Finding Real Bugs in Big Programs with Incorrectness Logic

This is a draft paper of August 2021 describing work that has not yet been published.

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

Recently, Incorrectness Logic (IL) has been advanced as a logical theory for proving the presence of bugs. Its proof theory supports a compositional approach to reasoning which mirrors the compositionality of Hoare logic, where knowledge that a bug exists can be inferred from local properties established independently for program parts. This promise of the compositionality of IL for bug catching is, however, so far unrealized: reasoning about bugs was done by hand, in a merely suggestive way, in the early work.

In this paper, we take a step towards realizing the promise of the compositional reasoning principles of IL as a foundation for practical program analysis. Specifically, we develop Pulse-X, a new, automatic program analysis for memory bugs based on ISL, a recent synthesis of IL and separation logic. To investigate the effectiveness of Pulse-X, we compare it to Infer, a related, widely used analyzer which has proven useful in industrial engineering practice, demonstrating how Pulse-X is competitive for speed and, when zooming in on a particular large program (OpenSSL), finds more true bugs with fewer false positives. Using Pulse-X, we have also found 15 new real bugs in OpenSSL, which we have reported to OpenSSL maintainers and have since been fixed.

1 INTRODUCTION

Incorrectness Logic (IL) [O’Hearn 2019] describes principles for reasoning about the presence of bugs as a dual to those of Hoare’s logic (HL) for reasoning about their absence. As well as being able to describe existing bug catching approaches, including traditional testing and symbolic execution, the proof theory of IL mirrors the compositionality of HL, where proofs of compound programs are built from (proven) specifications of its parts. In a retrospective on 50 years of Hoare logic, Apt and Olderog [2019] identify this compositionality as the principal reason for HL’s remarkable influence.

In this paper we take steps towards realising the potential of the IL compositionality, by developing an automatic program analyser, Pulse-X, which we apply to real programs. Our analyser is closely inspired by the Infer tool which is in deployment at Facebook, Amazon, Microsoft and other companies. In particular, Pulse-X adopts Infer’s approach to compositionality based on the biabductive proof principle [Calcagno et al. 2011]. However, rather than building on classic separation logic [O’Hearn et al. 2001] as Infer did, Pulse-X builds instead on the recent Incorrectness Separation Logic (ISL, [Raad et al. 2020]); and rather than reporting bugs based on failing to prove their absence, Pulse-X instead reports bugs based on positive proofs of their presence.

Program Analysis Context. Before describing our results, we recall some of the relevant program analysis context. The theory of static program analysis has traditionally most often adopted a global program perspective: analyses assume they are given the whole program to be analysed, compute an approximation (usually an over-approximation) of the program runs, and then use this information to issue bug reports [Cousot and Cousot 1977; Rival and Yi 2020]. Although there are notable exceptions (e.g., [Cousot 2001]), much less attention has been paid to the foundations of compositional static analysis. 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 the code review process. 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 globally on 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.

The Problem of Compositional Bug Reporting. One of the central problems in developing static analyses—particularly compositional analyses that are deployed automatically as part of CI—is figuring out when to report bugs to developers. This is difficult for several reasons.

First, the fast, compositional static analyses deployed in CI usually do not enjoy a "soundness for bug-catching" theorem. If they are sound for correctness (proving the absence of bugs) then any alarm they report will be the result of a failure to prove the program correct, and not of having found a bug with certainty. On the other hand, static analysis tools that are sound for incorrectness (proving the presence of bugs) have been developed, e.g., symbolic execution, but they usually work in a whole-program fashion and (thus far) have been too slow for CI on large codebases.

Second, there is the even more fundamental question of what constitutes a "bug". For global analyses, we can appeal to the classic distinction between true positives and false positives: a bug report is deemed a true positive if the bug can be exercised in some concrete execution of the program starting from a `main()` function or other specified entry points, and is otherwise deemed a false positive. On the other hand, compositional analyses only examine code snippets, not whole programs, so they must report bugs based only on local information about a snippet, without knowing the global program context in which it is used. In some cases such as libraries, an enveloping "whole program" simply does not exist. In the absence of such contextual information, however, it is not clear how to even define whether a bug reported by a compositional analysis is a true positive or not.

For example, a compositional analysis might deduce that a function `foo` is safe to execute when passed a nonzero argument, but that it will incur a memory safety error (e.g., null-pointer-dereference) when passed zero (0). Without knowing the calling contexts of `foo`, does this "bug" count as a true positive or not, and should it be reported?

