CBAT: A Comparative Binary Analysis Tool

Chloe Fortuna, JT Paasch
STR
{chloe.fortuna,j.paasch}@str.us

Sam Lasser, Philip Zucker
Draper
{slasser,pzucker}@draper.com

Chris Casinghino
Jane Street
ccasinghino@janestreet.com

Cody Roux
Amazon Web Services
codyroux@amazon.com

***Abstract*—Modifying a binary program without access to the original source code is an error-prone task. In many cases, the modified binary must be tested or otherwise validated to ensure that the change had its intended effect and no others—a process that can be labor-intensive. This paper presents CBAT, an automated tool for verifying the correctness of binary transformations. CBAT's approach to this task is based on a differential program analysis that checks a relative correctness property over the original and modified versions of a function. CBAT applies this analysis to the binary domain by implementing it as an extension to the BAP binary analysis toolkit. We highlight several features of CBAT that contribute to the tool's efficiency and to the interpretability of its output. We evaluate CBAT's performance by using the tool to verify modifications to three collections of functions taken from real-world binaries.**

I. Introduction

Software engineers often need to modify deployed code to fix bugs, remove bloat, and add new security features. In some cases, they must perform these modifications without access to the original source code—for example, because the software depends on closed-source libraries, or because it includes legacy components for which source code is no longer available. This process of *late-stage software customization* is difficult and error-prone [6], and in high-assurance settings, the modified software must undergo recertification to ensure that the changes did not introduce new bugs [16]. Late-stage software customization, and certification of the results, could be made more efficient and robust with the help of automated tools for *comparative analysis of binary programs*: i.e., tools for verifying that a modified binary has the same semantics as the original one, modulo any changes that were introduced intentionally.

In this work, we present the Comparative Binary Analysis Tool (CBAT), a formal methods-based framework for verifying the correctness of binary code transformations. CBAT's novel approach to this problem relies on *differential program analysis* techniques that are built atop an open-source *binary analysis toolkit*. The goal of differential program analysis is to check *relative* correctness properties over pairs of programs: in the case of CBAT, over the original and modified versions of a program. We have adapted the approach to the binary domain by implementing CBAT as an extension to the Binary Analysis Platform (BAP) [4], a toolkit for binary analysis. BAP enables CBAT to analyze binaries for a range of architectures by lifting them to a platform-independent intermediate language.

CBAT's primary component is the `wp` differential program analyzer. The `wp` tool operates on the original and modified versions of a function. It also takes in a relative correctness specification expressed in first-order logic. The tool performs a *weakest precondition* (WP) analysis [8] over these inputs to compute a logical formula that is true if and only if the specification holds. The `wp` tool then dispatches this condition to a Satisfiability Modulo Theories (SMT) solver. The solver either returns a *countermodel*—an initial machine configuration that causes the two functions to behave in a way that violates the correctness specification—or it reports that it can find no such countermodel.

This work makes the following contributions:

1) We present the `wp` differential program analyzer, a tool that supports safe binary patching by checking the correctness of a modified binary function relative to the original version.
2) We highlight three features of `wp` and its ecosystem that support efficient program analyses and interpretable analysis results: (a) an *interactive debugger* for exploring why a given countermodel leads to a correctness property violation; (b) *integration with multiple SMT solvers* for fast assertion checking and decoding of solver-produced countermodels; and (c) *function summaries* for tractable analysis of functions that call other functions.
3) We evaluate `wp`'s performance on three data sets of functions taken from real-world binaries.

CBAT is an open source project. The development is available online [1]; it includes a tutorial and interactive exercises.

This paper is organized as follows. In §II, we review background information on weakest precondition and differential program analyses. In §III, we introduce `wp` via an example interaction with the tool. In §IV, we describe several key features of CBAT's user interface and underlying implementation. Finally, we evaluate the performance of `wp` in §V and survey related work in §VI.

II. Background

A. Weakest Precondition Analysis

Weakest precondition (WP) analysis was introduced by Dijkstra [8]. The analysis checks assertions of the following

This work is sponsored by ONR/NAWC Contract N6833518C0107. Its content does not necessarily reflect the position or policy of the US Government and no official endorsement should be inferred.

