PATE Verifier User Manual (Version 0.9)

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

April 3, 2025

The PATE verifier is a static relational verification tool designed to ensure that security micropatches applied to binaries are safe and free from unintended consequences. The verifier generates detailed reports that precisely identify the conditions under which two binary versions exhibit identical behavior. This equivalence information can reveal any unforeseen side effects of a binary patch, such as changes to program behaviors beyond those intended by the original fix.

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 User Manual describes the PATE verifier as of February 2024, and is maintained in the PATE source repository[1].

The PATE verifier is a static relational verifier for binaries that builds assurance that micropatches have not introduced 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. 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 debugging 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`[2] tool for a possible approach to adding DWARF debug information.

PATE is designed to analyze binaries produced by applying a micropatch at the binary level where overall the pair of binaries are "mostly" similar. It is more difficult for PATE to compare binaries with substantially different control flow due to the challenges in finding corresponding slices to align for analysis. Major control flow divergences may require user input to guide PATE in its analysis process. We have observed that in some cases, compiling two programs with similar source (such as making a small source-level modification and rebuilding) can result in binaries that are more different than the small source code change might imply, as a result of various modern compiler heuristics and optimizations.

Currently, the verifier supports 32-bit ARM binaries, and 64-bit and 32-bit PowerPC binaries.

2 Installing the PATE Verifier

2.1 Recommended: Docker Container Image

We recommend using PATE via Docker.

If available, load the Docker image file from a file using:

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

Otherwise, build the Docker image from the PATE source repo. Building PATE is a memory-intensive process. We build the Docker image in an environment with 16 GB of RAM available to

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

Docker. Build the image using the following commands:

```
git clone git@github.com:GaloisInc/pate.git
cd pate
git submodule update --init
docker build --platform linux/amd64 . -t pate
```

PATE may subsequently be used via Docker, such as:

```
docker run --rm -it \
  --platform linux/amd64 \
  -v "$(pwd)"/tests/integration/packet:/target \
  pate \
  --original /target/packet.original.exe \
  --patched /target/packet.patched.exe \
  -s parse_packet
```

Please see later sections for detailed usage information.

2.2 Alternative: Building from Source

Alternatively, PATE may be compiled from source and run on a host system. The PATE tool is written in Haskell and requires the GHC compiler[3] and cabal[4] to build.

The current version of PATE is developed using GHC version 9.6.2 and cabal version 3.10.2.0. Additionally, the verifier requires an SMT solver to be available in `PATH`. We recommend `yices`. The `z3` and `cvc4` solvers may also work but are not regularly tested with PATE.

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
./pate.sh --help
```

Note that `./pate.sh` should be used to build PATE (not `cabal build` or similar).

Once built, invoke PATE locally using the `pate.sh` script.

3 Invoking the PATE Verifier

Once PATE has been built as described in Section 2, use the Docker container (or `pate.sh` script if building locally from source) to invoke PATE.

Invoking PATE presents the user with an interactive terminal user interface for analyzing a pair

[3]`https://www.haskell.org/ghc/`
[4]`https://www.haskell.org/cabal/`

of binaries. Users provide at least the paths to two (the “original” and “patched”) binaries at the command line when invoking PATE. Additionally, we recommend also specifying the desired entry point, using `-s <symbol>`. Providing the specific function of interest will enable PATE to begin reasoning closer to the relevant functionality of interest, greatly reducing the total analysis time required compared to starting from the program entry point. An example invocation using Docker looks like:

```
docker run --rm -it \
  --platform linux/amd64 \
  -v "$(pwd)"/z:/z \
  pate \
  --original /z/my.original.exe \
  --patched /z/my.patched.exe \
  -s start_function_name
```

Users may also wish to provide “hint” inputs that provide metadata that maps symbol names to addresses, easing analysis of stripped binaries. Please see the listing below for details.

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
--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
  --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
  --ignore-segments ARG    Skip segments (0-indexed) when loading ELF
  --json-toplevel          Run toplevel in JSON-output mode (interactive mode
                          only)
  --read-only-segments ARG Mark segments as read-only (0-indexed) when loading
                          ELF
  --script FILENAME        Path to a pate script file. Provides pre-defined
                          input for user prompts (see
                          tests/integration/packet-mod/packet.pate for an
                          example and src/Pate/Script.hs for details)
  --assume-stack-scope     Add additional assumptions about stack frame scoping
                          during function calls (unsafe)
  --ignore-warnings ARG    Don't raise any of the given warning types
  --always-classify-return Always resolve classifier failures by assuming
                          function returns, if possible.
  --add-trace-constraints  Prompt to add additional constraints when generating
                          traces.
  --quickstart             Start analysis immediately from the given entrypoint
                          (provided from -s)
```

3.1 –assume-stack-scope

This flag causes PATE to assume that out-of-scope stack slots (i.e. slots that are past the current stack pointer) are always equal between the two binaries. Although in actuality they may have differing out-of-scope stack contents, programs that exercise proper stack hygiene should never access those values. By assuming they are equal, PATE avoids creating spurious counter-examples where under-specified pointers alias inequivalent, but out-of-scope, stack slots. This is a fairly strong assumption that can have unintended side effects, including assuming *false* when out-of-scope stack contents are *provably* inequivalent.

4 The PATE Terminal UI

Once PATE has been invoked at the command line (see Section 3), the user is presented with an interactive terminal user interface.

Internally, PATE maintains a tree representing the state of the analysis. The interactive interface allows users to inspect the tree by selecting from a list of options based on the users current "location" in the tree. The user makes a selection by entering a number representing the node of interest and hitting enter, or by entering a command.

