PATE Verifier User Manual (Version 0.9)

Galois (pate-darpa-amp@galois.com)

July 28, 2023

The PATE verifier is a static relational verifier for binaries with the goal of proving that two binaries have the same observable behaviors. It is intended for proving that security micropatches applied to binaries are safe. When it cannot, the verifier provides detailed explanations that precisely characterize the difference in behavior between the two binaries. The verifier is intended to be usable by *domain experts*, rather than verification experts, and its explanations are designed to be in domain terms as much as possible. The verifier is intended to enable users to:

- precisely reason about the effects of patches applied to binaries,
- explain potential differences in observable behaviors, accounting for all possible execution paths, and thus
- reduce the time required to develop safe binary patches.

Contents

1 Introduction

This is the User Manual for the PATE verifier, updated to be consistent with the software release as of the end of AMP Phase 2 (7/28/23). This manual can be found within the repository snapshot comprising the code release, in the directory `docs/usermanual`. Much of the material in this document concerning how to build and run the verifier, as well as how to apply it to demonstration examples, can also be found within the repository. In the sections below, we provide pathnames for the corresponding documentation files to be found in the software release.

Currently, the verifier supports PowerPC and AArch32 binaries (currently requiring statically linked ELF binaries).

The PATE verifier is a static relation verifier for binaries that builds assurance that micropatches have not had any adverse effects. The verifier is a static relational verifier that attempts to prove that two binaries have the same observable behaviors. When it cannot, the verifier provides detailed explanations that precisely characterize the difference in behavior between the two binaries. The verifier is intended to be usable by *domain experts*, rather than verification experts, and its explanations are designed to be in domain terms as much as possible. After applying a micropatch to a binary, domain experts can apply the verifier to ensure that the effects are intended.

Note that while the verifier attempts to prove that the original and patched binaries have the same observable behaviors under all possible inputs, it is expected that they do not (or the patch would have had no effect). When the two binaries can exhibit different behaviors, the verifier provides the user with an explanation of how and where the behavior is different.

If DWARF information is available in either the original or patched binary, the verifier will use that information to improve diagnostics. Currently, function names, function argument names, local variable names, and global variable names can be used to make diagnostics more readable, for example, by replacing synthetic names with their source-level counterparts. If working with binaries that do not come with DWARF debug information natively, see the `dwarf-writer`[1] tool for a possible approach to adding DWARF debug information.

Note that recompiling a binary with a source patch applied can work for the purposes of the analysis, but can introduce complexities in cases where the compiler substantially rearranges code in response to the patch (which is common). When the compiler re-arranges code, the verifier has a more difficult time aligning the code in the original and patched binaries, which can lead to confusing or unhelpful diagnostics.

2 Building the PATE Verifier

The pate tool is written in Haskell and requires the GHC compiler (version 8.6-8.10) and the cabal build tool to compile. Building from source can be accomplished as follows:

```
git clone git@github.com:GaloisInc/pate.git
cd pate
git submodule update --init
cp cabal.project.dist cabal.project
cabal configure pkg:pate
```

[1] https://github.com/immunant/dwarf-writer

```
pate.sh --help
```

The verifier also requires an SMT solver to be available in "PATH". The default is "yices", but "z3" and "cvc4" are also supported. The verifier can also be built as a Docker image:

```
docker build . -t pate
```

The correctness of the build can be confirmed by

```
docker run --rm -it -p 5000:5000 -v `pwd`/tests:/tests pate \
     --original /tests/aarch32/const-args.original.exe \
     --patched /tests/aarch32/const-args.patched.exe
```

3 Running the PATE Verifier

3.1 Running From a Docker Container

If you have a Docker image containing the verifier, load and run it:

```
docker load -i /path/to/pate.tar
```

To run the verifier via Docker after this::

```
docker run --rm -it pate --help
```

3.2 Usage

To run the verifier on existing test binaries in the software release. From the root directory of the repository:

```
docker run --rm -it -v `pwd`/demos:/demos pate \
    -o /demos/challenge09/static/challenge09.original.exe \
    -p /demos/challenge09/static/challenge09.patched.exe \
    --original-bsi-hints /demos/challenge09/static/challenge09.json \
    --patched-bsi-hints /demos/challenge09/static/challenge09.json \
    -e ContinueAfterFailure \
    --save-macaw-cfgs /demos/challenge09/static/CFGs/ \
    -s "main" \
    -b /demos/challenge09/static/challenge09.toml
```

