Finding Real Bugs in Big Programs with Incorrectness Logic

QUANG LOC LE, University College London and Meta, UK
AZALEA RAAD, Imperial College London and Meta, UK
JULES VILLARD, Meta, UK
JOSH BERDINE, Meta, UK
DEREK DREYER, MPI-SWS, Germany
PETER W. O'HEARN, Meta and University College London, UK

Incorrectness Logic (IL) has recently been advanced as a logical theory for compositionally proving the *presence* of bugs—dual to Hoare Logic, which is used to compositionally prove their *absence*. Though IL was motivated in large part by the aim of providing a logical foundation for bug-catching program analyses, it has remained an open question: is IL useful only retrospectively (to explain *existing* analyses), or can it actually be useful in developing *new* analyses which can catch real bugs in big programs?

In this work, we develop Pulse-X, a new, automatic program analysis for catching memory errors, based on ISL, a recent synthesis of IL and separation logic. Using Pulse-X, we have found 15 new real bugs in OpenSSL, which we have reported to OpenSSL maintainers and have since been fixed. In order not to be overwhelmed with potential but false error reports, we develop a compositional bug-reporting criterion based on a distinction between latent and manifest errors, which references the under-approximate ISL abstractions computed by Pulse-X, and we investigate the *fix rate* resulting from application of this criterion. Finally, to probe the potential practicality of our bug-finding method, we conduct a comparison to Infer, a widely used analyzer which has proven useful in industrial engineering practice.

CCS Concepts: • **Software and its engineering** → **Correctness**; • **Theory of computation** → **Automated reasoning**; **Program analysis**; **Pre- and post-conditions**; **Program verification**.

Additional Key Words and Phrases: bug catching, incorrectness proving, incorrectness logic, compositionality

ACM Reference Format:
Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. 2022. Finding Real Bugs in Big Programs with Incorrectness Logic. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 81 (April 2022), 27 pages. https://doi.org/10.1145/3527325

1 INTRODUCTION

Incorrectness Logic (IL) [O'Hearn 2019] has recently been advanced as a logical foundation for *compositionally* proving the *presence* of bugs. In this paper, we show that this foundation is not merely of theoretical interest, but moreover has the potential to be practically useful as a basis for developing new static analyses. Building on prior work on biabduction and Incorrectness Separation Logic (ISL, [Raad et al. 2020]), we define a new compositional bug-catching analyser, Pulse-X, which we have applied to find *real* bugs in *big* programs (notably, 15 new and confirmed bugs in OpenSSL).

Authors' addresses: Quang Loc Le, University College London and Meta, UK, loc.le@ucl.ac.uk; Azalea Raad, Imperial College London and Meta, UK, azalea.raad@imperial.ac.uk; Jules Villard, Meta, UK, jul@fb.com; Josh Berdine, Meta, UK, josh@berdine.net; Derek Dreyer, MPI-SWS, Saarland Informatics Campus, Germany, dreyer@mpi-sws.org; Peter W. O'Hearn, Meta and University College London, UK, p.ohearn@ucl.ac.uk.

2475-1421/2022/4-ART81
https://doi.org/10.1145/3527325

Before we present Pulse-X in more detail, however, let us first review the key characteristics of IL that make it such a promising foundation for bug-catching.

Compositionality. A compositional analysis is one in which each part of the program is analysed *locally*—i.e., independently of the global program context in which it is used. The analysis result of a composite program is then computed directly from the analysis results of its constituent parts [Calcagno et al. 2011]. In recent CACM articles, industrial developers of static analyses at Facebook [Distefano et al. 2019] and Google [Sadowski et al. 2018] described several concrete advantages of compositional static analyses in practical deployments, the overarching one being that compositionality enables analyses to be usefully integrated into *code review*. According to the Facebook article, industrial codebases evolve at such a high velocity that, for a bug-finding analysis to be deployable as part of a code review process, it must execute in *minutes* and not hours; thus, any analysis that requires traversal of an entire large program would likely be too slow. By contrast, since compositional analyses work locally on code snippets rather than whole programs, they can be run quickly (in minutes) on *diffs* (snippets of code that change as part of a pull request). This agility makes it possible for compositional analyses to be run automatically as part of continuous integration (CI) systems and provide timely information to developers performing code review.

IL shows promise as a foundation for compositional static analyses in large part because, like Hoare Logic, it too is compositional: it enables one to prove compositional specifications of a procedure's behavior given only conditional information about the context in which that procedure is called. But this is only half of the equation: the other half is *under-approximate reasoning*.

Under-approximation. One of the most successful compositional static analyses is the Infer tool [Facebook 2021], in deployment at Facebook, Amazon, Microsoft, and other companies. Infer's use of *biabduction* provides a powerful technique for inferring compositional specifications for memory-manipulating procedures [Calcagno et al. 2011]. However, Infer was based on classic separation logic (SL) [O'Hearn et al. 2001], a logic of *over-approximation*—meaning that SL specifications prove the *absence* of bugs. As O'Hearn [2019] notes, this has led to a fundamental mismatch between the logical foundation of Infer and its practical deployment: Infer infers specifications that establish the absence of bugs, yet it is used as a bug-catcher (to report the presence of bugs). To bridge this gap and decide when to report bugs, Infer employs *heuristics* to determine when its failure to prove the absence of bugs gives sufficient cause to merit a bug report. These heuristics have been refined over time based on empirical evaluation of developer feedback on the bug reports produced by the tool, but they lack a clear formal foundation for why or whether they work.

In contrast to Hoare/separation logic, IL is *under-approximate*—meaning that its specifications establish the *presence* of bugs (under conditions about the environment). In particular, ISL triples, which have the form $[\mathit{presumption}]\ f\ [\epsilon : \mathit{result}]$, are interpreted as the dual of Hoare triples: they assert that any final state satisfying the result condition *result* is reachable (with exit condition ϵ) by executing f starting in some initial state satisfying *presumption*. Here, ϵ can be either *ok* (indicating normal termination) or *er* (indicating erroneous termination). In the latter case, we refer to the triple as an *error spec* because it establishes that f has a provably erroneous execution.

Thus, the very natural high-level idea underlying our analyser Pulse-X is to start from Infer but swap out its separation logic foundation with one based on Incorrectness Separation Logic (ISL, [Raad et al. 2020])—that is, to marry the effective biabductive inference algorithm of Infer together with the under-approximate foundation of ISL, seeing as the latter makes for a much better fit with the goal of bug-catching.

The Problem of Compositional Bug Reporting. A key problem remains, however, namely: when should Pulse-X report bugs? Pulse-X will generate a variety of ISL error specs for any given

procedure (under *er*), some of which may correspond to real bugs, but many of which will not because they only correspond to real bugs *under some conditions about the calling context.*

A common example would be a function such as `deref(x) { *x = 10; }` which simply dereferences its argument: if it is passed a null pointer, it will crash. Correspondingly, one of the ISL triples that Pulse-X would infer for `deref(x)` is the error spec `[x = NULL] deref(x) [er: x = NULL]`. In most situations, however, we would want to consider `deref` harmless—and not report any bug for `deref` *per se*—since the function may simply have an implicit precondition that it is always called with a non-null argument. Of course, if `deref` were actually called with null, we would then want to report a bug. Without knowing the calling contexts of such a function, though, how can we say that our error spec for `deref` is "real" or not? Should it be reported as a bug?

Latent vs. Manifest Errors. To address the bug-reporting problem, our key idea is to distinguish between two types of bug reports (ISL error triples): *manifest bugs* and *latent bugs*. Manifest bugs are context-independent: their presence does not depend on any particular preconditions which may or may not hold in the calling context of the procedure being analysed. We show how to check these context-independence conditions algorithmically as part of our analysis. If a bug does not satisfy these conditions, we refer to it as latent.

Given the classification into manifest vs. latent bugs, Pulse-X's general policy is to only report bugs to developers if they are manifest (but see exceptions below). To be clear, this does not mean that latent bug specs are useless—far from it: a latent bug spec for f may be used compositionally to compute manifest bug specs for code that calls f. For example, for the function `deref(x)` mentioned earlier, Pulse-X would compute an error spec whose presumption is that `x` is `NULL`; however, this error spec is *not* context-independent, and thus it would be classified as a latent bug. If, however, another function `foo` were to (without context-dependent conditions of its own) call `deref` with argument `NULL`, we would then report a manifest bug for `foo`.

To formalise the claim that manifest bugs are indeed context-independent, we establish that Pulse-X satisfies a *True Positives Property*:[1]

> If a procedure `f` within a complete program has a manifest error, then either `f` is dead code (not reachable from `main()`) or there exists a concrete trace from `main()` to the error.

This provides a very strong justification for reporting an error to a programmer, even when the global program context is not known.

Memory Leaks. The existing ISL theory [Raad et al. 2020] does not support memory leaks and thus cannot identify leaky programs as erroneous. As memory leaks constitute an important class of bugs (and can lead to program crashes due to out-of-memory errors), we extend the ISL theory to account for memory leaks. Accordingly, we extend Pulse-X to generate erroneous specifications for leaky programs. Interestingly, however, memory leaks constitute an exception to our general bug-reporting criterion: for memory leaks we report *both* latent and manifest bugs. This is because as we discuss in §2.3, if a function contains a memory leak, very rarely would this be the fault of a caller. Specifically, in the case of latent memory leaks (where the leak arises only when the calling context satisfies certain conditions), even though a caller may happen to avoid leaky paths, the leak is a bug just the same. Moreover, leaks can be difficult to observe (especially for an end user), but easy to fix. As such, we opt to report both latent and manifest memory leaks in Pulse-X. For similar reasons, we also report latent null-pointer-dereference errors occuring in the `main()` function.

[1] *Standard Caveat.* Our soundness theorem says that, under assumptions encapsulated in a formal model, the computed abstractions are under-approximate with no false positives. However, in a real implementation there can still be false positives, in cases that lie outside the theoretical model assumptions. We did encounter false positives in practice, all due to outside-the-theory issues with unknown code, but this was the minority of findings (see §4.2 and §5).

Evaluation. To validate the effectiveness of Pulse-X, in §5 we present an experimental evaluation, showing that our rigorous bug-reporting criterion is competitive with the heuristic approach of Infer. We ran both Infer and Pulse-X on OpenSSL versions from 2015 and 2021. We chose OpenSSL as Infer had previously reported several bugs on OpenSSL in 2015 (fixed by OpenSSL developers), and thus it was natural to consider how Pulse-X would fare. We observed that Pulse-X has a higher *fix rate* (fixed bugs/total bugs found) than Infer on 2015 OpenSSL. Pulse-X also found 15 new bugs in present-day OpenSSL, which have since been confirmed and fixed. We also compared the performance of Pulse-X against that of Infer on a number of large programs; our comparison suggests that although Pulse-X is currently an academic tool not deployed in an industrial CI system, its performance characteristics are close enough to Infer to suggest that it could indeed be deployed in such systems in the future.

While we concentrate on fixed and new bugs in OpenSSL in this paper, a slightly relaxed version of our manifest criterion (see §3.3) has been incorporated into a sibling program analyzer (Pulse) being developed at Meta, where it led to a large enough reduction in false positives to allow null pointer checks to be turned on in production and applied to codebases with millions of lines of code. This has resulted in tens of new bugs being found and fixed during code review, with fix rate over 80%.

Contributions. Our technical contributions are as follows. (1) We have formulated the notion of latent vs. manifest errors to improve bug reporting and minimise noise (§3). (2) We have extended the ISL theory to detect memory leaks (§3). (3) We have developed an analysis tool, Pulse-X, underpinned by our extended ISL theory, which Pulse-X uses our notions of manifest errors for improved bug reporting (§4). (4) We show how to check manifest error conditions algorithmically in Pulse-X (§3). (5) We have evaluated Pulse-X on 2.8 millions of lines of code from 10 big programs, found 15 new true bugs in OpenSSL, and demonstrated that Pulse-X's accuracy and performance are competitive with those of the state-of-the-art Infer tool, indicating its suitability for CI deployment (§5). The evaluation of Pulse-X is provided as a companion artifact [Le et al. 2022].

Outline. The remainder of this article is organised as follows. In §2 we present an overview of our analysis and several representative bugs found by Pulse-X. In §3 we discuss the formal model underpinning Pulse-X. In §4 we present the Pulse-X analysis algorithm. In §5 we evaluate the performance of Pulse-X in terms of its scalability, bug report accuracy, and number of bugs found. We discuss related work in §6 and conclude in §7.

2 PULSE-X OVERVIEW AND REPRESENTATIVE BUG EXAMPLES

Rather than start with theory, we begin with bugs, three of them drawn from running Pulse-X on OpenSSL. The first two are real bugs illustrating different challenges for compositional analysis and reporting, while the third is a false positive which is excluded by our reporting criterion. To prepare the reader for the upcoming technical development, we make several remarks along the way regarding the Pulse-X theory and tool architecture.

2.1 Bug 1: A Null Dereference Bug

Listing 1 shows a null-pointer-dereference (denial-of-service) vulnerability we discovered in OpenSSL's `ssl_excert_prepend` function. The line starting with + denotes our proposed patch. The code begins by calling `app_malloc`, which in turn calls `CRYPTO_malloc` in its body. `CRYPTO_malloc`is a *malloc wrapper*, a wrapper for a C-standard `malloc`, used throughout OpenSSL instead of a standard `malloc`. The returned pointer is set to `0` on line 4. Pulse-X found that the `malloc` wrapper could return `NULL` and thus a null-pointer-dereference may occur on line 4. We reported this bug with our proposed patch to OpenSSL.