Workshop on Binary Analysis Research (BAR) 2024
1 March 2024, San Diego, CA, USA
ISBN 979-8-9894372-0-7
https://dx.doi.org/10.14722/bar.2024.23009
www.ndss-symposium.org

```
enum msg_ty {
  NAV    = 42,
  LOG    = 59,
  DEPLOY = 64,
  ...
};

void process_message(enum msg_ty t, msg m) {
  switch (t) {
  case NAV:
    adjust_heading(m.data);
  case LOG:
    log_current_status(m.status);
    break;
  case DEPLOY:
    deploy_payload();
    break;
  ...
  }
}
```

(a) Original version of `process_message`. A `break` statement is deliberately omitted from the `NAV` case so that each course alteration is immediately followed by a logging event.

```
enum msg_ty {
  NAV    = 42,
  // LOG is no longer defined
  DEPLOY = 64,
  ...
};

void process_message(enum msg_ty t, msg m) {
  switch (t) {
  case NAV:
    adjust_heading(m.data);

  // LOG case removed

  case DEPLOY:
    deploy_payload();
    break;
  ...
  }
}
```

(b) Modified version of `process_message` in which logging has been removed. The change introduces unintended behavior: each `NAV` event is immediately followed by a `DEPLOY` event.

Fig. 1: C source code for two versions of the `process_message` function that processes a command signal and invokes the corresponding task handler. A `msg` (the second argument to `process_message`) is a struct with `status` and `data` fields; we omit its definition for brevity because it is the same in both programs.

form: if program c begins execution in a state that satisfies precondition P, and c terminates, then the final state necessarily satisfies postcondition Q. We write such an assertion as $\{P\}\, c\, \{Q\}$. WP checks an assertion by working backwards through each program statement to compute the *weakest precondition*: the predicate P' that satisfies $\{P'\}\, c\, \{Q\}$ while placing the fewest possible restrictions on the initial state. The original assertion $\{P\}\, c\, \{Q\}$ holds iff P implies P'.

B. Differential Analysis via Program Composition

CBAT relies on the key insight that to check a relative correctness property over two programs, one can compose the programs and then check a corresponding property over the composed program [14]. For example, consider the following two versions of a program fragment:

(p_1) `int x = 3; if (y) {x = 5;}`
(p_2) `int x = 5; if (y) {x = 7;}`

One might wish to verify that for any `y` value, p_2 computes a greater `x` value than p_1. To verify this fact, CBAT renames variables so that the two fragments do not interfere with each other, and it builds the following composed program:

($p_1 \circ p_2$)
```
int x1 = 3; if (y1) {x1 = 5;}
int x2 = 5; if (y2) {x2 = 7;}
```

The tool then checks the following assertion about this composed program: if `y1` $=$ `y2` at the start of execution, then `x1` $<$ `x2` when the program ends. If this assertion holds, it implies the relative correctness property stated above.

III. CBAT's Differential Analysis by Example

We introduce the `wp` tool with an example involving a binary patch for a simplified spacecraft control program.

A. A Binary Patch-Induced Bug

The spacecraft controller's `process_message` function (Figure 1a) processes command signals and dispatches tasks to the appropriate handlers: a `NAV` signal alters the craft's course, `LOG` writes the craft's current status to a log, `DEPLOY` deploys a research instrument, etc. In our example, the programmer has decided that each course alteration should be logged and has therefore omitted a `break` statement from the `NAV` case of the `switch` statement. As a result, execution falls through from the `NAV` case to the `LOG` case, and each navigation event is followed by a logging event.

Suppose that maintainers of this code perform late-stage software customization to remove the logging feature, producing the function in Figure 1b. (We show the modified function in source format for readability, but the patch is applied to an x86-64 binary.) Now the program has a bug: execution falls through from the `NAV` case to the `DEPLOY` case, and each navigation event will inadvertently deploy the research instrument.

B. Invoking and Controlling WP Analysis

The program's maintainers or an external certifier may wish to validate the change by checking various relative properties of the two program versions. For example, because the change removes functionality, a reasonable property is that the modified program should call a subset of the functions that the original program calls. To check this property, the certifier can invoke `wp` as shown in the example terminal session in Figure 2. The `wp` tool is implemented as a plugin to BAP and thus invoked via the command `bap wp`. The final two arguments to the command, `orig_prog` and `mod_prog`, are the original and modified versions of the example binary, respectively. The remaining arguments have the following meanings:

```
user@host:~/cbat-example$ bap wp \
> --func=process_message \
> --show=paths \
> --compare-func-calls \
> orig_prog mod_prog
Evaluating precondition.
Checking precondition with Z3.

Property falsified. Counterexample found.

Model:
        ...
        RSP    |->   0x000000003f8000c0
        RDI    |->   0x000000000000002a
        ...
```

Fig. 2: Example invocation of `wp` on binaries `orig_prog` and `mod_prog`. The tool checks that each run of the modified `process_message` function makes a subset of the function calls in the corresponding run of the original version. The tool determines that the binaries violate this relative correctness property, and it outputs a countermodel (shown in partial form) in which register RDI holds 0x2a = 42. When the two versions of `process_message` begin execution from this configuration (i.e., when the first argument is a `NAV` value), they will behave in a way that violates the specification.