In industrially deployed compositional analysis tools (such as those presented in the aforementioned CACM articles), the choice of when to report bugs is thus typically implemented using heuristics, which are refined over time based on empirical evaluation of developer feedback on the bug reports produced by the tool. Moreover, the success of these tools is measured not in "true positive rate" or "false positive rate" but rather in fix rate, namely the proportion of reported bugs fixed by developers. Fix rate is a useful empirical metric of the analysis effectiveness, but it leaves open a natural research question: Can we develop useful compositional bug-finding tools, and characterise their effectiveness, in a more formally rigorous way?  

1 The Google article [Sadowski et al. 2018] uses the term "effective false positive" for a report a developer does not fix, but this should not be confused with the established objective concept of false positive as a spurious bug claim.
Contributions. In this paper, we make a significant step forward in answering the above question. Specifically, we develop a compositional program analysis that is similar in spirit and deployability to existing, empirically effective analyses, but that (unlike those analyses) is also backed by rigorous foundations concerning the abstractions it computes and the bugs it reports.

We take as our starting point the original foundation of Infer [Calcagno et al. 2011], one of the analyses described in the Facebook paper. Infer’s use of bi-abduction provides a powerful technique for inferring compositional specifications for memory-manipulating procedures. However, as O’Hearn [2019] has noted, there is a fundamental mismatch between the logical foundation of Infer and its practical deployment. On the one hand, the theory of Infer is based on separation logic [O’Hearn et al. 2001], a logic of over-approximation—meaning that separation logic specifications prove the absence of bugs. On the other hand, Infer was deployed in practice as a bug catcher, for which we really want a logic of under-approximation—meaning that we want to prove the presence of bugs.

Our first step is thus to replace over-approximation by under-approximation: we obtain a compositional program analysis, called Pulse-X, which infers proofs much like the original Infer does, but they are proofs in Incorrectness Separation Logic (ISL) [Raad et al. 2020] rather than classic (over-approximate) separation logic. Pulse-X calculates a collection of under-approximate ISL triples of the form \([\text{presumption}] \ f \ [ \ e : \ \text{result}]\), which are interpreted as the dual of Hoare triples: they assert that any final state satisfying the result condition \(\text{result}\) is reachable (with exit condition \(\epsilon\)) by executing \(f\) starting in some initial state satisfying \(\text{presumption}\). Here, \(\epsilon\) can be either \(\text{ok}\) (indicating normal termination) or \(\text{er}\) (indicating erroneous termination, i.e., a program error).

By trading over- for under-approximation in the computed abstractions, we know that Pulse-X guarantees the existence of bugs under some conditions—our analysis infers error specifications of the form \([\text{presumption}] \ f \ [ \ e : \ \text{result}]\) for each function—but this leaves us with the key question of when to report them to developers. To this end, we describe conditions on \(\text{presumption}\) and \(\text{result}\) under which the error \(\text{er}\) is guaranteed to be context-independent, meaning intuitively that it does not depend on particular preconditions which may or may not hold in the calling context of \(f\). We show how to check these context-independence conditions algorithmically as part of our analysis, and thereby only report error specifications that satisfy them; we refer to such context-independent bugs as manifest bugs. On the other hand, we refer to non-manifest bugs as latent bugs. Even when we do not report them directly to developers, latent bugs nevertheless play an important role in bug detection and reporting: a latent bug of \(f\) may be used compositionally to compute manifest bug specifications for code that calls \(f\).

For example, for function \(\text{foo}\) mentioned earlier, Pulse-X would compute an error specification under the presumption that its argument is \(\theta\); however, this error specification is not context-independent, and thus it would be classified as a latent bug. If, however, another function \(\text{bar}\) were to (without context-dependent conditions of its own) call \(\text{foo}\) with argument \(\theta\), we would then report a manifest bug for \(\text{bar}\).

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

If a subprogram of a complete program has a manifest error, then either the subprogram is dead code (not reachable from \(\text{main()}\)) or there is a trace in the concrete semantics from \(\text{main()}\) 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. Interestingly, we also found that it makes sense to report some, though not all, latent bugs, in particular those for memory leaks; see §2.3 for discussion.
Standard Caveat. Our soundness theorem says that, under assumptions encapsulated in a formal model, the computed abstractions will be under-approximate and there will be 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-theory issues with unknown code, but this was the minority of findings (see §4.2 and §5). End of Standard Caveat.

Another key contribution we make is an experimental evaluation of Pulse-X, which indicates 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 for our evaluation because Infer had reported several bugs on OpenSSL back in 2015, which were fixed by OpenSSL developers, and thus it was natural to consider how Pulse-X would fare. We found that Pulse-X found more true bugs and fewer false ones than Infer. 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 a scientific experimental 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 industrial CI systems in the future.

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.

Pulse-X is an eXperimental version of the Pulse program analyzer being developed at Facebook, and was developed as a fork of Pulse. We’ll describe the relationship between Pulse-X and Pulse in more detail as part of the related work section, §6.

