphs (ICFG) in this paper which is a combined CFG of all CFG construction procedures. Constructing the static control flow of the binary was not a challenge but resolving indirect control flow transfers, for example, control flow dependent on an address table or data in memory/registers was more challenging. They solved this by first building a CFG that contained hell nodes and hell edges as an abstraction for unknown targets and control flow transfers, i.e. indirect calls. This CFG is called a conservative CFG and is the predecessor of a more precise CFG. Static analysis was used to refine the conservative CFG by trying to resolve each existing hell node and hell edge in the intermediate conservative CFG. Two types of hell edges are described, outgoing and incoming. Outgoing are edges that point to a hell node and incoming are edges that point from a hell node.

Another paper by Kinder et al. [5] presents the tool Jakstab (Java toolkit for static analysis of binaries) which is used for static analysis and model checking on executable files. The main reason behind developing Jakstab was for the detection of malicious code in executable files. Jakstab has the capability to generate CFGs that have more indirect jumps resolved than the leading disassembly tool at the time, IDA Pro [6]. This was accomplished by performing an iterative CFG construction process that interleaves disassembly rounds with dataflow analysis which in turn produces a more complete and accurate CFG. At the time there were two categories of disassemblers: *Linear sweep disassemblers*, for example, GNU objdump [7] and *Recursive traversal disassemblers* such as IDA Pro. *Linear sweep disassemblers* translates machine code into assembly sequentially and *Recursive traversal disassemblers* use depth first search to decode a program. What Kinder et al. acknowledged was that recovering indirect jumps and in particular function pointers required dataflow analysis on incomplete CFGs. Therefore they defined a new type of disassembler: an *iterative disassembler* which enabled iterative CFG construction.

The process of creating a CFG in Jakstab starts from the entry point of the executable. From there constants stored in registers and memory are propagated and folded to resolve indirect targets. If an address of an indirect write cannot be resolved an assumption is made that every memory cell can become undefined. Stub procedures are created in the CFG if a call to a shared library is made. All known parts of the CFG is constantly propagated and folded to resolve as many indirect jumps as possible. Sometimes during the iterative process new incoming edges to existing nodes can be discovered, rendering the CFG to be incorrect. If that

scenario occurs the whole process has to be restarted. When a target has been resolved it is scheduled to be disassembled in the next iteration of the CFG reconstruction process.

Both papers discussed in this section focus on the importance of indirect jump resolution for increasing CFG completeness. This paper contributes to more complete CFGs in Angr by resolving an indirect jump occurring in some Android binaries.

3 Background

For understanding the process of answering this papers research question, background knowledge of used concepts is required. Section 3.1 gives an introduction to CFGs and the recovery of CFGs in addition to a more specific explanation of CFGs in Angr. Thereafter an explanation of indirect jump resolution is presented in Section 3.2. Section 3.3 presents an overview of the Python framework Angr, followed by an explanation of Angr SimProcedures in Section 3.4. Lastly, an introduction to the Android Bionic library is presented in Section 3.5.

3.1 Control Flow Graphs

A control flow graph (CFG) is used for static analysis of software, which in turn is a form of analysis that analyses a program without executing it [8]. A CFG is constructed of nodes and edges where the nodes are blocks of program instructions called basic blocks and the edges are jumps between the blocks [8]. Together they form a graph that describes the control flow of a given program or function which can be used for static analysis. The creation of a CFG is done by recursively analyzing and disassembling basic blocks of a program. Each block is traversed looking for possible exit points to new blocks. If an exit is found the new block is added to the graph. When there are no more exits left the CFG construction process terminates [8]. There is one known challenge when it comes to CFG recovery and that is indirect jumps. A direct jump has a known target encoded into the instruction whereas an indirect jump happens when control flow is redirected to a target located in a register or memory location.

When recovering a CFG, the main objective is to find as many of the indirect jump targets as possible. A CFG has two properties determining the success of the recovery technique; Soundness and Completeness. **Soundness** is determined by the percentage of potential control flow transfers that were generated for the CFG. A completely sound CFG recovery technique resolves all possible control flow transfers for a CFG while a partially sound CFG recovery technique only resolves a subset of all potential transfers [8]. A **complete** recovery technique generates a CFG where all edges are possible control flow transfers [8].

