`pate` User Manual (Version 0.5)

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

January 25, 2022

The `pate`[1] tool 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, `pate` provides detailed explanations that precisely characterize the difference in behavior between the two binaries. `pate` 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 `pate` 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 (prior to testing), and
- reduce the time required to develop safe binary patches.

Contents

1 Introduction **2**
1.1 Quick Start . . . 2
1.2 Building `pate` . . . 2
1.2.1 Docker . . . 2
1.2.2 Building from Source . . . 3
1.3 Improving Analysis Results . . . 4

2 Options **4**
2.1 Block Info Format . . . 5
2.2 Interactive UI . . . 6
2.3 Debugging Options . . . 6

3 Proving Equivalence **7**
3.1 Code Alignment . . . 7

4 Inlining Calls **8**

[1] The name `pate` comes from the project name Patches Assured up to Trace Equivalence

1 Introduction

The `pate` verifier is a static relation verifier for binaries that builds assurance that micropatches have not had any adverse effects. `pate` is a static relational verifier that attempts to prove that two binaries have the same observable behaviors. When it cannot, `pate` provides detailed explanations that precisely characterize the difference in behavior between the two binaries. `pate` 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 `pate` to ensure that the effects are intended.

Note that while `pate` 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, `pate` can give the user either:

- a *differential summary* (Section 3) that explains the conditions under which the two binaries exhibit different behaviors, or
- a *memory differential* (Section 4) that explains how memory values will differ at binary locations with different behaviors.

1.1 Quick Start

The `pate` verifier is a command line tool, but provides an optional web UI. A typical use looks like:

```
BINDIR=/path/to/binaries
pate --original $BINDIR/original.exe \
     --patched $BINDIR/patched.exe \
     --interactive --port 5000 \
     --proof-summary-json report.json
```

This command runs the verifier on `original.exe` and `patched.exe` and generates a JSON report describing any differences between the two. It also starts up a local webserver on port 5000 that enables the user to interactive examine the proof that the verifier constructs. Sections 3 to 4 explain the exact properties verified by `pate`, along with the semantics of the summaries they report.

1.2 Building `pate`

The `pate` verifier can be built through either Docker or from source; this section includes instructions for both.

1.2.1 Docker

Using a Pre-Built Docker Image If you have a pre-built Docker image, it can be loaded as follows:

```
# Assuming that the distributed Docker image file is pate.tar
docker load < pate.tar
```

Building the Docker Image To build the Docker image from scratch, use the following commands:

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

Using the Docker Image To run `pate` from the Docker image, use a command like the following:

```
docker run --rm -it -v /path/to/binaries:/binaries pate \
  --original /binaries/original.exe \
  --patched /binaries/patched.exe \
  --proof-summary-json /binaries/report.json
```

While the Docker image contains all of the support files necessary to run the verifier, extra arguments are required to make the local filesystem (and the binaries to verify) accessible to the running Docker container. In this example, we can use the `-v` option to map a directory on the local filesystem into the Docker container, which both enables the verifier to read the binaries and persist a JSON report outside of the container.

1.2.2 Building from Source

The `pate` verifier is written in the Haskell programming language. Building it requires the GHC compiler[2] (versions 8.8 through 9.0) and the Cabal[3] build system, both of which can be installed via ghcup[4].

```
ghcup install ghc 8.10.7
ghcup install cabal 3.6.2.0
export PATH=$HOME/.ghcup/bin:$PATH

git clone git@github.com:GaloisInc/pate.git
cd pate
git submodule update --init
ln -s cabal.project.dist cabal.project
cabal configure -w ghc-8.10.7
cabal build pkg:pate
```

Note that running the verifier will require the yices SMT solver[5] to be in the user's `PATH`. The Docker image contains the necessary solvers to run `pate`.

[2] https://www.haskell.org/ghc/
[3] https://www.haskell.org/cabal/
[4] https://www.haskell.org/ghcup/
[5] https://yices.csl.sri.com/

1.3 Improving Analysis Results

If DWARF information is available in either the original or patched binary, `pate` 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`[6] 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, `pate` has a more difficult time aligning the code in the original and patched binaries, which can lead to confusing or unhelpful diagnostics.

2 Options

The full list of options supported by `pate` is documented in this section.

