dAngr: Lifting Software Debugging to a Symbolic Level

Dairo de Ruck
DistriNet, KU Leuven
3001 Leuven, Belgium
dairo.deruck@kuleuven.be

Jef Jacobs
DistriNet, KU Leuven
3001 Leuven, Belgium
jef.jacobs@kuleuven.be

Jorn Lapon
DistriNet, KU Leuven
3001 Leuven, Belgium
jorn.lapon@kuleuven.be

Vincent Naessens
DistriNet, KU Leuven
3001 Leuven, Belgium
vincent.naessens@kuleuven.be

***Abstract*—Debugging is a fundamental testing technique that directly interacts with the functionality and current state of a running program. It enables the debugger to step through a program and meanwhile inspect registers and memory as part of the program state. When debugging, variables and parameters are assigned concrete values resulting in a specific program path to be explored. This makes software testing time-consuming and at the same time requiring substantial expertise. On the other hand, symbolic debugging can explore multiple paths by replacing concrete input values by symbolic ones and choose the paths to be explored.**

`angr` is a dynamic symbolic execution (DSE) platform that can be programmed to symbolically execute a binary program with selected, possibly symbolic inputs. The binary is lifted to an intermediate, architecture independent representation, preparatory to the symbolic execution.

This paper presents `dAngr` a tool that builds upon `angr`, a symbolic execution platform, enabling the user to debug binaries by means of GDB-like commands, and enhances this experience by means of symbolic execution and binary analysis capabilities. We also abstract the `angr` framework and symbolic execution by utilizing these commands. The power of `dAngr` is demonstrated on multiple examples including capture-the-flag challenges with different levels of complexity.

I. Introduction

Symbolic execution is a program assessment technology that is applied for various purposes. Formal verification [1], software testing [2], smart contract assessment [3], [4], [5], [6] and vulnerability detection [7], [8], [9] are just a few examples. The technique explores the state space of a program by executing the program with symbolic values instead of concrete ones. In this paper, we focus on the software testing and vulnerability detection capabilities of symbolic execution. Symbolic execution can explore multiple program paths at once, amongst others, paths that are hard to reach without prior knowledge of concrete values that would reach such a branch. In contrast to traditional testing mechanisms, it can explore the state space of a program by means of symbolic values. Moreover, the symbolic execution engine can generate test cases that cover different parts of a program, and support bug detection in programs.

This technology was introduced by King et al. [10] in the 1970s, and has been a research subject since. There are many symbolic execution engines that have been developed over the years, each with their own strengths and weaknesses, for both static, concolic, and execution-generated engines. Together with `KLEE` [2], `S2E` [11], and `Manticore` [6], `angr` [12] is one of the most popular symbolic execution engines in the field. Like `Manticore` and `S2E`, `angr` can execute x86 and ARM binaries without access to the source code. Unlike the others, `angr` can assess binaries compiled to more obscure architectures, like MIPS and PowerPC. `angr` itself is more an extensible and versatile framework than a tool. It requires a customized Python program to analyze a specific binary. The code that interacts with it, and how it interacts with the binary, needs to be written by the user. This means that it has no predefined objective, and thus the user must define the analysis using the interfaces `angr` exposes.

This paper introduces "debugging with `angr`" (or `dAngr`). `dAngr` is an open-source[1] CLI tool that extends `angr` with GDB-like debugging capabilities, allowing the user to assess a binary with symbolic execution. This has several advantages over traditional debugging techniques. First, it can deal with symbolic values. This enables the user to explore the entire program space, including hard to reach branches, and evaluate the required inputs to take that specific path. This is very useful to assess unintended program behavior. Second, the user can define a custom entry state, starting the symbolic execution at any point in the binary. Hence, testers can easily select the parts of the program to be executed. In the context of large or complex binaries, assessors can execute critical functions directly and skip irrelevant parts. Finally, because `angr` lifts the binary to VEX IR, the debugger can be applied across multiple architectures.

This paper is structured as follows. In Section II we discuss the broader work in the field of symbolic execution, binary analysis tools and the combination thereof. Subsequently, we provide an overview of design objectives in Section III. In Section IV we discuss the implementation and features of

Workshop on Binary Analysis Research (BAR) 2025
28 February 2025, San Diego, CA, USA
ISBN 979-8-9919276-4-2
https://dx.doi.org/10.14722/bar.2025.23014
www.ndss-symposium.org

[1]GitHub project https://github.com/angr-debugging/dAngr
PyPi package https://pypi.org/project/dAngr/

dAngr more specifically, the features and necessary modifications made to accommodate a debugger that satisfies those objectives. Next, we evaluate dAngr on a set of example binaries, highlighting the effectiveness of the tool and its limitations, in Section V. Finally, we conclude the paper and point to current, and future work (in progress), in Section VI and VII.

II. Background and related work

Symbolic execution has gained significant attention in recent years due to its ability to explore program behavior systematically, making it a valuable tool for software testing, verification and security analysis. At its core, symbolic execution treats program inputs as symbolic variables rather than concrete values. This enables exploration of multiple execution paths and the detection of potential issues.

Symbolic execution can be broadly categorized into static symbolic execution (SSE) and dynamic symbolic execution (DSE).

SSE translates program fragments into formulas that represent the behavior of all possible paths within the fragment. This approach is often used in program verification tools such as ProVerif, Dafny and Verifast to formally prove properties such as the absence of vulnerabilities or concurrency issues [13], [14], [15].

DSE, on the other hand, generates path-specific constraints during each step of the execution. This approach is particularly well-suited for bug detection and test case generation. Tools such as KLEE, SE2 and angr employ DSE to uncover potential runtime issues like buffer overflows or division-by-zero errors [2], [11], [12].

In DSE, symbolic and concrete execution can be interleaved. This hybrid model, called concolic execution, is not always necessary. For instance, angr is capable of operating in a fully symbolic mode, enabling comprehensive path exploration without concrete values.

