3D Visualization of Symbolic Execution Traces

Jan Zielasko[1] Sören Tempel[2] Vladimir Herdt[1,2] Rolf Drechsler[1,2]
[1]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
[2]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
Jan.Zielasko@dfki.de, tempel@uni-bremen.de, vherdt@uni-bremen.de, drechsler@uni-bremen.de

***Abstract*—Symbolic execution is a powerful software testing technique for finding bugs in complex software. Unfortunately, following the symbolic execution and understanding its results is challenging. However, since symbolic execution is commonly not complete (i.e. due to path explosion) it is important to understand the limitations of the performed analysis. Otherwise, insufficiently tested code parts may not be identified and bugs remain unnoticed. Prior work attempts to address this problem via 2D visualizations which communicate properties of the performed analysis to the verification engineer. Since symbolic execution requires a visualization of several properties, such 2D visualizations often lack important information or end up being dense and difficult to understand.**

In order to overcome this limitation, we propose a novel 3D visualization of symbolic execution which allows visualizing additional properties via the third dimension. For this purpose, we have implemented a 3D visualization for the symbolic execution of RISC-V machine code and evaluate this implementation by comparing it to an existing 2D visualization. Our results demonstrate that the third dimension allows us to include additional information which is not captured by the existing 2D visualization. In order to stimulate further research on 3D visualization of symbolic execution, we have released our implementation as open source software.

I. Introduction

Understanding, analyzing and verifying programs is a challenging task and the difficulty only increases with the complexity of the code. An emerging software verification technique to verify the correctness of complex software is symbolic execution. Symbolic execution attempts to explore all reachable execution paths through a given program based on symbolic input variables. Unfortunately, the number of execution paths in a program grows exponentially with the number of branches in the code. This problem is referred to as *path explosion* in existing literature and one of the central challenges faced by existing symbolic execution engines [1, Section 1.2]. Due to the path explosion problem, enumerating all reachable paths is not feasible in the common case. For this reason, manual intervention or adjustments to the exploration strategies are often necessary to ensure the exploration of interesting paths through the program.

In order to tweak the symbolic exploration it is central to understand the results of a prior performed symbolic execution. Interpreting the results and understanding which parts of the program have not been sufficiently explored and why is challenging. For example, the symbolic execution engine may be focusing on exploring the same recursive function or loop over and over again, thus not exploring code outside of it. If the results are not understood correctly, the exploration is not adjusted accordingly and critical bugs in the software may remain unnoticed.

Prior work presents 2D visualization to aid the verification engineer in interpreting the results of a performed symbolic execution [2], [3], [4]. Designing a clear and easy to understand visualization for the execution of a program is challenging. The majority of prior work on 2D visualizations uses a graph-based visualization to represent the control flow of the tested program. However, such 2D graphs often exclude important information (e.g. statistics for each executed path) or become dense and difficult to comprehend. Symbolic execution extends the concrete execution of programs by one dimension, which contains the different paths through the same program. A possible solution to handle the greater amount of information that needs to be visualized, is to also extend the visualization by one dimension.

This paper proposes an approach to visualizing the symbolic execution of programs in three dimensions. To the best of our knowledge, this is the first publication proposing a 3D visualization for symbolic execution. Furthermore, we contribute an animation system to create a visualization that can replay the execution and update the created scene with each symbolic execution step, combining the advantages of static and dynamic visualizations. This allows us to visualize information about changes in the system state during execution as well as the number of times and the order individual sections of the program were explored in. We contribute a modular open source implementation of this system which is based on SymEx-VP [5], an existing symbolic execution engine for RISC-V machine code available at GitHub [6]. We have modified SymEx-VP to support the automatic generation of symbolic execution traces, which capture all relevant symbolic execution runtime information in a dedicated XML format. Furthermore, we contribute a visualization of generated traces as 3D scenes using Blender [7]. We evaluate our proposed 3D visualization by conducting a case-study comparison with SymNav [2] (a popular 2D graph-based visualization). Our results demonstrate that we can visualize additional important information about the performed symbolic execution via the third dimension in our visualization. As such the 3D visualization can support the verification engineer in getting a better understanding on the results of a performed symbolic execution and identify limitations or problems of applied symbolic exploration strategies. The visualization is also well suited for use in education. We provide our framework as open source to further stimulate education and research on 3D visualization of symbolic execution[1] [2].

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project Scale4Edge under contract no. 16ME0127 and within the project VerSys under contract no. 01IW19001 and within the project ECXL.

978-1-6654-7332-3/22/$31.00 ©2022 IEEE

[1]https://github.com/agra-uni-bremen/symex-trace-vp
[2]https://github.com/agra-uni-bremen/symex-3D

II. Related Work

There exist a large number of different program execution visualization tools, each suited for a different use case. Ghidra [8] is a popular reverse engineering tool that supports a broad range of architectures and use cases. It uses a 2D graph based visualization, which is used by most existing program execution visualization tools. They split the executed code into coherent control flow blocks and connect them with edges. CFGExplorer [9] is another example for a program analysis tool that uses this approach, but explores a novel graph layout.

Most tools can be differentiated by the abstraction level or programming language they are designed for and whether they create a static or a dynamic execution visualization. A dynamic visualization for symbolic execution, as presented by SED [4], is often used alongside a debugger and creates a visualization while the program is executed, which is updated with each step. In a static visualization approach, the visualization is independent from the execution environment, as all information is collected during execution and fed into the visualization component after the execution has finished. SymNav [2] is one of the most sophisticated 2D symbolic execution visualization tools and provides a visual analytics environment that aids the user in understanding and following different aspects of the symbolic execution. It is designed to be used by security analysts to fine-tune symbolic exploration of complex malware and software to understand the program and discover vulnerabilities. As such it targets programs only available in binary form. In contrast, SEViz [3] is a symbolic execution visualization tool, that operates on .NET source code.

