Bangr: Binary Ninja + angr

Kevan Baker, Daniel R. Tauritz, Samuel Mulder
Auburn University
kevan@auburn.edu, dtauritz@acm.org, szm0211@auburn.edu

***Abstract*—Binary analysis tools work better together. In the case of static analysis, symbolic execution tools are used to explore possible execution paths in a binary and decompilers are used to view binary code. In this paper, we discuss the bridging of these two types of tools, using state-of-the-art tools Binary Ninja and angr. We present a work-in-progress plugin for Binary Ninja named Bangr which integrates features of angr. With our plugin, we demonstrate how coupling angr and Binary Ninja enables answering questions that Binary Ninja cannot answer on its own. We further demonstrate the utility of having a graphical interface for angr, and conclude with a discussion on the Bangr plugin.**

I. Introduction

As binary analysis tools grow in complexity, our ability to perform introspection on their analysis becomes increasingly important. Disassembly and decompilation techniques have come far, but they can only answer so much. Symbolic execution [11] is a technique useful in static binary analysis for evaluating binary code with symbolic values; it can answer questions such as reachability, and possible values that a machine could have when executing a binary are easier to track. Symbolic execution is useful in answering questions about a binary program without concretely running the program. For example, symbolic execution can be used to solve capture-the-flag (CTF) challenges. In this paper we make the case that improving the interoperability of reverse engineering tools can provide such introspection on symbolic execution.

Understanding symbolic execution can be difficult for newcomers, as it is a formal method which behaves differently from concrete execution, with several major complex components. Symbolic states are created which follow the instructions in the binary and rely upon the use of an SMT solver, such as Z3 [8], to determine whether such states are feasible or not. The process of following instructions in the binary places constraints upon symbolic states, such as when assigning a value to a register, or a check on a register before branching on a guarded conditional branch. Some paths through binaries may have contradictory constraints making states unsatisfiable to SMT solvers. This process can be used to answer questions about a program without executing the program. For example, in a CTF challenge, there are multiple paths through the binary and typically only a single path to the flag. As symbolic execution follows all possible paths through the program, it is able to identify the path containing the flag. The state which followed the path to the flag can then send its constraints to the SMT solver to determine the value of the flag.

In the process of symbolic execution, path constraints that symbolic states collect are not made to be human readable [1], but instead are structured to be parsed by SMT solvers. Symbolic execution is also not typically performed directly on the assembly level, but on a lifted abstraction of the binary. Looking at disassembly, there are differences between what registers are expected at a specific address versus what the symbolic execution engine interacts with due to the intermediate representation (IR) used. As shown in Fig. 1, optimizations performed on the IR can result in differences at an instruction's address between disassembly and what the symbolic execution engine angr [15] actually executes.

To better visualize this process of dynamic symbolic execution in static binary analysis, we propose a coupling between the state-of-the-art Binary Ninja [16] and angr [15] tools via a plugin for Binary Ninja, allowing a user to interact with varying levels of IR presented by Binary Ninja, as well as to interact with the IR used by angr. We call this plugin "Bangr".[1] We integrate our plugin with Binary Ninja and angr because of their maturity and their ability to analyze binary software. angr provides facilities for its symbolic states to track historical actions on the state, including when constraints are added, registers are read from or written to, and more. Binary Ninja is a state-of-the-art binary analysis tool which shows different IRs of binaries and is extensible through plugins. The advantages of using Binary Ninja is that it already provides a convenient graphical user interface (GUI) which can be integrated with further plugins written in Python, the language used in angr.

Our goals for this plugin are to allow users to interact with angr and its symbolic states via their history and the IR the states execute over. Additionally, we want to allow for users to be able to reproduce their exploration in the GUI by keeping track of the user's actions and allowing for exporting to a standalone angr script. The novelty of this work is that to the best of our knowledge, there does not yet exist a graphical tool which allows users to view the history of what a state executed in symbolic execution, and query that state's history for constraints and temporary variables/register values over the course of symbolic execution via the context of the IR executed by the engine. Our tool provides the facility to select

[1] Bangr is available at [4]