We plan to make Pulse-X together with our experiments publicly available at a later date, and before this draft paper is published. For the time being, certain supplemental material may be accessed at http://www0.cs.ucl.ac.uk/staff/p.ohearn/RealBigDraftSupplemental.txt; this includes our bug reports and links to pull requests for fixes against recent OpenSSL, as well as the Infer bugs discussed in the experimental section of the paper. Additionally, a technical appendix containing proofs ommitted from the main body of the paper is at http://www0.cs.ucl.ac.uk/staff/p.ohearn/RealBigDraftAppendix.pdf.

2 PULSE-X OVERVIEW AND REPRESENTATIVE BUG EXAMPLES

Rather than start with theory we begin with bugs, four of them drawn from running Pulse-X on OpenSSL. The first three are real bugs illustrating different challenges for compositional analysis and reporting, while the fourth 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 Bugs 1 and 2: Null Dereference Bugs

Listing 1 shows a null-pointer-dereference (denial-of-service) vulnerability we discovered in OpenSSL’s ssl_excert_prepend function. The lines starting with + denote our proposed patch. The code begins by calling app_malloc, which in turn calls CRYPTO_malloc in its body. The CRYPTO_malloc function is a malloc wrapper, i.e., a wrapper for a C-standard malloc, used throughout OpenSSL instead of a standard malloc. The returned pointer is set to 0 on line 6. Pulse-X found that the malloc wrapper could return NULL and thus a null-pointer-dereference may occur on line 6. 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. 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.

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

\[
\begin{align*}
&\text{[emp \land \text{true}]} \text{ app malloc(sz, what)} \quad \text{[ok: ret \rightarrow \text{nil} \land \text{true}]} \\
&\text{[emp \land \text{true}]} \text{ app malloc(sz, what)} \quad \text{[ok: \exists X. ret \rightarrow X \cdot X \rightarrow \rightarrow \land \text{true}]} \\
\end{align*}
\]

Note that we inform our Pulse-X analyser (using a flag) that `CRYPTO_malloc` is a malloc for OpenSSL, but then Pulse-X automatically discovers a procedure summary for `app_malloc` and many other `CRYPTO_malloc` wrappers.

---

**Listing 1.** A null pointer dereference error in OpenSSL.

```c
int ssl_excert_prepend(SSL_EXCERT **pexc) {
    SSL_EXCERT *exc = app_malloc(sizeof(*exc), "prepend cert");
    if (exc == NULL)
        return 0;
    memset(exc, 0, sizeof(*exc));
    ...
}
```

**Listing 2.** Part of error trace of the bug in Listing 1.

```c
apps/lib/s_cb.c:959: error: Nullptr Dereference
apps/lib/s_cb.c:957:23: in call to `app_malloc`
955. static int ssl_excert_prepend(SSL_EXCERT **pexc)
956. {
957.     SSL_EXCERT *exc = app_malloc(sizeof(*exc), "prepend cert");
         ^
958.     memset(exc, 0, sizeof(*exc));
959. test/testutil/apps_mem.c:16:16: in call to `CRYPTO_malloc` (modelled)
14. void *app malloc(size_t sz, const char *what)
15. {
16.     void *vp = OPENSSL malloc(sz);
         ^
17.     return vp;
14. void *app malloc(size_t sz, const char *what)
15. {
16.     void *vp = OPENSSL malloc(sz);
         ^
17.     return vp;
```
Using the first specification for `app_malloc(sz, what)` on line 2, when calling `memset` (on line 6) the analyser uses a built-in summary for `memset` which includes the following error specification:

\[
[a \mapsto \text{nil} \land \text{true}] \quad \text{memset}(a, b, c) \quad [er: a \mapsto \text{nil} \land \text{true}]
\]

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

\[
[\text{emp} \land \text{true}] \quad \text{ssl_excert_prepend(pexc)} \quad [er: \text{emp} \land \text{true}]
\]

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 does not contain `abort`. It turns out that another wrapper with the same name `app_malloc`, located in file `apps/lib/apps.c`, does abort in the case of NULL result, which was the one the maintainer had in mind, but it was not the version called in Listing 1, leading to this bug. We returned to the maintainer with the trace information, and they agreed with us and acknowledged our bug report as genuine. However, although they agreed with the bug report, they did not accept our proposed fix and they preferred a different one. In a separate pull request https://github.com/openssl/openssl/pull/15836 the developer changed the wrapper in `test/testutil/apps_mem.c` to be one that never returns NULL, as it executes `abort()` when the embedded call to `OPENSSL_malloc()` inside the wrapper returns NULL (see Listing 3). Many of the malloc wrappers we have encountered in OpenSSL do have the ability to return NULL, but others, such as this one, do not. After applying the developer’s preferred fix, for the NULL case our analysis computes an `abort` triple for `app_malloc()` and for `ssl_excert_prepend()`. Since `abort` triples are a special form of `ok` triples representing a program exit, Pulse-X does not produce the error in `ssl_excert_prepend()`.