-h,–help
: Show the help text

-o,–original EXE
: The path to the original binary on disk

-p,–patched EXE
: The path to the patched binary on disk

-b,–blockinfo FILENAME
: A file containing additional information for the verifier (see Section 2.1 for details)

–proof-summary-json FILE
: A file to save important proof results to in JSON format

-V,–verbosity ARG
: The verbosity of logging output (default: Info, alternative: Debug)

–log-file FILE
: A file to save debug logs to (the verbosity is controlled by the `--verbosity` flag)

-i,–interactive
: Start a web server providing an interactive view of results

-p,–port PORT
: The port to run the interactive visualizer web server on (default: 5000)

–original-source FILE
: The source file for the original program (for visualization in the interactive UI)

–patched-source FILE
: The source file for the patched program (for visualization in the interactive

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

UI)

-m,–ignoremain
Don't add the main entry points to the set of function equivalence checks

–solver ARG
The SMT solver to use to solve verification conditions. One of CVC4, Yices, or (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)

–no-dwarf-hints
Do not use DWARF debug information in the original and patched binaries to improve diagnostics

–save-macaw-cfgs DIR
Save intermediate code discovery results (as printed CFGs) to the provided directory (for debugging purposes)

–solver-interaction-file FILE
Save interactions with the SMT solver during symbolic execution to this file (for debugging purposes)

2.1 Block Info Format

The `--blockinfo=FILE` argument enables users to provide extra information to the verifier to improve its results. The file it requires has the following format:

```
PatchData { patchPairs = [(origEntryAddr, patchedEntryAddr)]
          , ignoredPointers = ([(origDataAddr, origDataSize)]
                               , [(patchedDataAddr, patchedDataSize)]
                               )
          , equatedFunctions = [(inlineOrigAddr, inlinePatchedAddr)]
          }
```

where the values on the right-hand sides of the assignments are concrete values to be filled in. Each field has the following meaning:

`patchPairs`
A list of pairs of addresses. Each pair of addresses specifies function addresses in the original and patched binaries, respectively, that the user asserts to be equivalent. The first entry is used as the program entry point if the user has specified the `--ignoremain` option. Otherwise, the analysis will start from the program entry points declared in the ELF header.

`ignoredPointers`
Together with the `equatedFunctions` field, this enables the inline callee functionality described in Section 4. The elements of the pair are lists of memory regions whose contents should be ignored while verifying equivalence of two programs (specifically under the scope of a pair of equated functions). The first element of the pair are ignored memory regions for the original binary, while the second element are ignored memory regions

for the patched binary. Precisely, each location specification is of the form *(dataAddr, dataSize)*, where the *dataAddr* is the address of a pointer that points to *dataSize* bytes of memory. This is an important subtlety: it really refers to the address of a pointer, and the buffer referred to is the buffer at the address pointed to. This enables the inline callee feature to refer to dynamically allocated memory that does not have a fixed global memory location.

`equatedFunctions`
: A list of pairs of function addresses that are intended to be equivalent for the purposes of the inline callee feature.

2.2 Interactive UI

`pate` provides an interactive UI for monitoring proof progress and for exploring proof results. To enable it, pass the `--interactive` flag (optionally using the `--port` flag to control the port that the web server will listen on). Visiting `localhost` on the specified port will bring up the interactive UI, which serves two purposes:

1. Streaming analysis events into a virtual console in the browser window
2. Taking snapshots of the automatically generated equivalence proof, enabling exploration of the proof goals and the results of attempting to prove them

The interactive view enables visualization of the proof graph, which mirrors the call graph of the program. Clicking on any node in the proof graph displays detailed information about the corresponding code, as well as the proof obligations at that location. If source code is provided with the `--original-source` and `--patched-source` options, function source code will be shown for each node in the proof graph.

2.3 Debugging Options

There are a number of options useful for debugging the `pate` verifier. The `--save-macaw-cfgs` option enables users to save intermediate artifacts from code discovery for later inspection. This is most useful for identifying code discovery failures, unsupported instructions, incorrect semantics, or control flow within functions.

The `--solver-interaction-file` option enables users to save the interactions between the `pate` verifier and the SMT solver. This can be useful for diagnosing slow queries or SMT solver bugs (e.g., invalid models). The recorded interaction session can be replayed by simply running the SMT solver directly on it. It also records SMT solver responses (e.g., models from counterexamples).

3 Proving Equivalence

The primary feature of `pate` is statically verifying that two binaries, and original binary and a patched binary, have the same observable behaviors. It does this by constructing a compositional proof of equivalence between the two binaries. The proof aligns the loop-free and call-free code sequences in the binaries and computes a sufficient set of pre- and post-conditions for each pair of code sequences to exhibit identical behavior. Each of the post-conditions in the proof are *proof obligations*; if all of the proof obligations are satisfied, the two programs have the same observable behaviors. `pate` then traverses the proof to automatically discharge all of the proof obligations using an SMT solver.