Workshop on Binary Analysis Research (BAR) 2026
27 February 2026 , San Diego, CA, USA
ISBN 978-1-970672-08-4
https://dx.doi.org/10.14722/bar.2026.23015
www.ndss-symposium.org

```
IMark: 0x4012c0
Put rax = 0x402027
IMark: 0x4012c7
Put rdi = 0x402027
Put rip = 0x4012ca
IMark: 0x4012ca
t5 = I64 Get(rsp)
```

```
004012c0  lea   rax, [rel data_402027]
004012c7  mov   rdi, rax  {data_402027, "Incorrect!"}
004012ca  call  puts
004012cf  mov   eax, 0x1
004012d4  jmp   0x4012ea
```

Fig. 1. Screenshot of Bangr showing the difference between the Pyvex (left) and disassembly (right). At address *0x4012c7*, the Pyvex puts the constant directly into *rdi*, without using *rax* as seen in the disassembly. This is due to optimization done on the IR.

registers and temporary variables used in the Pyvex IR used by angr to query against the history of symbolic states and ask what values were in these registers or temporary variables if the state visited them.

In summary, the goals of Bangr are:

- Viewing the Pyvex IR used by angr
- Allowing for interaction with symbolic execution via angr
- Supporting the querying of symbolic states' history, including constraints, temporary variables, register values, and basic block history
- Writing out a standalone angr script replicating what was done in the plugin

II. Background

In the process of symbolic execution, symbolic states are created which represent the machine architecture including memory and storage. The states hold symbolic values which are influenced by the evaluation of the binary code. Initially, symbolic values are uninitialized, but are constrained as the program is evaluated. For example, when initializing a blank state, all registers in the machine architecture are uninitialized and unconstrained. When following the binary code, the states' registers and memory are initialized similar to how the program would initialize the real registers and memory in concrete execution. One notable limitation of symbolic execution of binaries is that command-line arguments cannot all be simulated, as there are potentially infinite variations on command-line arguments that could be provided, whether the binary supports or handles as many or not. Instead, users can specify the length in bytes of symbolic command-line arguments, and the number to initialize.

Symbolic states gather constraints on their memory as they move through the code, such as adding the constraint of a guard at a guarded branch condition. The number of symbolic states active at a time can grow exponentially, as at each guarded branch, a new state is generated to follow each path. In a loop, that means a split at each evaluation of the loop conditional. This problem, known as "state space explosion", often limits the practicality of symbolic execution [5]. Yet, many techniques exist which can be implemented alongside symbolic execution to mitigate this issue [3], [5], [7]. These techniques are incorporated in modern symbolic execution tools, such as angr.

angr [15] is a symbolic execution engine for binary analysis. It allows for the loading of binary code, and subsequent analysis of the binary using a modular framework. angr can create control flow graphs, provide disassembly and decompilation, and initiate and manage symbolic states within the binary. Symbolic states can be initialized with symbolic values, or users can specify values to be held in specific locations in the state, such as providing a specific value to a specific register and seeing what angr will return. angr stores values as "bitvectors", which represent the bits in memory as a vector. Bitvectors can have symbolic or concrete values depending on their context. When using angr's symbolic execution engine, states that are stepped through the binary accumulate constraints on themselves as they explore paths in the binary. These constraints are sent to an SMT solver, normally Z3 [8], which informs angr whether a state is satisfiable. angr allows for usage of different SMT solvers. The facility used by angr to manage states is called a "simulation manager", which manages states using "stashes" such as "active", "deadended", and "found". A simulation manager is created with one state set in the active stash. States are moved through the binary by "stepping", which moves a state through a single basic block. When a state cannot progress anymore, such as being at the end of main, it is moved into the "deadended" stash. When exploring for a specific objective, such as a specific address in the binary or a specific string in stdout, states that reach these objectives are moved into the "found" stash.

Interaction with angr is typically accomplished via Python scripts, as angr itself does not ship with a GUI and is used as a library. This has the advantage of extensibility within angr, as users can add their own exploration techniques as well as select different exploration techniques provided by angr. Users can extend other aspects of angr as well, with the ability to create analysis passes, breakpoints for inspection, and more.

Binary Ninja [16] is a proprietary static analysis framework used in reverse engineering of binary program, and is a state-of-the-art disassembler and decompiler in active development. This tool provides a GUI for the user to interact with and allows for easier viewing of the binary. Binary Ninja uses multiple levels of its own intermediate language to view the binary code, including views similar to Rust and C. Selections between graph views and linear views are available, allowing for visualization of control flow. Binary Ninja also includes its own debugger which integrates with GDB or other similar debuggers, but does not use symbolic execution in debugging. While Binary Ninja allows users to query for possible values in registers, and uses static analysis when resolving disassembly and control flow, Binary Ninja itself is not a symbolic execution platform.