- The `--func=process_message` argument tells `wp` to compare the two binaries' respective versions of the `process_message` function.
- The `--show` flag controls the output that `wp` produces. In this case, the `paths` argument to `--show` tells `wp` that if the tool determines that the correctness property in question is false, it should output a visual representation of the function execution paths that violate the property. We give examples of such paths in Section III-C.
- The `--compare-func-calls` flag specifies that every function call in a run of the modified binary should also occur in a corresponding run of the original binary.

C. Finding and Interpreting a Countermodel

The `wp` tool is able to determine that the original and modified versions of `process_message` violate the `--compare-func-calls` specification. In such a case, the tool produces a *countermodel* demonstrating why the specification is false. A countermodel is a mapping from machine registers and memory locations to their contents; when the two functions begin executing from this configuration, their behavior violates the assertion.

A portion of the `wp`-produced countermodel for our example appears in Figure 2; the countermodel maps register RDI to the value 0x2a (42). RDI holds the first argument to a called function according to the x86-64 calling conventions, and 42 is the integer representation of the `msg_ty` value `NAV`. Therefore, this countermodel reveals that when the modified version of `process_message` receives a `NAV` value as its first argument, it calls a function that the original version does not (namely, `deploy_payload`).

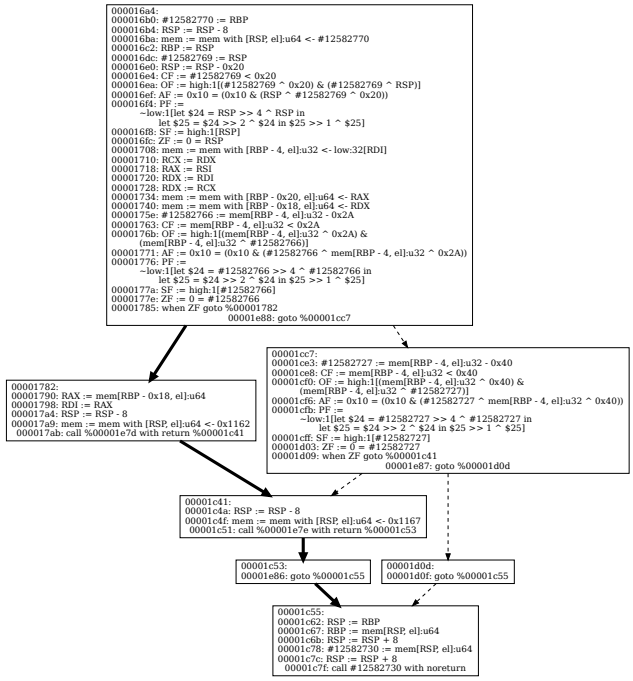

Fig. 3: Example `wp`-produced control flow graph in which highlighted edges represent the function's countermodel-induced execution path. Basic blocks are sequences of statements in BAP Intermediate Language (BIL).

The mechanism by which a `wp`-produced countermodel leads to a property violation may not be immediately obvious. To help the user understand the relationship between countermodel and violation, `wp` can produce a visual trace of the countermodel-induced execution path through each function. When `wp` is invoked with the `--show=paths` argument and it detects a countermodel, the tool outputs a control flow graph (CFG) for each analyzed function. In each CFG, the execution path induced by the countermodel is highlighted, enabling the user to trace the behavior that violates the specification. Figure 3 shows the highlighted CFG that `wp` produces for the modified version of `process_message`.

When a static depiction of control flow provides inadequate insight into an analyzed function's behavior, the user can trace the function's execution dynamically with `bildb`, CBAT's interactive debugger; see Section IV-A for details.