3.3 Command Line Options

The verifier accepts the following command line arguments::

```
  -h,--help                Show this help text
  -o,--original EXE        Original binary
  -p,--patched EXE         Patched binary
  -b,--blockinfo FILENAME  Block information relating binaries
  -s,--startsymbol ARG     Start analysis from the function with this symbol,
                           otherwise start at the program entrypoint
  -d,--nodiscovery         Don't dynamically discover function pairs based on
```

```
                        calls.
--solver ARG            The SMT solver to use to solve verification
                        conditions. One of CVC4, Yices, or Z3
                        (default: Yices)
--goal-timeout ARG      The timeout for verifying individual goals in seconds
                        (default: 300)
--heuristic-timeout ARG The timeout for verifying heuristic goals in seconds
                        (default: 10)
--original-anvill-hints ARG
                        Parse an Anvill specification for code discovery
                        hints
--patched-anvill-hints ARG
                        Parse an Anvill specification for code discovery
                        hints
--original-probabilistic-hints ARG
                        Parse a JSON file containing probabilistic function
                        name/address hints
--patched-probabilistic-hints ARG
                        Parse a JSON file containing probabilistic function
                        name/address hints
--original-csv-function-hints ARG
                        Parse a CSV file containing function name/address
                        hints
--patched-csv-function-hints ARG
                        Parse a CSV file containing function name/address
                        hints
--original-bsi-hints ARG Parse a JSON file containing function name/address
                        hints
--patched-bsi-hints ARG Parse a JSON file containing function name/address
                        hints
--no-dwarf-hints        Do not extract metadata from the DWARF information in
                        the binaries
-V,--verbosity ARG      The verbosity of logging output (default: Info)
--save-macaw-cfgs DIR   Save macaw CFGs to the provided directory
--solver-interaction-file FILE
                        Save interactions with the SMT solver during symbolic
                        execution to this file
--proof-summary-json FILE
                        A file to save interesting proof results to in JSON
                        format
--log-file FILE         A file to save debug logs to
-e,--errormode ARG      Verifier error handling mode
                        (default: ThrowOnAnyFailure)
-r,--rescopemode ARG    Variable rescoping failure handling mode
                        (default: ThrowOnEqRescopeFailure)
--skip-unnamed-functions Skip analysis of functions without symbols
```

4 Interacting with the Verifier

Once the REPL (Read-EVAL-Print loop) has started, the first step of the analysis is to select an entry point to start from. By default, the verifier starts verifying from the formal program entry point. This is often not very useful (and can be problematic for complex binaries with a large `_start` that causes problem for our code discovery). Additionally, for changes with a known (or at least expected) scope of impact, analyzing just the affected functions is significantly faster. To instead specify an analysis entry point, passing the `-s <function_symbol>` option will start the analysis from the function corresponding to the given symbol. Note that this requires function symbols to be provided for the binaries (either as embedded debug symbols or separately in one of the hint formats)::

```
docker run --rm -it -v‘pwd‘/tests:/tests/hints pate \
           --original /tests/01.elf \
           --patched /tests/01.elf \
           --original-anvill-hints /tests/01.anvill.json \
           --patched-anvill-hints /tests/01.anvill.json \
           -s main
```

The user can then select the desired entry point:

```
Choose Entry Point
0: Function Entry 0xdead
1: Function Entry "my\_fun" (0xfeed)
?> 1
```

The top level of the interactive process (reachable via the`top` command) is a list of all analysis steps that were taken, starting from the selected entry point. Each pair of address and calling contexts defines a unique toplevel proof "node". A given address and context may appear multiple times in the toplevel list, corresponding to each individual time that the address/context pair was analyzed. The latest (highest-numbered) entry corresponds to the most recent analysis of an address/context.

Each entry point is associated with an equivalence domain: a set of locations (registers, stack slots and memory addresses) that are potentially not equal at this point. Locations outside of this set have been proven to be equal (ignoring skipped functions). The analysis takes the equivalence domain of an entry point and computes an equivalence domain for each possible exit point (according to the semantics of the block).