In this paper, we focus on dynamic symbolic execution, which forms the foundation of the symbolic execution engine used in our debugger.

During *Dynamic Symbolic Execution*, program memory and state are symbolically and concretely traced, enabling detailed analysis of potential erroneous interactions. Previously unseen variables (e.g., memory, registers, ...) are initially *un*constrained, meaning they could embody any value within the domain of the variable. The symbolic values are propagated throughout the program, thereby accumulating constraints with each operation. When it encounters a branching condition, symbolic execution creates states for each potential path. For instance, an if statement may result in two states in which the branch condition is added to the constraints of one state, and its negation to the other state. By systematically solving constraints for unexplored paths, DSE ensures thorough testing and the discovery of hard-to-find issues. When a bug is detected, symbolic constraints capturing the program state at the point of failure can be solved to generate concrete inputs, facilitating efficient bug reproduction and debugging.

As the symbolic engine explores the state space using the program's control flow, the engine essentially navigates through the program's Control Flow Graph (CFG). The choice of the next state after each step heavily impacts exploration performance. Breath-first-search (BFS) approaches prioritize the oldest states to be handled first, while depth-first-search (DFS) keeps working with the most recently generated states. As a result, BFS has difficulties in memory usage as it may result in a large amount of states kept in the engines memory, while, on the other hand, DFS may get stuck in loops.

A. Debugging tools

Debugging allows the assessor to understand the inner workings of the binary. Popular examples are the GNU Debugger (GDB) [16], Microsoft Visual Studio and lldb [17]. Debugging enables the user to gain an unusual insight into the binary, going from program flow, to register workings, and memory values at any point in the debugging session. Experienced users are able to manipulate the program under analysis by defining their own values into registers and memory regions, testing the limits of the program.

We identify the following important features of command-line debuggers:

- **Breakpoints** allow the user to interact with the program during execution. Breakpoints can be set at specific lines or functions in the source code or addresses in the binary, with the possibility of adding a conditions. The program execution pauses when it reaches the breakpoint (matching the condition, when specified).
- **Step Execution** enables the user to execute the program one line or instruction at a time, which helps in closely monitoring the program's behavior.
- **Variable Inspection** provides the ability to inspect the values of variables at different points during program execution.
- **Stack Inspection** allows the user to inspect the stack at any point during the program execution, which helps to understand what variables and calls are stored on the stack.
- **Backtrace** displays the call stack at any point during the program execution, which helps in understanding the sequence of function calls that led to a particular state.
- **Core dumps** allow the user to analyze snapshots of a program's memory at the time of a crash.
- **Thread Debugging** provides tools to debug multi-threaded applications, including setting breakpoints in specific threads and inspecting thread-specific data.

GDB is extended with several plugins over time. Popular plugins include gdb-peda [18] and gdb-gef [19]. They visualize information, such as the stack, current instruction and registers, including color coding and dereferencing addresses. This results in a user-friendly and readily available display of information, enabling reverse engineers to monitor the current state of the program. In addition, GDB allows for

simple arithmetic operations and conversions, which are handy for pointer arithmetic and offset calculations. We keep these features in mind when designing dAngr, as we want to provide the user with a similar experience like GDB. GDB, by default, can only debug binaries intended for the same architecture as the host machine. Debugging binaries for different architectures requires utilizing emulation software like QEMU, which increases the complexity of the debugging process.

Symbolic debugging tools are relatively rare. Most existing symbolic debugging tools are primarily designed to assist in resolving issues encountered during software verification or formal proofs. Notable examples include symbolic debugging tools developed for the KEY-project (SED), Verifast and Gillian [20], [15], [21]. While SED and Gillian could be used during software development and testing, it is not their primary goal, and defer greatly from standard debugging approaches.

B. Reverse engineering tools

Reverse engineering tools are used to disassemble and decompile binaries. Ghidra [22], IDA [23], and BinNinja [24] are popular reverse engineering tools, providing a more readable representation of the binary. In addition, they support debugging functionality, which makes these tools very useful for reverse engineers. These tools can be extended with plugins to further enhance their functionality.

Both Ghidra and IDA have been extended to interface with angr. angryGhidra [25] is a plugin made for Ghidra, which allows users to leverage angr's functionality and rely on symbolic execution to explore the binary. The plugin exposes some angr functionalities, but lacks the ability to debug a binary with symbolic execution, as the plugin constrains most of angr's functionalities by only presenting the user the ability to execute snippets of code with concrete arguments. Another example is angrdbg [26]. This is a package that synthesizes a state for angr based on a debugging session in tools like GDB, IDA and Radare2. This is useful to mimic the program state when invoking a specific part of the binary, for example, skipping an unsupported function. In contrast to our approach, it does not create an abstraction layer over angr, nor does it provide an interface to create a debugging experience that leverages the advantages symbolic execution brings with it. Finally, angr-cli [27] is a Python package that creates an interactive CLI environment, with the possibility to get a visual representation in the style of gdb-gef. Since it is a package, users are required to write Python code to configure and define the angr project. The package is also limited in the available angr features it exposes.

To our knowledge, there is no tool that allows debugging with symbolic execution in a GDB-like manner.

III. General overview

While symbolic execution was initially introduced as a technique for program testing [28], its primary applications have evolved to include testcase generation, vulnerability detection, program verification, and formal proofs. Software debuggers on the other hand, are widely used, particularly during the testing phase of software development.

dAngr aims to bridge these two paradigms by combining the strengths of symbolic execution and debugging. We chose angr as our symbolic execution engine over alternatives like KLEE, Manticore and S2E for a few reasons:

1) angr is a comprehensive and extensible framework for binary analysis,
2) it is architecture-independent and supports less common architectures such as MIPS,
3) it can analyze binaries without requiring source code, making it ideal for reverse engineering and security research.