D. Aside: Predefined and Custom Specifications

The `--compare-func-calls` property used in our example is one of several predefined relative correctness specifications that `wp` makes available. Additional examples of predefined specifications are as follows:

- `--trip-asserts` ensures that execution never reaches calls to `__assert_fail()`.
- `--compare-post-reg-values=r1,r2,...` asserts that when the functions terminate, registers `r1,r2,...` hold the same values in both final states.

- `--check-invalid-derefs` assumes that all memory dereferences in the original binary are valid, and it checks that all additional dereferences in the modified binary are valid (i.e., they involve locations on the stack or heap).

The user can also provide custom specifications to `wp` via `--precond` and `--postcond` flags that take arguments in SMT-LIB syntax [2], a standard input language for SMT solvers. For example, one could invoke `wp` with the following argument to `--postcond`:

```
(assert (= RAX_orig RAX_mod))
```

In this specification, `RAX_orig` and `RAX_mod` refer to x86-64 return register RAX in the original and modified functions, respectively. The specification asserts that RAX should hold the same value at the end of both functions' executions; in other words, the functions should return the same value. (Note that one could also express this property with a predefined specification by calling `wp` with the argument `--compare-post-reg-values=RAX`).

A limitation of CBAT's support for custom specifications is that the user must be familiar with both SMT-LIB and the target architecture's ABI. A question for future work is how to enable non-expert users to write expressive specifications.

IV. CBAT Features and Implementation Highlights

In this section, we highlight three key features of CBAT: an *interactive debugger* for tracing countermodel-induced function behavior; *integration with multiple SMT solvers* for optimizing solver performance; and *function summaries* for modeling the behavior of called functions.

A. Interactive Debugger

The BAP Instruction Language Debugger, or `bildb`, is a CBAT component that helps users understand why a `wp`-produced countermodel violates the correctness specification provided to `wp`. The debugger executes a lifted binary function in an interactive manner; the user can step forward and backward through the function's execution, load values into machine registers and memory addresses, and set breakpoints.

When `wp` produces a countermodel, the user can invoke the debugger with the `wp`-generated machine configuration as the starting point to observe how that configuration produces behavior that violates the specification. To begin this process, the user directs `wp` to save the countermodel in a `bildb`-compatible format via the `--bildb-output` flag. For example, adding the argument `--bildb-output=init.yml` to the `wp` invocation in Figure 2 stores the countermodel in the YAML file `init.yml`. The user then launches `bildb` with this initial configuration by running the command shown in Figure 4. As the debugger's excerpted output shows, register RDI is mapped to the value 0x2A—the `NAV` argument that leads to a property violation—at the start of execution.

The `bildb` tool relies on BAP's engine for *microexecution* of code fragments in a virtual environment without user-provided input. Microexecution does not require the host machine to be able to execute the binary under analysis; this property makes the tool a useful alternative to traditional debuggers like `gdb`.

```
user@host:~/cbat-example$ bap \
> --pass=run \
> --bildb-debug \
> --bildb-init=init.yml \
> orig_prog
BIL Debugger
Starting up...

Architecture
Type: x86_64
Address size: 64
Registers:
R10    R11    ...

Initialized state
Variables:
...
RDI    : 0x2A
...

Entering subroutine: [%00000fc2] _start
...
Entering block %000003f5
...
>>> (h for help)
```

Fig. 4: Example invocation of the `bildb` debugger on binary `orig_prog`, starting from the `wp`-produced countermodel stored in file `init.yml`. In the initial machine configuration (shown in partial form), register RDI holds 0x2A, the integer representation of a `NAV` value. The user enter commands at the prompt to step to different instructions or breakpoints, display or modify various components of machine state, etc.

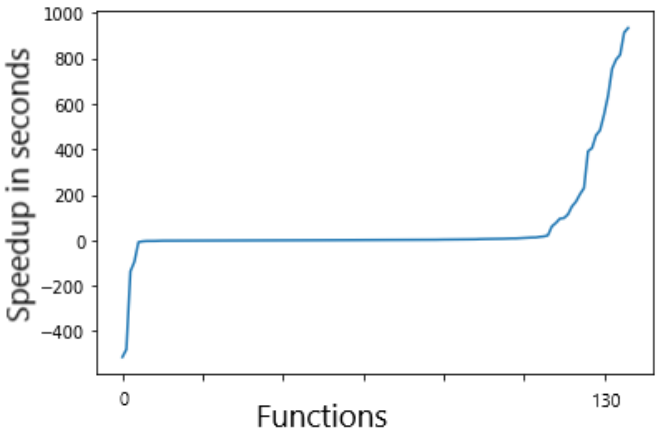

Fig. 5: The Boolector SMT solver's per-problem speedup over Z3 on 130 verification problems. Problems were sampled randomly from the Lift/Recompile `wp` benchmark described in Section V. While the two solvers are on par for most instances, Boolector is faster on average. Boolector is at least 20s slower on four functions, and at least 20s faster on 21 functions.