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

We found malloc wrappers to be implicated in a number of bugs in OpenSSL, but the reasoning needed after NULL is assigned is not always so straightforward as in the case above. To see this, consider the `DH_check_pub_key` function in Listing 4, where the code allocates new memory and assigns it to pointer q by calling the `BN_new` function (another malloc wrapper). After allocating memory for `ret` successfully, this function also assigns `ret->d` to NULL (on line 18). On line 6, the code sets 1 to the buffer pointed to by q by calling `BN_set_word`. On line 25, `ret->d` is dereferenced, 3The full trace is available in the supplementary material.

```
void *app_malloc(size_t sz, const char *what) {
    void *vp;
    if (!TEST_ptr(vp = OPENSSL_malloc(sz))) {
        TEST_info("Could not allocate %zu bytes for %s\n", sz, what);
        abort();
    }
    return vp;
}
```

triggering a null-pointer-deference on line 6. To find this bug, non-trivial heap information is needed; in our analyser this information is provided by ISL formulae representing abstract heaps.

Listing 4. OpenSSL null pointer bug in `DH_check_pub_key`.

```
int DH_check_pub_key(const DH *dh, const BIGNUM *pub_key, int *ret) {
    BIGNUM *q=NULL;
    ... 
    q=BN_new();
    if (q == NULL) goto err;
    BN_set_word(q,1);
    ... 
}
```

2.2 Bug 3: A Memory Leak

One particularly interesting bug involved the `s_server` application, implementing a generic SSL/TLS server which listens for connections on a given port using SSL/TLS. Pulse-X discovered a memory leak in function `www_body` (Listing 5). Once allocated, the SSL socket `ssl_bio` is pushed at the end of the linked list `io` (on line 6), and all sockets and buffers linked by the list `io` are freed afterwards. Pulse-X found that, the memory allocated at line 4 would be leaked. When we submitted the patch, an OpenSSL maintainer warned us that our fix might cause a double-free-error and suggested that assigning `ssl_bio` to `NULL` after the push would be the correct fix. However, rerunning Pulse-X on the patch proposed by the maintainer revealed that if the push fails (i.e., `ssl_bio` points to an allocated heap and it has not been pushed at the end of the

Listing 5. OpenSSL memory leak bug in `www_body`.

```
static int www_body(int s, int stype, int prot, unsigned char *context) {
    ... 
    io = BIO_new(BIO_f_buffer());
    ssl_bio = BIO_new(BIO_f_ssl());
    ... 
    BIO_push(io, ssl_bio);
    ... 
    BIO_free_all(io);
    + BIO_free(ssl_bio);
    return ret;
}
```
list io), then the leak occurs after the added assignment. This leak report was indeed valid because
the memory pointed to by ssl_bio was not accessible from any pointer after the added assignment,
and consequently the memory would not be freed afterwards. 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. Indeed, the maintainer’s fix would have been correct if the push had been
always successful. After reporting this new observation to the maintainer, they agreed with our
proposed fix.

This bug was a challenging one to find. The www_body contains 426 lines of code, and is fairly
complicated: 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 "biabduction" 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 un-
foldings). 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.

For an example of a latent bug, the potential null dereference in function file_ctrl (Listing 6) is
classified as latent in OpenSSL-1.0.1h and has not been fixed in the current revision OpenSSL-3.0.0.

Our analysis discovered a null-pointer-dereference on line 182, 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 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 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
null-pointer-dereferences. 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
1.0.1h (from 2015) experiments, there were 42 latent leaks that were fixed later and 5 latent leaks
that were not.

In more detail, our strategy is to report:

- all manifest null-pointer-dereferences, no matter where in the program;

---

Commit 147ed5f9def86840c9f6ba512e63a890d58ac1d6.
We describe the abstract states of our program analyser Pulse-X.

As we demonstrate in more detail in §5.2, our experimental evidence validates this strategy: on OpenSSL 1.0.1h, Pulse-X found 306 issues in all (latent and manifest), it reported 63 of the 306 issues (those satisfying the three properties above), 53 of the 63 have been subsequently fixed with 5 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 describe the abstract states of our program analyser Pulse-X, certain separation logic 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 we 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. We assume two (countably infinite) sets of program variables, Var, and logical variables, LVar, such that Var \( \cap \) LVar = \( \emptyset \), a set of heap (memory) locations, Loc, and a set of values, Val, such that Loc \( \cup \) \{nil\} \( \subseteq \) 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.