Binary Ninja and angr are extensible platforms that can have plugins written in Python. This language allows for common ground between the two tools, facilitating easy interaction.

III. Design of Bangr

The plugin is designed with two user interface (UI) components in mind: a "sidebar" and a "view". The sidebar allows users to interact with angr by creating symbolic states, while the view shows the user the Pyvex IR used by angr during symbolic execution. The plugin's commands are registered within the plugin such that they are accessible in the "Plugins" section of the menubar of Binary Ninja, allowing users to create simulation states and interact with the simulation manager without directly interacting with the sidebar. When registering plugin commands, Binary Ninja supports context-sensitive commands with custom functions, allowing for conditionally showing commands only when appropriate. Thus the user can select to initialize a call state only when they have selected a valid address within the binary. Similarly, users can also only set an address as an objective to explore in symbolic execution when it is a valid address.

The following subsections describe the plugin options available via the menu, the sidebar, and the view respectively.

A. Plugin Menu

The plugin menu is created to be context-aware. Binary Ninja provides facilities to validate options which will disable buttons whose actions would be invalid under certain contexts. On a new project, users start by initializing a symbolic state which can then be interacted with. When a new symbolic state is initialized, a simulation manager is also initialized which is given the new state. Additional options include a "help" button and the option to save an equivalent angr script which allows for the creation of repeatable angr runs outside of Binary Ninja. With a separate script, users new to angr are able to learn more of the structure of a typical angr script, and more advanced users can use the output script as a starting point to analyze a binary. Any angr features which are not exposed via the plugin can be written into the exported script.

The bulk of the plugin menu is divided into options for symbolic states and the simulation manager. The following subsections will detail each.

1) Symbolic States: Users can initialize either an "entry" state, which initializes a symbolic state at the entry point of the binary, a "full init" state, which initializes a symbolic state at a special area within angr which simulates the dynamic loader before passing to the entry point, and a "call" state, which allows the user to set up a state at an arbitrary address in the binary. A call state is useful for under-constrained symbolic execution of a specific function. When the user selects a call state to initialize on a function address, the function prototype provided by Binary Ninja is given to angr to assist in creating the call site on the symbolic state. After initializing a state, users can toggle options angr provides for states, such as zero filling unconstrained memory or registers. The plugin also provides the ability to view the currently active options on the selected state. Users can additionally query a state's solver to evaluate the value of any bitvector which was initialized for command-line arguments to the state. The constraints placed upon a state during symbolic execution are used to determine the value of queried bitvectors.

2) Simulation Manager: Once the user has initialized a state, they are given options for interacting with the "simulation manager" used by angr which governs the symbolic states. The user can select techniques for the simulation manager to use via the plugin menu. The techniques available include veritesting [3], setting a maximum amount of memory that can be used, using depth-first search, and more. There are also options for exploration with the simulation manager. Users can select an address to explore to or avoid during exploration, or search for a state which has a user-specified string in its standard output or provide a string to avoid in exploration. Finally, there is also the option to simply "step" the simulation manager, which has all active states evaluate one basic block.

B. Sidebar

Functionality in the sidebar is managed with buttons which connect directly to the same functions that are available in the plugin menu. While there are instances where buttons are only available in the sidebar, such as displaying register values in decimal or hexadecimal, the buttons which control and affect analysis were created to tie directly to the plugin menu's commands in order to have a consistent user experience.

In the sidebar, users have the option to initialize a symbolic state at the entry point of the binary with a button. Other options for initializing states are available in the plugin menu. Once a state has been initialized, the user is able to interact with the simulation manager that is created for the state. The available options to step states through the program include manually stepping each state in the simulation manager by one basic block, querying the simulation manager to explore for a state that contains a string in its output, or querying the simulation manager to explore for a state that reaches a specific address. After the simulation manager performs one of these tasks, the view in the sidebar of each state is updated to reflect all the states currently managed by the simulation manager. The user can then select a state from the simulation manager, and select one of the other tabs in the sidebar to view more information about the state, such as the state's history of basic blocks visited, constraints, and register values. As seen in Fig. 2, register values can show the output of a function. Users can view register values as hexadecimal or decimal, and can see any bitvectors that have been initialized by the user when creating the state. Additionally, a tab is available in the sidebar which is context-sensitive and connected to the view. This tab displays values of registers or temporary variables that the user selects from the Pyvex view and shows the amount of times that register or temporary variable was visited, and what values it had. It also shows whether the access was a read or a write.

