cozy: Comparative Symbolic Execution for Binary Programs

Anonymous Authors

***Abstract*—This paper introduces cozy, a tool for analyzing and visualizing differences between two versions of a software binary. The primary use case for cozy is validating “micropatches”: small binary or assembly-level patches inserted into existing compiled binaries. To perform this task, cozy leverages the Python-based angr symbolic execution framework. Our tool analyzes the output of symbolic execution to find end states for the pre- and post-patched binaries that are *compatible* (reachable from the same input). The tool then compares compatible states for observable differences in registers, memory, and side effects. To aid in usability, cozy comes with a web-based visual interface for viewing comparison results. This interface provides a rich set of operations for pruning, filtering, and exploring different types of program data.**

I. Introduction

Much of today’s infrastructure is built on a foundation of legacy software; maintaining and securing this software is a critically important task. Patching legacy software must sometimes take place at the binary level due to loss of source code, build toolchain/environment “bit rot,” or limitations on the deployment system (for example, bandwidth-limited systems in contested environments). Under these conditions, software maintainers sometimes deploy software *micropatches*: minimal assembly-level changes that fix a bug or add functionality. Due to the low-level nature of binary patches, it can be difficult to reason about their effects on program behavior.

In theory, one could gain confidence that a patch has made all and only the desired changes by using a variant of *comparative symbolic execution*[1] (CSE) [1], [2]. In other words, one could run the pre- and post-patched programs on symbolic input in order to identify inputs that cause the programs to behave differently or violate a relative correctness specification. However, two challenges limit CSE’s suitability for validating real-world binary patches. First, existing CSE techniques target source code or idealized high-level languages. Second, CSE results can be difficult to interpret. CSE typically produces a formal description of the programs’ semantic differences; this description can be complex when the programs under analysis are large, when the patch produces a large change in program behavior, or both.

[1]Prior work sometimes refers to “differential symbolic execution” [1] and “relational symbolic execution” [2]. Throughout this work, we use “comparative symbolic execution” as a generic term for this family of techniques.

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.2025.23xxx
www.ndss-symposium.org

This work presents cozy, a tool that provides insight into the effects of binary patches by identifying and visualizing semantic differences between binary programs. The tool has two main components: (1) a *symbolic execution framework* for analyzing pairs of binaries, and (2) a *visualization engine* for displaying and exploring CSE results. The cozy approach to CSE involves running two programs on symbolic input in order to identify pairs of final machine states that are *compatible*: reachable by the same input. Differences in program behavior can then be characterized in terms of differences between compatible states. One attractive feature of this approach is that unlike some comparative analyses, cozy CSE does not require a correctness specification as input. This feature is useful when the analyst does not know in advance how the programs should differ. In such a case, the analyst can examine compatible state pairs manually or check various specifications against the pairs in a post hoc manner.

Because cozy targets a scenario in which source code is unavailable, it must be able to symbolically execute binary programs. To achieve this goal, the tool builds upon the angr [3] binary analysis platform.

In summary, this work makes the following contributions:

- We present the cozy comparative symbolic execution (CSE) framework, a novel adaptation of CSE to the binary domain.
- We present the cozy graphical interface for visualizing the results of CSE and for exploring the effects of binary patches on program behavior.

cozy is an open-source Python package. The tool can be installed via the Python Package Index (PyPI) [4]; its source code and documentation are available on GitHub [5].

II. Example

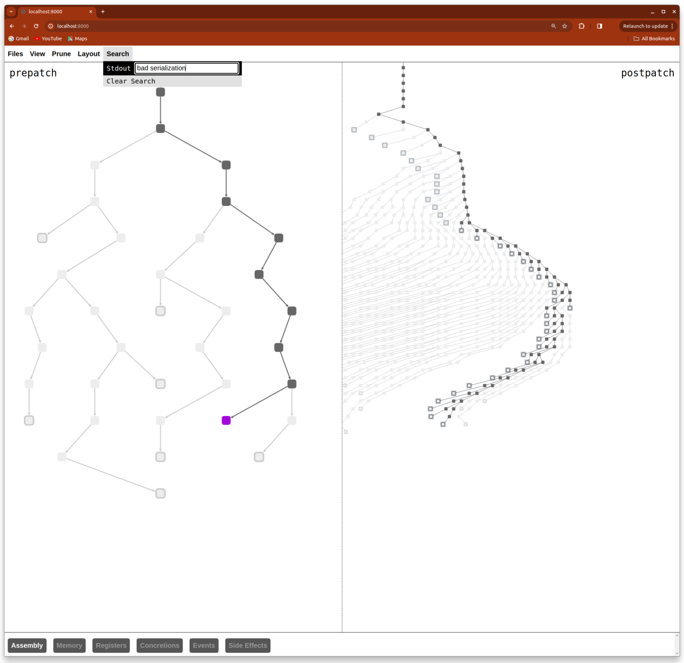
We introduce cozy with an example that involves two attempts at patching a vulnerable binary. cozy helps the user discover that while the first patch fixes the vulnerability, it also introduces unintended behavior. The tool then confirms that the second patch fixes the vulnerability without producing unintended behavior.

The example program, shown in Figure 1a,[2] is a simplified database server interface. The program’s update function

[2]While cozy operates directly on binaries, we present the program as pseudo-C source code for ease of understanding. An executable version of this example is available on the cozy GitHub repo [5].

```
1  void update(char *serialized) {
2    // begin patch
3    if (num_semicolons(serialized) > 2 ) {
4      puts("bad serialization!"); exit(1); }
5    // end patch
6    char *command = strtok(serialized, ";");
7    char *role = strtok(NULL, ";");
8    char *data = strtok(NULL, "");
9    if ((command is not "DELETE"|"STORE") ||
10       (role is not "root"|"guest" )) {
11     puts("bad input!"); exit(1); }
12   if ((command is "DELETE") && (role is "root")) {
13     delete(data);
14   } else if (command is "STORE") {
15     store(data);
16   } else {
17     puts("permission denied");
18   }
19   exit(0); }
20 int main(int argc, char **argv) {
21   char *command = argv[1], role = argv[2],
22     data = argv[3];
23   int len = strlen(command) + strlen(role) +
24     strlen(data) + 8;
25   char *serialized = malloc(len * sizeof(char));
26   sprintf(serialized, "%s;%s;%s", command,
27           role, data);
28   update(serialized); }
```

(a) Pseudo-C code for a simplified database front end. A user with root access is allowed to both store and delete data; a user with guest access is only allowed to store data. The original version of the program, which excludes lines 2–5, has a command injection vulnerability that enables a malformed command string to bypass the prohibition on guest deletions. Lines 2–5 are an overly restrictive patch that fixes the vulnerability but also rejects valid data payloads.