Differences between the two binaries may be reported into two ways:

- Observable trace differences – library calls or writes to distinguished memory regions which have different contents or occur in different orders
- Control flow divergence –significant divergence in control flow between the programs (i.e. one program returns while the other calls a function)

For example, if a function `my_fun` calls `my_sub_fun`, we might see the following toplevel output.

```
0: Function Entry "my\_fun" (0xfeed) (User Request)
1: Function Entry "my\_sub\_fun" (0xdeef) [ via: "my\_fun" (0xfeef) ]  (Widening Equivalence Domains)
2: 0xdeff [ via: "my\_sub\_fun" (0xdeef) <- "my\_fun" (0xfeed) ] (Widening Equivalence Domains)
4: Return "my\_sub\_fun" (0xdeef) (Widening Equivalence Domains)
5: 0xfeef [ via: "my\_sub\_fun" (0xdeef) <- "my\_fun" (0xfeed) ] (Widening Equivalence Domains)
6: Return "my\_fun" (0xfeed) (Widening Equivalence Domains)
```

The contents of a node can be inspected by entering the corresponding number at the prompt. A toplevel node contains sub-nodes corresponding to each possible "exit" that was discovered. For example suppose we entered `2` at the prompt above. The components of that node as displayed are:

- `2:` - the number of this node, enter at the prompt to navigate to it and see a detailed view of the analysis step
- `0xdeff` - the address that this analysis started from
- `[ via: "my_sub_fun" (0xdeef) <- "my_fun" (0xfeed) ]` - the calling context for this node, shown as the trace of function calls taken on the path to it (most recent call is first, left-to-right).
- `(Widening Equivalence Domains)` - the reason this analysis step was taken. This is provided by whichever previous analysis step scheduled this node to be processed.

The *prompt* indicates the status of the current node as follows:

- `*>` current node still has some active task running
- `?>` current node requires user input
- `!>` current node has raised a warning
- `x>` current node has raised an error
- `>` current node, and all sub-nodes, have finished processing

Similar to the prompt, nodes may be printed with a suffix that indicates some additional status as follows:

- `(*)` node still has some active task running
- `(?)` node requires user input
- `(!)` node has raised a warning
- `(x)` node has raised an error

A status suffix indicates that the node, or some sub-node, has the given status. e.g. at the toplevel the prompt `x>` indicates that an error was thrown during some block analysis, while the corresponding node for the block will have a `(x)` suffix.

Navigation Commands:

- `#` - navigate to a node, printing its contents
- `up` - navigate up one tree level
- `top` - navigate to the toplevel
- `goto_err` - navigate to the first leaf node with an error status
- `next` - navigate to the highest-numbered node at the current level

Diagnostic Commands:

- `status` - print the status of the current node
- `full_status` - print the status of the current node, without truncating the output
- `ls` - print the list of nodes at the current level
- `wait` - wait at the current level for more results. Exits when the node finishes, or the user provides any input

When the prompt is `?>`, the verifier is waiting for input at some sub-node. To select an option, simply navigate (i.e. by entering `#`) to the desired choice. For example, `goto_prompt` - navigate to the first leaf node waiting for user input.

5 Demonstration scenario

This section contains a detailed walk through of a demonstration of using the verifier to verify a patch to AMP Challenge 10.

From the repository root directory, run the verifier in its docker container:

```
docker run --rm -it -v ‘pwd‘/demos/may-2023/challenge10:/challenge10 pate \
           --original /challenge10/challenge10.original.exe \
           --patched /challenge10/challenge10.patched.exe \
           -b /challenge10/challenge10.toml \
           --original-bsi-hints /challenge10/challenge10.json \
           --patched-bsi-hints /challenge10/challenge10.json \
           --original-csv-function-hints /challenge10/challenge10.csv \
           --patched-csv-function-hints /challenge10/challenge10.csv \
           -s transport\_handler
```

The file `challenge10.toml` specifies additional metadata needed for verification. Specifically it instructs the verifier to start decoding the binary in thumb mode.

The file `challenge10.json` contains symbol information extracted using the BSI tool. This information is used to identify functions that should use stub semantics rather than be analyzed (e.g. libc calls).

The file `challenge10.csv` contains manually-defined symbol information that was not automatically extracted from the BSI tool. This is necessary to handle some PLT stubs that were not identified.

The last line tells the verifier to start the analysis at the function corresponding to the symbol `transport_handler`, which is known from the BSI symbol data. This is the function in which the patch appears.

Once the verifier starts printing output it can be interrupted at any time by pressing ENTER, but will continue processing in the background. Its results can then be interactively inspected as they become available.

See Section 4for more information on how to interact with the verifier's Read-Eval-Print loop.

Step 1: Select the entry point and wait

Select `1` to start the analysis from the `transport_handler` function.

```
Choose Entry Point
0: Function Entry segment1+0x3ba9
1: Function Entry "transport\_handler" (segment1+0x400c)
?> 1
.....................
0: Function Entry "transport\_handler" (segment1+0x400c) (User Request) (!).
...
```

The verifier then proceeds to print out each analysis step until user input is required.