In the Angr framework, there are two kinds of control flow graphs, CFGEmulated and CFGFast. CFGEmulated is generated using symbolic execution while CFGFast is generated using static analysis [2]. This thesis covers work that is only dependent on CFGFast, therefore CFGEmulated will not be described further. The procedure that Angr uses to recover a CFGFast consists of five steps starting from an entry point (an entry block) or a user-defined point [2]:

1. The basic block of the entry point is converted into the intermediate representation VEX. Thereafter all exits are collected, including jumps, calls, returns and continuations to other basic blocks.
2. All exits are checked for constant addresses, if found a new edge is added to the CFG. The destination block is then queued to be analyzed later.

3. If an exit is considered a function call, the destination block is regarded as the beginning of a new function and if the function usually returns, the block following the call is examined as well.

4. If an exit is of type return, the function returns, and the edges of the CFG are altered appropriately.

5. Indirect jump resolution is performed if the jump has no constant destination.

3.2 Indirect jump resolution

Indirect jumps are jump instructions where the target is not a constant address, unlike direct jumps where the address is part of the jump instruction. Instead, the address is stored in a register or in memory [9]. There are three categories of indirect jump calls [8]:

Computed Computed jumps are calculated in the code. An example of such a computed jump is when the application calculates an index from values stored in a register or memory in order to look up an address in a jump table.

Context-sensitive These jumps depend on the application context. For example when a function takes a callback as an argument that decides the next jump.

Object-sensitive Similar to the context-sensitive jump. Polymorphism in object-oriented languages requires virtual tables for virtual functions in order to determine a jump target. These jumps are determined at run time and are dependent on the type of object that is passed to the function.

During the construction of a CFGFast Angr uses several techniques for resolving indirect jumps. For intra-functional control flow transfers, Angr performs lightweight alias analysis, data flow tracking and other predefined strategies. Angr also includes jump table identification and indirect call target resolution for resolving indirect jumps [2, p.12].

3.3 Angr framework

Angr [10] is an open-source tool used for binary analysis and gives the user equipment to perform symbolic execution and static analysis on binaries. The tool is able to analyze binaries from multiple system architectures. Some of the built-in functionality for binary analysis are Fast Control Flow Graphs (CFGFast), Emulated Control Flow Graphs (CFGEmulated), Value Flow Graphs (VFG), and Data Dependency Graphs (DDG) [1].

To support static analysis for multiple CPU architectures Angr uses the intermediate representation (IR) VEX which is borrowed from Valgrind [11]. Before any static analysis is performed, the binary is lifted to VEX IR which abstracts away all architecture-specific differences into one common representation. Doing so enables Angr to write static analysis for all supported CPU architectures in one go instead of separate implementations for all architectures [12].

3.4 Angr SimProcedures

The class SimProcedure in Angr enables the user to modify the behavior of a program by hooking a function with a replacement procedure. Every time the address of that function is processed by the execution pipeline, the SimProcedure hooked to that function is executed instead. The reason for using SimProcedures is to avoid path explosion which happens when new exits grow exponentially. Initially, Angr used SimProcedures mostly for replacing library functions that were known to cause path explosion but it can also be used for replacing functions with a simpler function with the same outcome [13].

To help out with static analysis that involves SimProcedures it is possible to set three class variables for the SimProcedure. These are NO_RET, ADDS_EXITS and IS_SYSCALL. When NO_RET is set to true the function does not return, ADDS_EXITS is set to true if there is additional control flow other than returning and IS_SYSCALL is set if the function is a system call. The ADDS_EXITS variable is of interest when building a CFGFast in order to resolve indirect jumps. If it is known that the function replaced by the SimProcedure adds new exits, static analysis and heuristics are required in order to resolve those exits. In order to add new exits the function `static_exits` has to be implemented for the specific SimProcedure. The `static_exits` function takes one argument which is a list of Intermediate Representation Super Blocks (IRSB) [12] which are to be statically analyzed in order to return a list of tuples (address, jumpkind, namehint) containing the exits found [14]. These can later be used for constructing a more complete CFGFast of a program.

3.5 Android Bionic library

The Android Bionic library is a custom implementation of the C library libc and was developed for Android devices. There were three main reasons for the development of Bionic: the first reason was to solve a problem regarding software licensing, the second reason was to reduce size because of the limited hardware capabilities of mobile devices and the third reason was to increase speed because of the limited CPU power of mobile devices [15]. Bionic supports Linux kernels exclusively and the specific architectures supported are: arm, arm64, riscv64, x86, and x86-64 [16].