4.1 Example Usage

Launch PATE on the binaries in the `tests/integration/packet/` directory with:

```
docker run --rm -it \
  --platform linux/amd64 \
  -v "$(pwd)"/tests/integration/packet:/target \
  pate \
  --original /target/packet.original.exe \
  --patched /target/packet.patched.exe \
  -s parse_packet
```

Once launched, PATE presents the user is presented with a list of entry points and makes a selection:

```
Choose Entry Point
0: Function Entry "_start" (segment1+0x435)
1: Function Entry "parse_packet" (segment1+0x590)
?> 1
```

We indicate user input with a yellow highlight. Here the user enters `1`, selecting `parse_packet`. This starts PATE's analysis at this point, and the user sees output of:

```
..
0: Function Entry "parse_packet" (segment1+0x590) (User Request)
1: segment1+0x5c0 [ via: "parse_packet" (segment1+0x590) ]
   (Widening Equivalence Domains)
2: segment1+0x600 [ via: "parse_packet" (segment1+0x590) ]
   (Widening Equivalence Domains)
3: Return "parse_packet" (segment1+0x590)
   (Widening Equivalence Domains)
4: segment1+0x63c [ via: "parse_packet" (segment1+0x590) ]
   (Widening Equivalence Domains)
5: segment1+0x658 [ via: "parse_packet" (segment1+0x590) ]
   (Widening Equivalence Domains)
6: segment1+0x698 [ via: "parse_packet" (segment1+0x590) ]
   (Widening Equivalence Domains)
7: segment1+0x6d8 [ via: "parse_packet" (segment1+0x590) ]
   (Widening Equivalence Domains)
```

```
8: Function Entry "check_crc" (segment1+0x530)
   [ via: "parse_packet" (segment1+0x6ec) ]
   (Widening Equivalence Domains)
9: Return "check_crc" (segment1+0x530)
   [ via: "parse_packet" (segment1+0x6ec) ]
   (Widening Equivalence Domains)
10: segment1+0x6ec [ via: "parse_packet" (segment1+0x590) ]
    (Widening Equivalence Domains)
Handle observable difference:
0: Emit warning and continue
1: Assert difference is infeasible (defer proof)
2: Assert difference is infeasible (prove immediately)
3: Assume difference is infeasible
4: Avoid difference with equivalence condition
5: Avoid difference with path-sensitive equivalence condition
?> 4
```

PATE represents the analysis as a tree that can be navigated by the user. 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 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 this set have been proven to be equal (ignoring skipped functions). The analysis takes the equivalence domain of an address/context pair and computes an equivalence domain for each possible exit (according to the semantics of the block).

At this point in the example, PATE is asking how a detected observable difference should be handled. The user selects `4` to capture the difference in the equivalence condition, and PATE continues its analysis:

```
0: Function Entry "parse_packet" (segment1+0x590) (User Request)
1: segment1+0x5c0 [ via: "parse_packet" (segment1+0x590) ]
   (Widening Equivalence Domains)
2: segment1+0x600 [ via: "parse_packet" (segment1+0x590) ]
   (Widening Equivalence Domains)
3: Return "parse_packet" (segment1+0x590)
   (Widening Equivalence Domains)
4: segment1+0x63c [ via: "parse_packet" (segment1+0x590) ]
   (Widening Equivalence Domains)
5: segment1+0x658 [ via: "parse_packet" (segment1+0x590) ]
   (Widening Equivalence Domains)
6: segment1+0x698 [ via: "parse_packet" (segment1+0x590) ]
   (Widening Equivalence Domains)
7: segment1+0x6d8 [ via: "parse_packet" (segment1+0x590) ]
```

```
   (Widening Equivalence Domains)
8: Function Entry "check_crc" (segment1+0x530)
   [ via: "parse_packet" (segment1+0x6ec) ]
   (Widening Equivalence Domains)
9: Return "check_crc" (segment1+0x530)
   [ via: "parse_packet" (segment1+0x6ec) ]
   (Widening Equivalence Domains)
10: segment1+0x6ec [ via: "parse_packet" (segment1+0x590) ]
    (Widening Equivalence Domains)
11: segment1+0x71c [ via: "parse_packet" (segment1+0x590) ]
    (Widening Equivalence Domains)
12: segment1+0x71c [ via: "parse_packet" (segment1+0x590) ]
    (Re-checking Block Exits)
13: segment1+0x6ec [ via: "parse_packet" (segment1+0x590) ]
    (Propagating Conditions)
14: segment1+0x6ec [ via: "parse_packet" (segment1+0x590) ]
    (Re-checking Block Exits)
15: Return "check_crc" (segment1+0x530) [ via: "parse_packet" (segment1+0x6ec) ]
    (Propagating Conditions)
16: segment1+0x71c [ via: "parse_packet" (segment1+0x590) ]
    (Widening Equivalence Domains)
17: Return "parse_packet" (segment1+0x590) (Widening Equivalence Domains)
18: segment1+0x758 [ via: "parse_packet" (segment1+0x590) ]
    (Widening Equivalence Domains)
19: Function Entry "parse_data" (segment1+0x558)
    [ via: "parse_packet" (segment1+0x774) ]
    (Widening Equivalence Domains)
20: segment1+0x584 [ via: "parse_data" (segment1+0x558)
      <- "parse_packet" (segment1+0x774) ]
    (Widening Equivalence Domains)
21: Return "parse_data" (segment1+0x558)
    [ via: "parse_packet" (segment1+0x774) ]
    (Widening Equivalence Domains)
22: segment1+0x774 [ via: "parse_packet" (segment1+0x590) ]
    (Widening Equivalence Domains)
23: Return "parse_packet" (segment1+0x590)
    (Widening Equivalence Domains)
24: segment1+0x5c0 [ via: "parse_packet" (segment1+0x590) ]
    (Re-checking Equivalence Domains)
25: Return "parse_packet" (segment1+0x590)
    (Widening Equivalence Domains)
26: segment1+0x600 [ via: "parse_packet" (segment1+0x590) ]
    (Re-checking Equivalence Domains)
27: segment1+0x658 [ via: "parse_packet" (segment1+0x590) ]
    (Re-checking Equivalence Domains)
28: segment1+0x698 [ via: "parse_packet" (segment1+0x590) ]
    (Re-checking Equivalence Domains)
Continue verification?
```

```
0: Finish and view final result
1: Restart from entry point
2: Handle pending refinements
?> 0
```

The user selects `0` to finish and view the final result.