This section outlines the capabilities and features of angr that dAngr relies upon, the modifications we made to facilitate the debugging, and the design considerations employed to reduce the complexity, and improve user experience with symbolic execution.

A. The angr Framework

Since the seminal paper of Shoshitaishvili et al. [12], angr has undergone significant evolution. It now supports an increasing amount of architectures, boasts overall stability and efficiency, and includes advanced features such as automatic ROP chain generation, and automated exploit generation.

angr combines several key components to deliver a powerful symbolic execution framework.

- CLE loader: The CLE loader [29] initializes a project by loading the binary and its libraries into an abstract memory model. This serves as the foundation for subsequent analysis.
- VEX IR Lifting: The binary is lifted to VEX IR [30], a RISC-like intermediate representation. This abstraction decouples the symbolic execution from architecture-specific details, enabling cross-platform analyses. angr includes the ability to lift binaries from a wide range of architectures.
- Symbolic Execution Initialization: Users define symbolic variables (bit-vectors) and assign them to memory locations or registers. An initial state is specified, serving as the entry point for symbolic execution.
- Simulation Manager: This component manages the actual symbolic execution by propagating symbols throughout the program, maintaining access to program states produced during the execution. It facilitates in-depth analysis of memory, basic blocks, and program behavior.
- Control Flow Graph (CFG) Generation: angr supports CFG generation, enabling users to visualize program structure and identify interesting locations for further analysis or steer the exploration of the program..

While dynamic symbolic execution engines, such as KLEE, may operate at the instruction level, angr executes by default at the granularity of basic blocks. However, angr can be

configured to step with instruction-level precision, offering greater flexibility for specific use cases. These foundational features underpin the design and functionality of dAngr, allowing it to combine the power of symbolic execution with the interactivity of debugging.

B. dAngr's Design Objectives

The primary goal of dAngr is to create a user-friendly interactive debugger leveraging symbolic execution. The design objectives of dAngr are as follows:

1) *Interactive debugging*: The debugger combines the key capabilities of angr and GDB through a comprehensive command line interface (CLI). It offers an interactive experience, by providing instant responses to easily understandable commands, enabling users to manipulate and control execution seamlessly. Users are not required to redo time-consuming preparatory steps, such as loading the project or generating a control flow graph, ensuring an efficient debugging process.
2) *Dynamic feedback*: The tool offers dynamic feedback by allowing users to inspect the program state at any point during execution. Features such as breakpoints enable users to halt execution and visualize or inspect program elements, including variables, registers, the stack and memory. This realtime feedback facilitates a deeper understanding of the program behavior.
3) *Flexibility and Customizability*: The tool is designed to be versatile and extendable, making it suitable for various tasks such as reverse engineering, vulnerability detection, and software debugging. Users can expand its functionality using custom dAngr, Python, and Bash scripts, enabling tailored workflows to meet specific needs.
4) *Minimal Knowledge Requirement*: The debugger is user-friendly and designed for users with limited symbolic execution knowledge. It includes features such as example challenges, a comprehensive help function, and a straightforward interface for interacting with symbolic states.
5) *Ease of Experimentation*: It prioritizes an intuitive environment for experimentation with the debugger and the binary. This is particularly crucial in time-sensitive contexts such as capture-the-flag (CTF) competitions, where users need to interactively explore binaries and understand how the symbolic execution engine operates without facing unnecessary obstacles.

IV. Implementation and Discussion

We discuss the functionalities of dAngr, highlighting how it differs from regular debugging and how symbolic execution enhances the debugging process. We also discuss the steps taken to increase the flexibility and user-friendliness by introducing custom features and abstractions over the angr framework.

A. dAngr as a Regular Debugger

dAngr integrates the core features of symbolic execution with the main functionalities of a traditional debugger such as GDB. By leveraging symbolic execution and the angr framework, dAngr offers advantages over regular debugging, which we explore by comparing standard debugging features in a symbolic debugging context.

a) Breakpoints: In regular debuggers, breakpoints can be set on several attributes such as function names, line number and source file combinations, or on instruction addresses.

dAngr supports these methods via filter functions which check specific conditions during symbolic execution. Users can write their own filter functions or utilize pre-defined filter functions (i.e., filters on the address of an instruction, on a function name or on a line number). In addition, users can specify a breakpoint for halting execution when a particular string is printed to the standard output (stdout).

Conditional breakpoints in regular debuggers allow execution to halt based on a combination of a location and a condition. dAngr is more flexible and supports logical combinations of any type of filters, including custom defined filter functions.

Breakpoints in dAngr have an advantage over regular breakpoints, due to the manner symbolic execution handles branch statements (e.g., if-statements). In regular debugging, the debugger pauses the execution whenever a breakpoint is hit. Breakpoints that reside inside a branch consequently dependent on the branch-condition. For example, a breakpoint in the true-branch, does not get hit whenever the condition evaluates to false. In dAngr, breakpoints are independent of these conditions. The symbolic execution engine can explore both paths regardless of the condition. The engine simply adds constraints to the paths based on the branch taken. As a result, the breakpoint gets hit regardless of the required condition, which allows the assessor to evaluate the required values to reach past that breakpoint.

b) Step execution: Traditional debuggers allow stepping through the program at the instruction level, as well as stepping into, or over function calls. dAngr implements these functionalities with a key difference: by default it steps per basic block, reflecting angr's default behavior. However, single stepping is possible. Additionally, dAngr provides a run command, which continues program execution until either a breakpoint is hit or the program terminates.

c) Variable and stack inspection: Inspecting variables and memory is crucial for understanding the execution behavior. In angr, this involves accessing symbolic representations of registers and memory (including the stack and the heap) which are represented as bit-vectors. However, interpreting these potentially symbolic bit-vectors may be challenging for users unfamiliar with symbolic execution.