4 Contribution

This paper's contribution is to increase the completeness of CFGFasts in Angr created from Android binaries. Specifically, binaries compiled with the Android Bionic library, containing the `__libc_init` program startup function. The reason behind the work is that CFGFasts generated from binaries containing the Bionic `__libc_init` function cause an exception in Angr, stating that the function `static_exits` for the SimProcedure `__libc_init` is not yet implemented. By identifying affected binaries, implementing the function `static_exits`, and verifying its implementation, an increase in completeness for CFGFasts generated from Android binaries in Angr is achieved. By doing so, static analysis of Android binaries is improved and enables more precise analysis of such binaries.

5 Methodology

To answer the research question, how implementing the static_exits function for the SimProcedure __libc_init affect the completeness of CFGFasts in Angr, experiments were performed. Before doing the experiment, binaries causing the error had to be obtained and the static_exits function had to be implemented. Section 5.1 describes the process of identifying coveted binaries and Section 5.2 explains how static_exits was implemented. Finally, Section 5.3 goes through the verification process using the binaries identified from Section 5.1 and the implemented static_exits function explained in Section 5.2 in order to answer the research question.

5.1 Identify affected binaries

Finding the kind of binaries reproducing the error stating that the static_exits function is not implemented, is accomplished by understanding the Bionic function __libc_init more thoroughly. Because the SimProcedure __libc_init replaces the Bionic function with the same name, binaries compiled with the Bionic library will trigger the error. Identifying such binaries is therefore achieved by creating CFGFasts for Android binaries in Angr and checking if the error was reproduced.

5.2 Implementing static_exits

Before implementing the function static_exits it is necessary to understand what the function __libc_init does and which potential control flow transfers performed. The source code documentation [17] describes the function as a launch function called after the dynamic linking of the program is finished. The function takes four arguments; raw_args, onexit, slingshot and structors_array_t. The argument of interest here is Slingshot, which is a function pointer to the main function of the program and is used for the startup of the application. This is the only exit performed by __libc_init and is, therefore, the only exit that static_exits needs to resolve. The implementation developed took inspiration from another static_exits implementation located in the SimProcedure __libc_start_main in Angr.

As described in Section 3.4 the static_exits function takes one argument; a list of basic blocks executed previous to the SimProcedure. Statically analyzing these blocks makes it possible to find addresses of indirect jump targets stored in registers or memory during the execution of the SimProcedure. The implementation proposed here consists of four steps:

1. Create a blank state and set up a stack pointer.
2. Execute each block using the Angr default engine which executes a basic block and returns an object with references to each successor block. If a successor is found it is

stored in a state variable.

3. Dump the arguments of the found state using the right calling convention for the specific architecture.

4. Return a dictionary on the form (address, jumpkind, namehint) containing all exits found from the analysis.

The dictionary returned from `static_exits` is used in the construction of CFGFast and will further increase the completeness of the CFG.

5.3 Verification

This section presents two verification techniques: static analysis and visual analysis of CFG-Fasts. Visual analysis of a CFGFast suits smaller binaries that generate reasonably sized graphs while static analysis is required for larger binaries that generate too large graphs.

The static analysis technique consists of generating two CFGFasts for a binary compiled with the `__libc_init` function. One CFGFast that does not resolve exits from `__libc_init`, and one CFGFast that does resolve exits using the new `static_exits` implementation. By statically analyzing both CFGFasts and checking if the outgoing edge from the `__libc_init` node has the main function as its target, or if the `__libc_init` node does not have any outgoing edges, verification of the `static_exits` function could be performed. If there was an edge connecting the `__libc_init` node and the main function node, the `static_exits` function was implemented correctly. Verifying that the edge actually had the main function as its target was achieved by checking all outgoing edges from `__libc_init` and checking if the target address matches the first block of the main function. Ten binaries compiled with the Bionic library were analyzed using this technique.

Visual verification is similar to static analysis but instead of analyzing the CFGFast statically, an image of the CFGFast is examined for edges connecting `__libc_init` and main. Two CFGFasts were generated, one with resolved exits from `__libc_init` and one without. By checking that an edge from `__libc_init` and main was added in the graph with resolved exits from `__libc_init`, verification of the implementation could be done. Only one binary file was analyzed using the visual analysis technique.