Looking beyond the visualization of program execution there exist a range of similar domains such as hardware or software visualization. It has been shown that most classic visualization approaches in the domain of hardware/software co-visualization follow a similar 2D graph approach [10], which often runs into problems due to the complexity of the visualized data [11]. This resulted in a number of new approaches animating the 2D visualization or extending the visualization to three dimensions. In the context of software maintenance, CodeCity [12] and EvoStreets [13] use a 3D approach, to try to solve the task of visualizing vast amounts of information about software systems. Both represent an object-oriented software system as a virtual city with EvoStreets extending this visualization by additionally incorporating the evolution of the system into the visualization. The experiment conducted by Wettel [14] shows that such a 3D visualization can lead to a statistically significant increase in efficiency when interacting with such a visualization compared to the use of state of the practice tools that support reverse engineering and program comprehension. Another experiment compared the differences between viewing and interacting with the 3D EvoStreets visualization in conventional 2D environments and virtual reality with head-mounted displays [15]. The results show that users operate with approximately the same efficiency when interacting with the 3D visualization using orthographic projection with keyboard and mouse, 2.5D projection with keyboard and mouse and virtual reality headsets with hand-held controllers.

III. Background on Symbolic Execution

Symbolic execution is a formal program analysis technique that can explore many possible execution paths through the program at the same time without requiring concrete inputs. During a symbolic program execution, inputs are treated as symbolic. As such, an input value does not consist of a concrete value, but is instead internally represented using abstract symbolic expressions and constraints, which can be used to construct real input instances with the use of a constraint solver [1]. To run a program on symbolic values, the program is executed by an interpreter with symbolic execution capabilities.

When using dynamic symbolic execution to explore a program, the interpreter begins executing the program similar to a normal execution, with the difference, that some instructions might operate on symbolic values. Whenever the execution reaches an expression that contains a symbolic value which could cause a change of the program counter that depends on the symbolic value, for example a branch instruction, the symbolic expression solver checks, whether the branch is feasible or not. If the solver decides that both paths are feasible with the current symbolic value and the path constraints, it forks the current program state and the symbolic execution can follow both possible paths. For each branch, the symbolic value responsible for the branch is assigned a path constraint that limits the symbolic value in accordance to the expression that caused the branch. Afterwards, each branch can be executed independently from the other and can spawn any number of new forks itself. If the execution of a path terminates, either by simply reaching a valid end state or reaching an erroneous state for which it was previously specified that it should not be reached, the constraint solver can return an input instance that causes a concrete execution of the program to follow the same path to the same end state [16].

In practice, the number of possible paths that are discovered by the symbolic execution grows exponentially, leading to the so called path explosion problem. An additional factor that may limit the scalability symbolic execution is the complexity of symbolic constraints that have to be maintained and solved along each execution path [17]. For this reason, symbolic execution is usually applied to smaller units of code or only limited path lengths [18]. This makes symbolic execution well suited to exhaustively explore a huge number of different program paths in small programs or only a limited number of blocks or functions in larger programs.

IV. Methodology

In this section, we present our proposed methodology on 3D visualization of symbolic execution using the RISC-V *Instruction Set Architecture* (ISA) as a case-study.

A. Overview

Our approach is based on the visualization of symbolic execution traces. Fig. 1 shows an overview on our proposed 3D visualization framework, which consists of three major components, that handle:

1) Trace Generation (top left),
2) Trace Analysis and Optimization (middle right), and
3) Trace Visualization (bottom).

Each component was designed in a modular way to allow extension of the framework or adaption to other ISAs or visualization engines with minimal changes to the other components.

As the underlying symbolic execution engine we use SymEx-VP which is freely available on GitHub [6]. We have added a trace generation backend to SymEx-VP which generates trace files for symbolically executed programs. The central component of SymEx-VP is the *Instruction Set Simulator* (ISS) which decodes and simulates the individual instructions, while the symbolic execution engine and its internal SMT solver handle the symbolic exploration of the program. For this reason, the changes necessary to implement

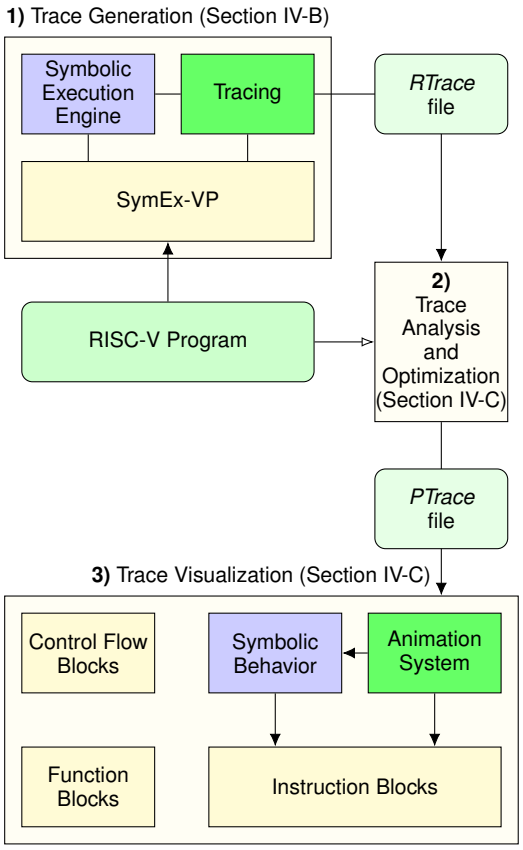

Fig. 1. Overview on the architecture of our porposed 3D visualization framework

tracing to collect detailed information about an executed program are mostly confined to these two components.