As shown in Fig. 1, we model a Pulse-X world as a pair \( (\eta, \sigma) \), comprising an environment \( \eta \) and a state \( \sigma \). Intuitively, the environment tracks the values associated with program and logical variables, while the state models the stack and the heap.

We use \( x, y, \ldots \) as metavariables for program variables; \( X, Y, \ldots \) for logical variables; \( v \) for values; \( e \) for expressions; \( B \) for Boolean expressions; and \( \tau \) for a designated program variable recording the return value at procedure exits. We further write \( \rightarrow \) to denote a tuple of variables; and \( \rightarrow \) to denote a tuple of expressions. Lastly, we assume an expression interpretation function, \( \llbracket \cdot \rrbracket : \text{Exp} \rightarrow \text{Env} \rightarrow \text{Val} \), and a Boolean interpretation function, \( \llbracket \cdot \rrbracket : \text{BExp} \rightarrow \text{Env} \rightarrow \text{Val} \), respectively evaluating the values of expressions and Boolean expressions against a given environment.
We present the Pulse-X which the state can be split into two disjoint sub-states, one satisfying \( \nu \vdash \), Vol. 1, No. 1, Article . Publication date: August 2021.

loops via backward variant rules in incorrectness logic [O'Hearn 2019] might be studied in separate work.

need in an under-approximate analysis; see [O'Hearn 2019] for a discussion. However, using inductive predicates to leap

5 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.
[emp] skip [ok: emp]
[emp] error() [er: emp]

\[ (\forall x \in \text{pvars}(\pi) \ x i \mapsto X_i) \land \pi[X_i/x_i] \] assume(\pi) \ [ok: (\forall y \in \text{pvars}(e) \ (x) \ y i \mapsto Y_i \ast x \mapsto X)] \land V = e[Y_i/y_i][X/x]

\[ x := e \ [ok: (\forall y \in \text{pvars}(e) \ (x) \ y i \mapsto Y_i \ast x \mapsto V)] \land V = e[Y_i/y_i][X/x]

\[ x \mapsto X \ast y \mapsto Y \ast Y \mapsto V \ x := [y] \ [ok: x \mapsto V \ y i \mapsto Y \ y \mapsto V] \]

\[ y \mapsto Y \ast Y \mapsto nil \ x := [y] \ [er: y \mapsto Y \ y \mapsto nil] \]

\[ x \mapsto X \ast X \mapsto Y \ y \mapsto Y \ x := [y] \ [ok: x \mapsto X \ y \mapsto Y \ y \mapsto Y] \]

\[ x \mapsto X \ast X \mapsto Y \ x := [y] \ [er: x \mapsto X \ y \mapsto Y] \]

\[ x \mapsto X \land X = nil \ x := [y] \ [er: x \mapsto X \ y \mapsto nil] \]

\[ x \mapsto X \ast malloc() \ x := malloc() \ [ok: \exists L. x \mapsto L \ast L \mapsto V \ y \mapsto true] \]

\[ x \mapsto X \ x := malloc() \ [ok: \exists L. x \mapsto L \ast L \mapsto nil] \]

\[ x \mapsto X \ast X \mapsto V \ x \mapsto X \ [fre(x) \ [ok: x \mapsto X \ast X \mapsto \ y \mapsto X] \]

\[ x \mapsto X \ast X \mapsto] \ x \mapsto X \ [fre(x) \ [er: x \mapsto X \ast X \mapsto X] \]

\[ x \mapsto X \ast X \land X = nil \ x \mapsto X \ [fre(x) \ [er: x \mapsto X \ast X \mapsto nil] \]

\[ x \mapsto X \ast ret \mapsto V \ x := ret \mapsto X \ [ok: x \mapsto X \ast ret \mapsto X] \]

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 \land true.

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 \( \eta \). 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 \( \langle p \rangle \ C \langle e : q \rangle \). As described in [Raad et al. 2020], a triple \( \langle p \rangle \ C \langle e : q \rangle \) is valid, written \( \models \langle p \rangle \ C \langle e : q \rangle \), iff every state in \( q \) is
reachability under the exit condition $\epsilon$ (which may be $ok$ or $er$) by executing $C$ on some state in $p$.

Note that $[p] C [\epsilon : false]$ is vacuously valid as false denotes an empty state set.