6 Results

The results gathered from the verification of `static_exits` in Section 5.3 are presented in this section. Results from two kinds of verification techniques, visual and static analysis of CFGFasts, are presented.

6.1 Visually obtained results

Figure 1 shows a zoomed-in part of a full CFGFast recovered from a binary where `static_exits` is not implemented. The figure shows three small CFGFasts and a small part of a fourth CFGFast. The second CFGFast from the left shows control flow for `__libc_init` and the partial CFGFast to the right shows part of the control flow of the main function. The full CFGFast can be found in appendix A.1.

Figure 1: CFGFast of Android fauxware binary. `static_exits` in `__libc_init` not implemented.

Figure 2 shows a zoomed-in part of a full CFGFast recovered from a binary where `static_exits` is implemented. There are three CFGFasts, the one in the middle is only shown partially and contains control flow from `__libc_init` to the main function of the program. The full CFGFast for Figure 2 can also be found in appendix A.1.

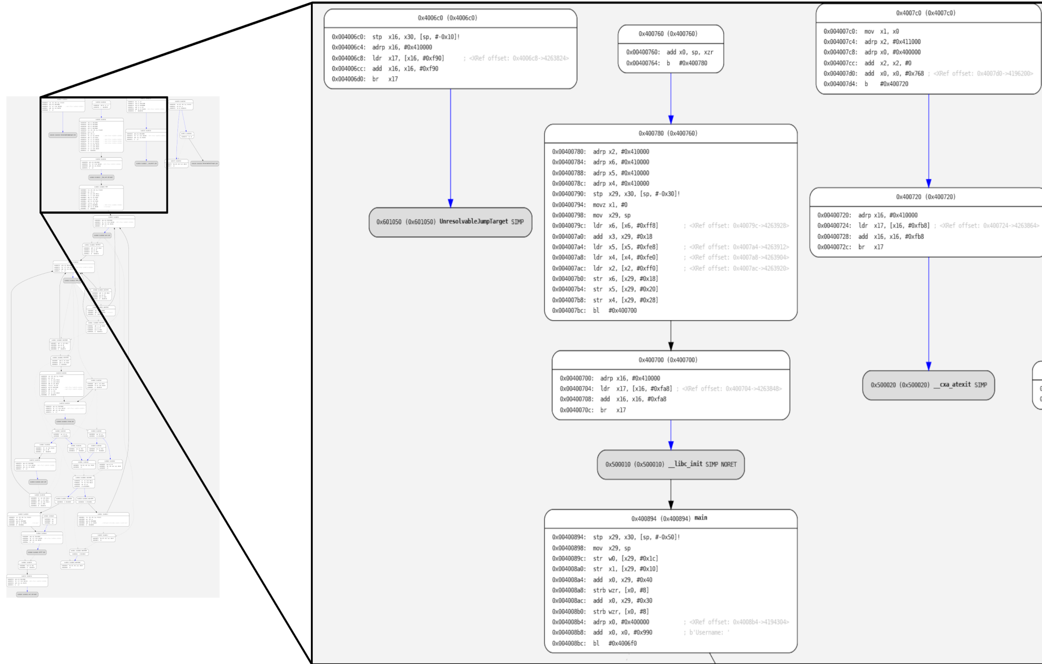

Figure 2: CFGFast of Android fauxware binary. `static_exits` in `__libc_init` implemented.

6.2 Results from static analysis

Results from statically analyzing CFGFast binaries compiled with the Bionic library are presented in this section. Table 1 shows that ten out of ten CFGFasts resolved exits from `__libc_init` to main using the new implementation of `static_exits`, described in Section 5.2.

Table 1 Table displaying results from analysis of ten CFGFasts generated from Android binaries. Resolves exit is a true or false value indicating if the jump from `__libc_init` to main was resolved.

Binary name	Exit Resolved
dexdump	yes
grep	yes
iptables	yes
logcat	yes
ndc	yes
netd	yes
oatdump	yes
ping	yes
reboot	yes
sh	yes

7 Discussion

The results shown in Section 6 can be used to answer the research question for this thesis. Results were gathered using two different techniques; visual and static analysis of CFGFasts. These results will be discussed and help answer how implementing `static_exits` for the SimProcedure `__libc_init` affects the completeness of CFGFasts in Angr. This section also discusses the correctness of the `static_exits` function.