(b) `cozy` visual comparison of the pre- and post-patched versions of the Figure 1a program. Trees on the left and right represent possible execution paths for the pre- and post-patched programs, respectively. The purple node on the left represents a violation of the assertion that a guest cannot delete data—i.e., `cozy` finds an input to the pre-patched binary that breaks the "no guest deletions" rule. The right pane shows paths through the post-patched binary that are triggered by the same input. All such paths, as well as some additional paths that are *not* triggered by the input, have a square endpoint indicating that they print "bad serialization!" (the patch's error message). In other words, the patch rejects all vulnerability-triggering inputs, but it rejects some benign inputs as well.

Fig. 1: Program with patch (1a) and `cozy` visualization of CSE results for the pre- and post-patched program versions (1b).

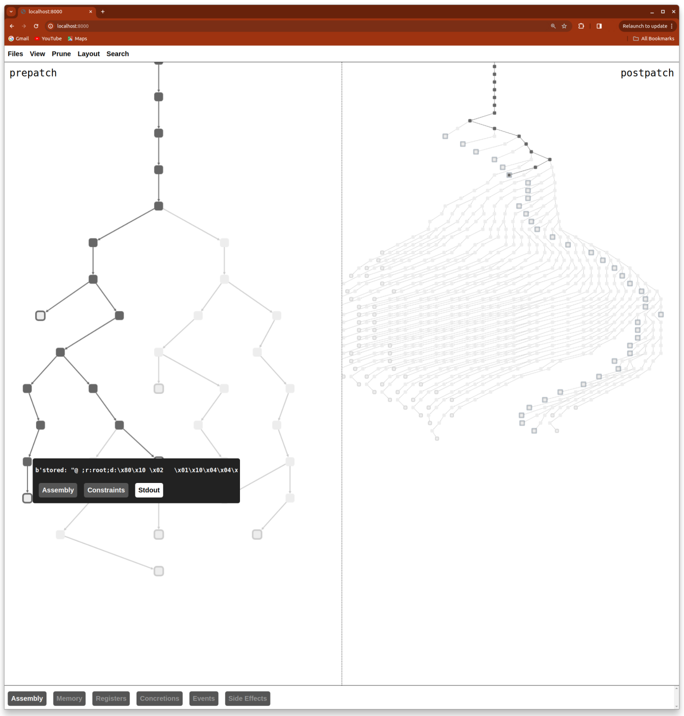
takes a serialized string containing arguments `command`, `role`, and `data` separated by semicolons. The `command` argument must be "STORE" or "DELETE", the `role` argument must be "root" or "guest", and `data` can be any string. `update` either (a) stores `data` to the database, (b) deletes `data` from the database, or (c) rejects the input as invalid, depending on the values of `command` and `role`. A "DELETE" command is only allowed when the role is "root"; the check on line 12 enforces this restriction. The `main` function serializes the command line arguments into a single semicolon-delimited string (line 17) and passes the string to the `update` function.

The original binary, which corresponds to the pseudocode in Figure 1a minus highlighted lines 2–5, has a command injection vulnerability: if the `role` argument is "guest" but the `command` argument is the string "DELETE;root", then the serialization-deserialization process incorrectly allows a guest to delete data.

Lines 3–4 in Figure 1a show an incorrect patch, which reports an error if the serialized string contains more than two semicolons. While this patch fixes the vulnerability, it is overly restrictive because semicolons should be allowed in the `data` payload argument, and the patch disallows such payloads.

To validate this change, the patch author runs `cozy` on the pre- and post-patched binaries. Doing so produces the visualization in Figure 1b. The trees in the left and right panes represent execution paths through the pre- and post-patched binaries, respectively. The operator has used a `cozy` feature to assert that the `delete` function should never be called when the `role` command line argument is "guest" (see Section III-E for details on assertions). The left (pre-patch) pane includes a purple node that indicates an assertion violation; in other words, `cozy` identifies a path through the pre-patched binary that corresponds to a command injection attack.

The user has clicked on the violation node in the left pane, which highlights all *compatible* paths in the right pane. Two paths are *compatible* when there is at least one concrete input that causes execution to proceed down both paths (see Section III-B for a detailed discussion of compatibility). Additionally, we have searched for paths in the right pane that print the string "bad serialization" (the error message that the patch produces); all such paths have a larger square endpoint.


prepatch
postpatch
Files View Prune Layout Search
Assembly
Constraints
Stdout
Assembly Memory Registers Concretions Events Side Effects

```
b'stored: "@ ;r:root;d:\x80\x10 \x02   \x01\x10\x04\x04\x
```

Fig. 2: To understand why the first patch attempt rejects valid input, the user finds a violation-free path through the pre-patched binary that is compatible with a "bad serialization" path through the post-patched binary.

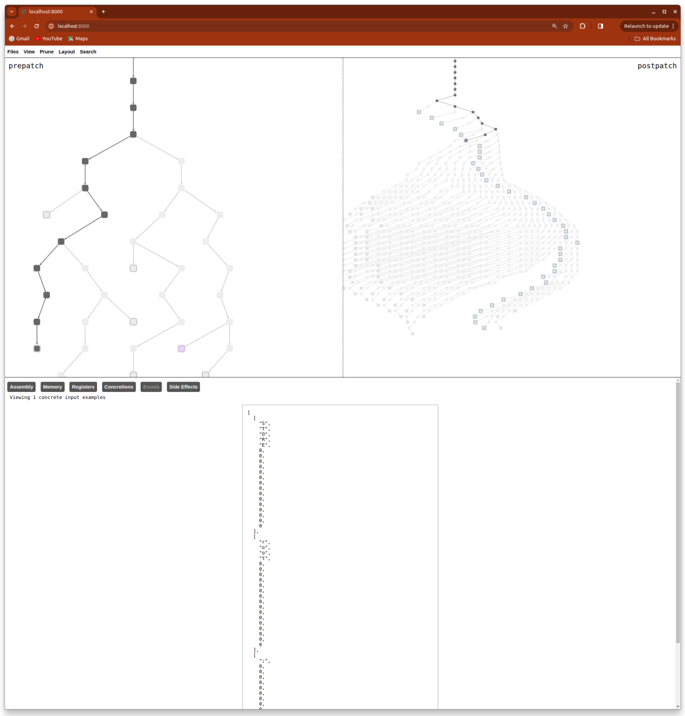
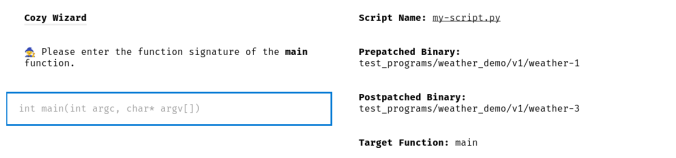

prepatch
postpatch
Files View Prune Layout Search
Assembly Memory Registers Concretions Events Side Effects
Viewing 1 concrete input examples

Fig. 3: `cozy` generates a concrete input that exercises the paths from Figure 2. The input `command`="STORE", `role`="ROOT", `data`=";" is valid (`data` is allowed to contain semicolons), but the patch rejects it.