**Definition 3.1 (ISL triples).** An (under-approximate) ISL triple $[p] C [\epsilon : q]$ is valid, written $\models [p] C [\epsilon : q]$, iff:

$$\models [p] C [\epsilon : q] \overset{\text{def}}{=} \exists q \subseteq [C, \epsilon ([p])]$$

where $[[\cdot]]_\epsilon : \text{Comm} \to \mathcal{P}((\text{World} \times \text{World}))$ denotes the command semantics under $\epsilon$, defined (in the technical appendix, http://www0.cs.ucl.ac.uk/staff/p.ohearn/RealBigDraftAppendix.pdf) as a state transition system analogously to that in [Raad et al. 2020].

### 3.2 Manifest Errors

We begin by recalling the ISL CONS and FRAME rules (below), as the placements of entailment and $\ast f$ assertions in the upcoming discussion implicitly appeals to these rules. Note that the entailments in the premise of CONS are in the opposite direction compared to those of their counterpart in Hoare logic; this is due to the under-approximate nature of ISL and our Pulse-X analysis.

<table>
<thead>
<tr>
<th>FRAME</th>
<th>CONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[p] C [\epsilon : q]$</td>
<td>$p' \vdash p$</td>
</tr>
<tr>
<td>$[p \ast r] C [\epsilon : q \ast r]$</td>
<td>$[p'] C [\epsilon : q']$</td>
</tr>
<tr>
<td>$q \vdash q'$</td>
<td>$[p] C [\epsilon : q]$</td>
</tr>
</tbody>
</table>

Recall from §2 that in order to minimise noise when reporting bugs and eliminate false positives, we introduce the notion of manifest errors. Intuitively, a manifest error denotes a valid procedure summary $[p] C [er : q]$ that (1) can be applied within any calling context (i.e., regardless of the state at call site); and (2) when applied, it always yields an erroneous execution terminating in a state satisfying an extension of $q$. Put formally, given any calling context $r$, if $\text{sat}(r)$ holds, then (1) there exists $f$ such that $p \ast f \vdash r$, i.e., the summary precondition $p$ can be extended with a frame $f$ to match the context $r$; and (2) $\text{sat}(q \ast f)$ also holds: the summary postcondition $q$ can be analogously extended with frame $f$. (Note that the entailment instance $\vdash$ here is in the opposite direction to what one would use in over-approximate analysis.) Condition (2) ensures that $[p] C [er : q]$ is not vacuously valid as otherwise $\text{sat}(q \ast f)$ would not hold.

**Definition 3.2 (Manifest errors).** A valid error triple $\models [p] C [er : q]$ denotes a manifest error iff for all $r$, if $\text{sat}(r)$ holds, then there exists $f$ such that $p \ast f \vdash r$ and $\text{sat}(q \ast f)$ also holds.

The following theorem shows that this notion of manifest errors ensures that executing $C$ on any calling context state $s$ can terminate erroneously in some state $s'$ satisfying an extension of $q$ (i.e., satisfying $q \ast \text{true}$). In other words, the manifest error is reachable from any input state. This is formulated, for simplicity, without taking procedure parameters into account, but in any case forms the semantic basis of our approach to compositional reporting. The full proof of Theorem 3.3 is given in the supplementary material.

**Theorem 3.3.** For all manifest errors $\models [p] C [er : q]$:

$$\forall s. \exists s'. (s, s') \in [C]_{er} \land s' \in \{q \ast \text{true}\}$$

In the following by a “complete program” we mean one with $\text{main}()$ in which there are no undefined, free function names.

**Corollary 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 $\text{main}$) or $[\text{main}]_{er} \neq \emptyset$.

---

Note that given any context state $s$, there exists an assertion $r$ that $s$ satisfies, namely $r \dashv \{s\}$. 

, Vol. 1, No. 1, Article . Publication date: August 2021.
**Manifest Errors and Backwards Under-Approximate Triples.** Note that Theorem 3.3 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 $[p] C [\epsilon : q]$, stating that $p$ under-approximates the states that can be reached from $q$ when executing $C$ backwards (see [Möller et al. 2021], Section 5). We formalise and prove this relation to backwards, under-approximate triples in the supplementary material. A related concept is studied in [Ball et al. 2005] (and elsewhere) under the name must transitions, in the reachability analysis of [Asadi et al. 2021], and was 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 Def. 3.2 is practically infeasible as it involves finding a suitable frame $f$ for each arbitrary context $r$, while also ensuring $\text{sat}(q * f)$. To remedy this, we identify four conditions that are simpler to check, do not quantify over all contexts, and if satisfied ensure that a given triple $\models [p] C [\epsilon : q]$ denotes a manifest error.