Step 2: Choose a synchronization point

During the analysis of the block starting at `0x4114` the analysis encounters a control flow divergence. This is an expected result of the patch, which, as shown in Figure 1, has inserted a trampoline starting at `0x4128`. If the verifier is polling for output this will appear automatically, otherwise if the output was interrupted we can navigate to prompt by executing `top` followed by `goto_prompt`:

```
?>goto\_prompt
Control flow desynchronization found at: GraphNode segment1+0x4114
                 [ via: "transport\_handler" (segment1+0x400c) ]
```

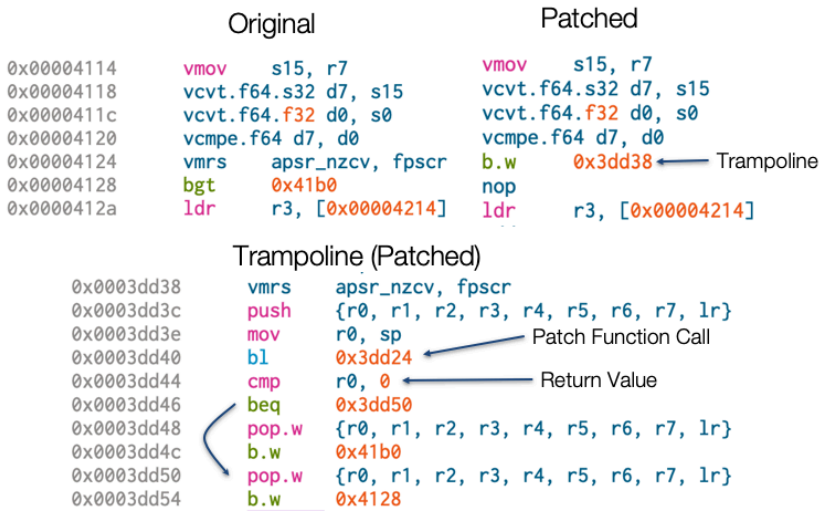

Figure 1: Comparison of aligned regions in original and patched binaries for Challenge 10

```
0: Ignore divergence (admit a non-total result)
1: Assert divergence is infeasible
2: Assume divergence is infeasible
3: Remove divergence in equivalence condition
4: Choose synchronization points
5: Defer decision
?>
```

Again, referring to Figure 1 we can see where this happens in the code. We can check the context of this choice by executing `up` then `up` to see the node that was being processed when this prompt was created.::

```
?>up
...
?>up
segment1+0x4114 [ via: "transport\_handler" (segment1+0x400c) ] (Widening Equivalence Domains)
0: Widening Equivalence Domains
1: Modify Proof Node
2: Predomain
3: Observably Equivalent
4: Block Exits (?)
5:   Call to: "puts" (segment1+0x33ac) Returns to: "transport\_handler"
     (segment1+0x41b8) (original) vs. Call to: segment1+0x3dd24
    Returns to: "transport\_handler" (segment1+0x3dd44) (patched) (?)
?>
```

Here we see that, from `0x4114` there are disagreeing block exits. Specifically in the original program the block can exit with a call to `puts` while the patched function exits with a call to the anonymous function at `0x3dd24` (the inserted patch function).

To handle this, we need to instruct the verifier to perform a single-sided analysis on each program, and specify the point at which control flow re-synchronizes. Specifically, we need to provide instruction addresses for the original and patched programs where, if execution reaches these addresses, both programs will resume in lockstep (i.e. all possible block exits (function calls) will be equal). We navigate to the prompt with `goto_prompt` and select `4: Choose synchronization points`.

We are then prompted to provide a pair of program points by selecting from a list of instructions. With a separate analysis we can determine that the required synchronization points are `segment1+0x3dd44 (patched)` and `segment1+0x4128 (original)`. As shown in Figure 1 at address `0x3dd44` (in the inserted trampoline), the patched program mirrors the branch instruction at `0x4128` in the original program.

Select these instructions from the list (one at a time) and the analysis will then continue.

Step 3: Generate an equivalence condition

The top-level nodes produced after this point are suffixed by `(original)` or `(patched)`, indicating which single-step analysis they correspond to. After some analysis, the verifier prompts with another control flow desynchronization.

```
Control flow desynchronization found at: GraphNode segment1+0x4128 (original) vs.
     segment1+0x3dd44 (patched) [ via: "transport\_handler" (segment1+0x400c) ]
0: Ignore divergence (admit a non-total result)
1: Assert divergence is infeasible
2: Assume divergence is infeasible
3: Remove divergence in equivalence condition
4: Choose synchronization points
5: Defer decision
?>
```

This desynchronization indicates that control flow may still diverge between the original and patched programs after the synchronization point we provided. This is exactly the intended result of our patch: after this point the program control flows *may* be equal (i.e., in the case where the patch has simply recovered the original behavior of the program), but they may also be unequal (i.e., in the case where the patch has modified the program behavior).