Executing a program with SymEx-VP generates a raw trace file (*RTrace*, top right of Fig. 1) which contains all necessary information to later reconstruct all important events. Saving the trace in an intermediate file allows the user to process and visualize the trace at any later point in time independently of the system running the backend. Additionally, the RTrace uses an easy to read and understand format based on XML [19], which makes it possible to gain information about the symbolic execution from the trace file itself. Fig. 2 shows an excerpt from an RTrace file containing one step XML element (lines 1,4), which contains the information about a single executed *ADD* instruction. In line 2, it lists the opcode name, the destination and source register name (*rd*, *rs1*, *rs2*) and whether they contain a concrete (*C*) or symbolic (*S*) value and lastly the symbolic behavior (*beh*), which is *none* in this example as all operands are concrete.

In the second step, the RTrace file is then fed into the trace analysis and optimization component, which consists of a set of Python scripts that analyze and optimize the data for visualization. As an optional feature, the source code and program binary can also be fed into the analysis step together with the RTrace file, which results in additional information about high level control flow, that enables annotations with source code lines. The processed data is then written into a processed trace file (*PTrace*, middle right of Fig. 1), which is also based on XML, but has a more complex structure than the RTrace file.

The third component is the trace visualization engine (Section IV-D). We implemented the visualization step as an addon for

```
1 <step pc="100d8" step="417">
2 <instruction opcode="ADD" rd="a5  (x15) C" rs1="a4
      (x14) C" rs2="a5  (x15) C" beh="none">
3 </instruction>
4 </step>
```

Fig. 2. Excerpt from the RTrace file generated from the execution of the example program used in Section V

Blender, which allows the import of PTrace files into the software, and automatically creates a 3D scene. The addon creates 3D objects for control flow blocks, function blocks, that mark function boundaries and single executed instructions. To color the created objects, we define a number of procedural materials. In addition to creating a 3D scene from the trace, we also use Blender's keyframe-based animation system to animate the instruction blocks. This makes it possible to visualize changes in the symbolic state of the system during execution and allows us to replay the complete execution of all discovered paths by the symbolic execution engine. As an add-on, this component is easy to install and use and was tested for compatibility with Blender versions 3.0 and 3.1.

More details on the three subsequent steps of our framework are provided in the following.

B. Trace Generation

To obtain a trace that allows following the concrete execution of a program, it suffices to save information about instructions and events that can not be determined from the program's binary. For the RISC-V standard instruction set only jumps and branches need to be traced, as the concrete execution of all other instructions is either unconditional or can be determined from the program's binary [20, p.5]. For symbolic execution, however, we must trace every executed instruction, as any of them might operate on symbolic values in memory or registers.

The traced data varies for each type of instruction, but always contains information about its effect on the symbolic state of the system. The symbolic behavior of an executed instruction can be classified into the following categories:

- A fully concrete operation has no symbolic behavior (category **none**).
- A concrete result may **destroy** a previously symbolic value.
- A symbolic result may **update** a previously already symbolic value.
- The operation can **create** a new symbolic value.
- A symbolic result may **overwrite** a symbolic value.
- Other **special** cases with regard to symbolic execution that concretize a symbolic value into a concrete value[3].

Since this approach does not necessarily rely on the program's binary to reconstruct the program's execution, it also works in cases where the actual executed code is not directly available in the compiled binary file since it traces each instruction that is actually decoded and executed by the SymEx-VP. If the source code of the executed binary is available it is used to identify the corresponding source code lines for each executed instruction. This information is then annotated on the individual instructions or coherent control flow blocks and included in the generated RTrace file. We designed the RTrace file to fulfill several requirements:

1) The trace file should be easy to generate and parse.
2) The contents should be human readable.

[3]For example, a multiplication with zero will always yield a zero result independent of the (symbolic) input value. Hence, a concrete result is produced in any case.

3) The trace size should be minimal.

XML strikes a good balance between these three requirements and was therefore used as the format for the RTrace file.

C. Trace Optimization

The RTrace file obtained by instrumenting the SymEx-VP contains all information necessary to reconstruct the symbolic program execution. In the next step, the RTrace is processed, analyzed and subsequently stored in the PTrace format, a format with similar requirements to the RTrace, but optimized for visualization and additionally containing all information acquired during the analysis step. This separation keeps the trace analysis independent from the execution environment or visualization framework. This means, the analysis implementation can easily be replaced by a different implementation without affecting the execution tracing step. Despite both steps being separated, any substantial modification to the information or structure in the PTrace will likely require changes in the visualization step as the visualization is based on the data provided.

The goal of the analysis step is to collect global information about each run as well as the complete symbolic exploration. This includes the range of accessed memory and PCs to allow a better placement and optimization of the visualization. This analysis step also removes all duplicate sections from the traced data, which drastically reduces the size of the trace. The duplicate sections are a result from the implementation of the symbolic exploration engine. SymEx-VP does not fork the execution at branch points and instead restarts the simulation for each new assignment of input variables. As such, multiple symbolic execution runs will have the same execution up until a certain branch point. The common steps are identical and not needed to understand or follow the execution. In contrary, they only clutter the scene and make it more difficult to understand the symbolic execution in the visualization. For this reason, all duplicate trace sections, that contain the exact same XML elements as already included in the parent run, are removed from the trace. To identify duplicate sections however, it is necessary to know the parent and child relation of all runs. This information is gained by constructing a binary tree using the run creation data from the RTrace.

If the binary contains DWARF [21] debug information and the source code is available, both files are used to extract high-level information about the execution. The collection of this information is optional and can later be used to add additional high-level visualization elements to the low-level visualization.