Cozy Wizard

🧙 Please enter the function signature of the **main** function.

```
int main(int argc, char* argv[])
```

Script Name: my-script.py

Prepatched Binary:
test_programs/weather_demo/v1/weather-1

Postpatched Binary:
test_programs/weather_demo/v1/weather-3

Target Function: main

Fig. 4: Interactive wizard that generates an application-specific `cozy` harness based on user input. As shown here, one input to the wizard is the type signature of the function that will serve as the entry point for symbolic execution.

We can immediately see that all paths compatible with the assertion violation print "bad serialization." In other words, the patch rejects all inputs that would have triggered the vulnerability. However, the right pane also shows several free-floating squares, which are paths that print "bad serialization" but are incompatible with the path to the assertion violation. Why is the patch rejecting serialized input that would not have violated the assertion?

To investigate further, we click one of the "bad serialization" matches in the right (post-patch) pane, and then hover over a compatible endpoint in the pre-patch pane, as shown in Figure 2. This sequence of actions corresponds to finding a violation-free path through the pre-patched binary that the patch would intercept. The standard output of the pre-patch endpoint shows that this path involves a store operation. If we click that store endpoint in the left (pre-patch) pane, we can ask `cozy` for concrete input(s) that triggers the corresponding paths. As shown in Figure 3, `cozy` synthesizes an input that indeed has a semicolon in the `data` argument and that is flagged as an error by the incorrect patch.

Finally, we replace the bad patch with a check in the `main` function that the `command` argument contains no semicolons, and we confirm that "bad command" errors arising from the new patch correspond to either (a) the assertion condition, or (b) "bad input" conditions in the prepatched binary.

III. Comparative Analysis

`cozy` uses symbolic execution to compare binary programs. The tool runs both programs on the same symbolic input until there are no remaining states to explore. Once symbolic execution is complete, `cozy` pairs each terminal state in the pre-patched binary with each compatible terminal state in the post-patched binary. For each compatible pair, `cozy` computes a diff of the pair's register contents, memory contents, and IO side effects. Once this process is complete, the user may either view the results in textual form or explore them via a graphical interface. In this section, we describe the program analysis that `cozy` implements, and we outline the user's options for controlling and customizing that analysis.

A. Setup

cozy typically runs in a *harness*: a Python script that first configures various cozy parameters and then invokes the tool on the target binaries. To streamline the process of creating an application-specific harness, cozy provides an interactive wizard that asks the user a series of questions about how the tool should perform its analysis (see Figure 4 for an example). The wizard generates a harness based on the user's responses. A typical harness performs the following steps:

1) Create cozy projects for both binaries. A project is an object that acts as an interface between cozy and a binary to analyze.
2) Define any hooks that are needed to model hard-to-emulate functions. Hooks are common on embedded system targets where the callee function performs a side effect that cannot be modeled in the angr emulation environment.
3) Create all symbolic variables that will be used during execution. Symbolic variables can represent function input as well as sources of nondeterminism. A common example of nondeterminism is when the program requires user input from stdin or over the network. For example, one may simulate the getchar function with a hook that returns a symbolic value.
4) Define a run function that takes a project as input and symbolically executes its underlying binary, using the hooks defined in step #2 along with any user-defined preconditions and initial memory values.
5) Call the run function once on the pre-patched binary and once on the post-patched binary to produce run results containing lists of deadended states.
6) Compare the run results to determine which state pairs are compatible, and then check each compatible pair for differences in registers, memory, and side effects.
7) Launch a web browser window that shows a visualization of the comparison results.

B. Compatible States

Core to cozy's analysis and visualization is the notion of *compatible* states. We say that two terminal states s and s' are *compatible* if there exists at least one concrete input that causes execution to terminate in state s in the pre-patch execution, and in state s' in the post-patch execution. We collect all compatible state pairs into the *Compatible* relation. More formally:

Definition 1 (Compatibility)**.**

$$\mathit{Compatible} \triangleq \{(s, s') \mid \mathit{compatible}(s, s')\}$$

$$\mathit{compatible}(s, s') \triangleq \mathtt{is\_sat}(s.\mathtt{constraints} \land s'.\mathtt{constraints})$$

where the notation $s.\mathtt{constraints}$ refers to the path constraints of terminal state s.[3]

[3]Note that angr stores memory and register contents separately from path constraints; cozy is built on top of angr and inherits this design choice.

Unsat core optimization: A naïve way to compute the *Compatible* set is to check all n^2 pairs of terminal states for joint satisfiability. cozy implements a memoization-based optimization to enhance performance. When $s.\mathtt{constraints} \land s'.\mathtt{constraints}$ is unsatisfiable for a pair of states (s, s'), cozy computes the *unsat core* and caches it. The unsat core is the minimal set of clauses for which the conjunction is unsatisfiable. Later, when we want to know if a new pair (s, s') is compatible, we first check if any previously discovered unsat core is a subset of the joint constraints $s.\mathtt{constraints} \land s'.\mathtt{constraints}$. If this check succeeds, then the joint constraints are immediately unsatisfiable, and we can skip the expensive call to is_sat. Since most state pairs are incompatible in practice, the unsat core optimization drastically reduces the number of SMT solver queries.

"No orphans" property: A desirable property of our analysis is the "no orphans" property; that is, every terminal state that the analysis reaches in one program should be compatible with at least one terminal state in the other program. The "no orphans" property supports intuitive user interaction, such that whenever the user selects a path in one program, at least one corresponding path in the other program is highlighted. The "no orphans" property holds for the symbolic execution strategies cozy implements: complete execution (the variant described so far) and incomplete concolic execution (Section III-F). Each filter that the user can apply to the states through the cozy interface (Section IV) also preserves this property. We now give a proof for the complete execution case:

Lemma 1 (No Orphans)**.** *After complete symbolic execution of two programs P and P', a terminal state s_i from P always has at least one compatible terminal state from P'.*

Proof. After complete exploration, the path conditions of the terminal state induce a disjoint complete partition over the set of possible inputs. Suppose that the input partition for the terminal states from P is $\{X_0, X_1, ..., X_n\}$ and that the input partition for the terminal states from P' is $\{Y_0, Y_1, ..., Y_m\}$.

Because the inputs are the same for both programs, we have the following union condition:

$$\bigcup_{i=0}^{n} X_i = \bigcup_{j=0}^{m} Y_j$$

Assume that for state s_i with corresponding non-empty input set X_i, the intersection with all P' input sets Y_j is empty. This is equivalent to saying that s_i is an orphan.

However, this would mean that there exists at least one concrete input $x \in X_i$ that cannot be found in the P' input $\bigcup_{j=0}^{m} Y_j$. This contradicts the previous union condition which says that the input sets must be equal. Therefore, the state s_i is not an orphan state. □

C. IO Side Effects

In addition to comparing programs' final states, the cozy user might wish to compare programs in terms of the *side effects* that they produce. To enable this use case, cozy has a subsystem for modeling IO side effects. Common examples of IO side effects that we have modeled in example programs include writing to stdout/stderr, writing to the network, and writing over a serial connection.