Since this desynchronization precisely describes the non-equal branching behavior, we can exclude it from our analysis by asserting its negation as our generated *equivalence condition*. This is option `3: Remove divergence in equivalence condition.`

After some analysis a similar prompt is given (corresponding to the inverse branching behavior), which we similarly handle by selecting `3` to assert the negation of this path condition.

The analysis then proceeds with this desynchronization omitted (and with a generated equivalence condition asserted at the synchronization point).

****

THIS IS THE POINT AT WHICH THE CURRENT DOCKER IMAGE (as of 7/27/23) throws an error. So below here is not verified against actual execution.

****

Step 4: Strengthening the equivalence domain

After some time, the analysis eventually halts with a prompt indicating that a control flow difference has been found at `0x4181`. With some investigation we can

determine that this difference is spurious. At the prompt, navigate to the toplevel node for `0x4181` via `up` then `up`, and select the option `2: Predomain`

```
?>up
..
?>up
segment1+0x4181 [ via: "transport\_handler" (segment1+0x400c) ]
          (Widening Equivalence Domains)
0: Widening Equivalence Domains
1: Modify Proof Node
2: Predomain
3: Observably Equivalent
4: Block Exits (?)
5:   Call to: "err" (segment1+0x33ec) Returns to: "transport\_handler"
    (segment1+0x4191)
6:   Call to: "err" (segment1+0x33ec) Returns to: "transport\_handler"
       (segment1+0x4191) (original) vs. Branch to: "transport\_handler"
       (segment1+0x402d) (patched) (?)
?>2
```

The output here indicates that, although control flow is synchronized between the programs, several registers as well as global memory values are excluded from the equivalence domain (i.e., not known to be necessarily equivalence at this point).

The source of this inequivalence can be traced to the instruction immediately following the synchronization point at `0x412a` (`top` then `25` then `2`). At this point, the equivalence domain has excluded `r0-r7` as well as the stack pointer and several stack slots. This spurious inequivalence is a result of the trampoline saving and then restoring these registers onto the stack before resuming normal control flow. The analysis has not retained enough context about the trampoline execution to automatically prove that this save/restore operation is sound.

We can instruct the verifier to strengthen the equivalence domain by explictly asserting that, at this program point, these registers are necessarily equivalent between the original and patched programs. At the node for `0x412a` (`top` then `25`), select the option `1: Modify Proof Node`. From this list we add an assertion by selecting `1: Assert condition`.

After providing this input , we are presented with the same control flow desynchronization prompt, which we now defer by selecting `4: Defer decision`, which will then present the prompt for the assertion we wish to add

```
Include Register:
0: r0
1: r1
2: r13
3: r2
4: r3
5: r4
6: r5
7: r7
8: Include Remaining Registers
9: Exclude Remaining Registers
?> 8
```

This is the list of registers that were excluded from the equivalence domain from `0x412a` Select `8` to include all of the given registers. This choice results in an assertion that all of the user registers are necessarily equal between the original and patched programs when they both reach `0x412a`. The analysis then proceeds by propagating the assertion up several nodes (indicated by the `Propagating Conditions` status), which is then eventually discharged. The subsequent proof nodes are then re-checked under this new assertion, and correspondingly strengthened equivalence domain.

Step 5: Propagating and interpreting the equivalence condition

The verifier can now complete the analysis, prioviding a proof that the programs are exactly equivalent under the generated equivalence condition. By default the condition is only asserted at exactly the location it is needed, however it can also be propagated to the entry point, in order to compute a sufficient condition at the beginning of the function call.

To do this, we navigate to the synchronization node (`top` then `57`) where we can see that an equivalence condition has been assumed. However this is only in terms of the condition registers at this point. Select `1: Modify Proof Node` and then `21: Propagate fully`.

Then select `2: Handle pending refinements` at the next prompt to handle the requested action. Once finished, the resulting equivalence condition can be examined by navigating to the node corresponding to the function entry point for `transport_handler`.

6 Acknowledgement

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) and Naval Information Warfare Center Pacific (NIWC Pacific) under Contract Number N66001-20-C-4027. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the DARPA & NIWC Pacific.