D. Trace Visualization

With all necessary trace data available, the next step is the visualization of the processed trace. The first challenge is to find a suitable visualization type, that can capture all aspects of the symbolic execution, and a framework, that can be used to create and display the visualization.

We chose a 3D tree approach as a base for our visualization with nodes representing the executed instructions and edges representing the control flow between them. An obvious control flow visualization approach would be to copy the design of proven 2D graph based visualizations and only extend it by one dimension. However, the graph based visualization approach does not accurately represent how the instructions are seen when viewing a disassembly of the binary. As this visualization aims to create a low-level instruction based visualization, we use a strictly linear visualization for the program counter with only one dimension instead, which results in a view similar to that of a binaries disassembly. Further, the graph also hides information about individual runs, because paths that reach the same program section after a point of divergence are joined again in the corresponding node. This compressed graph view is well suited for users who only want to understand the behavior of the program and are not interested in how the program was explored. As one of our goals is to aid the user in understanding the symbolic exploration, evaluating different exploration strategies and identifying problems, we chose to use an uncompressed tree based visualization. Fig. 3 shows how the different aspects are assigned to each axis. The X-dimension is used to visualize the program counter, with each grey cube representing a single executed instruction. The Y-dimension, is used to visualize the different paths through the program found through symbolic exploration. This makes it easy to compare individual paths and spot differences or diverging behavior quickly. In Fig. 3 the relation between the four different paths is visualized by the purple arrows connecting them. The Z-Dimension is used to visualize the current call-stack depth. This makes it possible to easily compare different passes through the same code in the same run. This is useful to understand how, for example, symbolic values are eliminated after calling the same function several times.

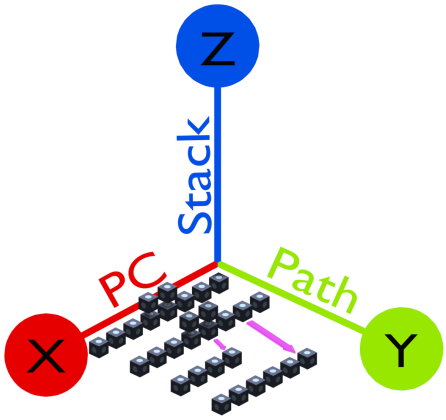

Fig. 3. Concept image showing how each of the different spacial dimensions is used to visualize information

The main aspects of the visualization backend, that have to be considered are the 3D scene creation process, shader/material capabilities, rendering and possible user interactions with the created scene. A detailed manual control over all the listed aspects after the scene was generated is essential to allow the user to customize the visualization or highlight specific details. For this reason, Blender was chosen as the visualization backend, as it offers all features that are necessary for automating complex 3D scene (e.g. [22], [23]) creation from a generated trace through its extensive Python API [24]. Blender makes it possible to create, place and view objects at arbitrary locations in 3D space. By using different shapes for the different instruction types (e.g. arithmetic instructions and branches) instead of annotating them with text, we create an abstract visualization of the program code, that makes it easy to identify interesting sections without cluttering the scene with unnecessary text. To visualize the different symbolic execution behaviors identified in Section IV-B, we utilize different colors and rotations for highlighting the individual instruction blocks. Each type of symbolic behavior (Section IV-B) is assigned a different color and rotation axis. We use green to highlight instructions

with the *create* behavior, red for *destroy*, blue for *update*, orange for *overwrite* and white for *special*, but the colors are freely customizable by the user. The color highlight is only applied to the inner section of all faces of the object's mesh, which leaves the remaining outer sections available for highlighting other aspects.

Blender also offers an advanced keyframe-based animation system which we leverage to incorporate dynamic aspects into our visualization scheme (to set the value of any property, e.g. the position, rotation, size or color, of any object at specific points in time and play an interpolated animation between them).

With the animation system, we can use time as an additional dimension to visualize the exploration of each path. We use this feature to animate the changes in the symbolic state of the system, which results in an animation that replays the complete symbolic execution of each path from start to finish. The animation begins at the start of the first discovered path, highlighting the first executed instruction as active with a bright green outline using the outer section mentioned above and a marker object placed above it. Similar to how a processor executes a binary, the marker transitions to the next executed instruction and marks it as active after a constant time interval. This continues until the execution reaches an instruction that leads to a new discovered path. From this point on the animation of the new path begins and both continue on until the execution is finished. If an active instruction has a symbolic behavior (Section IV-B), its color and rotation is updated accordingly. This makes it easy to spot sections in the visualization that contain instructions that operate on symbolic values and identify problematic functions that introduce multiple new symbolic values to the execution. In addition to the instruction objects, we also create objects representing control flow blocks that encapsulate and group coherent sections of each explored path.

V. Evaluation

In order to evaluate our proposed methodology we compare it to an existing 2D visualization using an example program as a case-study. For this purpose, we compare our novel 3D visualization with SymNav [2], an existing 2D visualization for symbolic execution (see Section II). The program used for this case-study is based on an extended symbolic execution example program from prior work [25, Listing 1]. It uses several different program constructs (like recursion and branches), thereby illustrating the visualization of different aspects of symbolic program execution.

In the following, we present the visualizations of our proposed approach (Section V-A) and SymNav (Section V-B) and then provide a discussion which compares both visualizations (Section V-C).

A. Proposed 3D Visualization

Our 3D visualization can be viewed from different configurable perspectives. In this section, we present three different views for the visualization of the aforementioned example program (Fig. 4, Fig. 5 and Fig. 6).