This bug may seem straightforward, and one might wonder whether a simplistic intra-procedural analysis could find it. The problem is that the analysis must understand that app_malloc is a malloc wrapper which can return NULL. Note that it is not scalable to ask the human (developer) to specify what the null-returning malloc wrappers are, as there might be many malloc wrappers in a given codebase, and new ones may be added with new pull requests.[2] A further interesting wrinkle is that not all malloc wrappers are the same: some wrappers never return NULL, and thus reporting bugs in these cases would lead to false positives, a point that was brought home to us vividly in an interaction with an OpenSSL maintainer which we now recall.

```
1  int ssl_excert_prepend(SSL_EXCERT **pexc){
2     SSL_EXCERT *exc= app_malloc(sizeof(*exc),
            "prepend cert");
3  +  if (exc == NULL) { return 0; }
4     memset(exc, 0, sizeof(*exc));
5     ...
6  }
```

Listing 1. A null pointer dereference error in OpenSSL.

Our analysis finds this bug by computing a procedure summary for app_malloc() which includes the two under-approximate ISL triples below:

$$[\mathsf{emp} \wedge \mathsf{true}]\ \texttt{app\_malloc(sz, what)}\ [ok\colon \mathsf{ret}{\mapsto}\mathsf{nil} \wedge \mathsf{true}]$$
$$[\mathsf{emp} \wedge \mathsf{true}]\ \texttt{app\_malloc(sz, what)}\ [ok\colon \exists X.\mathsf{ret}{\mapsto}X * X{\mapsto}{-} \wedge \mathsf{true}]$$

Using the first specification for app_malloc(sz, what) on line 2, when calling memset (on line 5) the analyser uses a built-in summary for memset which includes the following error specification:

$$[a{\mapsto}\mathsf{nil} \wedge \mathsf{true}]\ \texttt{memset(a,b,c)}\ [er\colon a{\mapsto}\mathsf{nil} \wedge \mathsf{true}]$$

Putting the two together, we obtain the following error specification for the entire procedure:

$$[\mathsf{emp} \wedge \mathsf{true}]\ \texttt{ssl\_excert\_prepend(pexc)}\ [er\colon \mathsf{emp} \wedge \mathsf{true}]$$

As we discuss in §3, this specification indicates that an error can happen under very general circumstances, no matter how ssl_excert_prepend() is called. This kind of general specification of a bug is what we call a *manifest error*, and we report such manifest errors to developers even when we do not have the calling context or an enveloping program containing a main() function.

When we first reported this bug to OpenSSL, an OpenSSL maintainer replied that it was a false positive as app_malloc aborts when the result is NULL. However, after inspecting the generated error trace of the bug shown in Listing 2, we were led to a definition of app_malloc in test/testutil/apps_mem.c which didn't call abort on NULL. [3]

```
1  apps/lib/s_cb.c:959: error: Nullptr Dereference
2  apps/lib/s_cb.c:957:23: in call to `app_malloc`
3    955. static int ssl_excert_prepend(SSL_EXCERT **pexc) {
4    956.     SSL_EXCERT *exc = app_malloc(sizeof(*exc), "prepend cert");
5  test/testutil/apps_mem.c:16:16: in call to `CRYPTO_malloc` (modelled)
6      14. void *app_malloc(size_t sz, const char *what) {
7      15.     void *vp = OPENSSL_malloc(sz);
```

Listing 2. Part of error trace of the bug in Listing 1. OPENSSL_malloc is a macro around CRYPTO_malloc.

[2]Note that, rather than having Pulse-X analyse the implementation of CRYPTO_malloc, we simply informed Pulse-X (using a flag) that CRYPTO_malloc behaves like standard malloc—the reason being that CRYPTO_malloc calls malloc in an obfuscated way via a global function pointer set at runtime. However, assuming that primitive spec for CRYPTO_malloc, Pulse-X automatically discovered procedure summaries for app_malloc and all the other malloc wrappers used by OpenSSL.
[3]The full trace is available in the supplementary material [Le et al. 2022].

It turns out that another wrapper with the same name `app_malloc`, located in `apps/lib/apps.c`, *does* abort in the case of `NULL` result and was the one the maintainer had in mind, but it was not the version called in Listing 1, leading to this bug. Equipped with this extra information, the maintainer acknowledged our bug report as genuine. However, they proposed an alternative fix instead of ours. In a separate pull request (https://github.com/openssl/openssl/pull/15836) the developer changed the wrapper in `test/testutil/apps_mem.c` to also call `abort` on `NULL` (see Listing 3). After applying the developer's preferred fix, Pulse-X gets rid of the path where `OPENSSL_malloc` returns `NULL` inside `app_malloc()`, and so in `ssl_excert_prepend()` too, since `abort` is modelled as a function that exits the program, Pulse-X no longer reports an error in `ssl_excert_prepend()`.

```
1  void *app_malloc(size_t sz, const char *what){
2      void *vp;
3      if (!TEST_ptr(vp = OPENSSL_malloc(sz)))
4          abort();
5      return vp;
6  }
```

Listing 3. Null-avoiding malloc() wrapper.

This story highlights the value of inter-procedural reasoning in explaining and fixing bugs that might deceptively seem simple.

2.2 Bug 2: A Memory Leak

One particularly interesting bug involved the `s_server` application, which implements a generic SSL/TLS server listening for connections on a given port using SSL/TLS. Pulse-X discovered a memory leak in the `www_body` function (Listing 4). Once allocated, the SSL socket `ssl_bio` is pushed to the end of the linked list `io` (line 6); all sockets and buffers in `io` are freed before returning. Pulse-X found that the memory allocated at line 4 would be leaked if `io` was itself `NULL` as in that case `BIO_push` would silently do nothing. When we submitted the patch (adding the line starting with + in Listing 4), an OpenSSL maintainer warned us that our fix might cause a double-free error and suggested assigning `ssl_bio` to `NULL` after the push instead. However, running Pulse-X on the patch proposed by the maintainer revealed that the leak still existed under the same condition: whenever the push failed. Furthermore, our proposed fix does not in fact cause a double-free error because the `BIO` library uses a reference count mechanism to prevent such an error. After reporting this new observation to the maintainer, they agreed with our proposed fix.

```
1   static int www_body(int s, int stype,
        int prot, unsigned char *context){
2       ...
3       io = BIO_new(BIO_f_buffer());
4       ssl_bio = BIO_new(BIO_f_ssl());
5       ...
6       BIO_push(io, ssl_bio);
7       ...
8       BIO_free_all(io);
9   +   BIO_free(ssl_bio);
10      return ret;
11  }
```

Listing 4. OpenSSL memory leak in `www_body`.

This bug was a challenging one to find. The `www_body` function contains 426 lines of fairly complex code: the list `io` is manipulated by a chain of function calls and multiple loops. In contrast to over-approximate techniques (such as that of Infer's analysis), which cannot reason precisely about the presence of bugs for looping programs, Pulse-X performs under-approximation and can reason precisely about the bugs within a bounded number of program paths (loop unfoldings). This example also illustrates a challenge in software testing. While a developer may write tests to try to make sure this function works correctly, it is perhaps not immediate that a test would exercise the condition that triggers the failure of `BIO_push(io, ssl_bio)`. As a direct consequence, this bug was nearly three years old (it also affects the stable release OpenSSL-1.1.1).

2.3 Latent and Manifest Bugs

A function has a latent error if it contains a bug that occurs only when its inputs satisfy certain conditions, i.e., the bug does not occur in all calling contexts. Otherwise the bug is manifest.

The potential null dereference in `file_ctrl` (Listing 5) is an example of bug classified as latent in OpenSSL-1.0.1h. It has not been fixed in the current OpenSSL-3.0.0 (commit 147ed5f).

```
1  long file_ctrl(BIO *b, int cmd, long num, void *ptr){
2     long ret = 1;
3     FILE *fp = (FILE *)b->ptr;
4     ...
5  }
```

Listing 5. Latent in `crypto/bio/bss_file.c`.

Our analysis found a null-pointer-dereference error on line 3, but did not report it to the user. This bug occurs only when the input `b` is `NULL`. Pulse-X would report this bug at a call site where pointer `b` is `NULL`. Indeed, this issue seems not to be a real bug and has not been fixed; the code of function `file_ctrl` is the same in the current revision of OpenSSL.

As this example illustrates, it seems undesirable to report latent null-pointer-dereferences (NPEs) to a programmer. Often, there are implicit assumptions on whether a pointer is allocated, which are not checked locally, and such assumptions can be inferred and pushed back to callers. Interestingly, however, the case of leaks is different. If a function contains a local memory leak, very rarely would this be the fault of a caller. A caller might happen to avoid a path with a leak, but the leak is a bug just the same. Furthermore, leaks can be difficult to observe (especially for an end user), but easy to fix. For these reasons, we have adopted a strategy of reporting latent leaks, but not NPEs. Our experience with this strategy has been good: OpenSSL developers reacted positively to latent leak reports, and legacy latent leaks had often been fixed: from our OpenSSL-1.0.1h experiments, there were 50% (7 out of 14) of latent leaks reported by Pulse-X that were fixed later.

In more detail, our strategy is to report:

- all manifest null-pointer-dereferences (like the one in §2.1), no matter where in the program;
- all memory leaks (like the one in §2.2), no matter where in the program; and
- all null-pointer-dereferences found in `main()`.

As we demonstrate in more detail in §5, our experimental evidence validates this strategy: on OpenSSL-1.0.1h, Pulse-X found 306 issues in all (latent and manifest); it reported 26 of the 306 issues (those satisfying one of the properties above); 19 of the 26 have been subsequently fixed with 3 not fixed and the remaining ones unknown (the procedures were removed totally). On direct inspection, we found that a high number of the remaining (latent) issues had not been fixed, and there was no reason to fix them (a human might reasonably label them false positives, had they been reported).

3 THE PULSE-X FORMAL MODEL

We present our ISL formulae, their semantics in a concrete state model, and the semantics of under-approximate triples. We give a formal definition of manifest bugs in terms of the assertions and triples, and establish a true positives result which underpins our bug reporting in Pulse-X. The result is a property of the triples and assertions themselves, and is independent of the way that the triples are computed. The next section describes an analysis algorithm for obtaining such triples.

The* Pulse-X *State Model. As shown in Fig. 1, we model a Pulse-X *world* as a pair (η, σ), comprising an *environment* η and a state σ. Intuitively, the environment tracks the values associated with

Pulse-X **Assertions**

$$\begin{array}{rll} \kappa ::= & \mathsf{emp} \mid x{\mapsto}X \mid X{\mapsto}V \mid X{\not\mapsto} \mid \kappa * \kappa & \text{spatial} \\ \pi ::= & \mathsf{true} \mid B \mid \pi \wedge \pi \mid \pi \vee \pi \mid \neg\pi & \text{pure} \\ \Delta ::= & \kappa \wedge \pi & \text{quantifier-free} \\ p, q, r ::= & \Delta \mid \exists X.\, \Delta & \text{top-level} \end{array}$$

Pulse-X **Model Domain**

$$\begin{array}{l} \eta \in \mathsf{Env} \triangleq (\mathsf{Var} \cup \mathsf{LVar}) \to \mathsf{Val} \\ \sigma \in \mathsf{State} \triangleq \mathsf{Loc} \rightharpoonup_{\mathrm{fin}} \mathsf{Val} \quad \text{with } \mathsf{Loc} \uplus \{\mathsf{nil}\} \subseteq \mathsf{Val} \\ s \in \mathsf{World} \triangleq \mathsf{Env} \times \mathsf{State} \end{array}$$

Pulse-X **Assertion Semantics**

$$\begin{array}{lll} (\eta,\sigma) \models \mathsf{emp} & \textit{iff} & \mathrm{dom}\,(\sigma) = \emptyset \\ (\eta,\sigma) \models x{\mapsto}X & \textit{iff} & \sigma = \{\eta(x) \mapsto \eta(X)\} \\ (\eta,\sigma) \models X{\mapsto}V & \textit{iff} & \sigma = \{\eta(X) \mapsto \eta(V)\} \\ (\eta,\sigma) \models X{\not\mapsto} & \textit{iff} & \sigma = \{\eta(X) \mapsto \bot\} \\ (\eta,\sigma) \models \kappa_1 * \kappa_2 & \textit{iff} & \exists \sigma_1, \sigma_2.\ \sigma = \sigma_1 \uplus \sigma_2 \wedge \\ & & (\eta,\sigma_1) \models \kappa_1 \wedge (\eta,\sigma_2) \models \kappa_2 \\ (\eta,\sigma) \models \kappa \wedge \pi & \textit{iff} & (\eta,\sigma) \models \kappa \text{ and } \eta \models \pi \\ (\eta,\sigma) \models \exists X.\, \Delta & \textit{iff} & \exists v.\ (\eta[X \mapsto v], \sigma) \models \Delta \end{array}$$

Fig. 1. The Pulse-X model domain, assertions and their semantics.

program and logical variables, while the state models the stack and the heap. We assume two (countably infinite) sets of *program variables*, Var, and *logical variables*, LVar, such that $\mathsf{Var} \cap \mathsf{LVar} = \emptyset$, a set of *heap (memory) locations*, Loc, and a set of values, Val, such that $\mathsf{Loc} \uplus \{\mathsf{nil}\} \subseteq \mathsf{Val}$. We further assume a standard interpreted language of *(program) expressions*, Exp, containing at least variables and values, and a standard interpreted language for *Boolean expressions*, BExp.

We use $x, y, \ldots$ as metavariables for program variables; $X, Y, \ldots$ for logical variables; v for values; e for expressions; ret for a designated program variable recording the return value at procedure exits; and $\overrightarrow{x}$ for a tuple of variables. Lastly, we assume an *expression interpretation function*, $[\![.]\!] : \mathsf{Exp} \to \mathsf{Env} \to \mathsf{Val}$, and a *Boolean interpretation function*, $[\![.]\!] : \mathsf{BExp} \to \mathsf{Env} \to \mathsf{Val}$, respectively evaluating the values of expressions and Boolean expressions against a given environment.