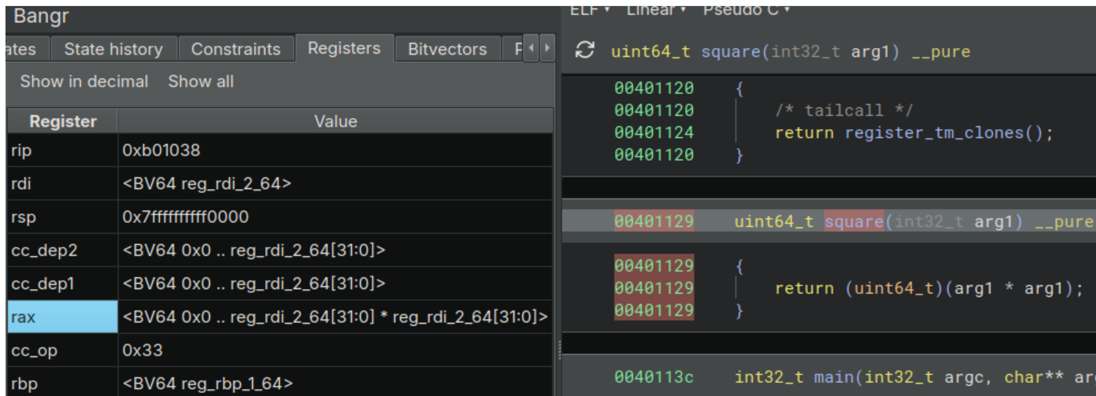

Fig. 2. The “Registers” view on the sidebar (left) and Binary Ninja’s Pseudo C on the right. Shown in register *rax* is the result of multiplying *rdi* by itself after stepping a call state initialized at the beginning of the *square* function

C. View

In the Pyvex view, users are able to see the Pyvex IR used in symbolic execution. This view is on a per-function basis, meaning that the user can only see one function at a time in the view. Users can navigate to different functions by using Binary Ninja’s standard facilities for navigation. This view uses Binary Ninja’s existing features to enable syntax-highlighting of instructions for human readability, and mimics Binary Ninja’s interactive address functionality, allowing users to navigate to different addresses in the binary by clicking on them within the view.

When the user has initialized and selected a symbolic state from the sidebar, they are able to select registers and temporary variables in the Pyvex IR which will be queried against the history of the currently selected symbolic state. If the symbolic state has been through the basic block containing the selection, the user will be presented with the values that the state had for each time the selection was visited and the sidebar’s tab to display this information will be automatically selected.

IV. Demonstration of Bangr

Given a simple binary, we will demonstrate solving a simple challenge for binary analysis tools with our plugin. The code for the binary can be seen in Listing 1. The main function of the program calls a function *check()* which accepts *argc*, and sets a function pointer *fp* to either function *f1()* or *f2()* based upon whether *argc* is odd or even.

Listing 1. Challenge Code

```
int f1() { return 1;}
int f2() { return 2;}
int f3() { return 3;}
int f4() { return 4;}
int (*fp)();
void set_fp(int v) {
  if (v % 2 == 0)
    fp = f1;
  else
    fp = f2;}
```

Name	Address	Section
_INIT_0	0x000401100	.text
f1	0x000401106	.text
f2	0x000401115	.text
f3	0x000401124	.text
f4	0x000401133	.text
set_fp	0x000401142	.text
main	0x000401178	.text

Fig. 3. Binary Ninja’s Symbols view, showing the addresses of the functions, including the two challenge solutions, *f1()* and *f2()*

```
int main(int argc, char** argv) {
  int x = 0;
  if (argc <= 0)
    return 1;
  set_fp(argc);
  x = fp() + f1() + f2() + f3() + f4();
  return x;}
```

This challenge is adopted from similar challenge code available at [14]. The goal of this challenge is to identify the possible values of the function pointer *fp* when called in the assignment to *x* after the call to *set_fp()* in *main()*. The challenge is made to be quickly answered by humans for ease of readability, yet challenging for tools to answer. The binary was compiled for x86_64 Linux with GCC given the flag *-no-pie*. Fig. 3 shows the addresses of the solutions to the challenge. In Fig. 4, Binary Ninja’s Medium Level Intermediate Language (MLIL) view is shown on main, with the line at *0x4011b1* highlighted showing the call to *fp*. Despite having backend analysis passes which include data-flow analysis, Binary Ninja is unable to determine the value of the call to *fp*. Fig. 5 shows the Low Level Intermediate Language (LLIL) view of the binary within Binary Ninja, with the cursor hovered over the call to *rax* which queries Binary Ninja for possible values for the register.