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) $p \equiv \text{emp} \land \text{true}$. 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 $\text{sat}(q * r)$ holds for an arbitrary $r$, we require that (2) $\text{sat}(q)$ hold, as otherwise $q * r$ would be rendered unsatisfiable. Note that condition (2) is not sufficient to ensure $\text{sat}(q * r)$ holds for an arbitrary $r$. To see this, consider $f()$ and $g()$ below and their valid error summaries:

$$f() \triangleq z := \text{malloc}(); \quad g() \triangleq \text{if } (x = 7) \text{ then error();}$$

$$S_1: \models [x \mapsto X \Rightarrow z \mapsto Z] f() [\epsilon: x \mapsto X \Rightarrow z \mapsto X \Rightarrow X \mapsto V]$$

$$S_2: \models [x \mapsto X \Rightarrow z \mapsto Z] f() [\epsilon: \exists Y. x \mapsto X \Rightarrow z \mapsto Y \Rightarrow Y \mapsto V \land Y = 7] \quad S_3: \models [x \mapsto X] g() [\epsilon: x \mapsto X \land X = 7]$$

Note that all summaries are valid and can be derived using the ISL proof system [Raad et al. 2020]. Specifically, ISL also includes the following axiom for malloc(), allowing the allocation to pick a specific location, namely $X$:

$$[z \mapsto Z] z := \text{malloc}() [\text{ok}: z \mapsto X \Rightarrow X \mapsto V] \quad \text{(MALLOC-ISL)}$$

This summary is a combination of the predefined summary of malloc() in Fig. 3 and an application of Cons 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 $\exists Y. x \mapsto X \Rightarrow z \mapsto Y \Rightarrow Y \mapsto V \land Y = 7$). Similarly, $S_3$ can be derived using the encoding of if and the predefined 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 $r \not\models X \mapsto V$, then $q_1 * r \equiv false$. In other words, for an error triple to be manifest, it must not require that malloc() allocate a specific location, i.e., the postcondition may not constrain the locations allocated via 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 $\exists Y. q$, we require that the heap locations in $k_q$ (i.e., the logical variables on the left-hand side of $\mapsto$ and $\not\mapsto$ assertions in $k_q$) be existentially quantified and thus not be the same as heap locations in the context. To this end, we define an auxiliary function, $\text{locs}(.)$, that computes the heap locations in a given spatial assertion $k_q$; we then require
that (3) $\text{locs}(κ_q) \subseteq \mathcal{X}_q$. 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 sat($q \ast r$) as the pure assertions in the postcondition may impede the satisfiability of $q \ast r$. 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 $q_2 \ast r$ unsatisfiable when $r \neq X \rightarrow V$. 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 $q_1 \ast r$ unsatisfiable when e.g., $r \neq X=2$. 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 $q \equiv \exists X_q. \kappa_q \land p_q$ and $\text{flv}(q) = \mathcal{Y}$, 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) sat($π_q[\mathcal{T}/\mathcal{Y} \cup \text{locs}(κ_q)]$) hold for all $\mathcal{T}$. Note that neither $S_2$ nor $S_3$ satisfies condition 4: in the case of $S_2$, sat($Y=X[v_1/X, v_2/Y]$) does not hold when $v_1 \neq v_2$; in the case of $S_3$, sat($X=7[\mathcal{v}/X]$) does not hold when $\mathcal{v} \neq 7$.

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 $\models \left[p\right] C [er : q]$ with $q \equiv \exists X_q. \kappa_q \land p_q$ denotes a manifest error if:

1. $p \equiv \text{emp} \land \text{true}$;
2. sat($q$) holds;
3. $\text{locs}(κ_q) \subseteq \mathcal{X}_q$, where $\text{locs}(.)$ is as defined below; and
4. for all $\mathcal{T}$, sat($π_q[\mathcal{T}/\mathcal{Y} \cup \text{locs}(κ_q)]$) holds, where $\mathcal{Y} = \text{flv}(q)$.

Note that 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 complexity of the satisfiability problem of equality logic, which is NP-complete. Nevertheless, checking these conditions is efficient in practice.

### 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, $a \in \text{Loc}$, that tracks all memory locations allocated thus far, as shown in Fig. 4. That is, we define states as $\text{State} \equiv (\text{Loc} \rightarrow \text{fin Val}) \cup \{(a) \rightarrow \mathcal{P}(\text{Val})\}$. Analogously, we extend the assertion language with the $a \rightarrow A$ and $\text{leaks}(S, L)$, where we use $A$ and $S$ as meta-variables for sets of logical variables. Specifically, the $a \rightarrow A$ denotes that the set of locations allocated thus far (at $a$) is given by $A$. We describe $\text{leaks}(S, L)$ 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 $\text{free}(.)$ 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 $\text{leaks}(S, L)$ 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
Extended Pulse-X Model Domain

\[ \eta \in \text{Env} \equiv (\text{Var} \rightarrow \text{Val}) \cup (L\text{Var} \rightarrow \text{Val} \cup P(\text{Val})) \]

\[ \sigma \in \text{State} \equiv (\text{Loc} \rightarrow _{\text{in}} \text{Val}) \cup (\{a\} \rightarrow P(\text{Val})) \quad \text{where} \quad a \in \text{Loc} \]