The PATE analysis tree can be navigated by the user with `top` to move to the top of the tree, numbers to navigate "into" nodes, `up` to move "up" a node, and `ls` to re-display the nodes available at a current level.

For example, to inspect the analysis results in the running example, the user may provide input as follows to view the equivalence condition(s):

```
  ...
  30: Final Result
  > 30
  Final Result
  0: Assumed Equivalence Conditions
  1:   segment1+0x6ec [ via: "parse_packet" (segment1+0x590) ]
  2:   segment1+0x71c [ via: "parse_packet" (segment1+0x590) ]
  3: Binaries are conditionally, observably equivalent
  4: Toplevel Result
  > 2
  segment1+0x71c [ via: "parse_packet" (segment1+0x590) ]
  0: ------
  1:   original
  2:   patched
  3: (Predicate) let -- segment1+0x734.. in not (and v67142 (not v67145))
  4: With condition assumed
  5:   Event Trace: segment1+0x71c .. segment1+0x754
  6: With negation assumed
  7:   Event Trace: segment1+0x71c .. segment1+0x754
  8: Simplified Predicate
  > 3
  let -- segment1+0x734
    v67145 = eq 0x0:[8] (select (select cInitMemBytes@66997:a 0) 0x11044:[32])
    -- segment1+0x734
    v67142 = eq 0x80:[8] (select (select cInitMemBytes@67121:a 0) 0x11045:[32])
 in not (and v67142 (not v67145))
```

This is a formal representation of an equivalence condition calculated by PATE, which is the result of the user choosing to "avoid" the detected observable difference. The condition is a predicate over the (symbolic) original and patched program states starting at a given address/context (in this case, starting at `0x71c` within the `parse_packet` function). In the binary ninja plugin (see 5) this predicate is post-processed to be more human-readable.

Continuing on, the user may view, for example, an example trace showing where the equivalence condition above does not hold:

```
> up
...
4: With condition assumed
5:   Event Trace: segment1+0x71c .. segment1+0x754
6: With negation assumed
7:   Event Trace: segment1+0x71c .. segment1+0x754
...
> 7
== Initial Original Registers ==
pc <- 0x71c:[32]
r0 <- 0x0:[32]
r1 <- 0x1:[32]
r11 <- Stack Slot: -8
r13 <- Stack Slot: -24
r14 <- 0x71c:[32]
r2 <- 0x1:[32]
...
== Original sequence ==
...
(segment1+0x724)
  Read 0x11044:[32] -> 0x80:[8]
  r0 <- 0x80:[32]
(segment1+0x728) Write Stack Slot: -21 <- 0x80:[8]
...
(segment1+0x734)
  Read 0x11045:[32] -> 0x80:[8]
  r0 <- 0x80:[32]
...
(segment1+0x748)
Read Stack Slot: -21 -> 0x80:[8]
r1 <- 0x80:[32]
...
== Initial Patched Registers ==
pc <- 0x71c:[32]
r0 <- 0x0:[32]
r1 <- 0x1:[32]
r11 <- Stack Slot: -8
r13 <- Stack Slot: -24
r14 <- 0x71c:[32]
r2 <- 0x1:[32]
...
== Patched sequence ==
...
(segment1+0x724)
  Read 0x11044:[32] -> 0x80:[8]
  r0 <- 0x80:[32]
```

```
(segment1+0x728) Write Stack Slot: -21 <- 0x80:[8]
...
(segment1+0x734)
  Read 0x11045:[32] -> 0x80:[8]
  r0 <- 0x80:[32]
...
(segment1+0x744) Write Stack Slot: -21 <- 0x0:[8]
(segment1+0x748)
  Read Stack Slot: -21 -> 0x0:[8]
  r1 <- 0x0:[32]
...
```

Here PATE has constructed a trace where memory addresses `0x11044` and `0x11045` are both assumed to contain `0x80` (negating the equivalence condition), and shown that this leads to a different value assigned to stack slot `-21`. This slot is provided as input to a subsequent call to `printf` and is therefore considered an observable difference.

See the following subsections for details about how to interpret and interact with the terminal user interface.

4.2 Status Indicators

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.

4.3 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

4.4 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 The PATE Binary Ninja Plugin

The PATE Binary Ninja plugin enables user to invoke and interact with PATE directly within the Binary Ninja reverse engineering platform.

The PATE plugin requires a commercially-licensed Binary Ninja installation. We have tested PATE with Binary Ninja stable version 4.0.5336.

5.1 Installation

First, install the PATE Docker container as described in Section 2.

Second, copy (or create a symlink from) the `pate_binja/` directory from the PATE source repo to your local Binary Ninja `plugins/` directory. Typically these are found in:

macOS `$HOME/Library/Application Support/Binary Ninja/plugins/`
Linux `$HOME/.binaryninja/plugins/`

The Binary Ninja plugin requires a relatively recent version of Python (3.10 or newer) and requires the `grpcio` package. If these are not present on the system, we recommend creating a Python `venv` or similar on your host, with something like:

```
python -m venv /path/to/new/virtual/environment
source /path/to/new/virtual/environment/bin/activate
pip install grpcio
```