The* Pulse-X *Assertions and Their Semantics. As shown in Fig. 1, Pulse-X assertions describe sets of worlds, and their semantics is given by a satisfiability relation, $\models$. The Pulse-X *assertions* (ranged over by p, q, r) are those of Calcagno et al. [2011], without inductive predicates and extended with the *invalidated* assertion $X{\not\mapsto}$ of ISL.[4] Pure assertions constrain the underlying environment η; their semantics is standard and omitted here. Analogously, spatial assertions constrain the shape of the state σ. Specifically, emp describes worlds in which the state is empty; $x{\mapsto}X$ describes worlds in which the state comprises a single location denoted by (the interpretation of) x containing the value denoted by X; analogously for $X{\mapsto}V$. Similarly, $X{\not\mapsto}$ describes worlds in which the state comprises a single invalidated location at X (i.e., X contains the designated value $\bot$), and $\kappa_1 * \kappa_2$ describes worlds in which the state can be split into two disjoint sub-states, one satisfying κ_1 and the other κ_2. The semantics of the remaining assertions is standard.

Notation. We define $(\!|p|\!) \triangleq \{s \mid s \models p\}$ and write $p \vdash q \overset{\text{def}}{\Leftrightarrow} (\!|p|\!) \subseteq (\!|q|\!)$. We write $p \equiv q$ for $p \vdash q \wedge q \vdash p$ and write $\mathrm{sat}(p)$ when p is satisfiable: $\mathrm{sat}(p) \overset{\text{def}}{\Leftrightarrow} (\!|p|\!) \neq \emptyset$. We write $\mathrm{fv}(p)$ (resp. $\mathrm{flv}(p)$) for the set of free variables (resp. free logical variables) in p. We use the standard substitution notation and write $p[v/X]$ for the assertion obtained from p by substituting v for X.

3.1 Pulse-X **Programs and Summaries**

We present the Pulse-X programming language in Fig. 2, which is similar to the intermediate language of Infer [Calcagno et al. 2011]. A program P comprises a sequence of *procedure (function)*

[4]Inductive predicates were used by Calcagno et al. [2011] to establish over-approximate loop invariants, which we do not need in an under-approximate analysis; see [O'Hearn 2019] for a discussion. However, using inductive predicates to leap loops via backward variant rules in incorrectness logic [O'Hearn 2019] might be studied in separate work.

$$
\begin{array}{rcll}
\mathsf{PDef} \ni \mathsf{pdef} & ::= & \mathtt{skip} \mid \mathtt{error()} \mid \mathtt{assume}(\pi) \mid x := e \mid x := [y] \mid [x] := y & \text{Predefined instructions} \\
 & & \mid x := \mathtt{malloc()} \mid \mathtt{free}(x) \mid \mathtt{return}(x) & \\
\mathsf{Inst} \ni \mathsf{inst} & ::= & \mathsf{pdef} \mid \mathtt{f()} & \text{Instructions} \\
\mathsf{Comm} \ni \mathsf{C} & ::= & \mathsf{inst} \mid \mathtt{local}\ x.\mathsf{C} \mid \mathsf{C};\mathsf{C} \mid \mathsf{C} + \mathsf{C} \mid \mathsf{C}^\star & \text{Commands} \\
\mathsf{FDec} \ni \mathsf{d} & ::= & \mathtt{f()} = \mathsf{C} & \text{Function declarations} \\
\mathsf{Prog} \ni \mathsf{P} & ::= & \mathsf{d}, \cdots, \mathsf{d} & \text{Programs}
\end{array}
$$

Fig. 2. The Pulse-X programming language.

declarations. A procedure declaration d is of the form $\mathtt{f()} = \mathsf{C}$, where C is a *command* describing the function body. Commands include *instructions* (inst), local variable declaration ($\mathtt{local}\ x.\mathsf{C}$), sequential composition ($\mathsf{C};\mathsf{C}$), non-deterministic choice ($\mathsf{C} + \mathsf{C}$) and Kleene (non-deterministic) iteration ($\mathsf{C}^\star$). An instruction is either a procedure call or a predefined instruction. The predefined instructions include skip, assume statements, assignment ($x := e$), heap lookup ($x := [y]$), heap mutation ($[x] := y$), memory allocation, memory disposal and return statements. Standard procedure calls in C – where a call allocates formals, assigns actuals and deallocates formals on procedure exit – can be encoded in terms of predefined instructions.

We use assume statements to encode if and while statements: $\mathtt{if}\ \pi\ \mathtt{then}\ \mathsf{C}_1\ \mathtt{else}\ \mathsf{C}_2$ is encoded as $(\mathtt{assume}(\pi); \mathsf{C}_1) + (\mathtt{assume}(\neg\pi); \mathsf{C}_2)$; $\mathtt{while}(\pi)\ \mathtt{do}\ \mathsf{C}$ is encoded as $(\mathtt{assume}(\pi); \mathsf{C})^\star; \mathtt{assume}(\neg\pi)$. Similarly, we define $\mathtt{assert}(\pi) \triangleq (\mathtt{assume}(\neg\pi); \mathtt{error()}) + (\mathtt{assume}(\pi); \mathtt{skip})$.

Our approach to formalism here is minimalistic rather than maximalistic: its aim is to include as little as possible in the language while still exposing key issues to explain/probe technically, rather than including as much as possible so as to cover more of an implemented analyser. In order to focus on the essential issues regarding the analysis algorithm and how it infers states, for brevity we focus on parameterless procedures. It would be straightforward to follow the treatment of parameters by Calcagno et al. [2011], and doing so would not provide any novel insights. We also assume that procedures are non-recursive. In practice they can also be subject to bounded unrolling, and that is what our implementation (implicitly) does.

Predefined Pulse-X ***Summaries as ISL Triples***. As we describe in §4, the Pulse-X algorithm uses the summaries (specifications) of predefined instructions to infer the specification of a given piece of code. The summaries of predefined instructions are given in Fig. 3 as ISL triples, and are adapted from those of Raad et al. [2020] with only minor changes to fit our formalism with environment η. In particular, these specifications are modified with $x{\mapsto}X$ to track the values X of program variables x. For example, the spec for assignment statements $x := e$ replaces variable x by the value X and other variables y_i by the values Y_i before actually applying the assignment.

Pulse-X ***Summaries as Valid ISL Triples***. As discussed in §2, Pulse-X computes procedure summaries as a set of *valid* ISL (under-approximate) triples of the form $[p]\ \mathsf{C}\ [\epsilon : q]$. As described in Raad et al. [2020], a triple $[p]\ \mathsf{C}\ [\epsilon : q]$ is *valid*, written $\models [p]\ \mathsf{C}\ [\epsilon : q]$, iff every state in q is reachable under the exit condition ϵ (which may be *ok* or *er*) by executing C on some state satisfying p. Note that $[p]\ \mathsf{C}\ [\epsilon : \mathsf{false}]$ is vacuously valid as false denotes an empty set of states.

Definition 3.1 (ISL triples). An (under-approximate) ISL triple $[p]\ \mathsf{C}\ [\epsilon : q]$ is *valid*, written $\models [p]\ \mathsf{C}\ [\epsilon : q]$, iff $(\!|q|\!) \subseteq [\![\mathsf{C}]\!]_\epsilon((\!|p|\!))$, where $[\![.]\!]_\epsilon : \mathsf{Comm} \to \mathcal{P}(\mathsf{World} \times \mathsf{World})$ denotes the command semantics under ϵ, defined in the supplementary material [Le et al. 2022], as a state transition system analogously to that in Raad et al. [2020].

$$\left[(\circledast_{x_i \in \mathsf{pvars}(\pi)}\, x_i{\mapsto}X_i) \wedge \pi[\overrightarrow{X_i}/\overrightarrow{x_i}]\right] \mathtt{assume}(\pi) \left[ok\colon (\circledast_{x_i \in \mathsf{pvars}(\pi)}\, x_i{\mapsto}X_i) \wedge \pi[\overrightarrow{X_i}/\overrightarrow{x_i}]\right]$$

$$\left[\begin{array}{l}(\circledast_{y_i \in \mathsf{pvars}(e)\setminus\{x\}}\, y_i{\mapsto}Y_i * x{\mapsto}X)\\ \wedge\ V = e[\overrightarrow{Y_i}/\overrightarrow{y_i}][X/x]\end{array}\right] x := e \left[\begin{array}{ll}ok\colon & (\circledast_{y_i \in \mathsf{pvars}(e)\setminus\{x\}}\, y_i{\mapsto}Y_i * x{\mapsto}V)\\ & \wedge\ V = e[\overrightarrow{Y_i}/\overrightarrow{y_i}][X/x]\end{array}\right]$$

$$[x{\mapsto}X * X{\mapsto}V * y{\mapsto}Y]\,[x] := y\,[ok\colon x{\mapsto}X * X{\mapsto}Y * y{\mapsto}Y]$$
$$[x{\mapsto}X * y{\mapsto}Y * Y{\mapsto}V]\,x := [y]\,[ok\colon x{\mapsto}V * y{\mapsto}Y * Y{\mapsto}V]$$

$$[\mathsf{emp}]\ \mathtt{skip}\ [ok\colon \mathsf{emp}] \qquad [\mathsf{emp}]\ \mathtt{error()}\ [er\colon \mathsf{emp}]$$

$$[x{\mapsto}X * X{\not\mapsto}]\,[x] := y\,[er\colon x{\mapsto}X * X{\not\mapsto}] \qquad [y{\mapsto}Y * Y{\not\mapsto}]\,x := [y]\,[er\colon y{\mapsto}Y * Y{\not\mapsto}]$$
$$[x{\mapsto}X \wedge X = \mathsf{nil}]\,[x] := y\,[er\colon x{\mapsto}X \wedge X = \mathsf{nil}] \qquad [y{\mapsto}Y \wedge Y = \mathsf{nil}]\,x := [y]\,[er\colon y{\mapsto}Y \wedge Y = \mathsf{nil}]$$

$$[x{\mapsto}X * X{\mapsto}V]\ \mathtt{free}(x)\ [ok\colon x{\mapsto}X * X{\not\mapsto}] \qquad [x{\mapsto}X]\,x := \mathtt{malloc()}\,[ok\colon \exists L.\ x{\mapsto}L * L{\mapsto}V \wedge \mathsf{true}]$$
$$[x{\mapsto}X * X{\not\mapsto}]\ \mathtt{free}(x)\ [er\colon x{\mapsto}X * X{\not\mapsto}] \qquad [x{\mapsto}X]\,x := \mathtt{malloc()}\,[ok\colon \exists L.\ x{\mapsto}L \wedge L = \mathsf{nil}]$$
$$[x{\mapsto}X \wedge X = \mathsf{nil}]\ \mathtt{free}(x)\ [er\colon x{\mapsto}X \wedge X = \mathsf{nil}] \qquad [x{\mapsto}X * \mathsf{ret}{\mapsto}V]\ \mathtt{return}(x)\,[ok\colon x{\mapsto}X * \mathsf{ret}{\mapsto}X]$$

Fig. 3. Predefined Pulse-X summaries as ISL triples, where pvars(.) returns the program variables of an expression or a statement; for brevity, we omit the pure assertion true and write p in lieu of $p \wedge \mathsf{true}$.

3.2 Manifest Errors

Recall from §2 that to minimise noise when reporting bugs and eliminate false positives, we introduce the notion of *manifest errors*. Intuitively, a manifest error denotes a valid summary $[p]$ C $[er\colon q]$ that (1) can be applied within *any calling context* (i.e., regardless of the state at the call site); and (2) when applied, it always yields an erroneous execution terminating in a state satisfying an extension of q. That is, executing C on any state s terminates erroneously in some state s' (i.e., $(s, s') \in [\![\mathsf{C}]\!]_{er}$) satisfying an extension of q ($s' \in (\!|q * \mathsf{true}|\!)$); i.e., the manifest error is reachable from any input state. This is formulated in Def. 3.2 (without taking procedure parameters into account for simplicity) and forms the semantic basis of our approach to compositional reporting.

Definition 3.2. A valid error triple $\models [p]\ \mathsf{C}\ [er\colon q]$ denotes a *manifest error* iff:

$$\forall s.\ \exists s'.\ (s, s') \in [\![\mathsf{C}]\!]_{er} \wedge s' \in (\!|q * \mathsf{true}|\!)$$

Theorem 3.3 below establishes a sufficient logical condition for identifying manifest errors (see the supplementary material [Le et al. 2022] for its full proof). It states that a valid summary $[p]$ C $[er\colon q]$ denotes a manifest error if for any context r such that $\mathsf{sat}(r)$ holds[5], then (1) there exists f such that $p * f \vdash r$, i.e., the precondition p can be extended with a frame f to match the context r; and (2) $\mathsf{sat}(q * f)$ also holds: the postcondition q can be analogously extended with frame f. The latter ensures that $[p]$ C $[er\colon q]$ is not vacuously valid ($q \not\vdash \mathsf{false}$) as otherwise $\mathsf{sat}(q * f)$ would not hold.

Theorem 3.3. *A valid error triple* $\models [p]\ \mathsf{C}\ [er\colon q]$ *denotes a* manifest error *if:*

$$\forall r.\ \mathsf{sat}(r) \Rightarrow \exists f.\ p * f \vdash r\ \wedge\ \mathsf{sat}(q * f)$$

Finally, the following theorem follows from Def. 3.2 and states that an error specification of a manifest bug in procedure `f()` of a complete program is either reachable from the (top) `main()` procedure, or it is not reachable because `f()` is dead code. By a "complete program" we mean one with `main()` in which there are no undefined, free function names.

[5]Note that given any context state s, there exists an assertion r that s satisfies, namely the formula p_s that describes s cell by cell such that $(\!|p_s|\!) \triangleq \{s\}$.