Modeling IO side effects with `cozy` involves defining a *hook* for a side effect-producing function that simulates the function's behavior. When symbolic execution reaches a call to a hooked function, `cozy` runs the hook and stores the resulting side effect payload in the state. When a child state forks off from its parent, it obtains a copy of the parent's stored side effects. `cozy` keeps track of IO side effects over different channels (stdout, network, etc.). When the user examines compatible states in the UI, `cozy` visually aligns their side effects so that any differences are clear.

D. Observational Differences

Two compatible states with *observational differences*—i.e., differences in their register values, memory values, or side effects—indicate the existence of an input that causes the two programs to behave differently. Because such differences may be of interest to users, `cozy` checks each pair of compatible terminal states for equality of their registers, written memory, and IO side effects. Note that these state components may be a combination of concrete and symbolic values because `cozy` runs programs on symbolic input.

For a compatible pair (s, s'), register contents r in s and register contents r' in s' are observationally different when the following condition holds:

$$\texttt{is\_sat}(s.\texttt{constraints} \land s'.\texttt{constraints} \land r \neq r') \quad (1)$$

`cozy` constructs analogous conditions for memory writes and IO side effects, and it checks the conditions with an SMT solver. Because `cozy` targets a micropatch scenario in which differences between programs are small, the tool is able to use several optimizations that reduce the number of SMT queries it must perform. Registers and memory values are often entirely concrete or syntactically identical, so they can be compared for equality without a solver query.

`cozy` also employs a model-caching feature from `angr`'s built-in solver. When a formula like Condition 1 is satisfiable, `cozy` caches the model (concrete assignments that make the condition true). Later, when `cozy` needs to determine whether a different formula is satisfiable, the tool checks whether any of the cached models satisfy the formula before it attempts to construct a fresh model.

E. Directives

`cozy` supports several kinds of *directives*, which are special hooks that run when execution reaches a specified program address. A directive can be thought of as a breakpoint that runs a snippet of user-provided code—for example, to debug symbolic execution or provide extra information to the execution engine. `cozy` supports the following directives:

- **Breakpoint** pauses execution so that the program state can be inspected by user-provided Python code. When used in conjunction with a Python debugger, the simulation state can be inspected interactively.

```
def index_assertion(state: angr.SimState):
  index = state.regs.r2
  return (index.SGE(0) & index.SLT(BUFFER_SIZE))

session.add_directives(
  cozy.directive.Assert.from_fun_offset(
    project, "loop", 0x20,
    index_assertion, "index out of bounds"))
```

Listing 1: Example of creating an assertion for an array bounds check. At instruction loop+0x20, we assert that the index (stored in register r2) must be in range. Note: SGE means "signed greater or equal" and SLT means "signed less than."

- **Assume** attaches extra constraints to the program state when execution reaches a specified point.
- **Assert** by default operates like an assert in an ordinary programming language or testing environment. When `cozy` performs a complete symbolic exploration, an assert can be used to ensure that for all possible inputs, the provided condition cannot be falsified. A common example of an assertion states that an array index stored in a register is in bounds before it is used in an array operation. Listing 1 gives an example of such an assertion.
  When symbolic execution encounters an assertion directive, it splits the current state into two child states: one in which the assertion is triggered, and one in which it is not. The state with the triggered assertion is stashed, and it is not executed further.
- **Postcondition** is a special type of assert that executes after the simulated function returns.
- **Virtual print** produces an IO side effect on the virtual print channel, which is useful for debugging an execution trace within the program. This technique is analogous to a symbolic version of `printf` debugging.
- **Error** is a directive that is triggered whenever the program reaches a specified address. When execution reaches an Error directive, `cozy` stashes the current state; execution does not proceed further. This directive is useful for marking certain branches of the program as throwing an error.

F. Concolic Exploration

By default, `cozy` uses `angr`'s standard symbolic execution strategy of exploring non-terminal states in a breadth-first manner. As an alternative strategy, `cozy` provides a variant of *concolic execution* [6]. Concolic execution is desirable when the state space is large because it allows for incomplete exploration while still producing a set of final states that satisfy the "no orphans" property (Lemma 1).

In the typical concolic execution scenario as presented in the literature [7], the program first runs on a concrete input and generates an execution trace. Next, the program runs on symbolic input, which is forced to follow the concrete trace. After symbolic execution reaches a terminal state, a portion of the symbolic path condition is negated and a new concrete input is synthesized from this condition. This newly generated concrete input therefore exercises a different execution path.

cozy achieves results similar to those of ordinary concolic execution, but it uses a different exploration process. When child states are generated from a parent, cozy substitutes concrete inputs into their constraints; the tool then defers (halts exploration of) all children with constraints that evaluate to false. This approach obviates the need for separate concrete execution of the program; it fuses concrete and symbolic execution into a single process. This fusion decreases the engineering effort required to implement the concolic approach and integrate it with the existing complete exploration code.

Once symbolic execution reaches a terminal state, cozy uses one or both of the following heuristics to decide how to continue exploration:

1) **Termination Heuristic**: A termination heuristic determines whether cozy should halt concolic execution. The default termination heuristic says that concolic execution should continue until the exploration of state space is complete. cozy also enables the user to choose termination heuristics based on cyclomatic complexity and basic block code coverage metrics; these heuristics may lead to incomplete exploration. In addition, the user can define custom termination metrics.
2) **Candidate Heuristic** If the termination heuristic says that exploration should continue, cozy needs to decide which deferred state to explore next. Choosing a deferred state is equivalent to negating part of the path condition of a previous exploration.
The "trivial" candidate heuristic simply chooses an arbitrary deferred state from the list of options. cozy also provides a more complex n-gram branch coverage heuristic [8] that attempts to choose the deferred state with the most unique basic block address history.

Once the candidate heuristic chooses the next state to explore, cozy generates a new concrete input from that state's path constraints. cozy then feeds this concrete input into both programs under comparison by activating the appropriate deferred states (those with path conditions that are now satisfied). The program used to generate the concrete input alternates between the pre- and post-patch binaries to ensure that both versions of the function are being explored.

By feeding the same concrete input to both programs, cozy ensures that no orphaned states will be generated. This invariant is important because it ensures that any terminal state a user selects in the cozy UI is compatible with at least one state in the other program.

IV. Visualization

The cozy Graphical User Interface (GUI) is a simple web application. As shown in Figure 5, the GUI presents the user with three main interfaces: (1) a menubar; (2) a pair of panels displaying two symbolic execution trees; and (3) a "diff panel," which presents detailed comparative information once the user selects a pair of branches from the execution trees. In the remainder of this section, we describe the GUI's presentation of symbolic execution trees and its diff panel in more detail.