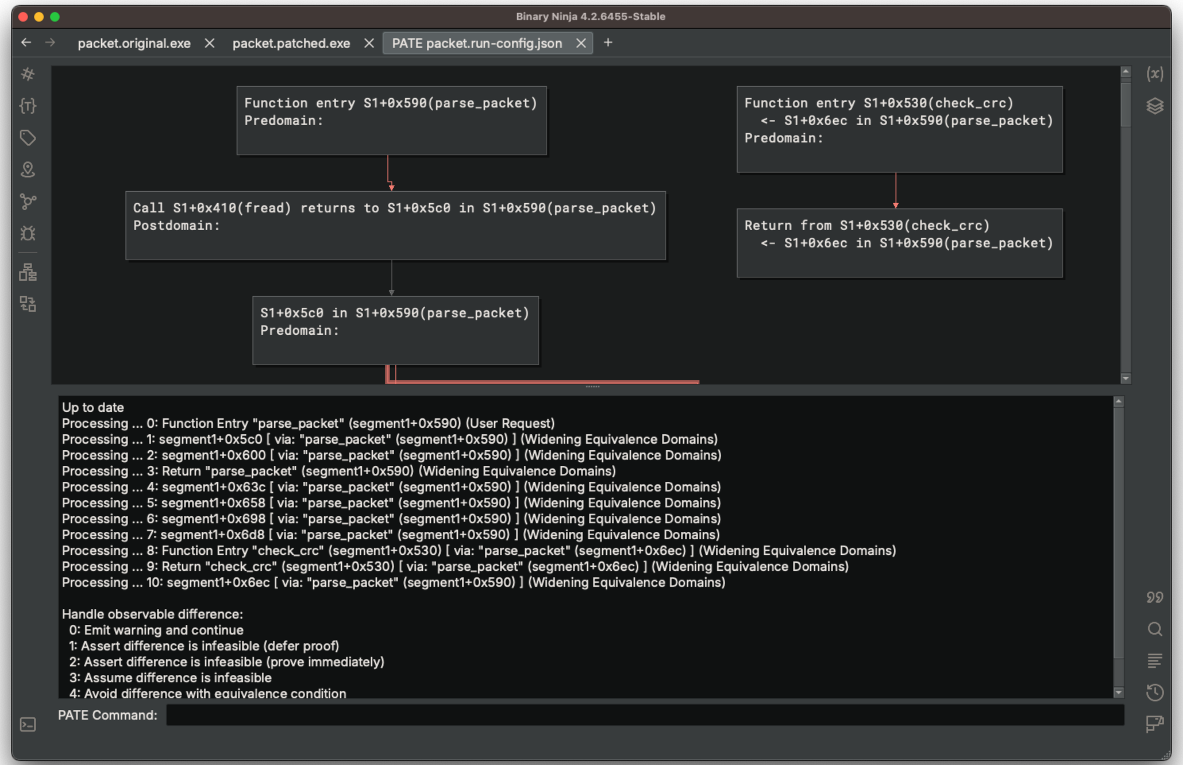
and then modifying Binary Ninja Python settings appropriately in the Binary Ninja Preferences list to point to this new environment. Specifically, check the settings for:

- Python Path Override
- Python Interpreter
- Python Virtual Environment Site-Packages

Once these steps have been completed, restart Binary Ninja. If the plugin is correctly installed and initialized, then the “Plugins” menu will now have a “Pate...” menu option.

5.2 Usage

In Section 4 we examined the `packet.exe` example using the PATE terminal UI. In this section, we describe analyzing the same example using the PATE Binary Ninja plugin.



```
Function entry S1+0x590(parse_packet)
Predomain:

Call S1+0x410(fread) returns to S1+0x5c0 in S1+0x590(parse_packet)
Postdomain:

S1+0x5c0 in S1+0x590(parse_packet)
Predomain:

Function entry S1+0x530(check_crc)
  <- S1+0x6ec in S1+0x590(parse_packet)
Predomain:

Return from S1+0x530(check_crc)
  <- S1+0x6ec in S1+0x590(parse_packet)
```

```
Up to date
Processing ... 0: Function Entry "parse_packet" (segment1+0x590) (User Request)
Processing ... 1: segment1+0x5c0 [ via: "parse_packet" (segment1+0x590) ] (Widening Equivalence Domains)
Processing ... 2: segment1+0x600 [ via: "parse_packet" (segment1+0x590) ] (Widening Equivalence Domains)
Processing ... 3: Return "parse_packet" (segment1+0x590) (Widening Equivalence Domains)
Processing ... 4: segment1+0x63c [ via: "parse_packet" (segment1+0x590) ] (Widening Equivalence Domains)
Processing ... 5: segment1+0x658 [ via: "parse_packet" (segment1+0x590) ] (Widening Equivalence Domains)
Processing ... 6: segment1+0x698 [ via: "parse_packet" (segment1+0x590) ] (Widening Equivalence Domains)
Processing ... 7: segment1+0x6d8 [ via: "parse_packet" (segment1+0x590) ] (Widening Equivalence Domains)
Processing ... 8: Function Entry "check_crc" (segment1+0x530) [ via: "parse_packet" (segment1+0x6ec) ] (Widening Equivalence Domains)
Processing ... 9: Return "check_crc" (segment1+0x530) [ via: "parse_packet" (segment1+0x6ec) ] (Widening Equivalence Domains)
Processing ... 10: segment1+0x6ec [ via: "parse_packet" (segment1+0x590) ] (Widening Equivalence Domains)

Handle observable difference:
  0: Emit warning and continue
  1: Assert difference is infeasible (defer proof)
  2: Assert difference is infeasible (prove immediately)
  3: Assume difference is infeasible
  4: Avoid difference with equivalence condition
```

Figure 1: *The PATE Binary Ninja Plugin. The top region is the PATE analysis overview graph. The bottom panel is used by the operator to interact with the PATE verifier.*

Once installed, invoke the PATE plugin from the Binary Ninja "Plugins → Pate..." menu option.