B. SMT Solver Integration

The `wp` tool relies on an SMT solver to check whether the weakest precondition formula that it computes is satisfiable. The tool's default solver, Z3 [7], offers both excellent performance and facilities for interpreting the results it produces. However, other solvers sometimes perform better on certain classes of formulas. For example, we observed that the Boolector SMT solver [17] achieved better average running time than Z3 on one of our benchmarks (see Figure 5). Here, we describe a novel technique for integrating `wp` with alternative solvers in a way that can boost performance while retaining the benefits of Z3's interpretation facilities.

The `wp` tool interacts with a solver at up to two points during an analysis: (1) when `wp` converts its internal representation of a precondition to the solver's input format; and (2) when `wp` expresses a solver-produced countermodel in terms of the input programs' variables and control flow to help the user understand why the assertion is false. Passing a precondition to an alternative solver is straightforward, because `wp` produces a formula in the solver-agnostic SMT-LIB format. However, when interpreting a countermodel, `wp` relies on a Z3-specific library for examining and querying countermodels. To obtain the performance benefits of an alternative solver A and the interpretability benefits of Z3, we run A in concert with Z3 as follows:

1) We first check the formula with solver A.
2) If A finds a countermodel, it outputs this model in a solver-agnostic SMT-LIB format. This output contains instantiations of the formula's logical variables that falsify the formula.
3) We then run Z3 on the original formula, adding assertions that set the logical variables to the values found by A. This step enables Z3 to find the same countermodel instantly.
4) We use CBAT's existing Z3 integration mechanism to translate the countermodel into information about the input programs to display to the user.

In cases where A is faster than Z3, this approach improves performance while still enabling `wp` to use Z3's tools for interpreting the countermodel.

C. Function Summaries

The `wp` tool's analysis is *intraprocedural*: it analyzes two versions of a single function. The tool's main approach to handling a function call within the function being analyzed involves using a *function summary*: a high-level specification of the called function's behavior. A function summary $\{P\}\, f\, \{Q\}$ asserts that if the machine's state (memory and registers) satisfies precondition P when function f is called, then the state satisfies postcondition Q after f returns. Function summaries enable `wp` to incorporate the effects of a called function f into its analysis without examining the body of f, which would make the analysis intractable (since f itself might call other functions).

CBAT includes both a library of nine predefined function summaries and facilities for defining custom summaries. The user can select which predefined summaries to use via a `--fun-specs` flag, which takes a list of summary names as an argument. When `wp` analysis reaches a called function with no custom summary, the tool uses the first summary from this list that applies to the function. Possible arguments to `--fun-specs` include the following:

- `chaos-caller-saved` asserts that a function can load arbitrary values into caller-saved registers.
- `verifier-error` asserts that a function triggers an error. This summary can be used to determine whether the function is reachable.
- `verifier-nondet` models the behavior of nondeterministic memory management functions such as `malloc`. This summary asserts that a function can return an arbitrary pointer.

The user can also define function summaries that express more fine-grained or application-specific properties. The user provides custom summaries to `wp` in SMT-LIB syntax via a `--user-func-specs` flag. For example, suppose that an x86_64 program includes a `div` function, which computes the integer division of its first argument by its second. To ensure that calls to `div` never result in an attempt to divide by zero, the user can invoke `wp` with the following argument to `--user-func-specs`:

```
div, (assert (not (= RSI (_ bv0 64)))),
     (assert true)
```

In this function summary, the precondition states that the second argument to `div` is not equal to zero (x86_64 register RSI holds the second argument to a called function, and `(_ bv0 64)` is a 64-bit representation of zero). The trivial postcondition `(assert true)` places no restrictions on program state after a `div` call.

V. Performance Evaluation for CBAT's WP Analysis

We evaluate the `wp` tool's performance by using it to verify modifications to three collections of binary functions. In each experiment, `wp` runs on the original and modified versions of a binary, checking that each modified function produces the same callee-saved register values as its analogue in the original binary. Such a check has four possible outcomes: `wp` can verify the property, find a countermodel, return "unknown" (indicating that the underlying solver could not verify or falsify the property), or time out (we use a one-kilosecond per-function time limit).