To simplify this, dAngr abstracts this complexity by providing intuitive commands for viewing the current state of memory, registers, and variables. This eliminates the need for manual interaction with angr in a Python shell or Python debugging session, improving user experience.

d) Visualization: Output visualization helps assessors follow program execution. dAngr captures and displays changes to file streams (e.g., stdout), allowing users to monitor output in real-time. Furthermore, angr only provides a full dump of the stream so far. dAngr only returns the output changes since last update (e.g., only the string printed at a certain step in the execution).

To further enhance visualization, dAngr provides a state overview inspired by tools like gdb-peda, gdb-gef and angr-cli. The interface highlights the most common registers, the stack and current basic block, with features like:

- Color-coding memory to allow an easy identification of the meaning of an address, mainly assigning colors to addresses located on the stack, heap or code segment.
- Recursive dereferencing of pointers, displaying the value they point to, which saves the assessor from having to manually dereference a pointer.
- Indicating symbolic values in light green or gray whether the value is initialized or not.

Displaying symbolic values, represented as constraint sets, can be challenging as these constraints may quickly become long and complex. dAngr addresses this by showing an evaluation of the symbolic value (a random concrete value that matches the constraints), alongside a truncated representation of the constraints.

Figure 1 shows the most important registers and their values at the top of the screen, followed by the stack and the current basic block.

B. dAngr as a Symbolic Debugger

Leveraging symbolic execution gives dAngr unique capabilities that go beyond traditional debuggers.

a) Selective execution: In traditional debugging, the user must define the instruction pointer to start execution at a particular location in the binary. This often results in segmentation faults if state initialization, including memory and registers, is incorrect. dAngr allows to start from any point in the binary, enabling assessors to focus on specific functions or regions. Moreover, when defining the entry state, function arguments can be passed as normal variables, with angr converting these variables in correct memory and register configurations, matching the calling convention derived by angr.

The ability of manually setting the initial state, is a feature that is invaluable for debugging large binaries or particularly interesting but hard to understand sections in the software.

b) Hooks and SimProcedures: Hooks in dAngr replace specific code regions with custom implementations, enabling users to: skip those regions (e.g., in the case of anti-debugging measures), replace unimplemented functions (e.g., interactions with hardware) or optimize symbolic execution by avoiding unnecessary complexity.

While regular debuggers can patch binaries, hooks offer a more flexible and temporary approach, which is particularly useful for symbolic debugging workflows.

c) Symbolic variables and constraints: Symbolic execution's ability to use symbolic variables instead of concrete ones allows for powerful analysis.

Symbolic variables accumulate constraints during execution, enabling the exploration of multiple paths. They facilitate targeted analysis by allowing users to add additional constraints (e.g., limiting a value's range). Symbolic variables are also useful to synthesize the required inputs to reach specific states, such as those containing vulnerabilities.

d) Platform independence: By lifting the binary to an intermediate representation in VEX IR, dAngr operates independently of the binary's architecture. This eliminates the need for platform-specific emulators like QEMU and simplifies debugging of binaries originating from embedded devices or uncommon architectures.

C. Customizability

dAngr interfaces with Bash and Python to increase flexibility.

- dAngr scripts: Users can write custom functions, that can be used like built-in dAngr commands. In combination with control structures like loops and if-checks, complex scripts can be written to automatically solve certain tasks, exploiting the capabilities provided by angr. This is achieved through a formally defined grammar for commands and control structures, supported by a lexer that validates inputs and provides immediate feedback to users about any violations or inconsistencies with that grammar.
- Python Interface: Users can call custom Python commands (prefixed with "!"), to perform specific computations not available through dAngr's commands. For example, applying regular expressions to variables in dAngr.
- Bash Integration: Users can execute bash commands (prefixed with "$"), for instance, to test discovered exploits.

These features allow users to automate complex workflows, extending dAngr's capabilities where needed.

D. Abstraction from angr

While angr offers extensive functionality, its complexity can overwhelm non-experts in the field of symbolic execution. dAngr abstracts common angr features into a simplified GDB-like command structure. Command suggestion and the help command provides an accessible list of available commands, their syntax and explanations.

At the time of writing, angr implements about 20 different path exploration techniques, with breadth-first-search as is default. They generally explore the state space in a way that deviates from regular program execution, making debugging less intuitive. Therefor, dAngr implements its own state exploration based on depth-first-search to better align with regular execution flows.

```
Legend: | Stack | Heap | Code | Instruction | Symbolic |
[------------------------ Registers -------------------------]
rax: 0x1f
rbx: 0x0
rcx: <0x7d> (<BV64 0x0 .. (if (if strlen_2_64 <= 0xff...)
rdx: <0x6b> (<BV64 0x0 .. (if (if strlen_2_64 <= 0xff...)
rsi: <0x34> (<BV64 0x0 .. (if (if strlen_2_64 <= 0xff...)
rdi: 0x403518
rbp: 0x0
rsp: 0x7fffffffeff30 --> <BV64 mem_7fffffffeff30_3_64{UNINITIALIZED}>
rip: 0x40083e
r8: <0x72> (<BV64 0x0 .. (if (if strlen_2_64 <= 0xff...)
r9: <BV64 reg_r9_4_64{UNINITIALIZED}>
r10: <BV64 reg_r10_5_64{UNINITIALIZED}>
r11: <BV64 reg_r11_6_64{UNINITIALIZED}>
r12: 0x0
r13: 0x0
r14: 0x0
r15: 0x0
[--------------------------- Stack ----------------------------]
0000| 0x7fffffffeff30 --> <BV64 mem_7fffffffeff30_3_64{UNINITIALIZED}> <- sp
0008| 0x7fffffffeff38 --> 0x400729
0016| 0x7fffffffeff40 --> <BV64 mem_7fffffffeff40_7_64{UNINITIALIZED}>
0024| 0x7fffffffeff48 --> 0x2
0032| 0x7fffffffeff50 --> 0x7fffffffeff90 --> <0x46544300656c6966> (<BV64 Reverse(arg1_0_536[535:512]) .. 0x...)
0040| 0x7fffffffeff58 --> 0x7fffffffeff95 --> <0x656854307b465443> (<BV64 Reverse(arg1_0_536[535:472])>)
0048| 0x7fffffffeff60 --> 0x0
0056| 0x7fffffffeff68 --> 0x0
[------------------------ Basic Block -------------------------]
0x40083e: xor edi, edi
0x400840: call 0x400570
```