The PATE plugin opens a window to select a *run configuration* file in JSON format. This file must end in the suffix `.run-config.json`. The run configuration must contain a key-value map with the following keys:

`original` (absolute or relative) file path to the original binary
`patched` (absolute or relative) file path to the patched binary
`args` a list additional arguments to pass to PATE

For example, the file `tests/integration/packet/packet.run-config.json` contains:

```
{
  "original": "exe/packet.original.exe",
  "patched": "exe/packet.patched.exe",
  "args": [
    "-s parse_packet"
    ]
  }
```

After selecting the run configuration file, the PATE plugin will open three Binary Ninja tabs: one for each of the original and patched binaries, and a third PATE *analysis* tab.

As shown in Figure 1, the PATE analysis tab is composed of two primary regions. The bottom portion is an interactive text view that corresponds to the terminal user interface. The user interacts with this region through the text input field at the bottom of the window. In the `packet.exe` example, the user enters the same commands described in Section 4 through the `Finish and view final result` step, at which point results will be rendered in the interactive Binary Ninja view.

The top portion of the PATE analysis tab is a graph view showing the current state of the PATE analysis. When PATE analysis is complete (via interaction in the bottom portion), the PATE graph shows rectangles for each slice of the program analyzed by PATE. Default-colored rectangles represent a pair of programs slices that were able to matched up between the original and patched binary. Green rectangles represent slices present only in the original program, if any. Blue rectangles represent slices present only in the patched program, if any. Right-click on a rectangle for options to jump directly to the relevant program location in the appropriate tab for the corresponding binary.

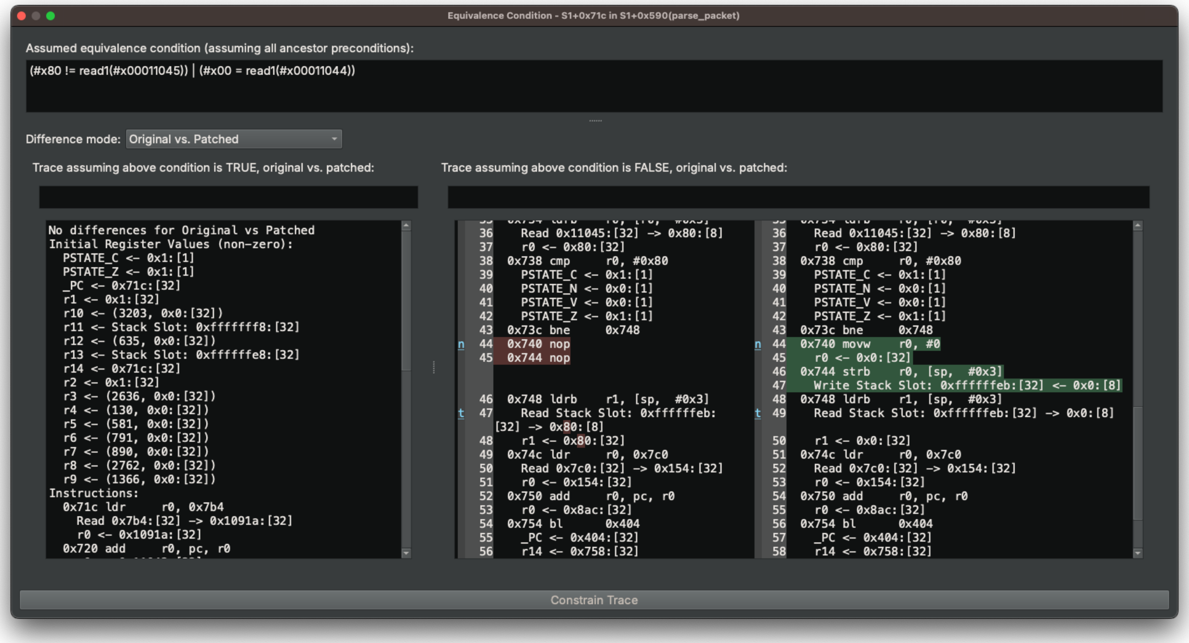
A red rectangle represents the "active" region, where PATE has found an equivalence condition. Right click and select "Show Equivalence Condition" to open the Equivalence Condition window. Please see Figure 2 for an example screenshot of the equivalence condition window in the PATE Binary Ninja plugin. An equivalence condition window includes:

- the expression describing the conditions under which programs behave equivalently
- a generated (concretized) trace through the program(s) showing an example where the equivalence condition is met
- a generated (concretized) trace through the program(s) showing an example where the equivalence condition is *not* met

Right-clicking on a rectangle in the PATE analysis graph and selecting "Show Instruction Trees" will open a new window showing basic blocks from the original program on the left and corresponding basic blocks from the patched program on the right. At the bottom will be a linearized representation of the instruction trees in the original and patched program, with colorized diff view. Instructions conditionally reachable through conditional control flow edges (if any) are prefixed by `+` or `-` to differentiate instructions in distinct branches. See Figure 6 for an example of the instruction tree view, with MCAD integration (described below).

5.3 Trace Constraints

The concretized traces shown in the equivalence condition window (see Figure 2) show a pair of *possible* traces through the program slices where the equivalence condition is either met or not met. In general there may be multiple paths through the program that may satisfy (or contradict) the equivalence condition. To observe alternative concrete traces, the final equivalence condition may be provided *trace constraints*, which restrict the values that PATE may choose when generating a concrete trace. When viewing the final equivalence condition, after the verifier has finished and the user has requested the final result, the "Constrain Trace" button at the bottom becomes enabled,


Equivalence Condition - S1+0x71c in S1+0x590(parse_packet)

Assumed equivalence condition (assuming all ancestor preconditions):

```
(#x80 != read1(#x00011045)) | (#x00 = read1(#x00011044))
```