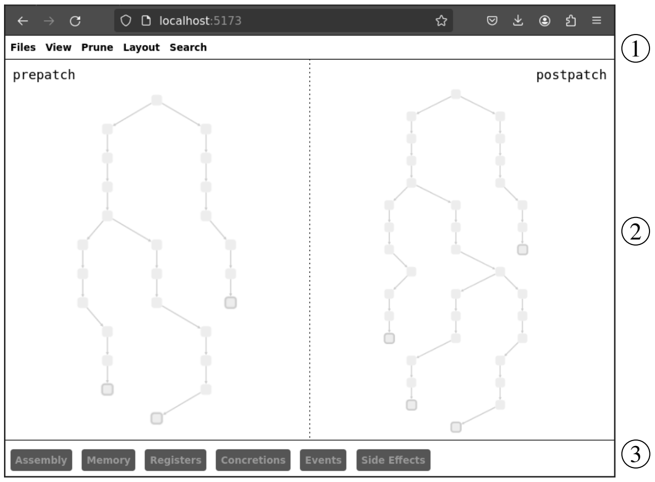


Fig. 5: The cozy GUI consists of (1) a menubar, (2) two panels displaying symbolic execution trees, and (3) a diff panel that enables the user to compare program branches across various dimensions.

A. Symbolic Execution Trees

A symbolic execution tree depicts the results of symbolically executing a given program with angr. The root is the initial program state, an internal node is the program state after execution of a basic block, and an edge is a symbolic execution step.

When analyzing symbolic execution results, the user needs a way to cut out extraneous noise. Typically, only a small subset of all of the possible paths through a program are of genuine interest. The cozy GUI offers three main mechanisms for focusing on the relevant parts of symbolic execution results: *highlighting*, *pruning*, and *compression*.

Several types of program states that are likely to be significant are automatically highlighted in the GUI. These include states that raised errors during execution, states at which a syscall or SimProc (modeled function) call occurred, states at which the program exceeded user-specified boundaries on loop iteration, and states at which a user-provided assert or postcondition failed. Different colors indicate different categories of potentially significant states. The color palate, and toggles to hide or show each type of state, are available under the "View" menu in the menubar.

Besides calling attention to relevant results, it can be helpful to filter out irrelevant results. cozy's main mechanism for filtering out irrelevancies is *pruning*. Pruning works as follows: cozy prunes (hides) each branch unless it is "interestingly related" to a compatible branch in the facing tree, where the user specifies (via the GUI) which relationships are interesting. For example, the user can indicate that two branches are interestingly related when their terminal states have different memory contents; pruning will then leave only the branches that differ from at least one compatible branch of the facing tree in terms of their final memory contents. The relations that the GUI checks are symmetric, so if a branch b survives pruning because it is partnered with a compatible branch b', then b' will survive as well. Therefore, pruning will never result in an orphaned branch.

Several pruning actions are available under the “Prune” menu. In addition to memory differences, `cozy` can check for differences in register contents as well as stdout and stderr output. The tool can also check whether at least one of two compatible branches ends with an error state, and whether at least one branch produces stdout that does not match a user-provided regular expression. In addition, the user can apply multiple pruning relations simultaneously, which results in pruning with the *conjunction* of the selected relations. We found that while it is possible in principle to apply arbitrary Boolean combinations of prunings, this approach makes for an excessively complicated UI. Hence, we restrict ourselves to the simpler case of pure conjunction.

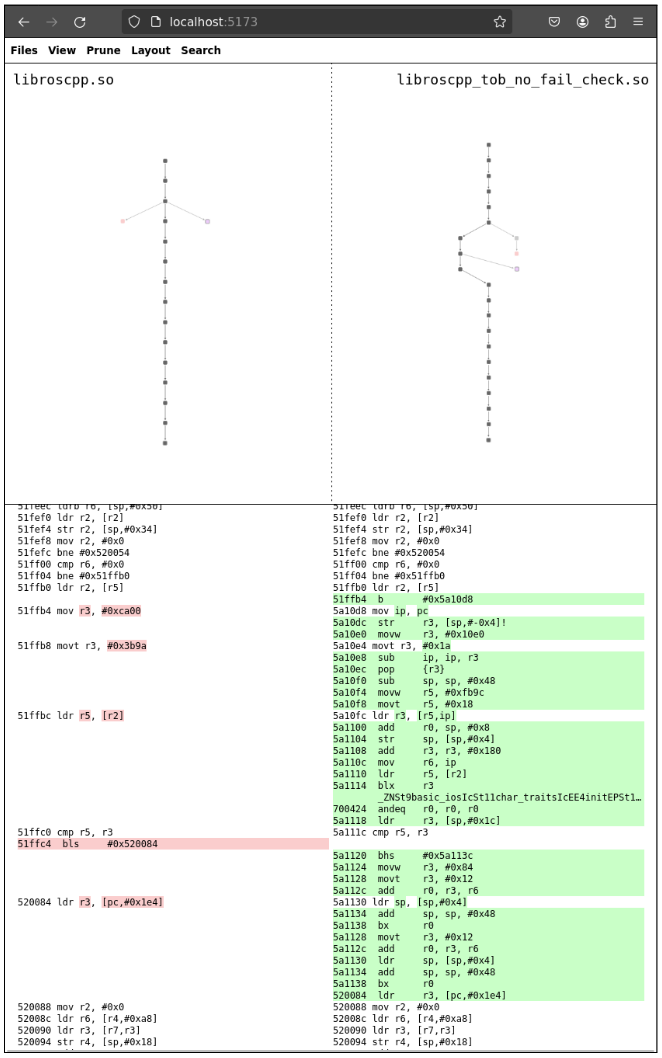
The final mechanism that `cozy` provides for sorting through the results of symbolic execution is *compression*: merging successive nodes that represent uninteresting or inevitable computation steps. There are two available compression levels: the user can (1) merge adjacent nodes that have identical constraints and (2) merge every node that has a unique child with that child node, eliminating all straight-line sequences of symbolic states.[4]

Besides sorting through branches using highlighting, filtering, and compression, the `cozy` user must be able to extract information from symbolic execution results. Within the GUI, there two primary features that expose information about a particular branch to a user. One of these features—tooltips—offers simple at-a-glance information about the nodes in a branch, taken in isolation. The other feature—`cozy`’s diff panel (Section IV-B)—looks at a branch in comparison with a compatible branch selected from the facing tree.

A tooltip appears when the cursor hovers over a node. Depending on the type of node, different kinds of information are available. A tooltip displays the following information: the assembly instructions provided by `angr`’s disassembler for the given state; the representation of those instructions in VEX [9] (the IR over which `angr` performs symbolic execution); the operative symbolic constraints; and concrete examples of possible contents of stdout and stderr. Special states—roughly those with special highlighting rules as described above—expose additional information. For example, error states expose error messages, and states that invoke SimProcs give the name of the function being hooked as well as the library that provides it.

To get a genuinely *comparative* analysis, however, a user needs to select two full branches as follows. First, the user clicks on the leaf of a candidate comparison branch, and that branch will be highlighted, along with all compatible branches in the facing tree. The user can then click on a compatible branch from the facing tree and begin to use the diff panel, as described in the next section.