Theorem 3.4 (True Positives Property). *If* `f()` *is a procedure in a complete program with a manifest error, then either* `f()` *is dead code (not reachable from* `main`*) or* $[\![\texttt{main}]\!]_{er} \neq \emptyset$.

Manifest Errors and Backwards Under-Approximate Triples. Def. 3.2 states that *every* (and not just some) state in the precondition can reach a state in the postcondition. As such, manifest errors curiously satisfy another interpretation of triples, namely that of *backwards, under-approximate triples* denoted as $\{\!|p|\!\}\ \mathsf{C}\ \{\!|\epsilon : q|\!\}$, stating that p under-approximates the states reachable from q when executing C backwards (see [Möller et al. 2021, §5]). We formalise and prove this relation to backwards, under-approximate triples in the supplementary material [Le et al. 2022]. A related concept was studied by Ball et al. [2005] under the name must$^+$ transitions, in the reachability analysis of Asadi et al. [2021], and referred to as "total Hoare triples" by de Vries and Koutavas [2011].

3.3 Algorithmically Identifying Manifest Errors

Note that directly checking the conditions in Theorem 3.3 is practically infeasible as it involves finding a suitable frame f for each *arbitrary* context r, while also ensuring $\mathsf{sat}(q * f)$. To remedy this, we identify four conditions in Theorem 3.5 that are simpler to check, do not quantify over all contexts, and if satisfied ensure that a given triple $\models [p]\ \mathsf{C}\ [er\colon q]$ denotes a manifest error. We proceed with an intuitive description of these four conditions.

First, to ensure that the triple can be applied in any context, we stipulate that the precondition impose no spatial or pure constraints on the underlying state. That is, we require (1) $p \equiv \mathsf{emp} \wedge \mathsf{true}$. This way, for any given context r we have $p * r \equiv r$ (and thus $p * r \vdash r$), yielding the frame r.

Next, to ensure that $\mathsf{sat}(q * r)$ holds for an arbitrary r, we require that (2) $\mathsf{sat}(q)$ hold, as otherwise $q * r$ would be rendered unsatisfiable. Note that condition (2) is not sufficient to ensure $\mathsf{sat}(q * r)$ holds for an arbitrary r. To see this, consider `f()` and `g()` below and their valid error summaries:[6]

```
f() ≜ z := malloc();
      if (z = x) then error()
```

$S_1\colon \models [x{\mapsto}X * z{\mapsto}Z]\ \texttt{f()}\ [er\colon x{\mapsto}X * z{\mapsto}X * X{\mapsto}V]$

$S_2\colon \models [x{\mapsto}X * z{\mapsto}Z]\ \texttt{f()}\ [er\colon \exists Y.\ x{\mapsto}X * z{\mapsto}Y * Y{\mapsto}V \wedge Y{=}X]$

```
g() ≜ if (x = 7) then error()
```

$S_3\colon \models [x{\mapsto}X]\ \texttt{g()}\ [er\colon x{\mapsto}X \wedge X{=}7]$

All summaries are valid and can be derived using the ISL proof system [Raad et al. 2020]. Specifically, ISL also includes the following axiom, allowing the allocation to pick a *specific* location, namely X:

$$[z{\mapsto}Z]\ z := \texttt{malloc()}\ [ok\colon z{\mapsto}X * X{\mapsto}V] \qquad \text{(Malloc-ISL)}$$

This summary is a combination of the predefined summary of `malloc()` in Fig. 3 and the under-approximate consequence rule, Cons, in ISL (where the direction of entailments in its premise are reversed [Raad et al. 2020]) to strengthen the post. As such, S_1 can be derived using the ISL rule for `malloc()` above, while S_2 can be derived using the predefined summary of `malloc()` in Fig. 3, followed by an application of Cons (as $\exists Y.\ x{\mapsto}X * z{\mapsto}Y * Y{\mapsto}V \wedge Y{=}X \vdash \exists Y.\ x{\mapsto}X * z{\mapsto}Y * Y{\mapsto}V$). Similarly, S_3 can be derived using the encoding of `if` and the summaries of `assume(.)` and `error()`.

Let q_1, q_2 and q_3 respectively denote the postconditions of S_1, S_2 and S_3. As mentioned above, in the case of S_1, `malloc()` allocates a *specific* location, namely that denoted by X, rendering $q_1 * r$ unsatisfiable for an arbitrary r: e.g., if $r \triangleq X{\mapsto}V$, then $q_1 * r \equiv$ false. In other words, for an error to be manifest, it must not require that `malloc()` allocate a specific location, i.e., the postcondition

[6] These functions have non-empty preconditions in their summaries because they manipulate global variables, so technically speaking they are already in violation of condition (1) of Theorem 3.3. However, this is only because, for simplicity, we have ignored global variables and function parameters in our formal presentation. The actual manifest criterion used by Pulse-X implements a less restrictive version of condition (1) which allows the precondition to merely assert that globals and function parameters exist so long as it does not constrain their contents.

may not constrain the locations allocated by `malloc()`. Intuitively, S_1 does not denote a manifest error as the specified error only occurs when `malloc()` allocates a specific location X (i.e., the location denoted by X is not allocated on the heap beforehand), and thus this error does not arise in contexts in which X is already allocated. To remedy this, when the summary postcondition is given by $\exists\overrightarrow{X_q}.\ \kappa_q \wedge \pi_q$, we require that the heap locations in κ_q (i.e., the logical variables on the left-hand side of $\mapsto$ and $\not\mapsto$ assertions in κ_q) be *existentially quantified* and thus not be the same as heap locations in the context. To this end, we define an auxiliary function, locs(.), that computes the heap locations in a given spatial assertion κ_q; we then require that (3) $\mathsf{locs}(\kappa_q) \subseteq \overrightarrow{X_q}$. Indeed, as we demonstrated above in Fig. 3, in Pulse-X we do not include the original ISL summary for `malloc()` in Malloc-ISL; instead we make a minor alteration by existentially quantifying the location allocated, thus ensuring that this condition holds for all summaries computed by Pulse-X.

Note that conditions (1–3) are still not sufficient to ensure $\mathsf{sat}(q * r)$ as the pure assertions in the postcondition may impede the satisfiability of $q * r$. Specifically, in the case of S_2, although the location allocated via `malloc()` is existentially quantified as Y in q_2, the pure part of q_2 requires $Y{=}X$, once again constraining the location allocated via `malloc()`, rendering $q_2 * r$ unsatisfiable when $r \triangleq X{\mapsto}V$. Note that $q_2 \equiv q_1$ and thus the error specified by S_2 is likewise not manifest. In the case of S_3, the postcondition q_3 constrains the value of x (tied to logical variable X) by requiring $X{=}7$, rendering $q_3 * r$ unsatisfiable when e.g., $r \triangleq X{=}2$. Intuitively, the error identified by S_3 does not denote a manifest error as it only arises in contexts in which $X{=}7$. To rule out summaries such as S_2 and S_3 as manifest errors, when the summary postcondition is given by $q \triangleq \exists\overrightarrow{X_q}.\ \kappa_q \wedge \pi_q$ and $\mathsf{flv}(q) = \overrightarrow{Y}$, we lastly require that the pure assertion π_q be satisfiable for *any choice* of allocated locations (ruling out summaries such as S_2) and *any choice* of free logical variables (ruling out summaries such as S_3). Put formally, we require that (4) $\mathsf{sat}(\pi_q[\overrightarrow{v}/\overrightarrow{Y} \cup \mathsf{locs}(\kappa_q)])$ hold for all $\overrightarrow{v}$. Note that neither S_2 nor S_3 satisfies condition 4: in the case of S_2, $\mathsf{sat}(Y{=}X[v_1/X, v_2/Y])$ does not hold when $v_1{\neq}v_2$; in the case of S_3, $\mathsf{sat}(X{=}7[v/X])$ does not holds when $v{\neq}7$.

We formalise the four conditions described above in Theorem 3.5, proving that they are sufficient for identifying manifest errors. The full proof of this theorem is given in the supplementary material.

Theorem 3.5 (Manifest errors). *An error triple* $\models [p]\ \mathsf{C}\ [er\colon q]$ *with* $q \triangleq \exists\overrightarrow{X_q}.\ \kappa_q \wedge \pi_q$ *denotes a* manifest error *if:*

(1) $p \equiv \mathsf{emp} \wedge \mathsf{true}$*;*
(2) $\mathsf{sat}(q)$ *holds;*
(3) $\mathsf{locs}(\kappa_q) \subseteq \overrightarrow{X_q}$*, where* $\mathsf{locs}(.)$ *is as defined below; and*
(4) for all $\overrightarrow{v}$, $\mathsf{sat}(\pi_q[\overrightarrow{v}/\overrightarrow{Y} \cup \mathsf{locs}(\kappa_q)])$ *holds, where* $\overrightarrow{Y} = \mathsf{flv}(q)$.

$\mathsf{locs}(\mathsf{emp}) \triangleq \emptyset \quad \mathsf{locs}(x{\mapsto}X) \triangleq \{x\} \quad \mathsf{locs}(X{\mapsto}V){=}\mathsf{locs}(X{\not\mapsto}) \triangleq \{X\} \quad \mathsf{locs}(\kappa_1 * \kappa_2) \triangleq \mathsf{locs}(\kappa_1) \cup \mathsf{locs}(\kappa_2)$

Conditions (1) and (3) can be checked in polynomial time, and the complexity of checking conditions (2) and (4) in our assertion language corresponds to the one of the satisfiability problem of equality logic, which is NP-complete. Nevertheless, checking these conditions is efficient in practice.

Relaxing the Manifest Condition. Although our notion of manifest errors above allows us to identify a large number of bugs in existing code (see §5), we have found that a weakening of the criterion can be useful to identify certain kinds of bugs. To see this, consider `h()` below:

`h()` $\triangleq$ `free(`x`);` $[x] := 2$	S_4: $\models$ $[x{\mapsto}X * X{\not\mapsto}]$ `h()` $[er\colon x{\mapsto}X * X{\not\mapsto}]$
	S_5: $\models [x{\mapsto}X * X{\mapsto}V]$ `h()` $[er\colon x{\mapsto}X * X{\not\mapsto}]$

Note that neither S_4 nor S_5 meet the conditions in Theorem 3.5: their preconditions require non-emp heap resources, namely $X{\not\mapsto}$ and $X{\mapsto}V$, respectively. In particular, the error in S_4 only arises

Extended Pulse-X **Model Domain**

$\eta \in \mathsf{Env} \triangleq (\mathsf{Var} \rightharpoonup \mathsf{Val}) \cup (\mathsf{LVar} \rightharpoonup \mathsf{Val} \cup \mathcal{P}(\mathsf{Val}))$

$\sigma \in \mathsf{State} \triangleq (\mathsf{Loc} \rightharpoonup_{\mathrm{fin}} \mathsf{Val}) \cup (\{\mathbf{a}\} \rightharpoonup \mathcal{P}(\mathsf{Val}))$ where $\mathbf{a} \in \mathsf{Loc}$

Extended Pulse-X **Assertions**

$\kappa ::= \cdots \mid \mathbf{a} \mapsto \mathbf{A}$

$\pi ::= \cdots \mid \mathsf{leaks}(\mathbf{S}, L)$

Extended Pulse-X **Assertion Semantics**

$(\eta, \sigma) \models \mathbf{a} \mapsto \mathbf{A}$ *iff* $\sigma = \{\mathbf{a} \mapsto \eta(\mathbf{A})\}$

$(\eta, \sigma) \models \mathsf{leaks}(\mathbf{S}, L)$ *iff* $\eta(L) \notin \mathsf{reach}(\eta(\mathbf{S}), \sigma)$

where $\mathsf{reach}(S, \sigma) \triangleq \bigcup_{i \in \mathbb{N}^+} S_i$ $\quad S_n \triangleq \left\{ l \in \mathsf{Loc} \,\middle|\, \exists k \in S_{n-1}.\ \sigma(k) = l \right\}$ $\quad S_0 \triangleq S$

$$\left[\mathbf{a} \mapsto \mathbf{A} * x \mapsto X\right]\ x := \mathtt{malloc()}\ \left[ok\colon \exists L.\ \mathbf{a} \mapsto \mathbf{A} \uplus \{L\} * x \mapsto L\right]$$

$$\left[\mathbf{a} \mapsto \mathbf{A} * x \mapsto X * X \mapsto V\right]\ \mathtt{free}(x)\ \left[ok\colon \mathbf{a} \mapsto \mathbf{A} \setminus \{X\} * x \mapsto X * X \not\mapsto\right]$$

Fig. 4. Updated ISL model and assertions for memory leak detection (above), where $\mathbf{A}, \mathbf{S}$ denote meta-variables for sets of logical variables and the extensions from Fig. 1 are highlighted; updated predefined summaries for memory leak detection (below), where the extensions from Fig. 3 are highlighted.

when x is deallocated ($X \not\mapsto$) and thus could easily constitute a false positive when x is allocated. By contrast, the error in S_5 *always* arises, *provided that* $[x] := 2$ is reached and executed, i.e., the execution is not terminated earlier due to a previous error at $\mathtt{free}(x)$ as captured by S_4. This is because executing $[x] := 2$ after it has been deallocated (by $\mathtt{free}(x)$) always results in an error, and the error in S_5 is non-manifest only because (due to the short-circuiting semantics of ISL) $[x] := 2$ may not be executed. We thus argue that errors such as S_5 are of interest to programmers.

To this end, we introduce a *relaxed* notion of manifest errors, whereby we allow non-emp heaps in the precondition. Specifically, as shown in Def. 3.6, we relax the first condition in Theorem 3.5 to allow for *positive heaps*, i.e., those without deallocated (negative) heap cells. Intuitively, this is because the non-emp heap resources in the precondition are only needed to execute earlier instructions *normally* (under *ok*) without encountering an error, and as shown by the predefined ISL summaries in Fig. 3, negative heap cells are only required for erroneous executions (under *er*).