Our benchmarks are based on the following transformations and binaries:

- The **RetroWrite** benchmark is based on an instrumentation pass [9] that prepares a binary for integration with the American Fuzzy Lop (AFL) fuzzing framework [20]. This pass was used to modify the Linux `base64` utility.
- The **Lift/Recompile** benchmark is based on a binary lifting and recompilation framework [10], [18] that was used to modify a version of GNU `tar`.
- The **Embrittle** benchmark is based on a tool [11] that performs various binary transformations, including shuffling blocks and renaming program elements. These transformations were applied to a second version of GNU `tar`.

Benchmark	# Functions	Verified	Countermodels	Unknowns	Timeouts
RetroWrite	145	123 (84.8%)	2 (1.4%)	0 (0.0%)	20 (13.8%)
Lift/Recompile	1171	804 (68.7%)	59 (5.0%)	7 (0.6%)	301 (25.7%)
Embrittle	1511	1143 (75.6%)	84 (5.6%)	62 (4.1%)	222 (14.7%)

Fig. 6: The total number of functions in each benchmark, and the results of using `wp` to compare the pre- and post-transformation versions of each function. For each input, `wp` can verify the transformation, find a countermodel suggesting that the transformation violates the correctness property, return an "unknown" value, or time out.

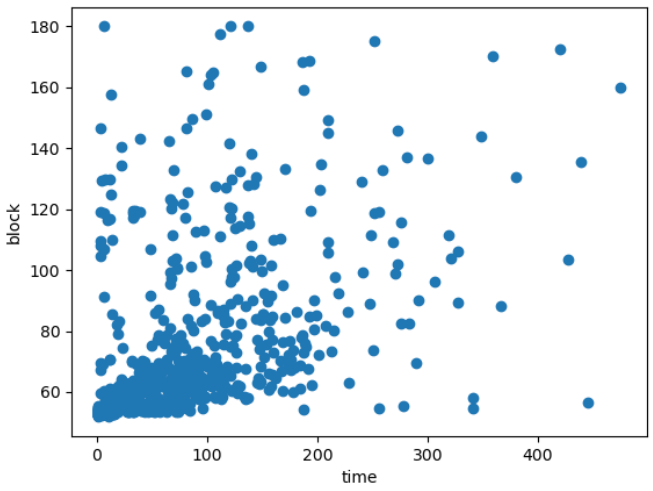

Fig. 7: Function size (measured in # basic blocks) vs. `wp` total runtime for a sample of problem instances from the Lift/Recompile benchmark.

Figure 6 lists the total number of functions in each benchmark, as well as the number of functions that result in each outcome. The `wp` tool verifies or finds a countermodel for 86%, 74%, and 81% of inputs in the three benchmarks, respectively. Further inspection is required to determine whether each countermodel that `wp` produces in these experiments represents a property violation or a false positive. In practice, checking a complex property of a large program with complete fidelity to the underlying machine is intractable, so tools like CBAT must simplify the machine model in a way that leads to some false positives.

It is difficult to count false negatives—property violations that the tool fails to detect—because doing so would require a source of ground truth about which modified functions are incorrect relative to the original ones. We merely note that for performance reasons, `wp` makes a number of standard simplifying assumptions that can produce false negatives. A classic example is loop unrolling: `wp` replaces each program loop with a finite, user-configurable number of instances of the loop body. As a result, `wp` can fail to detect violations that only occur after a number of loop iterations that is greater than the unrolling factor.

Interestingly, we observed that while the time required to compute a weakest precondition is closely correlated with function size, SMT solving time is *not* closely correlated with function size. Therefore, neither is `wp`'s total runtime, because solving time dominates total runtime. Figure 7 illustrates the apparent absence of a strong relationship between function size and total runtime for several hundred problem instances from the Lift/Recompile benchmark.

VI. Related Work

A. Differential Analysis

Lahiri et al. [14] describe a technique for *differential assertion checking* (DAC), or proving relative correctness between two program versions. The authors observe that performing this task is more efficient than proving absolute correctness of the later version. They implement the technique within the SymDiff semantic differencing tool [13]. Earlier work on reducing false positive rates of concurrent program verifiers [12] describes an approach to DAC for bounded programs. DAC involves composing two programs so that correctness can be expressed in terms of the *product program*; Barthe et al. [3] describe product programs in a general setting. CBAT adapts the framework of product program-based DAC to the binary domain.