In Fig. 4 the 3D scene is shown from a top-down perspective. This view eliminates one of the dimensions, resulting in a view similar to classical 2D visualizations. In this regard, Fig. 4 provides an overview of the paths discovered through symbolic execution. The six vertical columns each represent a different execution path through the program (p1-p6). The individual paths are ordered from left to right in the order they were explored. Each explored path reaches a different section of the program. The different program sections are identified by the horizontally-aligned blocks. Information about the blocks is encoded using their color. Each block executed by a given path contains a number of smaller and mostly grey instruction blocks (i) that represent single executed instructions and underneath, a number of larger, colored control flow blocks (c). The source and target of jumps (orange) and branches (green) are connected by curves (b), that, depending on the direction, are placed either on the left or right side of the instruction blocks. More information about these blocks is provided on the left-hand side of Fig. 4. Most importantly, each block belongs to a function block (f) which is labeled with the function name and identifies the beginning and end of a function. Slightly to the right, the program counter (d) for each instruction block is displayed. Furthermore, the line number for the corresponding instruction in the source code is provided.

Fig. 5 shows the same visualization from a different perspective, which makes it possible to see the stack depth information contained in the Z-dimension. From this perspective the recursion can be spotted in the top right section (R), which contains a stack of multiple instruction blocks for each program counter. In contrast, all other non-recursive sections contain only a single or no instruction block for each program counter. Fig. 4 shows the same section in the second most block from the top (R).

One additional difference between the two figures is the program counter plane that is enabled for Fig. 5. The program counter plane is a background overlay which makes it easier to associate individual instruction blocks with their respective program counter. This is achieved by alternating between a light grey and dark grey background color for each program counter. Compared to Fig. 4, the view presented in Fig. 5 also presents a closer view of the individual instruction blocks. In the lower left section (s) of Fig. 5, instruction blocks are highlighted in different colors on their inner section and rotated to visualize their symbolic behavior. The colors correspond to the behavior described in Section IV-A. The green color marks the *create* behavior, while red corresponds to *destroy*.

A zoomed in view of the section marked in Fig. 5 (M) can be seen in Fig. 6. The section contains the starting point of p2, p3, p4 and p5 which are connected to their parent path that spawned the new symbolic run by a purple arrow. For each run, the control flow block objects are created for every control flow block that is reached by that path. In contrast, the instruction objects are only created from the point onwards from which the path diverges from its parent to avoid cluttering the scene with duplicate information.

From the three different figures, it is easy to understand which code sections are reached by which path and which code sections are explored multiple times by different symbolic runs. In this example, the first seven control flow blocks are reached by all of the six different paths, but diverge from this point on. The dark orange block (beneath i) for example is only reached by p2, which results in none of the other paths possessing a block for this section. In the branch condition that caused p2, the symbolic value is compared to a magic value for equality, which effectively concretizes the symbolic value to this magic number for this path. This is also the reason why no other paths are spawned from p2 as the concrete value can never fulfill both edges of a branch condition. An aspect, which is difficult to spot in the still image is the starting point of p1. This point lies at the beginning of the main function at the top of the teal colored control flow block row (c). In the animated visualization it is clearly highlighted by a marker and a bright green color. The instruction objects for all but the first run are

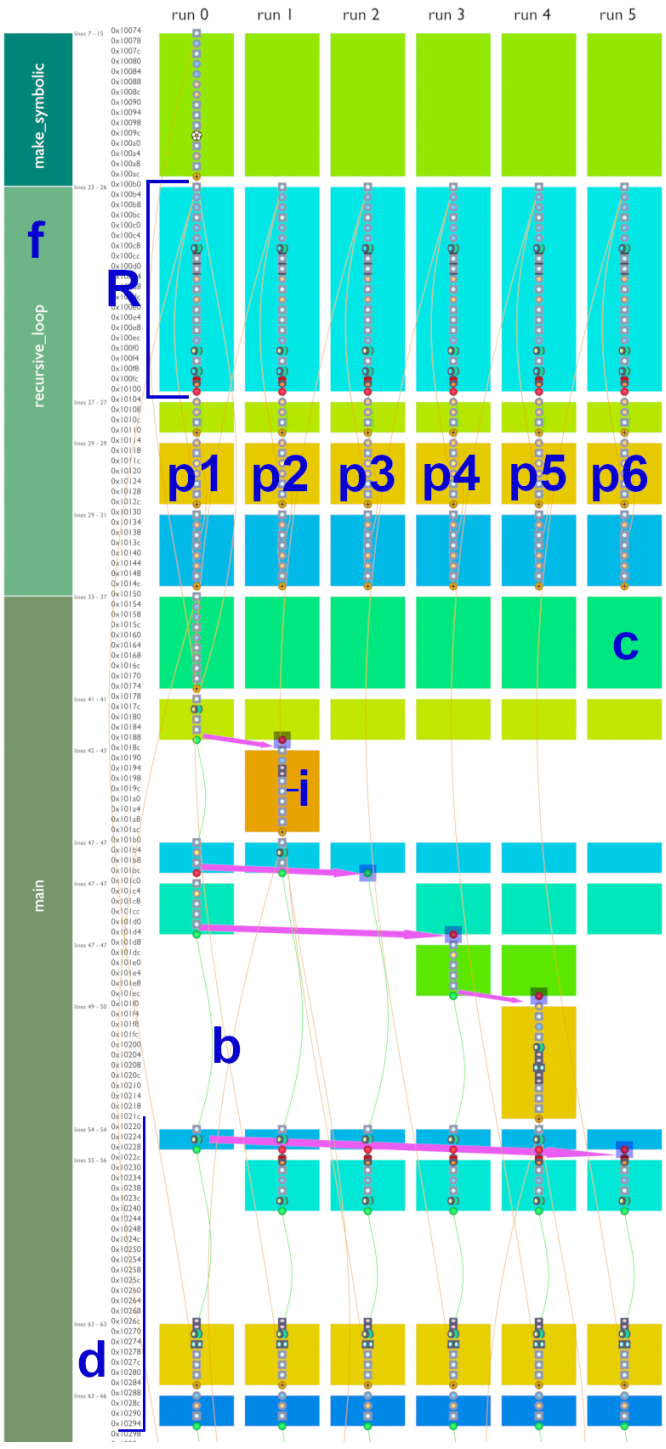

Fig. 4. Top down orthographic view of the 3D visualization of the example program

only created after the point they diverge from their parent. The `recursive_loop` section that lies before the PC of the point of divergence contains instruction objects for all paths, as execution jumps to this function after that point in time.