Using the Bangr plugin, this challenge can be solved by initializing an entry state with a symbolic value for *argc*. Once an entry state is initialized and the address of the call to *fp()* is selected in Binary Ninja, Bangr can be queried to explore to

```
ELF ▾  Linear ▾  Medium Level IL ▾
int32_t main(int32_t argc, char** argv, char** envp)
 5 @ 004011a3  rdi = rax_1
 6 @ 004011a5  set_fp(rdi)
 7 @ 004011aa  rax_2 = [&fp].q
 8 @ 004011b1  rax_3 = rax_2()
 9 @ 004011b3  rbx_1 = rax_3
10 @ 004011b5  f1()
11 @ 004011ba  rbx_2 = rbx_1 + 1
12 @ 004011bc  f2()
13 @ 004011c1  rbx_3 = rbx_2 + 2
14 @ 004011c3  f3()
15 @ 004011c8  rbx_4 = rbx_3 + 3
16 @ 004011ca  f4()
17 @ 004011cf  rax_4 = 4 + rbx_4
18 @ 004011d1  var_1c_1 = rax_4
19 @ 004011d4  rax = var_1c_1
20 @ 004011d4  goto 23 @ 0x4011dc
```

Fig. 4. Binary Ninja MLIL of the challenge code, with the call to *fp* at *0x4011b1*

```
ELF ▾  Linear ▾  Low Level IL ▾
int32_t main(int32_t argc, char** argv, char** envp)
 7 @ 00401197  if ([rbp - 0x24 {var_2c}].d s> 0) th

 8 @ 004011a0  eax = [rbp - 0x24 {var_2c}].d
 9 @ 004011a3  edi = eax
10 @ 004011a5  call(set_
11 @ 004011aa  rax = [&  UndeterminedValue
12 @ 004011b1  call(rax)
13 @ 004011b3  ebx = eax
14 @ 004011b5  call(f1)
15 @ 004011ba  ebx = ebx + eax
16 @ 004011bc  call(f2)
17 @ 004011c1  ebx = ebx + eax
18 @ 004011c3  call(f3)
19 @ 004011c8  ebx = ebx + eax
20 @ 004011ca  call(f4)
21 @ 004011cf  eax = eax + ebx
```

Fig. 5. Binary Ninja LLIL of the challenge code, with Binary Ninja displaying "UndeterminedValue" for the call to *rax* at *0x4011b1*

this address via either the plugin menu or sidebar. As there are only two possibilities for the call to *fp()*, setting the number of states to find at the selected address to two is sufficient. After exploration, Bangr will display two states in the "found" stash as seen in Fig. 8. By selecting one state at a time and selecting the "Registers" tab (see Fig. 2), the value of *rax* for each state can be viewed.

The Pyvex IR view can also be used to view the value of *rax*. After exploration, selecting a state from the "found" stash and opening up the state's history, one can see the list of basic blocks the state has stepped through. Upon selecting a basic block from the history, Binary Ninja will navigate to that address. When in the Pyvex view, selecting *rax* will show the history of *rax* in the sidebar at that instruction which accesses it, as shown in Fig. 7.

V. Related Work

Existing works seek to aid in understanding symbolic execution through angr. This section will describe relevant works which integrate interfaces with angr.

dAngr [9] shows a command line interface similar to the GNU debugger (GDB), and allows for users to symbolically execute a binary as they would through GDB. dAngr presents a

rax | 0x401115

Fig. 6. Value of *rax* in "found" state after execution

Bangr
sters | Bitvectors | Pyvex Temp Vars/Registers
register rax @ instruction 0x4011aa was accessed 1 time. It was a write with value <BV64 0x401106>