Definition 3.6 (Relaxed-manifest errors). A triple $\models [p]\, \mathsf{C}\, [er\colon q]$ denotes a *relaxed-manifest error* if:

(1) $p \equiv \kappa \wedge \mathsf{true}$ for some κ such that $\mathsf{nlocs}(\kappa) = \emptyset$, where $\mathsf{nlocs}(.)$ is defined below; and

(2) conditions 2-4 in Theorem 3.5 hold.

$\mathsf{nlocs}(\mathsf{emp}) = \mathsf{nlocs}(x \mapsto X) = \mathsf{nlocs}(X \mapsto V) \triangleq \emptyset \quad \mathsf{nlocs}(X \not\mapsto) \triangleq \{X\} \quad \mathsf{nlocs}(\kappa_1 * \kappa_2) \triangleq \mathsf{nlocs}(\kappa_1) \cup \mathsf{nlocs}(\kappa_2)$

Observe that S_5 is a relaxed-manifest error but not a manifest one, while S_4 is neither. In our experiments (§5), the relaxed condition allowed Pulse-X to find additional bugs in OpenSSL without introducing false positives. Note that Def. 3.6 describes sufficient *syntactic* conditions for relaxed-manifest errors (as with Theorem 3.5), rather than prescribing a *semantic* definition (analogous to Def. 3.2). This is because such syntactic conditions are easier to check algorithmically and are what we use in Pulse-X. Nevertheless, it is straightforward to formulate a semantic definition for relaxed-manifest errors.

3.4 Extending ISL to Support Memory Leaks

We next describe how we extend the ISL theory in Raad et al. [2020] to support memory leak detection. To this end, we assume a designated location, $\mathbf{a} \in \mathsf{Loc}$, that tracks all memory locations allocated thus far, as shown in Fig. 4. That is, we define states as $\mathsf{State} \triangleq (\mathsf{Loc} \rightharpoonup_{\mathrm{fin}} \mathsf{Val}) \cup (\{\mathbf{a}\} \rightharpoonup \mathcal{P}(\mathsf{Val}))$. Analogously, we extend the assertion language with the $\mathbf{a} \mapsto \mathbf{A}$ and $\mathsf{leaks}(\mathbf{S}, L)$, where we use $\mathbf{A}$ and $\mathbf{S}$ as meta-variables for sets of logical variables. Specifically, the $\mathbf{a} \mapsto \mathbf{A}$ denotes that the set of locations allocated thus far (at $\mathbf{a}$) is given by $\mathbf{A}$. We describe $\mathsf{leaks}(\mathbf{S}, L)$ shortly below.

Accordingly, we adapt the predefined summaries for memory allocation and disposal to track allocated locations, as shown at the bottom of Fig. 4. Specifically, the erroneous specifications (under *er*) for `malloc()` and `free(.)` remain unchanged (as in Fig. 3), while their normal specifications (under *ok*) are adapted as shown in Fig. 4 to additionally account for the set of allocated locations.

Detecting Memory Leaks. The leaks(S, L) assertion states that the location denoted by L is *leaked* in that it is not reachable from the starting points given by S. Intuitively, as we demonstrate shortly, we check for memory leaks at the end of each procedure call; as such, the starting points correspond to the *global program variables*, denoted by $\mathcal{G}$. That is, a procedure leaks a location L if L has been allocated (i.e., is tracked under $\mathbf{a}$) and is not reachable from $\mathcal{G}$ (i.e., leaks($\mathcal{G}, L$) holds). We then define the noLeaks assertion, denoting that no allocated location is being leaked, as follows:

$$\mathsf{noLeaks} \overset{\text{def}}{\Leftrightarrow} \exists \mathbf{A}.\ \mathbf{a} \mapsto \mathbf{A} \wedge \bigwedge_{L \in \mathbf{A}} \neg \mathsf{leaks}(\mathcal{G}, L)$$

To detect memory leaks, we insert `assert`(noLeaks) at the end of each procedure.[7] As such, if procedure `f()` leaks a location, this assertion fails, leading Pulse-X to report a memory leak for `f()`.

4 THE PULSE-X ANALYSIS ALGORITHM

Our Pulse-X program analysis finds a collection of ISL triples for a given procedure in a program, forming its *summary*. These summaries are then used to report errors, and to obtain (inductively) summaries for other procedures. We describe our Pulse-X analysis algorithm in terms of a proof search algorithm in ISL. We present it as a variation on predicate transformer semantics, one that generates pre-assertions (presumptions) on the way to producing a post-assertion (result). Following our presentation of the Pulse-X algorithm, we then describe our actual implementation of Pulse-X, how it differs from the idealised algorithm, and sources of false positives which arise from going outside the presumptions of the soundness theorem.

4.1 The Pulse-X Analysis Algorithm as Proof Search in ISL

Specification Tables. Our proof search algorithm carries around a *specification table*, T, that associates each instruction (i.e., a predefined instruction or a procedure call) inst with a set of ISL specifications. At the beginning of the algorithm, the specification table is only populated with the summaries of predefined instructions, and is extended incrementally with the procedure summaries along the way. As such, for each predefined instruction inst, the T(inst) is as given in Fig. 3; and for each function call `f()`, the T(`f()`) is of the form $[p_1]$ `f()` $[\epsilon_1 : q_1], \cdots, [p_n]$ `f()` $[\epsilon_n : q_n]$.

At the core of Pulse-X is the *evaluation function* Eval(p, C, T). Given a presumption p and a specification table T, the Eval(p, C, T) computes a set of tuples of the form (ϵ, m, q), such that:

If we extend p with the 'missing' resource m, then ϵ:q is a valid result of executing C on $p * m$.

This intuition is captured in the theorem below, where we write $T \models [p * m]\, \mathsf{C}\, [\epsilon : q]$ to denote that if the triples in T are valid ISL triples (see Def. 3.1), then the triple $[p * m]\, c\, [\epsilon : q]$ is also valid. We formalise the notion of Eval(p, C, T) shortly below.

Theorem 4.1 (Under-approximation Soundness). *For all* $p, \mathsf{C}, T, \epsilon, m, q$*:*

$$(\epsilon, m, q) \in \mathsf{Eval}(p, \mathsf{C}, T) \quad \textit{implies} \quad T \models [p * m]\, \mathsf{C}\, [\epsilon : q]$$

[7] Note that `assert`(noLeaks) does not fit the official syntax of the encoding of assert statements from earlier, because noLeaks is not a pure (heap independent) boolean. Instead of extending the syntax of booleans we can more simply regard `assert`(noLeaks) as a special command apart from the given instructions. Its semantics is that its *ok* relation sends a state to the same state if there are no leaks, and its *er* relation sends a state to the same state if there are leaks.

$$\mathsf{Eval}(p, \mathtt{skip}, T) \triangleq \{(ok, \mathsf{emp}, p)\}$$

$$\mathsf{Eval}(p, \mathtt{local}\ x.\mathsf{C}, T) \triangleq \left\{ (\epsilon, \exists Y.\, m, \exists Y.\, q') \,\middle|\, \begin{array}{l} (\epsilon, m, q) \in \mathsf{Eval}(y{\mapsto}Y * p \wedge Y{=}\mathsf{nil}, \mathsf{C}[y/x], T) \\ \text{and}\ \ q' = \textsc{prune}(y, Y, q) \end{array} \right\}$$
$$\text{where} \quad y, Y \text{ fresh in } p \text{ and } y \text{ fresh in C}$$

$$\mathsf{Eval}(p, \mathsf{C}_1; \mathsf{C}_2, T) \triangleq \{(\epsilon, m_1 * m_2, q) \,|\, (ok, m_1, q_1) \in \mathsf{Eval}(p, \mathsf{C}_1, T) \wedge (\epsilon, m_2, q) \in \mathsf{Eval}(q_1, \mathsf{C}_2, T)\}$$
$$\cup \{(er, m_1, q_1) \,|\, (er, m_1, q_1) \in \mathsf{Eval}(p, \mathsf{C}_1, T)\}$$

$$\mathsf{Eval}(p, \mathsf{C}_1 + \mathsf{C}_2, T) \triangleq \mathsf{Eval}(p, \mathsf{C}_1, T) \cup \mathsf{Eval}(p, \mathsf{C}_2, T)$$

$$\mathsf{Eval}(p, \mathsf{C}^\star, T) \triangleq \mathsf{Eval}(p, \mathsf{C}^{unrollings^*}, T) \qquad \text{where} \quad \mathsf{C}^{i+1} \triangleq (\mathtt{skip} + (\mathsf{C}; \mathsf{C}^i))$$

$$\mathsf{Eval}(p, \mathsf{inst}, T) \triangleq \{(\epsilon, m, b * f) \,|\, [a]\ \mathsf{inst}\ [\epsilon : b] \in T \wedge (m, f) \in \textsc{BiabDiscover}(p, a)\}$$

$$\mathsf{Eval}(p, \mathtt{assert}(\mathsf{noLeaks}), T) \triangleq \{(ok, \mathsf{emp}, p) \mid p \models \mathsf{noLeaks}\} \cup \{(er, \mathsf{emp}, p) \mid p \not\models \mathsf{noLeaks}\}$$

Fig. 5. The Eval function for pre/post discovery.

The* Eval *Function. We define the Eval function using the inductive rules shown in Fig. 5. These rules are designed in such a way that they maintain the result in Theorem 4.1, thereby ensuring the soundness of the Pulse-X analysis by construction. Note that these rules are an adaptation of ISL proof rules, oriented to a forwards-running symbolic execution. The Eval definition in the case of sequential composition ($\mathsf{C}_1; \mathsf{C}_2$) incorporates many of the key properties of the analysis. Specifically, we first execute the first command C_1 and generate its missing resource m_1. In the case where executing C_1 results in an error, the execution is halted (short-circuit semantics) and m_1 is returned as the missing resource. Otherwise, we continue with executing C_2 and generate its missing resource m_2, which is then combined with the missing resource of C_1 and returned as $m_1 * m_2$. To execute a path symbolically we use the sequential composition rule repeatedly, until the execution terminates normally, or we encounter an error (short-circuiting).

Note that the Eval definition of procedure calls (the inst case at the bottom of Fig. 5) uses the $\textsc{BiabDiscover}(p, q)$ function. This function is the biabduction notion from Calcagno et al. [2011].

$\textsc{BiabDiscover}(p, q)$ returns a set of pairs of the form (m, f) such that $p * m \models q * f$. That is, it abduces a frame f and anti-abduces a missing resource m. In the Eval rule for procedure calls at the bottom of Fig. 5, after abducing the frame f and anti-abducing the missing resource m via $\textsc{BiabDiscover}(p, q)$, the m is fed back as the missing resource (to be added to the presumption), while f is carried forward. The soundness of the procedure call rule follows from the frame and consequence proof rules of ISL. Note that our treatment of procedure calls is as in Calcagno et al. [2011], except that (1) we anti-abduce m while they abduce m; and (2) we abduce f while they anti-abduce f. This difference is due to the different direction of entailments in the premise of the rule of consequence in under-approximate reasoning of incorrectness logic (IL) and ISL, compared to the over-approximate reasoning of Hoare logic and separation logic (SL).

The rule for local variables utilizes a function $\textsc{prune}(y, Y, q)$ which takes program and logical variables and a symbolic heap as arguments, and returns a symbolic heap. Intuitively, it deallocates a cell denoted by y from q. The correctness property for $\textsc{prune}$ is that for some Y', the entailment $y{\mapsto}Y' * \textsc{prune}(y, Y, q) \wedge Y{=}\mathsf{nil} \models q$ holds. $\textsc{prune}$ can be implemented by simply stripping $y{\mapsto}-$ facts from a symbolic heap, and performing additional equivalence-preserving boolean simplifications if desired. The direction of the entailment in the correctness condition is a result of the reversed rule of consequence in IL, where shrinking the post is a sound operation.

As mentioned earlier, `assert(noLeaks)` is treated as a special instruction. It can be implemented by checking whether all locations in the symbolic heap are reachable from globals, where symbolic reachability takes pure facts into account. This has been a standard operation on symbolic heaps in SL program analyses [Distefano et al. 2006]. To be sound for bug catching, this calculation should be sound for concluding that there are leaks ($p \not\models \mathsf{noLeaks}$) – if the prover thinks there's an unreachable element in a symbolic heap, then there is in at least one concrete heap that satisfies it – while the prover can be complete but unsound for concluding that there are no leaks ($p \models \mathsf{noLeaks}$).

Lastly, the Eval rule for loops assumes a parameter, *unrollings*, for bounded loop unrolling, leading to a form of bounded model checking.

In order to convert the rules in Fig. 5 to ones that also bound paths as well as loop depths, we assume we are given another parameter, *width*, and accordingly adapt Eval so that it computes a sequence of tuples (ϵ, m, q) rather than a set. We then interpret the set comprehension and $\cup$ in the case of sequential composition so that the final sequence follows a "lexicographic" order of first selecting disjuncts resulting from C_1 in order, and for each of them collecting those resulting from C_2 second. Similarly, we interpret $\cup$ in the cases of non-deterministic choice so that it selects first from the disjuncts resulting from C_1 (thus favouring lower iteration counts when evaluating loops), then those from C_2, capping the length of the resulting sequence at a maximum of *width*.

Analogously, we interpret the set comprehension in the case of instructions (inst) to truncate after *width*, and assume the specification table and $\textsc{BiabDiscover}(p, q)$ give back sequences instead of sets. For brevity, we omit a more formal definition of width bounding and of the ordering implemented in the tool. Note that the order described briefly in the previous paragraph is the one implemented in Pulse-X and is but one of several possible; a more comprehensive exploration of the benefits of various orderings is left to future work.