B. Binary Analysis Toolkits

Binary analysis toolkits lift binaries to an intermediate representation (IR) and facilitate program analyses at the IR level. CBAT is implemented as a collection of plugins for the Binary Analysis Platform (BAP) [4], an open source toolkit. Another widely used toolkit is angr [19]. Submissions based on BAP and angr placed first and third, respectively, at the DARPA Cyber Grand Challenge [5], a competition among automated cyber defense systems. More recently, the Ghidra framework [15] developed and open-sourced by the NSA has emerged as a popular tool for binary analysis.

Acknowledgment

We thank Ivan Gotovchits for maintaining BAP and for extending it with new features in support of the CBAT project's research goals.

References

[1] "GitHub repository for the CBAT development," 2023. [Online]. Available: https://github.com/draperlaboratory/cbat_tools

[2] C. Barrett, A. Stump, C. Tinelli *et al.*, "The SMT-LIB Standard: Version 2.0," in *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories*, vol. 13, 2010, p. 14.

[3] G. Barthe, J. M. Crespo, and C. Kunz, "Relational Verification Using Product Programs," in *17th International Symposium on Formal Methods*, 2011. [Online]. Available: https://doi.org/10.1007/978-3-642-21437-0_17

[4] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A Binary Analysis Platform," in *Computer Aided Verification*, 2011. [Online]. Available: https://doi.org/10.1007/978-3-642-22110-1_37

[5] DARPA, "Cyber Grand Challenge Homepage," Available: https://www.cybergrandchallenge.com, 2016.

[6] ——, "Broad Agency Announcement: Assured Micropatching (AMP)," 2019. [Online]. Available: https://sam.gov/api/prod/opps/v3/opportunities/resources/files/5b8ab57b7efcbc34e18a46d1cbbe6fe2/download?&status=archived&token=

[7] L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2008. [Online]. Available: https://doi.org/10.1007/978-3-540-78800-3_24

[8] E. W. Dijkstra, "Guarded Commands, Nondeterminacy and Formal Derivation of Programs," *Communications of the ACM*, vol. 18, no. 8, pp. 453–457, 1975. [Online]. Available: https://doi.org/10.1145/360933.360975

[9] 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

[10] A. Flores-Montoya and E. M. Schulte, "Datalog Disassembly," in *USENIX Security Symposium*, 2020. [Online]. Available: https://dl.acm.org/doi/10.5555/3489212.3489273

[11] Galois, Inc., "Software Fault Encouragement project summary," 2017. [Online]. Available: https://galois.com/project/brittle/

[12] S. Joshi, S. K. Lahiri, and A. Lal, "Underspecified Harnesses and Interleaved Bugs," in *Principles of Programming Languages*, 2012. [Online]. Available: https://doi.org/10.1145/2103656.2103662

[13] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo, "SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs," in *Computer Aided Verification*, 2012. [Online]. Available: https://doi.org/10.1007/978-3-642-31424-7_54

[14] S. K. Lahiri, K. L. McMillan, R. Sharma, and C. Hawblitzel, "Differential Assertion Checking," in *9th Joint Meeting on Foundations of Software Engineering*, 2013. [Online]. Available: https://doi.org/10.1145/2491411.2491452

[15] National Security Agency, "Ghidra," 2019. [Online]. Available: https://ghidra-sre.org/

[16] S. Nelson, B. Fischer, E. Denney, J. Schumann, J. Richardson, and P. Oh, "Product-Oriented Software Certification Process for Software Synthesis," Tech. Rep. NASA/CR-2004-212819, 2004. [Online]. Available: https://ntrs.nasa.gov/api/citations/20040034049/downloads/20040034049.pdf

[17] A. Niemetz, M. Preiner, and A. Biere, "Boolector 2.0," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 9, no. 1, pp. 53–58, 2014.

[18] E. M. Schulte, J. Dorn, A. Flores-Montoya, A. Ballman, and T. Johnson, "GTIRB: Intermediate Representation for Binaries," *Computing Research Repository*, 2019. [Online]. Available: http://arxiv.org/abs/1907.02859

[19] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Krügel, 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

[20] M. Zalewski, "american fuzzy lop," 2017. [Online]. Available: https://lcamtuf.coredump.cx/afl/