Figure 1: State visualization at the end of the solution script for the google2016_unbreakable challenge [31]

V. Evaluation

To validate the effectiveness of `dAngr`, we tested it on a set of CTF challenges, 16 challenges from the `angr` documentation [32] and 17 from the Oregon `angr` CTF [33]. The Oregon CTF challenges are designed to highlight specific features of `angr`, whereas the challenges in the `angr` documentation are sourced from various CTFs and demonstrate the vast capabilities of `angr`, ranging from introductory to intermediate and complex challenges. Most of these examples demonstrate the effectiveness of symbolic execution in a reverse engineering scenario. However, there are examples that showcase the capabilities for vulnerability detection and even automated vulnerability exploitation.

These challenges illustrate that `dAngr` is capable to solve many challenges that are solvable with `angr`, lowering the initial hurdle that symbolic execution provides. This is achieved by employing the aforementioned higher level of abstraction, enabling new users to grasp the nuances of symbolic execution in an interactive GDB-like manner.

We highlight a selective execution example and two CTF-challenges that demonstrate the capabilities and the simplicity of using `dAngr` in both reverse engineering and exploitation scenarios. We use the solutions from the `angr` documentation to enable a direct comparison between the solutions. The challenges and respective `dAngr` solutions are available in the online repository.

A. AES Example

`dAngr` was created when encountering a (then suspected) vulnerability in real world firmware, during a reverse engineering study [34]. The firmware contained an insecure AES key generation vulnerability, which was used to encrypt privacy-critical communication. In particular, the key was derived from two variables which lacked sufficient randomness to be secure. While static tools like `Ghidra` provided valuable insights, decompilation was incomplete which left the code hard to interpret. Since this was a MIPS based binary, debugging the binary with traditional methods like GDB was complicated in the absence of a MIPS environment.

Using a combination of binary analysis and reverse engineering, the value of one of the two variables, and the structure of the other was discovered. Leveraging the selective execution in `dAngr`, a solution is easily found. For ethical reasons, the repository only includes a binary that resembles the binary from the case study. Listing 1 shows the solution for that binary.

The binary includes an "`obfuscated`" key generation mechanism of which the inputs are known (derived from the device configuration).

```
(dAngr)>  load "aes_example"
    Info:   Binary 'aes_example' loaded.
(dAngr)>  unconstrained_fill
    Info:   Fill with zeros.
(dAngr)>  set_function_prototype "char* obfuscate(
↪  char* , char* )"
    Info:   Function signature set for obfuscate
(dAngr)>  set_function_call
↪  "obfuscate('VerifySafeSecret','12345678910')"
    Info:   Function setup at 0x4012ff with
    ↪  memory:["Value 'VerifySafeSecret' stored at
    ↪  0x1000", "Value '12345678910' stored at
    ↪  0x2000"]
(dAngr)>  run
    Info:   Terminated.
(dAngr)>  to_str (get_return_value)
      3221225472
```

Listing 1: Solution of the AES real world example

After loading and initializing the binary, we set up our selective execution by defining a function prototype for the "`obfuscate`" function, which can be invoked by `dAngr`. Next, we harness that prototype to execute the function call, which is located somewhere in the binary. This invocation is conducted concretely, with the two variables that were identified earlier. That function call invokes the execution of the function, using the binary implementation itself, which in turn results in our key being returned.

While the example shows how a user can define a function prototype and call it with selected, concrete arguments, it is also possible to create an entry state, using the function address and the arguments, without defining the function prototype.

B. *Fairlight*

Next, we illustrate the capabilities of `dAngr` in a reverse engineering context, with *Fairlight*, a reverse engineering CTF-challenge that requires a valid key as a command-line argument. The binary employs various checks to validate the input. After these checks, the challenge provides feedback about the correctness of the key. Traditionally, this would be solved with a debugger by resolving these checks or writing Python code that leverages `angr` to find a state that reached the desired path.

We provide two listings. Listing 3 displays a solution in `angr` and Listing 2 shows a snippet of the terminal with the commands and their feedback using `dAngr`.

Firstly, we present the `dAngr` solution (Listing 2), followed with highlighted differences with the `angr` solution (Listing 3).

The first three lines initialize the symbolic execution engine, by loading the binary and creating a 14 byte symbolic bit-vector that represents the input key. Once that is done, we create an entry state, providing the name of the binary and the bit-vector as arguments. `dAngr` provides feedback after each command, i.e., the lines starting with "*Info:*". Next, we set a breakpoint on the basic block that jumps to the success message. This is done following the command "`breakpoint`", as seen in Listing 2.

```
(dAngr)>  load 'securityfest_fairlight'
    Info:   Binary 'securityfest_fairlight' loaded.
(dAngr)>  add_symbol argv1 0xe
    Info:   Symbol argv1 created.
    BV with size:112
(dAngr)>  set_entry_state args=["./fairlight",
↪  &sym.argv1]
    Info:   Execution will start at specified entry
    ↪  point.
(dAngr)>  breakpoint (by_address 0x4018f7)
    Info:   Address Filter: 0x4018f7 added to
    ↪  breakpoints.
(dAngr)>  exclude (by_address 0x4018f9)
    Info:   Address Filter: 0x4018f9 added to
    ↪  exclusions.
(dAngr)>  run
    Info:   Break: Address Filter: 0x4018f7.
(dAngr)>  evaluate &sym.argv1
      b'4ngrman4gem3nt'
```

Listing 2: Fairlight solution in `dAngr`

```
proj = angr.Project('securityfest_fairlight',
↪  load_options={"auto_load_libs": False})
argv1 = claripy.BVS("argv1", 0xe * 8)
initial_state =
↪  proj.factory.entry_state(args=["./fairlight",
↪  argv1])
sm = proj.factory.simulation_manager(initial_state)

sm.explore(find=0x4018f7, avoid=0x4018f9)

found = sm.found[0]
found.solver.eval(argv1, cast_to=bytes)
```

Listing 3: Fairlight solution with `angr` [35]

In this example, we break using an address filter to maintain similarity with the solution in `angr` and exclude the alternative path that leads to the failure message. Finally, symbolic execution is invoked until it reaches the breakpoint. Once the breakpoint is triggered, the execution halts and the user can inspect the current state. In this case, we evaluate the symbolic variable "*argv1*".