Fig. 7. The sidebar of Bangr, showing the value of *rax* for a state which executed an instruction at *0x4011aa*

novel abstraction layer over angr which allows users to interact with angr indirectly and dynamically. The provided GDB-like interface includes facilities for creating and inspecting states, setting "breakpoints", and various methods of interacting with the symbolic execution engine, providing a robust symbolic debugger. But, dAngr runs on the command line and does not allow the user to view the various IRs available via Binary Ninja, and is designed primarily as a debugger.

cozy [10] allows users to select two input binaries and analyze differences between them using a visualization. cozy uses angr in the backend, but is focused on showing differences in microcode patches between updates for binary code when source code is unavailable. Included with cozy is a wizard which allows for the setup and subsequent execution of a standalone script which uses angr. The tool gives users the ability to investigate paths through binaries to find states which align between the symbolic execution of two different binaries. While the user can see the disassembly and Pyvex used by angr, the focus is on differential analysis between binaries.

SENinja [6] is a plugin for Binary Ninja which allows users to perform symbolic execution across Binary Ninja's own LLIL and uses Z3 [8] as its background SMT solver. The paper lists angr as an inspiration, but does not choose to use angr, and instead creates its own methodology for symbolic execution. This has the advantage of choosing a different strategy for exploration during symbolic execution, where depth-first search is used. Additionally, the states are made to be copy-on-write, saving memory. However, the dedicated symbolic execution engine results in overhead and a need for a maintainer. Additionally, our plugin interfaces with angr which does support the depth-first search strategy. Our plugin further can export angr scripts which can be run standalone, outside of Binary Ninja.

angr-management [2] is a GUI for angr [15] which allows for interacting with angr. This GUI has views for disassembly, decompilation, and the IR used by angr, Pyvex. These can be viewed linearly and as graphs. The GUI supports instantiating a state and a simulation manager, but initializing an entry state does not allow for setting up bitvectors for command line arguments. While the GUI allows for saving an angr-db, a database of the project, it doesn't support writing out a standalone script. The additional IRs and features of Binary Ninja are missing as well.

TABLE I
COMPARISON OF TOOL CAPABILITIES

Tool	UI	State management	Pyvex View	Interactive state history	Output script
angr-management	GUI	Full	Yes	Semi	No
dAngr	TUI	GDB-like	No	Semi	No
SENinja	GUI	Full	No	No	No
Cozy	GUI	Semi	No	Yes	Yes
AngryGhidra	GUI	Semi	No	No	No
Bangr	GUI	Full	Yes	Yes	Yes

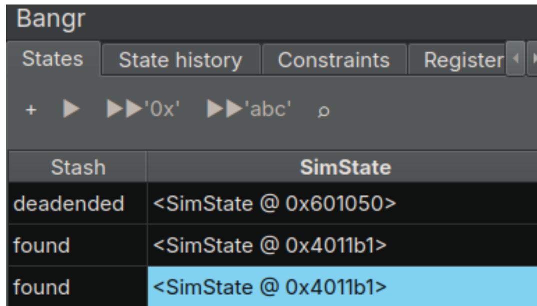

Fig. 8. The "States" view of Bangr, showing two found states and one deadended state

AngryGhidra [12] is a plugin for Ghidra [13] which allows users to interface with angr and Ghidra. The tool supports setting "find" and "avoid" goals for symbolic execution, and allows users to run the simulation manager with angr. But, the tool does not provide views into the Pyvex used by angr, nor does it interface with the temporary variables or register values in the history of the states in the simulation manager.

A summary of the capabilities of each tool is provided in Table I, detailing whether creation of entry states and call states are fully supported as well as inspection upon the states in the "state management" column. The "interactive state history" column indicates whether the history of the state is accessible and can be viewed. The ability of each tool to output a standalone script which can be executed outside the tool is also shown. While the tools all support some aspects, only Bangr includes full support for these aspects.

VI. Limitations and Future Work

Bangr is a work in progress, and as such there are some desirable features which are currently missing from the plugin. While the plugin offers customizability of angr's options, not all of the vast amount of options offered by angr are currently provided for the user. Future improvements would allow users to select options beyond just toggling state options and present more details on each option when the user is selecting them.

angr provides facilities for creating custom "SimProcedures" to call in place of functions. These SimProcedures are used in place of the Pyvex lifted from the binary, and can have the advantage of being easier to symbolically execute. For example, if there is a function that takes a string as an argument and returns a count of how many times a character occurs in the string, one could hook that function with a SimProcedure that returns a symbolic value and avoid potential state space explosion as angr tries to emulate all possible variable lengths of the symbolic string. Providing a facility for users to create and use their own SimProcedures in Bangr is a future step for the plugin.