A special case can be seen in the empty section at the bottom, which represents a section that contains unreachable code. In the source code, this section contains a check testing the symbolic variable for $(var > 0) \wedge (var < 0)$.

The visualization of symbolic behavior can be seen in the red, green and orange colored sections on some of the instruction blocks. To increase the contrast between instructions with and without symbolic behavior, the light intensity of the object illuminating the scene can be reduced. This was done for the detailed view in Fig. 6, which makes it easier to spot sections of interest in larger scenes. The differences in rotation used for symbolic behavior can be seen in the bottom left section in Fig. 5. This results in a different visual shape compared to other instruction blocks when viewed in 3D without losing the instruction type information encoded in the base shape (the rotated cubes stand out, while still being clearly identifiable as cubes).

The state of the visualization in all figures is shown at a point at which all symbolic runs were fully executed. Using Blender's animation system, the user can view the state of the symbolic execution at any point in time or replay the complete symbolic execution.

B. SymNav 2D Visualization

Fig. 7 shows the 2D CFG generated by SymNav for the same example program compiled for the x86 architecture. In the SymNav tool, the graph is shown in the main graph view after the program was fully explored and with the main function selected as active.

In the full user interface of SymNav, the left side displays general information about the symbolic exploration, while the right side allows the user to steer the symbolic exploration or filter the results. For this comparison, we only focus on the 2D graph visualization of the symbolic execution shown in Fig. 7.

The graph is a visualization of all paths found by the symbolic execution through the main function merged into one connected graph. Each node contained in the graph represents a control flow block, which are mostly identical to the control flow blocks created in our 3D visualization. SymNav allows the user to expand or collapse these blocks to either only contain the block start address or the complete assembly code of the block. Edges in the graph that are explored multiple times or by multiple symbolic execution runs are highlighted by using a greater line thickness. The user can navigate to different functions by clicking on the corresponding nodes in the graph, but only a single function can be viewed at the same time. The second control flow block (m) contains the x86 equivalent of the dark orange control flow block described in Section V-A, in which the symbolic value is compared to a magic value. The visualization of how the new path splits at the first block is similar, but in the 2D visualization, the paths join again into the same node. This graph clearly shows how each code section can be reached, but loses the information about the individual run, as it is not possible to understand which blocks the path following the second node (m) can actually reach.

C. Discussion

In this section, we compare the two different visualizations presented above. We mainly focus on the advantages of a 3D visualization and the general new aspects of this specific 3D visualization approach in comparison to classical 2D symbolic execution visualizations. For this reason we only compare the graph view from SymNav with our animated 3D scene as any of the additional SymNav windows can be integrated into our visualization tool in a similar fashion without affecting the main 3D view in any way.

SymNav uses a 2D representation for the control flow, which merges the information from all symbolic runs in a single graph. Compared to our approach, in which we visualize each individual