[4]Symbolic execution can add a constraint without branching when, for example, the result of adding the negation of the constraint is unsatisfiable.


localhost:5173
Files View Prune Layout Search
libroscpp.so
libroscpp_tob_no_fail_check.so

```
51feec ldrb r6, [sp,#0x50]
51fef0 ldr r2, [r2]
51fef4 str r2, [sp,#0x34]
51fef8 mov r2, #0x0
51fefc bne #0x520054
51ff00 cmp r6, #0x0
51ff04 bne #0x51ffb0
51ffb0 ldr r2, [r5]

51ffb4 mov r3, #0xca00


51ffb8 movt r3, #0x3b9a





51ffbc ldr r5, [r2]









51ffc0 cmp r5, r3
51ffc4  bls     #0x520084




520084 ldr r3, [pc,#0x1e4]








520088 mov r2, #0x0
52008c ldr r6, [r4,#0xa8]
520090 ldr r3, [r7,r3]
520094 str r4, [sp,#0x18]
```

```
51feec ldrb r6, [sp,#0x50]
51fef0 ldr r2, [r2]
51fef4 str r2, [sp,#0x34]
51fef8 mov r2, #0x0
51fefc bne #0x520054
51ff00 cmp r6, #0x0
51ff04 bne #0x51ffb0
51ffb0 ldr r2, [r5]
51ffb4  b       #0x5a10d8
5a10d8 mov ip, pc
5a10dc  str     r3, [sp,#-0x4]!
5a10e0  movw    r3, #0x10e0
5a10e4 movt r3, #0x1a
5a10e8  sub     ip, ip, r3
5a10ec  pop     {r3}
5a10f0  sub     sp, sp, #0x48
5a10f4  movw    r5, #0xfb9c
5a10f8  movt    r5, #0x18
5a10fc ldr r3, [r5,ip]
5a1100  add     r0, sp, #0x8
5a1104  str     sp, [sp,#0x4]
5a1108  add     r3, r3, #0x180
5a110c  mov     r6, ip
5a1110  ldr     r5, [r2]
5a1114  blx     r3
         _ZNSt9basic_iosIcSt11char_traitsIcEE4initEPSt1…
700424  andeq   r0, r0, r0
5a1118  ldr     r3, [sp,#0x1c]
5a111c cmp r5, r3

5a1120  bhs     #0x5a113c
5a1124  movw    r3, #0x84
5a1128  movt    r3, #0x12
5a112c  add     r0, r3, r6
5a1130 ldr sp, [sp,#0x4]
5a1134  add     sp, sp, #0x48
5a1138  bx      r0
5a1128  movt    r3, #0x12
5a112c  add     r0, r3, r6
5a1130  ldr     sp, [sp,#0x4]
5a1134  add     sp, sp, #0x48
5a1138  bx      r0
520084  ldr     r3, [pc,#0x1e4]
520088 mov r2, #0x0
52008c ldr r6, [r4,#0xa8]
520090 ldr r3, [r7,r3]
520094 str r4, [sp,#0x18]
```

Fig. 6: Diff panel showing the assembly instructions in the original (left) and patched (right) versions of a program. Red and green highlighting represents deletion and insertion, respectively.

B. *The Diff Panel*

The diff panel becomes available when the user selects a pair of compatible branches for deeper analysis. The types of comparisons that the diff panel supports can be grouped into three broad categories: comparisons of *event streams*, *terminal states*, and *concrete inputs*.

The sequence of nodes along a symbolic execution path corresponds to several different kinds of event streams: the stream of assembly instructions executed, the stream of read and write operations on memory and registers, and the stream of modeled IO effects. `cozy` compares these types of event streams using a familiar git-style line diff. An example of an assembly stream comparison appears in Figure 6, where it is possible to see the exact region where program execution passes through a small patch applied to a shared object file.

For each type of event stream comparison, when the user mouses over an event, the UI highlights the tree node that corresponds to that event. This behavior enables the user to intuitively connect the contents of the tree-view to the contents of the event stream. In some cases, the event stream also contextually exposes other types of information. For example, the stream of assembly instructions can provide the location in the original source that corresponds to a given line of assembly, if this information is recoverable from DWARF debug information in the binaries that `cozy` has analyzed.

In addition to event stream comparisons, `cozy` supports comparisons of terminal states. For example, `cozy` can compare the final memory contents of two compatible branches. This process may involve comparing symbolic values, since terminal states can contain symbolic values. In such a case, `cozy` checks whether the symbolic values in the states are logically equivalent. If they are, `cozy` reports this fact, and if they are not, `cozy` generates some *concretions* that illustrate a possible scenario in which the terminal states differ in spite of an identical initial state.

Finally, the diff panel can generate concrete inputs that exercise compatible branches under comparison. Compatibility guarantees the existence of an input that produces the two sequences of behaviors that the branches represent. The concretion view in the GUI's diff panel displays example inputs that are shared between the two compatible paths. This feature, in combination with `cozy`'s pruning functionality, make it possible to recover specific inputs that generate execution paths of interest, especially paths where behavior differs interestingly between the two binaries being compared.

Compatibility does not guarantee that *every* input that produces the behavior associated with the first branch also produces the behavior associated with the second branch, or vice versa. In cases where there are inputs that will produce the behavior of the first branch, but not the second (or vice versa), `cozy` also makes these inputs available, and in cases where no such inputs exist, `cozy` makes it clear that one of the two branches "refines" the other, or that the two branches are "equivalent," in the sense that they represent behaviors that are produced by exactly the same set of inputs.

V. Evaluation

We evaluated `cozy` by measuring the tool's execution time as it symbolically executed pairs of binary functions and checked a relative correctness property over each pair. The evaluation goals were as follows: (1) to observe `cozy`'s execution speed on widely used binary functions; (2) to demonstrate `cozy`'s ability to verify a desirable correctness property; and (3) to produce a set of `cozy` test harnesses that can serve as a reference point for other users of the tool.

A. Data Set

Each instance in the data set is a pair of binary functions:

1) A function f taken from a Linux `base64` binary
2) A modified version of f instrumented with code that supports coverage-guided fuzz testing

To create the data set, we used the RetroWrite binary rewriting tool [10] to instrument the `base64` binary with code that supports integration with the American Fuzzy Lop (AFL) fuzzer [11]. We then selected 15 functions from the original binary and paired them with their instrumented versions from the modified binary.

B. Correctness Property and Experimental Setup

Because the instrumentation only exists to support fuzzing, a function from the original binary should have the same observable behavior as its instrumented counterpart. We use `cozy` to verify this property as follows. First, `cozy` symbolically executes both functions and computes the set of compatible state pairs. Second, for each pair, `cozy` checks an assertion that the states agree in terms of their register and memory contents. If `cozy` can falsify this assertion, then there exists an input that causes the two functions to behave differently, and verification fails.

The precise formulation of state agreement depends on a function's return type. For example, if a function returns a 64-bit integer, then two compatible states hold equal return values when the full contents of their `RAX` registers are equal (`RAX` is the 64-bit return register for the x86-64 ISA). However, in the case of a function that returns a 32-bit integer, only the lower 32 bits of `RAX` (i.e., register `EAX`) must be equal across the states—the higher-order bits of `RAX` are allowed to differ. Parameter types place similar constraints on the functions' symbolic input. For these reasons, each data instance requires a custom test harness that captures function-specific behavior. These harnesses, along with our full evaluation framework, are included in the public `cozy` repository [5].