Difference mode: Original vs. Patched

Trace assuming above condition is TRUE, original vs. patched:

```
No differences for Original vs Patched
Initial Register Values (non-zero):
  PSTATE_C <- 0x1:[1]
  PSTATE_Z <- 0x1:[1]
  _PC <- 0x71c:[32]
  r1 <- 0x1:[32]
  r10 <- (3203, 0x0:[32])
  r11 <- Stack Slot: 0xfffffff8:[32]
  r12 <- (635, 0x0:[32])
  r13 <- Stack Slot: 0xffffffe8:[32]
  r14 <- 0x71c:[32]
  r2 <- 0x1:[32]
  r3 <- (2636, 0x0:[32])
  r4 <- (130, 0x0:[32])
  r5 <- (581, 0x0:[32])
  r6 <- (791, 0x0:[32])
  r7 <- (890, 0x0:[32])
  r8 <- (2762, 0x0:[32])
  r9 <- (1366, 0x0:[32])
Instructions:
  0x71c ldr      r0, 0x7b4
    Read 0x7b4:[32] -> 0x1091a:[32]
    r0 <- 0x1091a:[32]
  0x720 add      r0, pc, r0
```

Trace assuming above condition is FALSE, original vs. patched:

```
    Read 0x11045:[32] -> 0x80:[8]
    r0 <- 0x80:[32]
  0x738 cmp      r0, #0x80
    PSTATE_C <- 0x1:[1]
    PSTATE_N <- 0x0:[1]
    PSTATE_V <- 0x0:[1]
    PSTATE_Z <- 0x1:[1]
  0x73c bne      0x748
n 0x740 nop
  0x744 nop


  0x748 ldrb     r1, [sp,  #0x3]
t   Read Stack Slot: 0xffffffeb:
[32] -> 0x80:[8]
    r1 <- 0x80:[32]
  0x74c ldr      r0, 0x7c0
    Read 0x7c0:[32] -> 0x154:[32]
    r0 <- 0x154:[32]
  0x750 add      r0, pc, r0
    r0 <- 0x8ac:[32]
  0x754 bl       0x404
    _PC <- 0x404:[32]
    r14 <- 0x758:[32]
```

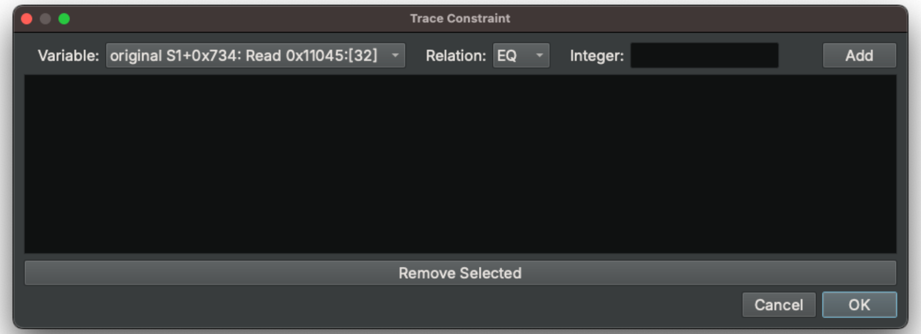
```
    Read 0x11045:[32] -> 0x80:[8]
    r0 <- 0x80:[32]
  0x738 cmp      r0, #0x80
    PSTATE_C <- 0x1:[1]
    PSTATE_N <- 0x0:[1]
    PSTATE_V <- 0x0:[1]
    PSTATE_Z <- 0x1:[1]
  0x73c bne      0x748
n 0x740 movw     r0, #0
    r0 <- 0x0:[32]
  0x744 strb     r0, [sp,  #0x3]
    Write Stack Slot: 0xffffffeb:[32] <- 0x0:[8]
  0x748 ldrb     r1, [sp,  #0x3]
t   Read Stack Slot: 0xffffffeb:[32] -> 0x0:[8]

    r1 <- 0x0:[32]
  0x74c ldr      r0, 0x7c0
    Read 0x7c0:[32] -> 0x154:[32]
    r0 <- 0x154:[32]
  0x750 add      r0, pc, r0
    r0 <- 0x8ac:[32]
  0x754 bl       0x404
    _PC <- 0x404:[32]
    r14 <- 0x758:[32]
```

Figure 2: *The PATE Binary Ninja plugin equivalence condition window.*

allowing constraints to be added to the given trace.

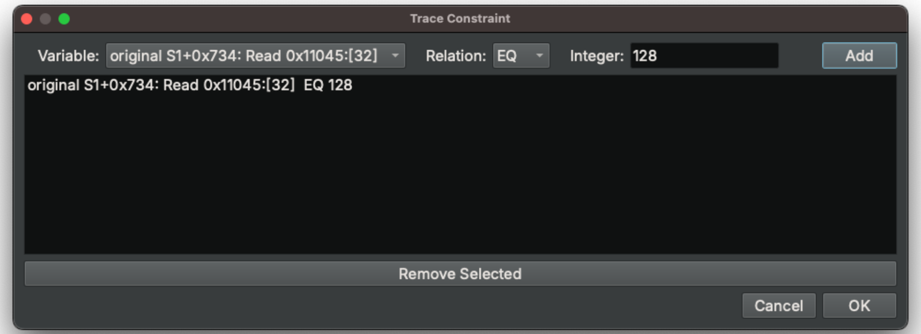
When "Constrain Trace" is clicked, the "Trace Constraint" window (see Figure 3) appears. The "Variable" dropdown is populated with the memory reads from both the original and patched programs. Notably each read is prefixed with the instruction address that the read occurred at, indicating that the constraint applies to the content of memory at that specific program point. The user then selects a relation and enters an integer value to compare the memory content against. Clicking "Add" will then add the specified constraint to the list below (see Figure 4).