Section 6.1 contains results from visual analysis of two CFGFasts generated from a binary containing the Bionic library. The expected behavior of `static_exits` is to resolve outgoing exits from `__libc_init`. These exits can be used when building CFGFasts in Angr from binaries using the Bionic `__libc_init` function. Figure 1 shows a CFGFast from such binary without `static_exits` implemented and as expected, there are no outgoing edges from the `__libc_init` node. Figure 2 on the other hand uses the implementation of `static_exits` developed for this thesis, and it is showing an outgoing edge from `__libc_init` to the main function of the application. This is the only expected exit from `__libc_init` which help verify the implementation of `static_exits`.

Section 6.2 shows results gathered from static analysis of CFGFasts recovered from 10 binaries containing the Bionic library. The verification methodology is described in Section 5.3. Results are shown in Table 1 and clearly show that the implementation of `static_exits` in the SimProcedure `__libc_init` resolves the expected exit for the ten binaries tested.

CFG completeness, as described in Section 3.1, is the percentage of possible control flow transfers recovered when generating a CFG from a binary file. By implementing the `static_exits` function, which resolves a new exit for CFGFasts recovered from Android binaries containing the `__libc_init` Bionic function, it is clear to say that the completeness of CFGFasts has increased. This impacts only the completeness of CFGFasts recovered from binaries compiled with the Bionic library using the `__libc_init` function.

7.1 Future work

Closely related work is to implement a SimProcedure in Angr for the function `__cxa_atexit` that exists in the GNU C Library (glibc) [18]. The SimProcedure is not implemented in Angr yet so implementing it and resolving its exits would further increase the completeness of Android binaries. This work would include verifying the solution as well. This problem is prevalent in Figures 1 and 2 as one of the CFGs contains the `__cxa_atexit` function which is not connected to the main CFG.

Broader work covers indirect jump resolution in general as it is a known issue when it comes to static analysis of binaries. The incompleteness of CFGs is most often caused by indirect jumps that are difficult and tedious to resolve.

8 Conclusion

By implementing the missing function `static_exits` in Angr, the completeness has increased for CFGFasts generated from binaries compiled with the function `__libc_init` from the Android Bionic library. Doing so has made the analysis of such binaries more precise and could help the detection of vulnerabilities in Android binaries in the future.

References

[1] "Angr documentation, "introduction"," https://docs.angr.io/en/latest/quickstart.html, accessed: 2023-04-21.

[2] "Angr documentation, "control-flow graph recovery (cfg)"," https://docs.angr.io/en/latest/analyses/cfg.html, accessed: 2023-04-21.

[3] P. Lantz and B. Johansson, "Towards bridging the gap between dalvik bytecode and native code during static analysis of android applications," in *2015 International Wireless Communications and Mobile Computing Conference (IWCMC)*. IEEE, 2015, pp. 587–593.

[4] B. De Sutter, K. De Bosschere, P. Keyngnaert, and B. Demoen, "On the static analysis of indirect control transfers in binaries," in *Proceedings of the international conference on parallel and distributed processing techniques and applications*, vol. 2. CSREA PRESS, 2000, pp. 1013–1019.

[5] J. Kinder and H. Veith, "Jakstab: A static analysis platform for binaries: Tool paper," in *Computer Aided Verification: 20th International Conference, CAV 2008 Princeton, NJ, USA, July 7-14, 2008 Proceedings 20*. Springer, 2008, pp. 423–427.

[6] hex rays, "Ida as a disassembler," https://hex-rays.com/ida-pro/ida-disassembler/.

[7] ——, "objdump," https://sourceware.org/binutils/docs/binutils/objdump.html.

[8] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE symposium on security and privacy (SP)*. IEEE, 2016, pp. 138–157.

[9] A. Westerberg, "Proof-producing resolution of indirect jumps in the binary intermediate representation bir," 2021.

[10] "angr, "angr"," https://angr.io/, accessed: 2023-06-01.

[11] Valgrind.org, "Valgrind," https://valgrind.org/.

[12] "Angr documentation, "intermediate representation"," https://docs.angr.io/en/latest/advanced-topics/ir.html, accessed: 2023-04-21.