C. Results

Using the process described above, we checked each function pair in the data set for equivalent observable behavior. The evaluation took place on a machine running Ubuntu 20.04 with an Intel i9-12900H processor and 64 GB of RAM. The results appear in Figure 7. The table shows symbolic execution time for the original and modified binaries, as well as comparison time, which includes time spent computing compatible states and comparing register and memory contents. `cozy` verifies that the instrumentation code leaves each function's observable behavior unaffected.

VI. Related Work

Computing differences between programs has a long history in the literature. Unlike the symbolic execution discussed here, the majority of previous tools operate on the textual or abstract syntax tree (AST) level [12]–[14], and do not attempt any actual simulation of the programs under analysis.

The `diff` utility [15] distributed with Unix based operating systems is one example of an early comparison program. `diff` reports differences in lines, and performs a longest common subsequence computation to attempt to align two text files. The `diff` utility is generic, in the sense that it will function over any programs that can be represented in text

Function Name	**# Terminal States**	**Symbolic Execution Time (s)**		**Comparison Time (s)**
		Original	Instrumented	
`base64_decode_alloc_ctx`	31	16.0683	41.1273	2.0919
`base64_decode_ctx`	31	15.1894	38.2189	2.0648
`base64_decode_ctx_init`	1	0.0108	0.0685	0.0451
`base64_encode_alloc`	17	6.8573	10.5143	3.9269
`base64_encode`	17	7.6123	12.4908	4.6590
`clone_quoting_options`	1	0.1203	0.1525	0.0461
`close_stdout`	1	1.4348	5.3738	0.0699
`close_stdout_set_file_name`	1	0.0098	0.0656	0.0469
`close_stdout_set_ignore_EPIPE`	1	0.0092	0.0629	0.0442
`close_stream`	58	6.2557	16.2690	4.0236
`decode_4`	29	4.3319	6.7472	1.5532
`deregister_tm_clones`	1	0.0125	0.0529	0.0453
`fadvise`	2	0.1976	0.1738	0.1010
`get_quoting_style`	1	0.1713	0.1030	0.0506
`isbase64`	1	0.1334	0.0849	0.0401

Fig. 7: Binary functions from a Linux `base64` utility, numbers of terminal states that `cozy` symbolic execution finds for them, and running times for `cozy` symbolic execution and verification. For each function, the original and instrumented versions have the same number of terminal states because the instrumentation code is branchless.

files. However this approach, because it does not understand the semantics, cannot be used to provide a rich understanding of program behaviour.

`cozy` does utilize a textual diff over the assembly trace (see Figure 6) of a program in the visualization interface. When two terminal states are selected, the assembly pane will give a linear list of instructions executed for that trace, in the format of color-coded line based diff.

The most relevant prior work to our approach is that of Person et al. in their paper on "Differential Symbolic Execution" [1]. Our approach differs in a number of key ways. First, we analyze binary programs, whereas Person's approach analyzes high-level Java programs. Second, the method by which we check for pair compatibility and report deltas differs. In Person's computation of the partition effect delta, path conditions are checked for strict equivalence using an "if and only if." This approach may detect inconsequential changes in control flow. Our approach is only concerned with observational differences—differences in registers, memory, and IO side effects after execution. It ignores differences at intermediate execution points that Person's tool would flag. Finally, our analysis of final register, memory and IO side effect content is more fine-grained than Person's approach, which has enabled us to create a novel visualization interface.

Person additionally discusses symbolic summary, which we do not utilize in our execution model. *Symbolic summaries* may be used to summarize the effects of common blocks of code. Additionally, *abstract summaries* may be used to skip execution of code that the two programs under comparison have in common. For example, a common code block B when fed identical inputs (registers and memory) will result in the two programs reaching an identical ending state, regardless of the actual execution that occurs within B.

C standard library hooks are one location where symbolic summaries are currently used in `cozy`. These hooks intercept calls to standard library functions, and perform the equivalent computation via a Python callback. The hooks are meant to simplify hard to execute standard library functions, typically resulting in far fewer child states.

Abstract symbolic summaries, while providing interesting benefits, do suffer from several drawbacks that makes them infeasible to use in `cozy`. Due to their black box nature, abstract symbolic summaries do not allow for fine-grained analysis of register and memory contents in terminal states. Additionally, abstract symbolic summaries, since they are essentially computation that is skipped, do not allow for generation of concrete example inputs that lead to selected terminal states. In our experience with the micropatching process, generating concrete example inputs is essential for aiding in understanding program behaviour.

Shadow symbolic execution is another body of work [16]–[18] that functions on principles similar to `cozy`. In shadow symbolic execution, an original and patched program are symbolically executed in lockstep until divergence is reached. Divergent program points are used to generate new test cases that exercise the impact of the patch. Divergence must be manually annotated by constructing a combined original and patched program via a special `change()` macro.

`cozy` differs in several key ways from shadow symbolic execution. `cozy` executes the original and patched binary in two separate symbolic execution runs, removing the need for manual `change()` annotations. Additionally, `cozy` operates on binary programs, whereas the literature on shadow symbolic execution has focused on Java, C, and C++ programs.

VII. Discussion

As part of the DARPA Assured Micropatching (AMP) program, `cozy` was used on a variety of third-party challenge problems. We have additionally created a variety of example programs designed to exercise different portions of the tool. In this section, we discuss our observations on the micropatching process and how `cozy` performs in the overall workflow.

The primary challenge of understanding micropatch behavior is making sense of the large volume of information that `cozy` generates. For all but the simplest programs, the textual report `cozy` generates is too cumbersome to understand. This fact led to the creation of the interactive visualization interface.

Direct examination of the symbolic values attached as state constraints, or stored in registers or memory is generally unhelpful. These symbolic expressions are typically large and too complex to be easily understood with manual inspection. `cozy`'s ability to generate example concrete inputs, for a pair of compatible states, has proven both intuitive and useful.

States with assertion failures are flagged with a purple color, making them easily visible in the tree view. One common workflow is to check that all assertions triggered in the prepatched program are not triggered in the postpatch program. Prepatched assertion failures should be compatible with postpatch states that jump to micropatch code. By exploring various execution traces, concrete examples, and comparisons, the operator can achieve a high degree of assurance that the micropatch is behaving exactly as intended.

The skills required to use `cozy` overlap with those needed to use `angr`. A rough understanding of assembly code is required to attach assertions at certain program points. The initial effort to apply `cozy` to two versions of an application is outlined in Section III-A. The top-level arguments must be constructed, which requires knowledge of the argument types and their memory layouts. One can obtain this information from original source code or from a reverse engineering tool like Ghidra [19].