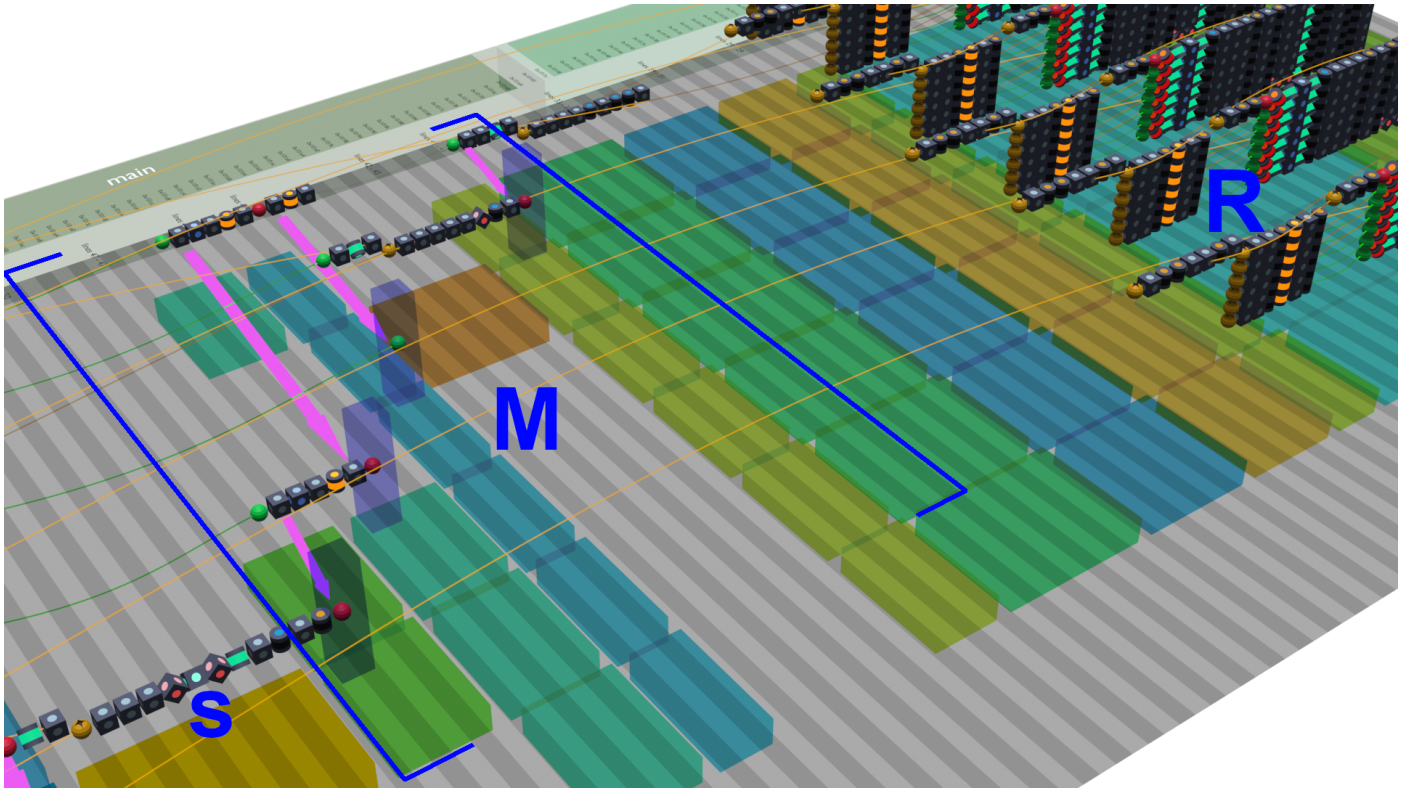

Fig. 5. 3D visualization of the symbolic execution of the example program

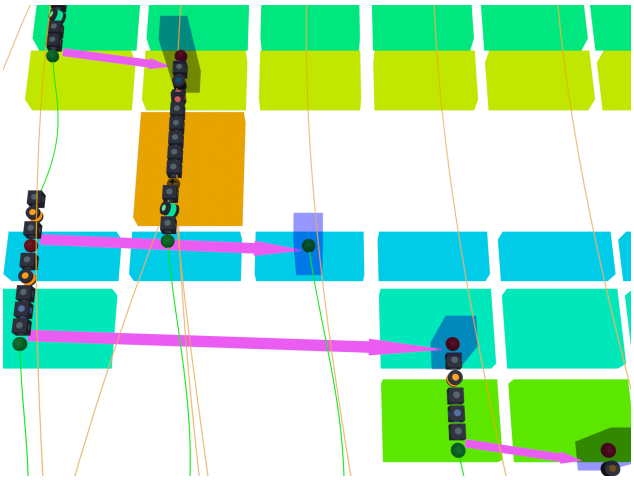

Fig. 6. Detailed view of the section marked M in Fig. 5

run, this compressed view is missing detailed information about how the symbolic execution engine explored the program. Using our visualization, it is possible to identify problems with the symbolic execution or evaluate different exploration strategies. By using a one dimensional, linear visualization for the program counter, we match the code representation as it is seen in a disassembly of the programs binary. This allows us to use the additional dimension for other aspects and also results in a number of features, for example, being able to spot unreachable program sections. On the other hand, a 2D graph representation compresses all the information, which makes it easier to understand the results of the complete symbolic execution and reduces the size of the visualization. For this reason, the choice of visualization for the program counter and control flow depends on the specific use case. SymNav does not visualize the symbolic behavior for the executed code in any way. While this information is not necessary to understand the results of the symbolic execution, it is essential to understand how those results arise. Using colors to encode this information is easy to understand in the visualization without cluttering the scene and could also be applied to SymNav's 2D visualization. SymNav does also not directly display information about problematic sections of the program. As shown in the example, visualizing the stack depth in the Z-dimension makes it trivial to spot time intensive recursion and also conveys general information about the control flow to the user. Fig. 4 effectively demonstrates that any 2D visualization can be expanded to a 3D visualization to allow visualization of additional information without sacrificing any aspect of the 2D visualization.

The animation component in our visualization is essential in making the propagation of symbolic values and the creation and start of runs easy to understand, as it guides the user through the 3D graph. In SymNav the user has to manually step through the 2D graph which still makes it possible to follow the execution, but much harder in comparison. A similar animation could be applied to SymNav's 2D graph although it is more difficult to apply to its compressed control flow graph.

Because of the additional dimensions, extending or adjusting the creation process of the animated 3D visualization for a specific use case is more difficult compared to that of a classical 2D graph.

VI. Conclusion

We have presented a novel approach for 3D visualization of symbolic execution traces and an animation system which visualizes changes in the symbolic state over time. Our proposed visualiza-

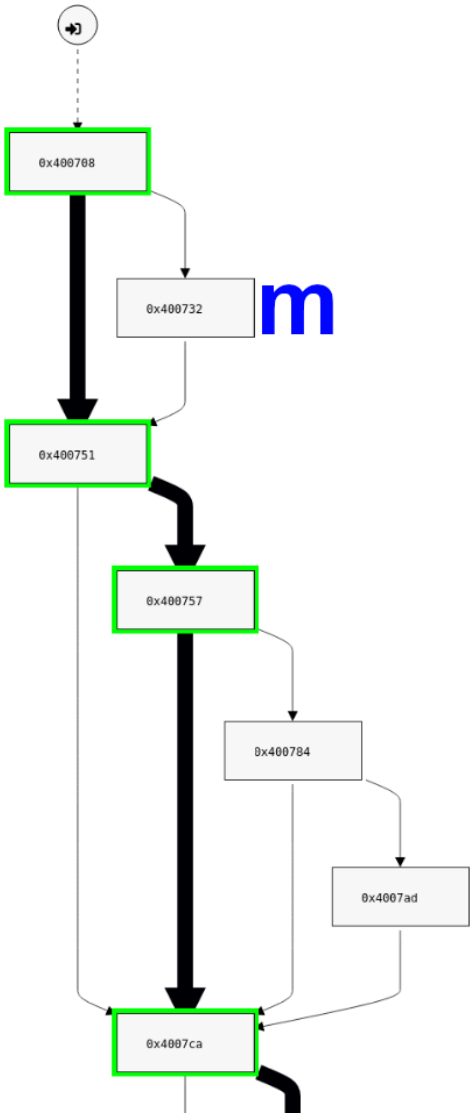

Fig. 7. Excerpt from the 2D graph created by symbolically executing the example program with SymNav