[13] E. Cheng, "Binary analysis and symbolic execution with angr," Ph.D. dissertation, PhD thesis, The MITRE Corporation, 2016.

[14] "Angr documentation, "hooks and simprocedures"," https://docs.angr.io/en/latest/extending-angr/simprocedures.html, accessed: 2023-04-21.

[15] Google, "Anatomy physiology of an android," https://sites.google.com/site/io/anatomy--physiology-of-an-android.

[16] A. O. S. Project, “Android 6.0 compatibility definition,” https://source.android.com/docs/compatibility/6.0/android-6.0-cdd.

[17] “AndroidXRef, ”libc$_{i}nit_{d}ynamic.c$”,′′ , accessed: 2023-04-22.

T. G. C. L. (glibc), “The gnu c library (glibc),” https://www.gnu.org/software/libc/.

A Appendix

A.1 Control flow graphs from Android fauxware binary

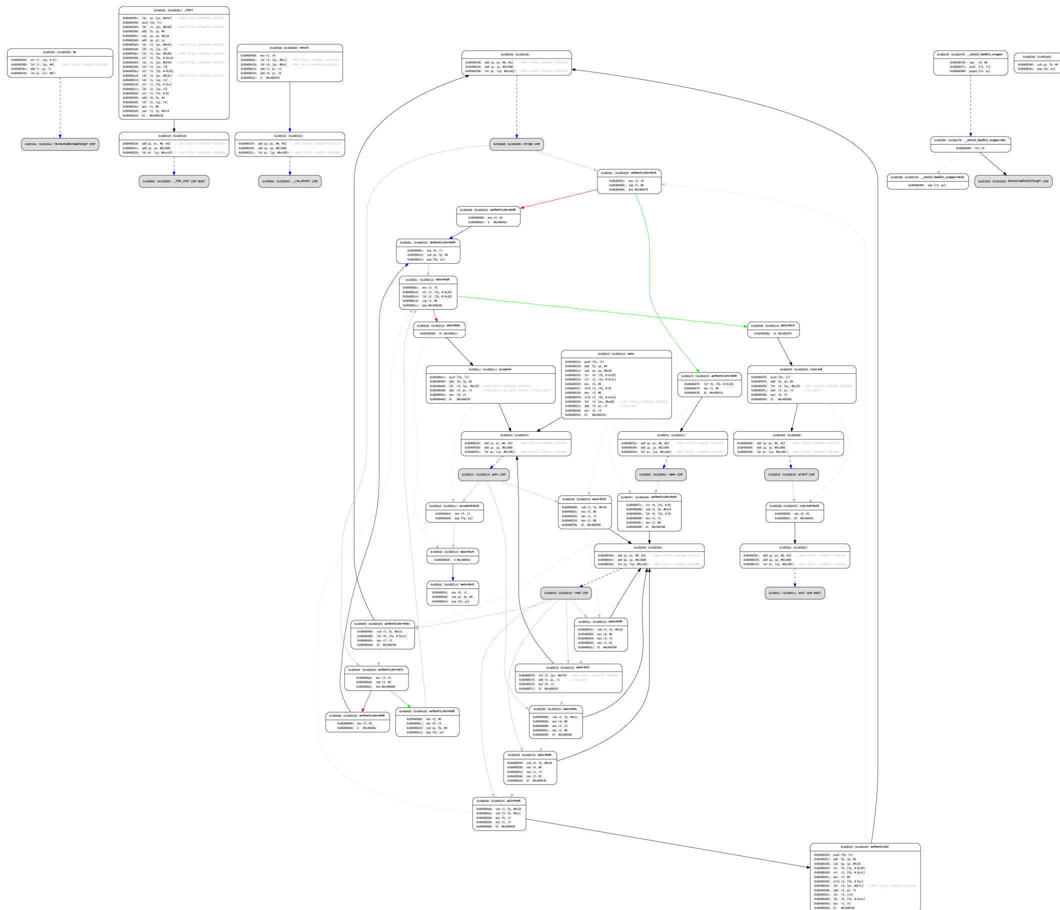

Figure 3: CFGFast of Android fauxware binary. `static_exits` in `__libc_init` not implemented.

Figure 4: CFGFast of Android fauxware binary. `static_exits` in `__libc_init` implemented.

UMEÅ UNIVERSITY