Remark 1. Our analysis algorithm is strikingly simple compared to the original biabductive analysis from Calcagno et al. [2011]. In particular, a key source of simplification relates to the handling of loops: to over-approximate loops, the original biabductive analysis had to seek fixed-points (i.e., to guess loop invariants). To describe loop invariants for linked structures, inductive definitions of predicates in one form or another are generally needed, and the treatment of inductive predicates is a challenging issue in over-approximate analysis for both (abductive) theorem proving and abstract semantics of instructions. In contrast, for our under-approximate analysis, we can avoid the issue entirely: our abstract domain does not include inductive predicates (apart from a special one for checking leaks), and we need not seek loop invariants because we are merely seeking to prove the existence of (finite) buggy executions. Instead, we use a simple fragment of the symbolic heaps used in separation logic analyzers which includes $\mapsto$ and $\not\mapsto$ assertions and pure boolean conditions, but not e.g., predicates to describe unbounded lists or trees. There are potential advantages to including inductive definitions to describe (backwards) loop variants, and that is a direction for research in the future, but we can use the simple method of loop unrolling and get useful results without needing to invent sophisticated abstract domains. So, an under-approximate biabductive analysis has a lower technical startup cost than an over-approximate one, and yet we will see in the next section that this low-startup-cost analysis nonetheless delivers comparable and often better practical results than analyses with higher startup costs.

A second source of simplification is the description of the analysis via the Eval function. The idea that hypotheses discovered during execution are sent back to the precondition is crystallized in the sequencing rule, which is nothing but a derived inference rule. So soundness of the analysis is obvious: the analysis is simply doing proof search using straightforwardly derivable inference rules. In contrast, the original biabductive analysis was presented in two forms – a denotational style from Calcagno et al. [2009] and a worklist presentation from Calcagno et al. [2011] – neither of which

had as simple a presentation or as direct a connection to proof theory. Our approach using Eval is related to the analysis description in Raad et al. [2020], but makes one further simplification in avoiding having a presumption/result pair describing the computation "up to now" as a parameter, letting the sequencing rule do the whole job.

These comparisons to the original carry over as well to other biabductive analyses that have appeared in the intervening years, including Le et al. [2014] and Fragoso Santos et al. [2020].

Remark 2. Our approach of using *width* and *unrollings* parameters for under-approximate bounding may be regarded as simple-minded in the extreme: we simply cut after a certain number of loop iterations or paths are explored. This under-approximation is sound in Incorrectness Logic, but different choices could be made too. For instance, dynamic testing tools, especially fuzzers, often employ more sophisticated exploration strategies, based on the observation that testing is a search problem [Harman et al. 2015]. Such strategies can also be used in under-approximate static tools such as Pulse-X. However, simple-minded can be very useful (especially in the early studies for an approach), as it provides baselines for future work to build upon, as well as a better understanding of the potential benefits a new approach has that are not due to more sophisticated strategies. Our goal therefore is to keep our approach simple in design and implementation, and to resist the urge to bring in more sophisticated search strategies, in order to gain insights and understanding. We were pleasantly surprised, in fact, that this simple bounding furnished as good experimental results as it did – see §5 – and expect that there is potential for going quite a bit further.

Generating Procedure Summaries. As mentioned above, our analysis algorithm begins with a specification table T that is initially only populated with the predefined summaries in Fig. 3, and extends T along the way with procedure summaries. To do this, given a procedure `f() = C`, we extend T by assigning the specifications in $\mathsf{Eval}(\mathsf{emp}, \mathtt{C}, T)$ to $T(\mathtt{f()})$. That is, we start with the empty state emp, thus assuming nothing about the initial state. Although our formalism focuses on parameterless procedures, it is straightforward to adapt it to account for parameters: rather than emp, we would then begin with $\overrightarrow{y \mapsto Y}$, where $\overrightarrow{y}$ denote the procedure parameters and $\overrightarrow{Y}$ denote the logical variables recording their initial values. We illustrate the analysis via the following example.

Example 4.2. We show how Pulse-X infers `foo()` summary in Fig. 6 and finds a null-pointer error (NPE). Specifically, `set(y,v)` dereferences the heap cell whose address is the content of the pointer `y` and updates its content to `v`. The NPE arises as follows: (1) on line 3, `malloc()` is called and x is assigned to an allocated heap cell; (2) x is dereferenced on line 4, assigning `nil` to its content; and (3) `set(`x`,1)` is called; as the content of x is `NULL`, this leads to a NPE.

Pulse-X algorithm infers summaries for `set(`y`,`v`)` and then `foo()`. Procedure `set(`y`,`v`)` uses two parameters, which are excluded from our formalism, but will help illustrate the workings of the algorithm. As discussed above, we simply record the initial value of y and v via $y \mapsto Y$ and $v \mapsto V$, respectively. Initially, the Pulse-X algorithm starts with the pre-state $y \mapsto Y * v \mapsto V$. As z is a local variable, it applies the local inference rule via Eval (Fig. 5) to obtain σ_0. Note: (1) for now, ignore the highlighted assertions in σ_0: they represent the anti-abduced assertion m, which will be computed by subsequent analysis steps; (2) to simplify the presentation, and as z and Z are distinct from other variables, we reused z and Z and did not replace z with a fresh variable.

On encountering the memory read on line 8, Pulse-X uses its predefined summary in Fig. 3 via Eval (Fig. 5). This in turn calls the biabductive procedure, BiabDiscover, to infer the frame f and the missing resource m. More concretely, Pulse-X uses the following *ok* summary of memory lookup (repeated from Fig. 3):

$$[z \mapsto Z * y \mapsto Y * Y \mapsto W]\ z := [y]\ [ok\colon z \mapsto W * y \mapsto Y * Y \mapsto W]$$

1: void foo(){
[emp ∧ true]
2: local x;
[x↦X ∧ X=nil]
3: x := malloc(); assume(x != NULL);
σ_3 ≡ [ok: ∃L. x↦L∗L↦V ∧ X=nil∧L≠nil]
4: [x] := NULL;
σ_4 ≡ [ok: ∃L. x↦L∗L↦nil ∧ X=nil∧L≠nil]
5: set(x, 1);
σ_5 ≡ [er: ∃L. x↦L∗L↦nil ∧ X=nil∧L≠nil]}

6: void set(y,v){
[y↦Y∗v↦V∗ Y↦W ∧ W=nil]
7: local z;
σ_0 ≡ [y↦Y∗v↦V∗z↦Z∗ Y↦W ∧ Z=nil ∧ W=nil]
8: z := [y];
σ_1 ≡ [ok: y↦Y∗v↦V∗z↦W∗Y↦W∧Z=nil∧ W=nil]
9: [z] := v;
σ_2 ≡ [er: y↦Y∗v↦V∗z↦W∗Y↦W∧Z=nil∧W=nil]
10: }

Fig. 6. An example illustrating how Pulse-X generates procedure summaries.

and poses the biabductive query BiabDiscover(y↦Y∗v↦V∗z↦Z∧Z = nil, z↦Z∗y↦Y∗Y↦W), which yields $\{(m, f)\}$ with $m \equiv Y{\mapsto}W$ and $f \equiv v{\mapsto}V \wedge Z = \text{nil}$. m is used to compute the pre-assertion and it is sent back to the beginning of the procedure, as denoted by the highlighted assertion $Y{\mapsto}W$, while f is combined with the post-condition of $z := [y]$ to obtain σ_1 (in Fig. 6). The other *er* specifications for $z := [y]$ are applied similarly and result in two additional (*er*) disjuncts being analysed.

Similarly, for $[z] := v$ on line 9, Pulse-X uses the following *er* summary of memory store in Fig. 3:

$$[z{\mapsto}W \wedge W = \text{nil}]\ [z] := v\ [er\colon z{\mapsto}W \wedge W = \text{nil}] \tag{1a}$$

poses the query BiabDiscover($y{\mapsto}Y * v{\mapsto}V * z{\mapsto}W * Y{\mapsto}W \wedge Z = \text{nil},\ z{\mapsto}W \wedge W = \text{nil}$), yielding $\{(m, f)\}$ with $m \equiv \text{emp} \wedge W = \text{nil}$ and $f \equiv y{\mapsto}Y * v{\mapsto}V * Y{\mapsto}W \wedge Z = \text{nil}$. m is for computing the pre-condition, and f is combined with the post-condition of $[z] := v$ to obtain σ_2 (shown in Fig. 6). The other specifications for $[z] := v$ are applied similarly and result in two additional disjuncts being analysed.

Finally, Pulse-X infers the following summary based on the precondition and σ_2.

$$[y{\mapsto}Y * v{\mapsto}V * Y{\mapsto}W \wedge W = \text{nil}]\ \texttt{set(y,v)}\ [er\colon y{\mapsto}Y * v{\mapsto}V * Y{\mapsto}W \wedge W = \text{nil}] \tag{1}$$

Similarly, it generates four other summaries for `set(y,v)` as follows.

$$[y{\mapsto}Y * v{\mapsto}V \wedge Y = \text{nil}]\ \texttt{set(y,v)}\ [er\colon y{\mapsto}Y * v{\mapsto}V \wedge Y = \text{nil}] \tag{2}$$

$$[y{\mapsto}Y * v{\mapsto}V * Y{\not\mapsto}]\ \texttt{set(y,v)}\ [er\colon y{\mapsto}Y * v{\mapsto}V * Y{\not\mapsto}] \tag{3}$$

$$[y{\mapsto}Y * v{\mapsto}V * Y{\mapsto}W * W{\mapsto}W']\ \texttt{set(y,v)}\ [ok\colon y{\mapsto}Y * v{\mapsto}V * Y{\mapsto}W * W{\mapsto}V] \tag{4}$$

$$[y{\mapsto}Y * v{\mapsto}V * Y{\mapsto}W * W{\not\mapsto}]\ \texttt{set(y,v)}\ [er\colon y{\mapsto}Y * v{\mapsto}V * Y{\mapsto}W * W{\not\mapsto}] \tag{5}$$

(2) and (3) (resp. (4) and (5)) correspond to the two disjuncts generated after line 8 (resp. line 9).

Pulse-X processes the statements of `foo()` similarly, where it uses the following summary of `x := malloc()` in Fig. 6: $[x{\mapsto}X]\ x := \texttt{malloc()}\ [ok\colon \exists L.\ x{\mapsto}L * L{\mapsto}V \wedge L \neq \text{nil}]$.

Note that, upon calling `set(x,1)` on line 5 from state σ_4, of the above five summaries, only (1) is applicable, for which the biabductive procedure returns the output σ_5, as shown in Fig. 6. As such, σ_5 is returned as the error post-condition, in accordance with the post-state of (1). In particular, Pulse-X infers the following *er* specification: $[\text{emp}]\ \texttt{foo()}\ [er\colon \exists L.\ L{\mapsto}\text{nil} \wedge L \neq \text{nil}]$.

Note that the above triple denotes a manifest error as it satisfies the four conditions of Theorem 3.5: (1) its precondition is emp ∧ true; (2) its postcondition is satisfiable i.e., sat(∃L. $L{\mapsto}$nil ∧ $L \neq$ nil) holds; (3) locs($L{\mapsto}$nil) = {L} and L is existentially quantified; and (4) sat($L \neq$ nil) holds. In the implementation, Pulse-X also generates an error trace that is based on the chain of function calls

(e.g., `foo();set(x,1)`) and their corresponding triples. If the trace, like the one in this example, includes a null-dereference *er* triple e.g., (1a), the error is an NPE.

In the cases of (2), (3), (4) and (5), the biabductive procedure returns an empty set, as their preconditions are not compatible with σ_4. For instance, the biabductive query generated from the precondition of (2) (after renaming variables) and σ_4 is as follows:

$$\textsc{BiabDiscover}(\exists L.\ \underline{x{\mapsto}L} * z{\mapsto}Z * \underline{L{\mapsto}\mathsf{nil}} * X\not\mapsto \wedge\ \underline{L \neq \mathsf{nil}},\ \underline{x{\mapsto}X} * v{\mapsto}1 * z{\mapsto}Z \wedge \underline{X = \mathsf{nil}})$$

As the underlined sections show that the $x{\mapsto}X \wedge X = \mathsf{nil}$ in the precondition of (2) is incompatible with $x{\mapsto}L * L{\mapsto}\mathsf{nil} \wedge L \neq \mathsf{nil}$ in σ_2 (they cannot be reconciled), this query returns the empty set.

4.2 Implementation Notes

Our implementation of Eval does not use proof search *per se*. Rather, it uses a worklist algorithm which maintains a set of (ϵ, m, q) triples at each program point [Jhala and Majumdar 2009]. Intuitively, m denotes the 'missing' resource that is to be sent back to the start of the procedure, and q denotes states that can be reached from the combination of m and the assumed function precondition. The relationship between the predicate-transformer presentations (e.g., Eval in Fig. 5) and the worklist descriptions is the same as in standard forwards program analysis, except for the presence of the additional component m.

Given an implementation of Eval, we take several further steps to obtain Pulse-X:

- We check the validity of predicate noLeaks at the end of each procedure, to detect memory leaks (see §3). This gives the effect of inserting `assert`(noLeaks) at the end.[8]
- We apply Eval to all procedures in a codebase, such that Eval is applied to callees before their callers. For brevity, we do not formalise this ordering as it is standard; however, we note that the analysis of callees before callers can be done lazily ("on-demand") as follows. When the summary for a callee procedure `f()` is needed for the analysis of the current procedure `g()`, if the summary of `f()` is already available in the specification table T, (i.e., `f()` has already been analysed) then we simply fetch it; otherwise, we pause the analysis of `g()`, generate the summary of `f()` and store it in T, and subsequently resume the analysis of `g()`.
- Once all procedure summaries have been generated and stored in the specification table, we examine the error specifications to determine which errors to report. This is a two-stage process. In the first stage, we filter out those null-pointer errors (NPEs) that are not manifest or not associated with `main()`, as well as those errors that have already been reported (our summaries are associated with metadata indicating their "reported" status). In the second stage, we report the remaining errors and update their metadata to indicate that they have been reported, so as to report errors as soon as we have ascertained they are manifest and to avoid reporting them again at procedure call sites.
- All this work is done at the level of an intermediate language with a front-end that maps C/C++ to the intermediate language. This front-end is provided independently by Infer-the-platform.