In practice, we do not expect the two programs to be equivalent, as the program was patched for a reason (i.e., to change its behavior). If the two binaries can exhibit different behaviors, `pate` identifies the part of the program where the difference can manifest, analyzes the counterexample provided by the SMT solver that exhibits the different behavior, and generates a *differential summary* that explains the conditions under which the difference in behavior is observable. A differential summary is a logical formula in terms of function arguments and global variables that precisely describes the conditions under which the two binaries exhibit different behavior. Formally, if one *assumes* that the differential summary is true, at the beginning of a function, the patched program will exhibit different behavior than the original program.

Listing 1: An example differential summary

```
let
v376575 = select cInitMem@372468:a 0
v423090 = select v376575 (bvSum R0:bv 0x4:[32])
v423130 = select v376575 (bvSum R1:bv 0x4:[32])
v423111 = select v376575 (bvSum R0:bv 0x2:[32])
v423104 = select v376575 (bvSum R0:bv 0x3:[32])
v423125 = bvSum (bvZext 16 v423111) (bvShl (bvZext 16 v423104) 0x8:[16])
v432396 = bvSlt v423125 0x0:[16]
v423131 = eq 0x0:[8] v423130
v432397 = and (not (eq 0x0:[2] (bvSelect 2 2 v423090))) v423131 v432396
 in not v432397
```

An example differential summary is shown in Listing 1. This formula describes a relationship between a number of values read from pointers passed via arguments in `R0` and `R1`. The differential summaries are available for inspection both in the interactive proof explorer UI (see Section 2 for details) and in the JSON reports generated by the verifier.

3.1 Code Alignment

To construct the equivalence proof between an original and patched binary, `pate` breaks each binary up into loop-free and call-free sequences of instructions. It then *aligns* the corresponding code sequences in the original and patched

binary. If this alignment is unsuccessful, the proof will ultimately fail, and will further more fail to produce useful differential diagnostics. The verifier is robust to many of the methods for inserting micropatches (e.g., redirecting execution to a patch location and jumping back). Trouble typically arises in the presence of larger code motion that can arise from recompilation, rather than micropatching. Additionally, any movement of mutable *data* (e.g., shifting the contents of the `.data` or `.bss` sections) is likely to cause catastrophic failures, as the verifier will be unable to find a meaningful comparison between the original and patched binaries. If a patch requires larger scale changes that do not fit the micropatching model, `pate` provides an additional verification mode that can deal with some types of larger changes, which is described in Section 4.

4 Inlining Calls

If the compositional equivalence proof method described in Section 3 cannot provide a useful diagnostic of a patch, `pate` supports *symbolic call inlining* as an alternative proof method that can be used to isolate larger binary differences and reason about them in a structured way. Examples of scenarios where this feature are useful include:

- If the behavior of the program has changed *unconditionally* (i.e., it always has different behavior in the patched binary); in this case, there is no sensible differential summary (besides `true`)
- As an example of that, consider replacing one function with another that has different behavior, but "acceptable" behavioral differences
- If precise reasoning about looping (but symbolically terminating) code is required
- If substantial code changes make one-to-one code alignment between the original and patched binaries impossible

The result of using this feature is a summary of the difference in memory post-states after the two equated functions return to their callers. This information enables analysts using `pate` to determine whether or not their patches have unintended affects on the memory of the program that may impact functionality.

To enable this feature for corresponding regions of code in the original and patched binaries (available at the function level), see the `--blockinfo` option and the file format described in Section 2.1. Inlining calls enables `pate` to reason about a larger scope within the input programs, which can improve precision compared to the compositional verification approach. It also enables more precise reasoning about loops. The tradeoff for using this feature is that it can be less scalable, depending on the nature of the code being inlined. We refer to it as *call inlining* because an entire sub-tree of the call graph is "inlined" into a single node within the compositional equivalence proof.

As a general guideline, when using this feature, the functions that should be "equated" using the `equatedFunctions` configuration option should be parents (formally, graph dominators) of all of the code that has been patched. Compared to the compositional proof, the inlined equated callees are:

1. Initialized with identical symbolic states
2. Each symbolically executed independently

After symbolic execution, `pate` attempts to prove that every byte in their memory post-states always has the same value. Any bytes that cannot be proven to be equal are summarized in different ranges of addresses and reported to the user. As an optimization, `pate` only attempts to prove that addresses that have been written to during symbolic execution are equal.