tion approach assists verification engineers in understanding the symbolic program exploration and its results, which help identifying code parts that have been insufficiently tested. Furthermore, we have presented a modular implementation of our proposed visualization based on Blender [7] and SymEx-VP [5]. In order to evaluate this implementation, we have performed a case-study comparing our 3D visualization approach with an existing 2D visualization from prior work. The results of this comparison indicate that more information about the performed symbolic exploration can be visualized using the third dimension without sacrificing clarity of the visualization. To the best of our knowledge, we have presented the first 3D visualization specifically for symbolic execution. To stimulate further research on this topic, we have released our 3D visualization framework as open source software. We plan to improve our proposed approach in future work by integrating additional information into our visualization (e.g. concretization) to further support the verification engineer in understanding crucial aspects of a performed symbolic execution.

REFERENCES

[1] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys*, vol. 51, no. 3, 2018.

[2] M. Angelini, G. Blasilli, L. Borzacchiello, E. Coppa, D. C. D'Elia, C. Demetrescu, S. Lenti, S. Nicchi, and G. Santucci, "Symnav: Visually assisting symbolic execution," in *16th IEEE Symposium on Visualization for Cyber Security*, 10 2019.

[3] D. Honfi, A. Voros, and Z. Micskei, "Seviz: A tool for visualizing symbolic execution," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, 2015, pp. 1–8.

[4] M. Hentschel, R. Bubel, and R. Hähnle, "Symbolic Execution Debugger (SED)," in *Runtime Verification*, B. Bonakdarpour and S. A. Smolka, Eds. Springer International Publishing, 2014, pp. 255–262.

[5] S. Tempel, V. Herdt, and R. Drechsler, "Symex-vp: An open source virtual prototype for os-agnostic concolic testing of iot firmware," in *Journal of Systems Architecture*, vol. 126, 2022, p. 102456.

[6] Group of Computer Architecture, University of Bremen, "symex-vp." [Online]. Available: https://github.com/agra-uni-bremen/riscv-vp

[7] Blender Foundation, "Blender," https://www.blender.org/, 2002.

[8] N. S. Agency, "Ghidra," https://ghidra-sre.org/, 2019.

[9] S. Devkota and K. Isaacs, "Cfgexplorer: Designing a visual control flow analytics system around basic program analysis operations," *Computer Graphics Forum*, vol. 37, pp. 453–464, 06 2018.

[10] R. Drechsler and J. Stoppe, "Hardware/software co-visualization on the electronic system level using systemc," in *2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID)*, 2016, pp. 44–49.

[11] R. Drechsler and M. Soeken, "Hardware-software co-visualization: Developing systems in the holodeck," in *2013 IEEE 16th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, 2013, pp. 1–4.

[12] R. Wettel and M. Lanza, "Visualizing software systems as cities," in *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, 2007, pp. 92–99.

[13] F. Steinbrückner, "Consistent software cities : supporting comprehension of evolving software systems," Ph.D. dissertation, BTU Cottbus - Senftenberg, 06 2013.

[14] R. Wettel, M. Lanza, and R. Robbes, "Software systems as cities: a controlled experiment," in *2011 33rd International Conference on Software Engineering (ICSE)*, 2011, pp. 551–560.

[15] M. Steinbeck, R. Koschke, and M. O. Rudel, "Comparing the evostreets visualization technique in two-and three-dimensional environments a controlled experiment," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, 2019, pp. 231–242.

[16] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[17] R. Majumdar and K. Sen, "Hybrid concolic testing," in *29th International Conference on Software Engineering (ICSE'07)*, 2007, pp. 416–426.

[18] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in *Proceedings of the 10th European Software Engineering Conference*, vol. 30, 09 2005, pp. 263–272.

[19] W3C, "Extensible Markup Language (XML) 1.0 (fifth edition) : W3C recommendation 26 november 2008," 2008. [Online]. Available: https://www.w3.org/TR/REC-xml

[20] I. R. Gajinder Panesar, *RISC-V Processor Trace*, 1st ed., https://riscv.org/specifications/, UltraSoC Technologies Ltd, 3 2020.

[21] Debugging Information Format Committee, "DWARF Debugging Information Format," UNIX International Waterview Corporate Center, Parsippany, NJ, USA, Tech. Rep. Version 4, 2010. [Online]. Available: http://www.dwarfstd.org/doc/DWARF4.pdf

[22] J. D. Durrant, "BlendMol: advanced macromolecular visualization in Blender," *Bioinformatics*, vol. 35, no. 13, pp. 2323–2325, 11 2018.

[23] A. et al., "Intuitive representation of surface properties of biomolecules using BioBlender," *Bioinformatics*, vol. 13, no. 16, 3 2012. [Online]. Available: https://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-13-S4-S16

[24] Blender Foundation, "Blender 2.93.6 Release Candidate Python API Documentation," https://docs.blender.org/api/current/, 2021.

[25] 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, pp. 138–157.