Finally, we note that our Pulse-X implementation departs in several ways from its formalism and from C/C++ semantics. In particular, our implementation handles calls to unknown procedures (e.g., library functions for which the code is not available, or calls to unresolved function pointers) by treating them as if they had no effect on the state (i.e., as `skip`), except that the return value and pointer arguments are assigned arbitrary values. This is not accounted for in our formalism. We also interpret arithmetic formulae using a theorem prover for rationals, rather than the fixed-precision

[8] We distinguish between a version of `assert`() known internally to the analysis and that of C. Programmers sometimes use `assert`() or `abort`() to indicate when they do not want to be warned, and we treat `abort` this way for OpenSSL, as in §2.1.

integers (and floats) of the machine. Both of these departures open up the possibility of real-world false positives, an issue we revisit in the next section.

5 EVALUATION

The goal of our experiments is to investigate the practical applicability of our approach, which has two primary subgoals: 1) to show that our bug reporting criterion is effective at identifying some interesting bugs without drowning that signal in many false alarms, and 2) to demonstrate that our approach scales to large codebases. For the first experiment, we ran Pulse-X on OpenSSL, a widely used implementation of the SSL/TLS protocols as well as a general-purpose cryptography library, and compared the accuracy of bugs found to that of Infer. Infer had previously reported bugs on OpenSSL-1.0.1h back in 2015 [O'Hearn 2015], so this provided an additional point of comparison, making OpenSSL a natural choice for this part. For the second experiment, we ran Infer and Pulse-X on a variety of publicly-available open-source C and C++ projects, and compared their runtimes. We conducted all our experiments on a Linux machine with 24 cores.

To study accuracy we follow Distefano et al. [2019] in looking at fix rate – the proportion of reports that have been fixed – in our evaluation. The fix rate concept is often used to judge the effectiveness of a deployment of an analysis in CI but, while Pulse-X is not deployed in this way, we can use the fix rate concept by looking at a legacy rather than new version of a codebase.

In particular, our evaluation is testing the following hypotheses.

- **Hypothesis H1**. On OpenSSL-1.0.1h Pulse-X has a superior fix rate to the present-day Infer.
- **Hypothesis H2**. Pulse-X finds new bugs worth fixing in current OpenSSL.
- **Hypothesis H3**. Pulse-X is broadly comparable with Infer in terms of performance, while reporting a comparable number of bugs.

Old Bugs. We consider OpenSSL-1.0.1h, the legacy version of OpenSSL from 2015. This project has 2.83 millions of bytes of IR code (MBoC) with 8,658 procedures and it took Pulse-X 4 minutes (CPU running time) to analyse. The results of these experiments are as follows:

- Pulse-X found 26 bugs, 19 of which had been fixed, including 9 manifest NPEs, 3 relaxed-manifest NPEs and 7 memory leaks, fix rate is 73%(= 19/26). Moreover, the unfixed issues involve either calls to unknown functions for which the code is unavailable or buggy procedures that no longer exist in OpenSSL-3.0.0-147ed5f (commit 147ed5f).
- The present-day Infer discovered 39 fixed bugs and 41 unfixed issues, for a fix rate of 49%(= 39/80). 8 of the 39 fixed bugs overlap with those found by Pulse-X. The other 41 issues flagged by Infer were classified as latent errors by Pulse-X and so were not reported by Pulse-X.

These results confirm hypothesis **H1**.

From **H1** we cannot conclude that Pulse-X is "better" than Infer. As is often the case [Pendergrass et al. 2013], the two analysis tools find different bugs: there are true (fixed) bugs found by each tool that the other misses. Rather, **H1** provides some evidence (scoped to OpenSSL) that Pulse-X's fix rate is not so bad as to rule out a CI deployment like Infer's (**H3** provides similar for performance).

Our reporting criterion filters the error specifications, reporting only some of them; we comment on the effect of this filtering, in light of this experiment. As we saw in §2.3, reporting *all* error specifications as alarms is obviously a bad idea. It would make Pulse-X report any time an address that was not allocated locally is accessed for the first time, due to the error specifications for load and store instructions that have invalidated assertions in their preconditions. To avoid obvious false positives, one might then think of reporting alarms whenever 1) there is no invalidated assertion in the precondition, and 2) if the error is due to a null pointer `X`, then `X=0` is not part of the precondition either. The latter is to say: we have not assumed a pointer was null, only to then complain about it

being null. This is a subset of the total number of error specifications, but still much greater than the manifest specifications. Pulse-X found 280 of these latent issues, more than 23 times the number of manifest issues reported. Upon inspection, most of these seem to be false positives: They are true specifications but might be considered as issues that are unlikely to be triggered in a global program. Indeed a high number of the 280 have not been fixed (like the one shown in §2.3).

So, the reporting criterion suppresses reports of issues which have not been fixed, and this is a positive indication. But, the manifest criterion is purposely extremely conservative, reporting NPEs only when they can occur in *all* contexts. It might be that some of the suppressed latent issues can be worth fixing, and we don't claim our criterion to be the ultimate one (perhaps there is no ultimate). Indeed, Listing 6 shows a bug in function `chopup_args` that had been fixed by OpenSSL developers and is classified as latent by Pulse-X. On line 6, the function allocates new memory (by calling `OPENSSL_malloc`) and assigns the resulting address to `arg->data`. As the allocation might fail, `arg->data` could point to `NULL`. Subsequently, the dereference on line 9 might cause an NPE. Pulse-X classified this issue as latent because this error occurs only when the condition on line 4 holds. Interestingly, that is the case in a call site to this function in `main`, as shown in Listing 7, which may be why, when Infer reported this bug to the developer in 2015, it consequently got fixed. In theory, Pulse-X could report this bug at the call site of `chopup_args` in `main` as a manifest issue. However, this call is only reachable after 20 conditional statements and 2 loops, and thus the disjunct limit was hit before that bug could be found.

```
1  int chopup_args(ARGS *arg, char *buf,
       int *argc, char **argv[]) {
2    int num,i;
3    ...
4    if (arg->count == 0) {
5      arg->count=20;
6      arg->data=(char**)OPENSSL_malloc(
           sizeof(char *)*arg->count);
7    }
8    for (i=0; i<arg->count; i++)
9      arg->data[i]=NULL;
10   ...
11 }
```

Listing 6. Latent error in chopup_args.

Comparing with the 15 bugs found by Infer and fixed in 2015, Pulse-X reported 4 overlapping fixed bugs, and present Infer discovered 4. Of the 11 fixed bugs not reported by Pulse-X, 5 were null dereferences found by Pulse-X but that were classified as latent. We are unsure why the remaining fixed leaks were not discovered, but it might have to do with the order of paths being processed. We are also unsure why the present-day Infer does not report 11 of the fixed bugs. The main lesson we can take from these numbers is that there might be room to loosen our reporting criterion, though this would have to be done in a way that avoids introducing bugs deemed not-worth-fixing.

```
1  int main(int Argc, char *ARGV[]) {
2    ...
3    arg.count=0;
4    ...
5    if (!chopup_args(&arg,buf,&argc,&
         argv)) break;
6    ...
7  }
```

Listing 7. Manifest error in `main` of openssl.c.

New Bugs. Given the positive results for Pulse-X on legacy OpenSSL, we decided to run it on the current beta OpenSSL-3.0.0-147ed5f, in case it found new bugs that would be worth fixing.

OpenSSL-3.0.0-147ed5f contains 8.55 MBoC with 22,979 procedures and it took Pulse-X 16 minutes (CPU running time) to analyse. Often, when judging the accuracy of an analysis, researchers use the concepts of true and false positive rates. While meaningful in theory, judging whether a bug is

a true or false positive can be error-prone and time-intensive. Below we speak of "thought-true" and "thought-false" bugs, to emphasize the (fallible) human judgement involved.

Pulse-X discovered 30 issues on 3.0.0. Manual inspection by us revealed what we thought were 15 real bugs (7 manifest NPEs, 1 relaxed-manifest NPE, and 7 memory leaks), as well as 5 maybe and 10 likely false positives. We then submitted the 15 thought-true bugs to the OpenSSL maintainers in two pull requests (https://github.com/openssl/openssl/pull/15834 and https://github.com/openssl/openssl/pull/15910). The maintainers agreed that all 15 bugs were legitimate, and our patches are now in the OpenSSL codebase (see the non-anonymous supplementary materials). We take the information that the OpenSSL maintainers judged these issues worth fixing as confirmation of hypothesis **H2**.

We discuss one false positive reported by Pulse-X in `ossl_do_ex_data_init` (Listing 8). This function assigns the pointer address of a field of `ctx` to variable `global` via the function call on line 2. The call on line 2 invokes three other functions, `pthread_getspecific`, `pthread_rwlock_init`, `pthread_once` whose code is unavailable, and so are unknown to Pulse-X. Our analysis replaces the result of an unknown function by a fresh logical variable, and as a consequence the analysis assumes that `global` points to a local heap cell instead of a global pointer. This leads Pulse-X to report a memory leak as a consequence of the allocation on line 4.

```
1  int ossl_do_ex_data_init(OSSL_LIB_CTX *ctx){
2      OSSL_EX_DATA_GLOBAL *global =
           ossl_lib_ctx_get_ex_data_global(ctx);
3      if (global == NULL) return 0;
4      global->ex_data_lock =
           CRYPTO_THREAD_lock_new();
5      return global->ex_data_lock != NULL;
6  }
```

Listing 8. Pulse-X false positive.

If we were to model `pthread_getspecific`, `pthread_once`, and `pthread_rwlock_init` then we could avoid this specific issue. But there are very many libraries, some not yet created, which would have to be modelled. Another way to address this issue would be to suppress any report that involves unknown code.[9] This would stamp out all the false positives observed with OpenSSL, and we could trumpet that we had obtained no false positives on OpenSSL, but it would also remove true positives: 5 of the 15 true bugs fixed in 3.0.0 involved unknown functions. Rather than artificially move to a position where we can blow that trumpet, we prefer to present the realistic situation (which arises from examples going beyond assumptions of theory) and not make a value judgement on whether it would be better to stamp out these false positives at the expense of false negatives; this is, in the end, an engineering rather than a scientific judgement, and we would prefer to be able to provide this data to engineers to help inform their judgement.

A run of Infer found 116 issues on 3.0.0, 7 of which overlap with those fixed bugs found by Pulse-X. We do not know exactly how many more of the Infer issues would be considered worth fixing by OpenSSL maintainers, but we noticed over 40 likely false positives and did not pursue the matter further. We emphasize that we are not making any comparative claim w.r.t. Infer here—our hypothesis H2 only mentions Pulse-X—and these numbers are given just for information.

We also remark that Coverity scan, scan.coverity.com, runs regularly on OpenSSL and does result in bug fixes [10]; as Coverity is proprietary, we do not know whether it finds the issues that Pulse-X does, but the 15 reported Pulse-X-issues had not been previously fixed at the time of our tool run.

[9] Interesting subtler approaches to unknown code have been investigated, a good example being [Das et al. 2015]; but, to our knowledge, no definitive solution has emerged and further research is warranted.

[10] https://scan.coverity.com/projects/tpm2-openssl

Scale. To test H3, we studied the scalability of Pulse-X and Infer on 10 large programs—including OpenSSL-1.0.1h, OpenSSL-3.0.0-147ed5f, wdt, bistro, SQuangLe, RocksDB, FbThrift, OpenR, Treadmill, and Watchman—measuring CPU running time and counting the number of issues reported by Infer and Pulse-X. The sizes of these projects range from 2.83 to 407 millions of bytes of IR code. Note that we used proprietary build targets for C++ projects. In the following, we give a brief summary. The experiments show that the running time is from 4 to 900 minutes and the number of bugs found is from 0 to 136 bugs. We also find that Pulse-X and Infer's performance are comparable, as are the number of issues both tools report. These results confirm hypothesis **H3**.

Threats to validity. Threats to validity correspond to the generalizability of our findings. Our study evaluates C/C++ projects over various kinds of applications and two kinds of bugs: NPEs and memory leaks. Still, this may not represent all kinds of projects and bugs and thus may affect our study's generalizability. Hence, the experimental evidence in this section should be interpreted in the appropriately cautious way: we do not claim that these results will carry over to arbitrary other codebases (as would be impossible to conclude from any finite experiments).

6 RELATED WORK

Our work builds directly on the biabductive compositional analysis of Infer [Calcagno et al. 2011], on Incorrectness Logic [O'Hearn 2019], and on Incorrectness Separation Logic (ISL) [Raad et al. 2020]. In their work introducing ISL, Raad et al. also presented an intra-procedural analysis. Our work goes beyond that of Raad et al. [2020] in four main ways. First, we formulate an inter-procedural analysis, using Eval (our predicate transformer-style semantics of Pulse-X) for summary inference, with an implemented inter-procedural analysis algorithm. Second, we provide an account of memory leaks. Third, we provide an experimental evaluation which is unusual and novel in style (based on fix rate instead of guessed false positives). Lastly, and most significantly, we propose a formally rigorous approach to the problem of compositional bug reporting via latent versus manifest errors.

More recently, Raad et al. [2022] introduce Concurrent Incorrectness Separation Logic (CISL) as an under-approximate analogue of Concurrent Separation Logic [OHearn 2007]. CISL is a general theoretical foundation for detecting *concurrency* bugs such as data races and deadlocks. As of yet, CISL is a theoretical work and the authors have left the development of bug detection tools underpinned by CISL to future work.