Trace Constraint

Variable: original S1+0x734: Read 0x11045:[32] Relation: EQ Integer:

Figure 3: *Empty trace constraint dialogue.*

Multiple constraints may be added at this point, or removed via the "Remove Constraint" button. Once the desired set of constraints has been provided, clicking "OK" will close the dialogue

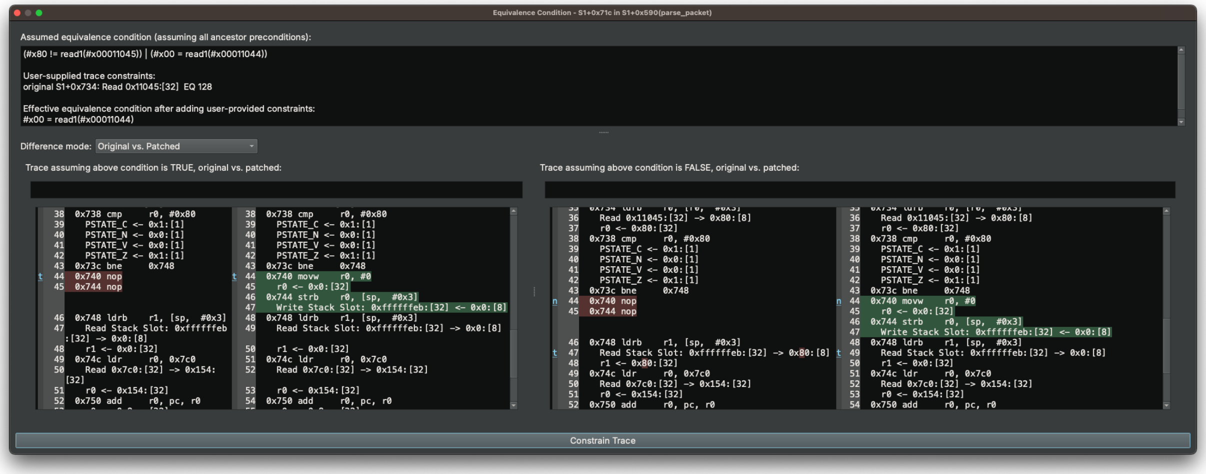


Variable: original S1+0x734: Read 0x11045:[32] Relation: EQ Integer: 128

```
original S1+0x734: Read 0x11045:[32]  EQ 128
```

Figure 4: *Trace constraint dialogue with added constraint.*

and present an updated equivalence condition window containing the now-constrained traces (see Figure 5). The equivalence condition text area lists the original equivalence condition, the user-supplied trace constraints, and the effective equivalence condition as a result of incorporating the user-provided constraints.



```
Assumed equivalence condition (assuming all ancestor preconditions):
(#x80 != read1(#x00011045)) | (#x00 = read1(#x00011044))

User-supplied trace constraints:
original S1+0x734: Read 0x11045:[32]  EQ 128

Effective equivalence condition after adding user-provided constraints:
#x00 = read1(#x00011044)
```

Difference mode: Original vs. Patched

Trace assuming above condition is TRUE, original vs. patched:

```
0x738 cmp     r0, #0x80
  PSTATE_C <- 0x1:[1]
  PSTATE_N <- 0x0:[1]
  PSTATE_V <- 0x0:[1]
  PSTATE_Z <- 0x1:[1]
0x73c bne     0x748
0x740 nop
0x744 nop


0x748 ldrb    r1, [sp,  #0x3]
  Read Stack Slot: 0xffffffeb
:[32] -> 0x0:[8]
  r1 <- 0x0:[32]
0x74c ldr     r0, 0x7c0
  Read 0x7c0:[32] -> 0x154:
[32]
  r0 <- 0x154:[32]
0x750 add     r0, pc, r0
```

```
0x738 cmp     r0, #0x80
  PSTATE_C <- 0x1:[1]
  PSTATE_N <- 0x0:[1]
  PSTATE_V <- 0x0:[1]
  PSTATE_Z <- 0x1:[1]
0x73c bne     0x748
0x740 movw    r0, #0
  r0 <- 0x0:[32]
0x744 strb    r0, [sp,  #0x3]
  Write Stack Slot: 0xffffffeb:[32] <- 0x0:[8]
0x748 ldrb    r1, [sp,  #0x3]
  Read Stack Slot: 0xffffffeb:[32] -> 0x0:[8]

  r1 <- 0x0:[32]
0x74c ldr     r0, 0x7c0
  Read 0x7c0:[32] -> 0x154:[32]

  r0 <- 0x154:[32]
0x750 add     r0, pc, r0
```

Trace assuming above condition is FALSE, original vs. patched:

```
  Read 0x11045:[32] -> 0x80:[8]
  r0 <- 0x80:[32]
0x738 cmp     r0, #0x80
  PSTATE_C <- 0x1:[1]
  PSTATE_N <- 0x0:[1]
  PSTATE_V <- 0x0:[1]
  PSTATE_Z <- 0x1:[1]
0x73c bne     0x748
0x740 nop
0x744 nop


0x748 ldrb    r1, [sp,  #0x3]
  Read Stack Slot: 0xffffffeb:[32] -> 0x80:[8]
  r1 <- 0x80:[32]
0x74c ldr     r0, 0x7c0
  Read 0x7c0:[32] -> 0x154:[32]
  r0 <- 0x154:[32]
0x750 add     r0, pc, r0
```

```
  Read 0x11045:[32] -> 0x80:[8]
  r0 <- 0x80:[32]
0x738 cmp     r0, #0x80
  PSTATE_C <- 0x1:[1]
  PSTATE_N <- 0x0:[1]
  PSTATE_V <- 0x0:[1]
  PSTATE_Z <- 0x1:[1]
0x73c bne     0x748
0x740 movw    r0, #0
  r0 <- 0x0:[32]
0x744 strb    r0, [sp,  #0x3]
  Write Stack Slot: 0xffffffeb:[32] <- 0x0:[8]
0x748 ldrb    r1, [sp,  #0x3]
  Read Stack Slot: 0xffffffeb:[32] -> 0x0:[8]
  r1 <- 0x0:[32]
0x74c ldr     r0, 0x7c0
  Read 0x7c0:[32] -> 0x154:[32]
  r0 <- 0x154:[32]
0x750 add     r0, pc, r0
```