The `angr` solution is very similar to the `dAngr` solution, each being seven commands. However, there are some differences. The first difference is that the `angr` solution is a Python program, as opposed to the interactive command-line nature of the `dAngr` solution. Another difference lays in the creation of symbolic variables. `angr` expects the length in bits, whereas `dAngr` requires the length in bytes. The last difference is the inspection of the state. In `angr` the user has to review the state from the `found` stash and evaluates the symbolic variable, whereas in `dAngr` the user evaluates the symbolic variable related to the current state directly.

Listings 4 and 5 show an alternative method to solve the *Fairlight* challenge. Instead of setting a breakpoint that searches for an address, the breakpoint filters on a string printed to the standard output stream.

In `dAngr` this is achieved by setting a breakpoint based on a stream and providing the string that needs to be found. In `angr`, a user has to create a function that inspects the

```
...
(dAngr)> breakpoint (by_stream "ACCESS GRANTED")
    Info:   Standard Stream Filter: 1 added to
    ↪  breakpoints.
(dAngr)> exclude (by_stream "ACCESS DENIED")
    Info:   Standard Stream Filter: 1 added to
    ↪  exclusions.
(dAngr)> run
    >   OK - ACCESS GRANTED: CODE{4ngrman4gem3nt}
```

Listing 4: Fairlight alternative solution in dAngr

```
...
def find(state):
        return b"ACCESS GRANTED" in
        ↪  state.posix.dumps(1)
def exclude(state):
    return b"ACCESS DENIED" in state.posix.dumps(1)

sm.explore(find=find, avoid=exclude)
```

Listing 5: Fairlight alternative solution with angr

state and checks for the string in the standard output stream (stdout). The latter is less intuitive and may result in errors, as the dump is the full stream output so far for the followed path. In the former case, it is as easy as breaking based on an address. Finally, dAngr prints the stdout to the console, containing the key. In angr the user has to write additional code to inspect and print the binary's standard output.

C. Simple AEG

We showcase an example where dAngr can be leveraged in an exploitation scenario, using the AEG challenge from the 2016 insomnihack CTF [36]. The dAngr solution can be found in Listing 6. To solve this challenge, a *Automated Exploit Generation* script (AEG-script) is written. The first part of the script searches the required input to gain control over the program counter (PC), and the second part leverages that input, to inject a payload and point the PC to the injected shellcode.

In the previous example, we displayed the dAngr solution by providing the commands with their feedback, in this example we show a dAngr script that solves the challenge. dAngr can execute dAngr "scripts". This is done by providing a path to a text file or Markdown script, as an argument to the dAngr command. The engine parses the markdown file, and sends the commands in the markdown code segments into the CLI. Alternatively, a user can paste the code segments into the prompt and dAngr handles the code at once.

The solution to this challenge is more intricate than the previous example. The binary contains a buffer overflow vulnerability. The angr framework automatically finds the vulnerability by saving unconstrained states. By default angr discards unconstrained states. The user has to instruct the simulation manager to keep the unconstrained states. Once a state reaches a buffer overflow angr's simulation manager moves that state to the unconstrained stash. In dAngr, the "keep_unconstrained" command enables this feature. In

```
load 'demo_bin'
keep_unconstrained
set_entry_state
↪  add_options=['REVERSE_MEMORY_NAME_MAP',
↪  'TRACK_ACTION_HISTORY']

verbose_step False

...

def fully_symbolic(state):
    fully_sym = True
    for s in (chop_symbol &reg.pc):
        if not (is_symbolic s):
            fully_sym = False

    return fully_sym

exploitable_state_index = -1
while exploitable_state_index < 0:
    step
    unconst = len (list_states unconstrained)
    for i in range(unconst):
        select_state i unconstrained
        if fully_symbolic &state:
            exploitable_state_index = i
            break
        else:
            move_state_to_stash i 'unconstrained'
            ↪  'pruned'

shell_code =
↪  0x6a68682f2f2f73682f62696e89e331c96a0b5899cd80
control_buffer_addr = get_controlled_buffer_addr 22


if satisfiable &reg.pc == control_buffer_addr &&
↪  &mem[control_buffer_addr->22] == shell_code &&
↪  control_buffer_addr > 0:
    add_constraint &mem[control_buffer_addr->22] ==
    ↪  shell_code
    add_constraint &reg.pc == control_buffer_addr

    to_hex (dump_stdstream stdin)

    print "Required input to spawn a /bin/bash
    ↪  shell: "
    println (dump_stdstream stdin)
else:
    println "Could not place the shellcode in the
    ↪  buffer and set the PC to the buffer address"
```

Listing 6: AEG solution in dAngr

the entry state, extra options are set that allow us to get the address of our input.

Next, we step through the program and search for an unconstrained state in which the program counter becomes fully symbolic.

Finally, to build an exploit, two supplementary constraints are added to the input. One that constrains our program counter to point to the input buffer, and the second to set our input buffer to contain our shellcode. Once the constraints are added, we can display the exploit string.