Since `cozy` uses symbolic execution as its base analysis, it inherits the challenges of that technique: path explosion, nontermination, and costly SMT queries. To mitigate the path explosion problem, we have implemented joint concolic execution (Section III-F). The concolic execution we have implemented may be used for incomplete exploration while preserving terminal state compatibility. The generation of "interesting" concrete inputs is still a weakness of this approach. Although the heuristics attempt to explore deferred execution states that have unique basic block histories, we cannot know what concrete inputs will lead to interesting *future* states.

SMT formulas can encode complex problems, some of which can be difficult to solve. For example, the following program encodes a simple function that checks for a counterexample to Fermat's Last Theorem:

```
input: Integers $a$, $b$, $c$, and $n$
begin
    if $n > 2$ and $a > 0$ and $b > 0$ and $c > 0$ and $a^n + b^n = c^n$:
        print("Wiles's proof of Fermat's Last Theorem was wrong!")
    end
end
```

If we were to run this program in symbolic execution with symbolic inputs, at the branch condition we would have to ask the SMT solver to find a proof for Fermat's Last Theorem. Clearly this is unfeasible for current solvers, and branching predicates can, in general, be undecidable.

Non-termination presents another problem for symbolic execution. It is obviously difficult, in general, to detect non-termination. In some programs, non-termination is a feature; for example, in event-handling loops. To deal with non-termination, we allow the user to place an upper bound on the number of times a loop executes. As a simple mechanism to avoid nontermination, `cozy` uses `angr`'s `LocalLoopSeer` exploration technique, which detects loops by recording the history of execution. If the upper bound on instruction iteration count is reached, we halt execution of that state and stash it. In the visualization, the spinning state can be seen as a downwards facing arrow.

In this paper, we haven't yet touched on the creation of formal specifications for intended patch behavior. Our initial work on the DARPA AMP program focused on this area. A formal specification boiled down to creating an SMT formula with an if-then-else (ITE) at the top level. The condition of the ITE determined when the patch changed program behavior, the true branch specified how the patch changed behavior, and the false branch specified that memory and registers must be identical in all other circumstances.

Tool operators had several complaints about creating these formal specifications: (1) the specifications were difficult to write, requiring the construction of complex SMT formulas; and (2) writing a formal specification was similar to writing the patch in the first place, so there were complaints about having to do the same work twice. Based on feedback from these third-party operators, we determined that an interactive, visualization-based approach would be more helpful.

The feedback loop created by the `cozy` tool is, in essence, an interactive way to explore the formal specification space. It is possible to use `cozy` to check directly that a patch changes behavior only in some specified way. This kind of formal verification is accomplished by writing a function that takes in a compatible state pair and returns an assertion condition. If `cozy` can falsify the assertion for any compatible state pair, then verification fails.

VIII. Conclusion

In this paper we have presented `cozy`, a Python-based framework built on top of `angr` that uses symbolic execution to detect observable differences in binary programs. The `cozy` project is designed to analyze micropatches, which are small binary or assembly-patches inserted into existing legacy programs. By using `cozy`'s novel visualization interface, the tool's operator can gain confidence that a given micropatch has its intended effect. Operators who already have experience with the `angr` symbolic execution framework will find it easy to get started with `cozy`. We hope that operators will find `cozy` useful as part of the verification step of the micropatch development process.

References

[1] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu, “Differential Symbolic Execution,” in *Foundations of Software Engineering (FSE)*, 2008. [Online]. Available: https://doi.org/10.1145/1453101.1453131

[2] G. P. Farina, S. Chong, and M. Gaboardi, “Relational Symbolic Execution,” in *Principles and Practice of Programming Languages (PPDP)*, 2019. [Online]. Available: https://doi.org/10.1145/3354166.3354175

[3] 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. [Online]. Available: https://doi.org/10.1109/SP.2016.17

[4] “`cozy` Python Package Index (PyPI) entry,” 2024, [Link removed for double-blind review].

[5] “GitHub repository for the `cozy` development,” 2024, [Link removed for double-blind review].

[6] P. Godefroid, N. Klarlund, and K. Sen, “DART: Directed Automated Random Testing,” *SIGPLAN Not.*, vol. 40, no. 6, p. 213–223, jun 2005. [Online]. Available: https://doi.org/10.1145/1064978.1065036

[7] K. Sen, “Concolic Testing,” in *Automated Software Engineering (ASE)*, 2007. [Online]. Available: https://doi.org/10.1145/1321631.1321746

[8] J. Wang, Y. Duan, W. Song, H. Yin, and C. Song, “Be Sensitive and Collaborative: Analyzing Impact of Coverage Metrics in Greybox Fuzzing,” in *Research in Attacks, Intrusions and Defenses (RAID)*, 2019. [Online]. Available: https://www.usenix.org/conference/raid2019/presentation/wang

[9] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, “Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware,” in *Network and Distributed System Security Symposium (NDSS)*, 2015. [Online]. Available: https://www.ndss-symposium.org/ndss2015/firmalice-automatic-detection-authentication-bypass-vulnerabilities-binary-firmware

[10] S. Dinesh, N. Burow, D. Xu, and M. Payer, “RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization,” in *IEEE Symposium on Security and Privacy*, 2020. [Online]. Available: https://doi.org/10.1109/SP40000.2020.00009

[11] M. Zalewski, “american fuzzy lop,” 2017. [Online]. Available: https://lcamtuf.coredump.cx/afl/

[12] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, “Fine-Grained and Accurate Source Code Differencing,” in *Automated Software Engineering (ASE)*, 2014. [Online]. Available: http://doi.acm.org/10.1145/2642937.2642982

[13] B. Fluri, M. Würsch, M. Pinzger, and H. Gall, “Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction,” *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 725–743, 2007.

[14] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, “Change Detection in Hierarchically Structured Information,” *SIGMOD Rec.*, vol. 25, no. 2, p. 493–504, jun 1996. [Online]. Available: https://doi.org/10.1145/235968.233366

[15] J. W. Hunt and M. D. MacIlroy, *An Algorithm for Differential File Comparison*. Bell Laboratories Murray Hill, 1976.

[16] H. Palikareva, T. Kuchta, and C. Cadar, “Shadow of a Doubt: Testing for Divergences between Software Versions,” in *International Conference on Software Engineering (ICSE)*, 2016. [Online]. Available: https://doi.org/10.1145/2884781.2884845

[17] Y. Noller, H. L. Nguyen, M. Tang, and T. Kehrer, “Shadow Symbolic Execution with Java PathFinder,” *ACM SIGSOFT Softw. Eng. Notes*, vol. 42, no. 4, pp. 1–5, 2017. [Online]. Available: https://doi.org/10.1145/3149485.3149492

[18] T. Kuchta, H. Palikareva, and C. Cadar, “Shadow Symbolic Execution for Testing Software Patches,” *ACM Trans. Softw. Eng. Methodol.*, vol. 27, no. 3, pp. 10:1–10:32, 2018. [Online]. Available: https://doi.org/10.1145/3208952

[19] NSA, “Ghidra,” https://www.ghidra-sre.org/, accessed 2024-06-13.