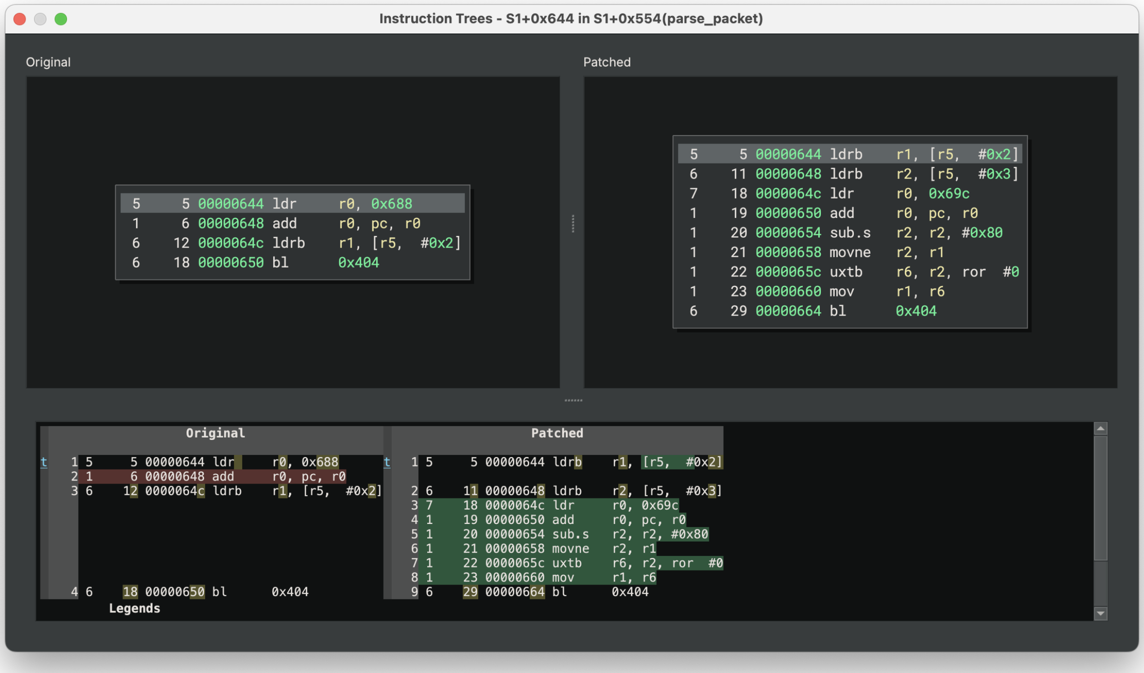
Figure 5: *Expanded equivalence condition window with constrained traces.*

5.4 Replays

Executing a run configuration file with the PATE plugin will produce a *replay* in a file named `lastrun.replay`, which can be preserved by renaming to another filename on disk. These files cache the user input and verifier output, enabling the user to load a previous interaction with PATE without having to re-execute the verifier analysis. If the user loads a replay rather than a run configuration, the process is mostly the same, except the command input field is already populated with the recorded user input, and the user just hits "enter" to proceed to the next step. Editing the input commands will have no effect, as the responses are recorded in the replay file. That is, the user cannot deviate from the pre-recorded responses from the PATE verifier to perform any analysis other than the one recorded.

5.5 MCAD Integration

MCAD[5] is a performance analysis tool that performs static prediction of instruction timings for sequences of instructions such as those identified by PATE. The details of the MCAD system is outside the scope of this user guide, but if the MCAD Docker container is available, the PATE Binary Ninja plugin will show MCAD-predicted cycle counts next to each instruction in the Instruction Trees view, as shown in Figure 6. Use the PATE plugin preference option “MCAD Docker image name” to specify the name of the MCAD Docker container on the host in order to enable MCAD integration in the PATE plugin.



```
5     5 00000644 ldr      r0, 0x688
1     6 00000648 add      r0, pc, r0
6    12 0000064c ldrb     r1, [r5,  #0x2]
6    18 00000650 bl       0x404
```

```
5     5 00000644 ldrb     r1, [r5,  #0x2]
6    11 00000648 ldrb     r2, [r5,  #0x3]
7    18 0000064c ldr      r0, 0x69c
1    19 00000650 add      r0, pc, r0
1    20 00000654 sub.s    r2, r2, #0x80
1    21 00000658 movne    r2, r1
1    22 0000065c uxtb     r6, r2, ror  #0
1    23 00000660 mov      r1, r6
6    29 00000664 bl       0x404
```

```
t  1 5     5 00000644 ldr      r0, 0x688      t  1 5     5 00000644 ldrb     r1, [r5,  #0x2]
   2 1     6 00000648 add      r0, pc, r0
   3 6    12 0000064c ldrb     r1, [r5,  #0x2]   2 6    11 00000648 ldrb     r2, [r5,  #0x3]
                                                  3 7    18 0000064c ldr      r0, 0x69c
                                                  4 1    19 00000650 add      r0, pc, r0
                                                  5 1    20 00000654 sub.s    r2, r2, #0x80
                                                  6 1    21 00000658 movne    r2, r1
                                                  7 1    22 0000065c uxtb     r6, r2, ror  #0
                                                  8 1    23 00000660 mov      r1, r6
   4 6    18 00000650 bl       0x404              9 6    29 00000664 bl       0x404
```

Figure 6: *PATE Binary Ninja MCAD integration showing predicted cycle counts (per instruction, cumulative) for instructions in original (left) and patched (right) basic blocks.*

[5]https://github.com/securesystemslab/LLVM-MCA-Daemon

6 Acknowledgements

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.