There are minor differences between the solution using angr [37], and the one leveraging dAngr. One of the notable differences is the way the dAngr solution handles states.

dAngr uses a single current state on which symbols, memory, registers, etc. are manipulated. Therefor, the unconstrained state is first selected to become the current state. This example shows that dAngr is capable of solving a complex challenges, even with the limitation in the capabilities of dAngr compared to angr.

The solutions to the other challenges can be found in our online repository[2].

D. Limitations

dAngr still has a number of limitations.

a) The angr framework: Most library calls have SimProcedures in the angr framework. But, sometimes the user has to write their own SimProcedures. This can be perfectly achieved in dAngr but may have a negative impact on the performance due to overhead from the additional dAngr code.

b) The design and implementation of dAngr: Another limitation inherent to the building blocks of dAngr is path explosion. Path explosion always provides a challenge for symbolic execution. dAngr is no exception to this, in angr resource consumption can be reduced by writing custom exploration techniques leveraging smart stash management. These are currently not available in dAngr.

Likewise, the DFS approach of dAngr can be problematic. For instance, in the case of large or infinite loops, too many states may be kept in memory leading to a crash due to resource hogging or even exhaustion.

The custom stepping approach checks for updates in the output in every step. This creates additional overhead, resulting in a slower performance compared to angr.

c) The abstractions made: A problem resulting from the abstractions, is limitation on functionalities and capabilities of angr exposed in dAngr. angr has many different features and functionalities, and new ones are still being developed. Providing access to each and every one of them, is a challenging and difficult task. The interface with Python can be used to overcome some of these limitations. However, the current interface does not expose the simulation manager or the current angr project in use, preventing direct invocation of these angr features through the Python interface.

d) Symbolic execution: Some problems are inherent to symbolic execution. These are difficult problems to solve in dAngr. Currently, symbolic execution provides no ability to execute different threads and simulate race conditions. Those bugs cannot be found using our debugger. Symbolic execution also struggles with cryptographic functions, due to their randomness. This limits the use cases where our tool can be applied.

VI. Work in Progress

In this paper, we demonstrate that dAngr leverages symbolic execution to offer powerful debugging capabilities that are hard or even impossible to achieve with traditional debuggers. However, dAngr is still work in progress, and several enhancements could be made to further improve its usability and efficiency. The following are (potential) areas of development.

a) Exploration Techniques: The fixed DFS approach limits the tool in how the binary can be explored. While it aligns well with normal program execution, it limits the tool's speed and scalability. To mitigate resource exhaustion, such as excessive memory or time consumption, dAngr could support the selection of alternative exploration techniques or the definition of custom strategies:

- Targeted Exploration: Supporting targeted exploration techniques [38], which prioritize the shortest paths to a target location, could guide the execution more efficiently toward breakpoints or specific program states.
- State Pruning and Merging: Incorporating techniques for pruning uninteresting states [39], [40], [41], [7] or merging regions of the binary that do not impact the symbolic store [42] could improve performance. Unfortunately, these techniques often produce more complex constraints, which could increase the computational burden on the constraint solver.

b) IDE Integration: While the GDB-style command line interface is comprehensible, debugging support in an IDE would make the tool even more appealing for both beginners and advanced users.

c) Dissemination and Education: Several hands-on sessions are planned in which students of both bachelor and master level will be educated in leveraging the dAngr symbolic debugger on several binaries and CTFs. This will serve as an introduction to advanced program analysis, reverse engineering, and the world of CTFs. This allows us to improve upon the user-friendliness of the tool, by aggregating and implementing their feedback.

VII. Conclusion

With dAngr, we set out to make the powerful capabilities of symbolic execution accessible to non-experts in vulnerability detection, reverse engineering, and debugging, while maintaining the intuitiveness and functionality of GDB-like debuggers. We achieved these goals by abstracting selected angr functionalities, and integrating them into comprehensive, user-friendly commands. Through an interactive interface, we reduced the reliance of Python scripting for utilizing angr, eliminating tedious and time-consuming steps such as reloading the binary for every execution attempt.

To enhance user interaction, we introduced features like state visualization, which provides an intuitive overview of memory and register states during debugging. Additionally, we implemented a DFS execution strategy, which mirrors regular program execution, making symbolic execution more comprehensible.

For flexibility, dAngr incorporates interfaces with both bash and Python. This allows users to extend its functionality for specific use-cases.

The support for custom functions and control flow structures allows dAngr to be used in a scripting-like manner, automat-

[2]GitHub project: https://github.com/angr-debugging/dAngr/tree/main/examples

ing parts of the debugging and vulnerability detection process. Furthermore, a command grammar checks for syntax errors, providing immediate feedback to the users and improving ease of use.

The combination of these features results in a versatile tool capable of both debugging and automating (parts of) the vulnerability detection and exploitation pipeline. We validated `dAngr` by solving 35 CTF-challenges and exploiting one real world vulnerability, demonstrating its usability, effectiveness and reduced expertise requirements. These example challenges also serve as learning material for new users, showcasing the potential of symbolic execution in debugging contexts.

REFERENCES

[1] R. B. Dannenberg and G. W. Ernst, "Formal program verification using symbolic execution," *IEEE Transactions on Software Engineering*, no. 1, pp. 43–52, 1982.

[2] C. Cadar, D. Dunbar, and D. Engler, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. USA: USENIX Association, 2008, p. 209–224.

[3] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, 2019, pp. 531–548.

[4] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1186–1189.