Currently, a view into the stack and heap of memory is not implemented in Bangr. While this does limit viewing the complete state, querying against the Pyvex to see what was in temporary variables and registers can be used to examine the state. Future work includes creating a way to view the stack and heap of a state.

The constraints view currently is not made to be more human-readable than the string representation of bitvectors supported by angr. While there is currently support for querying a basic block against the history of a state to see the constraints that were put into the state up until that point, future work can explore different ways of representing constraints to the user [1].

Currently, there is no way to see the overall exploration of the binary by all states. An additional view which shows a heatmap of blocks based upon the relative number of states visiting each block would improve the ability to visualize exploration through the binary.

VII. Discussion and Conclusion

In this paper, we present Bangr which uses the strengths of Binary Ninja to facilitate introspection of analysis performed by angr. This plugin for Binary Ninja allows for users to view the IRs offered by Binary Ninja, as well as the IR used by angr during symbolic execution. Users can instantiate symbolic states and perform symbolic execution, and inspect the states during and after execution, querying the history of states. This plugin provides additional insights into symbolic execution performed by angr by presenting a graphical interface to angr's complex and thorough functionality. With this plugin, we argue that there are untapped benefits to the coupling of state-of-the-art tools which are typically used independently of each other, and the benefit of a GUI with angr is also a benefit to Binary Ninja, as we have shown with our demonstration where Binary Ninja is unable to resolve a value in *rax* that angr can resolve. We additionally argue that having a standalone executable script output from the GUI has value in both understanding the underlying angr functionality within the plugin, and extending it past the capabilities of the plugin.

REFERENCES

[1] T. Amon and T. Loffredo, "Creating human readable path constraints from symbolic execution," in *Proceedings of the Workshop on Binary Analysis Research (BAR)*, 2020. [Online]. Available: https://dx.doi.org/10.14722/bar.2020.23006

[2] angr Development Team, "angr-management: The official angr gui," https://github.com/angr/angr-management, 2025, accessed: 2025-12-31.

[3] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with veritesting," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 1083–1094. [Online]. Available: https://doi.org/10.1145/2568225.2568293

[4] K. Baker, "Bangr," https://github.com/Program-Understanding/bangr_plugin, Code for Bangr plugin.

[5] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, May 2018. [Online]. Available: https://doi.org/10.1145/3182657

[6] L. Borzacchiello, E. Coppa, and C. Demetrescu, "Seninja: A symbolic execution plugin for binary ninja," *SoftwareX*, vol. 20, p. 101219, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352711022001376

[7] F. Busse, M. Nowack, and C. Cadar, "Running symbolic execution forever," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 63–74. [Online]. Available: https://doi.org/10.1145/3395363.3397360

[8] L. De Moura and N. Bjørner, "Z3: an efficient smt solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, p. 337–340.

[9] D. de Ruck, J. Jacobs, J. Lapon, and V. Naessens, "dangr: Lifting software debugging to a symbolic level," in *Proceedings of the Workshop on Binary Analysis Research (BAR)*, 2025. [Online]. Available: https://dx.doi.org/10.14722/bar.2025.23014

[10] C. Helbling, G. Leach-Krouse, S. Lasser, and G. Sullivan, "cozy: Comparative symbolic execution for binary programs," in *Proceedings of the Workshop on Binary Analysis Research (BAR)*, 2025. [Online]. Available: http://dx.doi.org/10.14722/bar.2025.23004

[11] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, p. 385–394, Jul. 1976. [Online]. Available: https://doi.org/10.1145/360248.360252

[12] Nalen98, "Angryghidra: Use `angr` in ghidra," https://github.com/Nalen98/AngryGhidra, 2024.

[13] National Security Agency, "Ghidra: Software reverse engineering framework," https://github.com/NationalSecurityAgency/ghidra, National Security Agency, 2019, open-source software for reverse engineering (accessed 2026-01-07).

[14] Program-Understanding, "Suns benchmark dataset," https://github.com/Program-Understanding/suns-dataset, open-source repository of challenge binaries.

[15] 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.

[16] Vector 35 Inc, "Binary ninja: Reverse engineering and binary analysis platform," https://binary.ninja/, interactive decompiler, disassembler, and API for binary analysis.