Dillig et al. [2012] develop a semi-automatic approach for whole-program bug reporting. Given a latent error, their approach infers constraints that capture the information their analysis is missing. These constraints are then presented to the user who can either discharge the error (false positive) or validate it (true positive). In contrast, our focus is on the development of an automatically checkable bug reporting criterion, which we believe is essential for scalability and practical deployment.

Long before the work on IL began, summary-based under-approximate analysis was studied in the context of symbolic execution by Godefroid and others [Godefroid 2007; Godefroid et al. 2010]. Godefroid et al. [2010] use an analogue of the under-approximate triple referred to as a must$^-$ transition [Ball et al. 2005]. Summaries were used in these works in an effort to speed up a global program analysis, and ultimately one falls back on dynamic analysis for soundness. These works present experimental results for device drivers up to just over 30k LOC, with running times in the hundreds of minutes, whereas we tackle programs in the hundreds of thousands of LOC. Summary-based analyses were subsequently implemented in the SAGE and PEX tools used in production at Microsoft, but are not used by default widely in their deployments, because other techniques were found which were better for fighting path explosion (Godefroid, personal communication). Note that we do not use summaries principally for efficiency (though they help), but rather as part of a begin-anywhere analysis which can be applied to partial programs (and thus

fits well with a CI deployment for analysing pull requests). Although we believe the authors of these works could have, technically, run the symbolic part of their analysis on partial programs, they did not develop an approach for compositional reporting as we do, and this would have been essential for automated deployment without the human intervention to filter reports.

Another direction in symbolic execution is "under-constrained" analysis, which uses symbolic execution to analyse individual functions without going back to `main()`. Representative tools are JPF [Khurshid et al. 2003], UC-KLEE [Ramos and Engler 2015], Sys [Brown et al. 2020], and JPF-Star [Pham et al. 2019]. Instead of abduction, these works use lazy initialisation [Khurshid et al. 2003] to infer procedure preconditions and use the preconditions for test case generation. These works are not summary-based, but do have the characteristic of Infer where execution can start from anywhere in a program.

The authors of UC-KLEE emphasise that the ability to run on individual procedures, without the complete program, opens up new possibilities for applying symbolic execution to real code, since execution on a global program might not reach many of the functions. When run on an OpenSSL target [Ramos and Engler 2015], UC-KLEE found five true memory leak bugs with 267 false positives (excluding 269 false positives in ASN.1 sub-library). When run on OpenSSL-1.0.1h, Pulse-X found seven leak bugs which were fixed (so we consider them true positives) with seven false positives. This is a rough comparison as the two tools were not run on exactly the same OpenSSL version. UC-KLEE uses a number of heuristics to rein in false positives, and there appears to be some similarity with the concept of manifest; e.g., the must-fail heuristic "identifies errors that must occur for all input values"; in contrast, manifest errors *may* occur for all input values. The heuristics are not defined precisely by Ramos and Engler [2015], making a precise comparison difficult, and numbers are given for heuristics other than must-fail but not must-fail itself. Their best-performing heuristic, belief-fail, led to a claimed true-positive ratio of 20%, considerably lower than what we have observed in our experiments (which is 73% for Pulse-X).

Sys [Brown et al. 2020] is a successor of UC-KLEE. Sys combines static analysis with under-constrained symbolic execution: it utilises static checkers to quickly uncover potential error sites and then runs under-constrained symbolic engines on these hot spots. Like Pulse-X, Sys finds real bugs in big (browser) code. In contrast, however, Sys does not describe a reporting criterion; rather, it uses what the Sys authors call "ad hoc, checkers-specific tricks" to rein in false positives.

Gillian [Fragoso Santos et al. 2020] is a separation logic reasoning platform that combines technologies for symbolic execution-based testing, verification, and biabductive compositional analysis. The compositional testing phase of Gillian currently works by using over-approximate rather than under-approximate summaries, but we expect it could be altered to follow our under-approximate approach. If so, given that Gillian is multi-language and parametric on the memory model of the target language, an expressive under-approximate reasoning platform might be obtained for C, JavaScript, and many other languages.

A distantly related line of work is harness generation for fuzzers (e.g., the work of Ispoglou et al. [2020]). When starting from `main()`, many parts of the code can be difficult to reach for a fuzzer. Harnesses are "fake `main` programs" which set up the environment and then call a collection of functions repeatedly. Automatic harness generation is similar in spirit to the precondition generation of Infer or UC-KLEE, but the harnesses (so far) can be more complex than preconditions. Harness generation can also suffer from generating spurious or infeasible fake `main` programs, which could uncover "bugs" that programmers regard as false positives. We are unsure whether harness generation and symbolic execution or compositional static analysis can affect one another by direct transfer of already-existing ideas; however, given their somewhat similar motivations, we expect some convergence in the future.

7 CONCLUSION

This paper has made steps towards realising the promise of Incorrectness Logic for defining *new* automated bug catching techniques, going beyond its ability to describe existing approaches. Our focus has been in a specific direction, compositionality, where program logics such as Hoare Logic and Separation Logic have had the most influence for correctness reasoning. This is not the only possible direction. New under-approximate abstractions and methods for synthesising (backwards) loop variants might also be approached, but those would be topics for other papers.

We defined a new compositional, under-approximate analysis, by merging ideas from compositional analysis based on Separation Logic [Calcagno et al. 2011] and Incorrectness SL [Raad et al. 2020]. This merging gives us sufficient information in the abstract semantics to reason soundly about the presence of bugs, and one of our main contributions is a rigorous reporting criterion which accesses this information, leading to a "true positives property" of compositional bug reporting on potentially incomplete program fragments. Analysis of incomplete program fragments (without a global program context) is relevant to the diff-time analysis reporting recently emphasised by Google and Facebook, but rests on theoretical definitions rather than heuristics alone.

A second main contribution is an experimental evaluation of the effectiveness of our method based on an implementation in a tool, Pulse-X. We compared to Infer, a state-of-the-art industrial program analysis. We found that Pulse-X is competitive on performance and, zooming in on OpenSSL, superior on accuracy (true/false bugs). We remark that Pulse-X is a scientific prototype that does not have the demonstrated industrial impact of Infer. Furthermore, there is no guarantee that our results on accuracy will carry over to arbitrary other codebases.[11] But, with these caveats, our results do provide some corroborative evidence of potential effectiveness.

The general point we wish to make is the positive potential of using sound under-approximate abstractions when reasoning compositionally. Beyond computing the abstractions, in this paper we used them to design bug reporting criteria. The criteria were validated theoretically as well as experimentally, using the concept of fix rate as a proxy for whether developers would consider the reports as true bugs. We certainly do not claim to have discovered the ultimate compositional reporting criteria in this paper but rather provided example criteria with associated experimental backing, criteria which we hope others will be motivated to improve, extend, or even replace.

ACKNOWLEDGMENTS

This research was supported in part by a UK's Engineering and Physical Sciences Research Council (EPSRC): Grant number EP/R006865/1, a UKRI Future Leaders Fellowship [grant no. MR/V024299/1], and a European Research Council (ERC) Consolidator Grant for the project "RustBelt", funded under the European Union's Horizon 2020 Framework Programme (grant agreement no. 683289).

REFERENCES

Ali Asadi, Krishnendu Chatterjee, Hongfei Fu, Amir Kafshdar Goharshady, and Mohammad Mahdavi. 2021. Polynomial reachability witnesses via Stellensätze. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 772–787. https://doi.org/10.1145/3453483.3454076

Thomas Ball, Orna Kupferman, and Greta Yorsh. 2005. Abstraction for Falsification. In *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3576)*, Kousha Etessami and Sriram K. Rajamani (Eds.). Springer, 67–81. https://doi.org/10.1007/11513988_8

Fraser Brown, Deian Stefan, and Dawson Engler. 2020. Sys: A Static/Symbolic Tool for Finding Good Bugs in Good (Browser) Code. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 199–216. https://www.usenix.org/conference/usenixsecurity20/presentation/brown

[11] Of course this is almost always the case with static analyses, which are tackling undecidable problems.

Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. 2009. Compositional Shape Analysis by Means of Bi-Abduction. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09)*. Association for Computing Machinery, New York, NY, USA, 289–300. https://doi.org/10.1145/1480881.1480917

Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2011. Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM* 58, 6, Article 26, 66 pages. https://doi.org/10.1145/2049697.2049700

Ankush Das, Shuvendu K. Lahiri, Akash Lal, and Yi Li. 2015. Angelic Verification: Precise Verification Modulo Unknowns. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9206)*, Daniel Kroening and Corina S. Pasareanu (Eds.). Springer, 324–342. https://doi.org/10.1007/978-3-319-21690-4_19

Edsko de Vries and Vasileios Koutavas. 2011. Reverse Hoare Logic. In *Software Engineering and Formal Methods*, Gilles Barthe, Alberto Pardo, and Gerardo Schneider (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 155–171.

Isil Dillig, Thomas Dillig, and Alex Aiken. 2012. Automated Error Diagnosis Using Abductive Inference. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 181–192. https://doi.org/10.1145/2254064.2254087

Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling Static Analyses at Facebook. *Commun. ACM* 62, 8 (jul 2019), 62–70. https://doi.org/10.1145/3338112

Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2006. A Local Shape Analysis Based on Separation Logic. In *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 3920)*, Holger Hermanns and Jens Palsberg (Eds.). Springer, 287–302. https://doi.org/10.1007/11691372_19

Facebook. 2021. https://fbinfer.com/

José Fragoso Santos, Petar Maksimović, Sacha-Élie Ayoun, and Philippa Gardner. 2020. Gillian, Part i: A Multi-Language Platform for Symbolic Execution. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 927–942. https://doi.org/10.1145/3385412.3386014

Patrice Godefroid. 2007. Compositional Dynamic Test Generation. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Nice, France) *(POPL âĂŹ07)*. Association for Computing Machinery, New York, NY, USA, 47–54. https://doi.org/10.1145/1190216.1190226

Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and SaiDeep Tetali. 2010. Compositional may-must program analysis: unleashing the power of alternation. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, Manuel V. Hermenegildo and Jens Palsberg (Eds.). ACM, 43–56.

Mark Harman, Yue Jia, and Yuanyuan Zhang. 2015. Achievements, Open Problems and Challenges for Search Based Software Testing. In *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*. IEEE Computer Society, 1–12. https://doi.org/10.1109/ICST.2015.7102580

Kyriakos K. Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. 2020. FuzzGen: Automatic Fuzzer Generation. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 2271–2287. https://www.usenix.org/conference/usenixsecurity20/presentation/ispoglou

Ranjit Jhala and Rupak Majumdar. 2009. Software Model Checking. *ACM Comput. Surv.* 41, 4, Article 21 (Oct. 2009), 54 pages. https://doi.org/10.1145/1592434.1592438

Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. 2003. Generalized Symbolic Execution for Model Checking and Testing. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Warsaw, Poland) *(TACAS'03)*. Springer-Verlag, Berlin, Heidelberg, 553–568.

Quang Loc Le, Cristian Gherghina, Shengchao Qin, and Wei-Ngan Chin. 2014. Shape Analysis via Second-Order Bi-Abduction. In *Computer Aided Verification*, Armin Biere and Roderick Bloem (Eds.). Springer International Publishing, Cham, 52–68.

Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter O'Hearn. 2022. Artifact and Appendix of Finding Real Bugs in Big Programs with Incorrectness Logic (supplementary material). https://doi.org/10.5281/zenodo.6342311

Bernhard Möller, Peter O'Hearn, and Tony Hoare. 2021. On Algebra of Program Correctness and Incorrectness. In *Relational and Algebraic Methods in Computer Science - 19th International Conference, RAMiCS 2021*.

Peter W. OHearn. 2007. Resources, Concurrency, and Local Reasoning. *Theor. Comput. Sci.* 375, 1-3 (apr 2007), 271–307. https://doi.org/10.1016/j.tcs.2006.12.035

Peter W. O'Hearn. 2015. https://mailing.openssl.dev.narkive.com/2DbkkYzD/openssl-org-3403-null-dereference-and-memory-leak-reports-for-openssl-1-0-1h-from-facebook-s-infer

Peter W. O'Hearn. 2019. Incorrectness Logic. *Proc. ACM Program. Lang.* 4, POPL, Article 10 (Dec. 2019), 32 pages. https://doi.org/10.1145/3371078

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings*. 1–19. https://doi.org/10.1007/3-540-44802-0_1

J.A. Pendergrass, S.C. Lee, and C.D. McConnell. 2013. Theory and Practice of Mechanized Software Analysis. In *Johns Hopkins APL Technical Digest, Volume 32, Number 2*. 499–508.

Long H. Pham, Quang Loc Le, Quoc-Sang Phan, Jun Sun, and Shengchao Qin. 2019. Enhancing Symbolic Execution of Heap-Based Programs with Separation Logic for Test Input Generation. In *Automated Technology for Verification and Analysis*, Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza (Eds.). Springer International Publishing, Cham, 209–227.

Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter O'Hearn, and Jules Villard. 2020. Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II* (Los Angeles, CA, USA). Springer-Verlag, Berlin, Heidelberg, 225–252. https://doi.org/10.1007/978-3-030-53291-8_14

Azalea Raad, Josh Berdine, Derek Dreyer, and Peter O'Hearn. 2022. Concurrent Incorrectness Separation Logic. *Proc. ACM Program. Lang.* 6, POPL (2022). Conditionally accepted.

David A. Ramos and Dawson Engler. 2015. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 49–64. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/ramos

Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from Building Static Analysis Tools at Google. *Commun. ACM* 61, 4 (March 2018), 58–66. https://doi.org/10.1145/3188720