[5] S. So, S. Hong, and H. Oh, "SmarTest: Effectively hunting vulnerable transaction sequences in smart contracts through language Model-Guided symbolic execution," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 1361–1378. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/so

[6] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," *CoRR*, vol. abs/1907.03890, 2019. [Online]. Available: http://arxiv.org/abs/1907.03890

[7] J. Vadayath *et al.*, "Arbiter: Bridging the static and dynamic divide in vulnerability discovery on binary programs," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 413–430. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/vadayath

[8] C. Yagemann, M. Pruett, S. P. Chung, K. Bittick, B. Saltaformaggio, and W. Lee, "ARCUS: Symbolic root cause analysis of exploits in production systems," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 1989–2006. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/yagemann

[9] C. Yagemann, S. P. Chung, B. Saltaformaggio, and W. Lee, "Automated Bug Hunting With Data-Driven Symbolic Root Cause Analysis," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. Virtual Event Republic of Korea: ACM, Nov. 2021, pp. 320–336. [Online]. Available: https://dl.acm.org/doi/10.1145/3460120.3485363

[10] L. A. Clarke, "A system to generate test data and symbolically execute programs," *IEEE Transactions on software engineering*, no. 3, pp. 215–222, 1976.

[11] V. Chipounov, V. Kuznetsov, and G. Candea, "The s2e platform: Design, implementation, and applications," *ACM Trans. Comput. Syst.*, vol. 30, no. 1, Feb. 2012. [Online]. Available: https://doi.org/10.1145/2110356.2110358

[12] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[13] B. Blanchet, "Modeling and verifying security protocols with the applied pi calculus and proverif," *Foundations and Trends® in Privacy and Security*, vol. 1, no. 1-2, pp. 1–135, 2016. [Online]. Available: http://dx.doi.org/10.1561/3300000004

[14] K. R. M. Leino, "Accessible Software Verification with Dafny," *IEEE Software*, vol. 34, no. 6, pp. 94–97, Nov. 2017, conference Name: IEEE Software. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8106874

[15] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens, "Verifast: A powerful, sound, predictable, fast verifier for c and java," in *NASA Formal Methods*, M. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 41–55.

[16] GNU-Project, "GDB: The GNU Project Debugger." [Online]. Available: https://www.sourceware.org/gdb/

[17] LLVM-Project, "LLDB." [Online]. Available: https://lldb.llvm.org/

[18] PEDA-Team, "Peda github repository," https://github.com/longld/peda, 2012.
[19] GEF-Team, "Gef github repository," https://github.com/hugsy/gef/, 2015.
[20] M. Hentschel, R. Bubel, and R. Hähnle, "The Symbolic Execution Debugger (SED): a platform for interactive symbolic execution, debugging, verification and more," *International Journal on Software Tools for Technology Transfer*, vol. 21, no. 5, pp. 485–513, Oct. 2019. [Online]. Available: https://doi.org/10.1007/s10009-018-0490-9
[21] N. Karmios, S.-E. Ayoun, and P. Gardner, "Symbolic Debugging with Gillian," in *Proceedings of the 1st ACM International Workshop on Future Debugging Techniques*. Seattle WA USA: ACM, Jul. 2023, pp. 1–2. [Online]. Available: https://dl.acm.org/doi/10.1145/3605155.3605861
[22] "Ghidra." [Online]. Available: https://ghidra-sre.org/
[23] "IDA Pro." [Online]. Available: https://hex-rays.com/ida-pro
[24] "Binary Ninja." [Online]. Available: https://binary.ninja/
[25] Nalen98, "Angryghidra github repository," https://github.com/Nalen98/AngryGhidra, 2020.
[26] A. Fioraldi, "Symbolic execution and debugging synchronization," 2020.
[27] F. Magin, "angr-cli github repository," https://github.com/fmagin/angr-cli, 2018.
[28] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976. [Online]. Available: https://dl.acm.org/doi/10.1145/360248.360252
[29] Angr-Team, "angr/cle," Jan. 2025, original-date: 2015-05-27T09:53:48Z. [Online]. Available: https://github.com/angr/cle
[30] "smparkes/valgrind-vex." [Online]. Available: https://github.com/smparkes/valgrind-vex/tree/master
[31] Google, "Google ctf," https://github.com/ctfs/write-ups-2016/tree/master/google-ctf-2016/reverse/unbreakable-enterprise-product-activation-150, 2016.
[32] Angr-Team, "angr/angr-examples: Example scripts using angr." [Online]. Available: https://github.com/angr/angr-examples
[33] J. Springer, "jakespringer/angr_ctf," Jan. 2025, original-date: 2017-07-05T20:50:19Z. [Online]. Available: https://github.com/jakespringer/angr_ctf
[34] V. Goeman, D. de Ruck, T. Cordemans, J. Lapon, and V. Naessens, "Reverse engineering the eufy ecosystem: A deep dive into security vulnerabilities and proprietary protocols," in *Proceedings of the WOOT Conference on Offensive Technologies (WOOT '24)*. WOOT Conference on Offensive Technologies, Aug 2024.
[35] chuckleberryfinn, "Fairlight solution," https://github.com/angr/angr-examples/blob/master/examples/securityfest_fairlight/solve.py, 2016.
[36] Insomnihack-Team, "Insomnihack/Insomnihack-2016," Aug. 2024, original-date: 2016-03-24T10:22:28Z. [Online]. Available: https://github.com/Insomnihack/Insomnihack-2016
[37] Angr-Team, "Angr-examples github repository," https://github.com/angr/angr-examples/blob/master/examples/insomnihack_aeg/solve.py, 2024.
[38] K.-K. Ma, "Directed Symbolic Execution," in *Static Analysis*. Springer Berlin Heidelberg, 2011, pp. 95–111, series Title: Lecture Notes in Computer Science.
[39] D. A. Ramos and D. Engler, "Under-Constrained symbolic execution: Correctness checking for real code," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 49–64. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/ramos
[40] L. Zhao, Y. Duan, H. Yin, and J. Xuan, "Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing," in *Proceedings 2019 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2019. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2019/02/ndss2019_04A-5_Zhao_paper.pdf
[41] C. Paduraru, M. Paduraru, and A. Stefanescu, "Optimizing decision making in concolic execution using reinforcement learning," in *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. Porto, Portugal: IEEE, Oct. 2020, pp. 52–61.
[42] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," *Acm Sigplan Notices*, vol. 47, no. 6, pp. 193–204, 2012.