Extended Pulse-X Assertions

\[ \kappa ::= \cdots \mid a \rightarrow A \]

\[ \pi ::= \cdots \mid \text{leaks}(S, L) \]

Extended Pulse-X Assertion Semantics

\[(\eta, \sigma) \models a \rightarrow A \iff \sigma = \{a \mapsto \eta(A)\} \]

\[(\eta, \sigma) \models \text{leaks}(S, L) \iff \eta(L) \notin \text{reach}(\eta(S), \sigma) \]

where \[ \text{reach}(S, \sigma) \triangleq \bigcup_{i \in \mathbb{N}^+} S_i \quad \quad S_n \triangleq \{l \in \text{Loc} \mid \exists k \in S_{n-1}. \sigma(k) = l\} \]

\[ S_0 \triangleq S \]

\[ [a \rightarrow A \ast x \mapsto X] \ x := \text{malloc}() \ [\text{ok} : \exists L. \ a \rightarrow A \cup \{L\} \ast x \mapsto L] \]

\[ [a \rightarrow A \ast x \mapsto X \ast x \mapsto V] \ \text{free}(x) \ [\text{ok} : a \rightarrow A \setminus \{X\} \ast x \mapsto X \ast x \mapsto V] \]

Fig. 4. Updated ISL model and assertions for memory leak detection (above), where A, 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.

shortly, we check for memory leaks at the end of each procedure call; as such, the starting points correspond to the \textit{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 a) and is not reachable from \( \mathcal{G} \) (i.e., \( \text{leaks}(\mathcal{G}, L) \) holds). We then define the \textit{noLeaks} assertion, denoting that no allocated location is being leaked, as follows:

\[ \text{noLeaks} \overset{\text{def}}{=} \forall A. \ a \rightarrow A \land \bigwedge_{L \in A} \neg\text{leaks}(\mathcal{G}, L) \]

To detect memory leaks, we insert \textit{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 \textit{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

\textbf{Specification Tables}. Our proof search algorithm carries around a \textit{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

\(^7\)Note that \textit{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 boolean we can more simply regard \textit{assert}(noLeaks) as a special command apart from the given instructions. Its semantics is that its \textit{ok} relation sends a state to the same state if there are no leaks, and its \textit{er} relation sends a state to the same state if there are leaks.
\[\text{Eval}(p, \text{skip}, T) \triangleq \{(\text{ok}, \text{emp}, p)\}\]

\[\text{Eval}(p, \text{local } x:C, T) \triangleq \left\{ (\epsilon, \exists Y. m, \exists Y. q') \mid (\epsilon, m, q) \in \text{Eval}(\text{if } Y \rightarrow Y \ast p \land Y = \text{nil}, C[y/x], T) \land \text{ and } q' = \text{prune}(y, Y, q) \text{ where } y, Y \text{ fresh in } p \text{ and } y \text{ fresh in } C \right\}\]

\[\text{Eval}(p, C_1; C_2, T) \triangleq \left\{ (\epsilon, m_1 \ast m_2, q) \mid (\text{ok}, m_1, q_1) \in \text{Eval}(p(C_1, T)) \right\} \cup \left\{ (\epsilon, m_2, q) \mid (\epsilon, m_2, q) \in \text{Eval}(p(C_1, C_2, T)) \right\}\]

\[\text{Eval}(p, C_1 + C_2, T) \triangleq \text{Eval}(p, C_1, T) \cup \text{Eval}(p, C_2, T)\]

\[\text{Eval}(p, C^*, T) \triangleq \text{Eval}(p, C^\text{unrollings}, T) \quad \text{where } C^{i+1} \triangleq (\text{skip } + (C; C^i))\]

\[\text{Eval}(p, \text{inst}, T) \triangleq \left\{ (\epsilon, m, b \ast f)\mid[a]\text{ inst }[\epsilon : b] \in T \land (m, f) \in \text{BiabDiscover}(p, a)\right\}\]

\[\text{Eval}(p, \text{assert(noLeaks)}, T) \triangleq \{(\text{ok}, \text{emp}, p) \mid p \models \text{noLeaks}\} \cup \{(\epsilon, \text{emp}, p) \mid p \not\models \text{noLeaks}\}\]

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

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 \([p_1] f() [e_1 : q_1], \ldots, [p_n] f() [e_n : q_n]\).

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

If we extend \(p\) with the ‘missing’ resource \(m\), then \(\epsilon:q\) is a valid result of executing \(C\) on \(p \ast m\).

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

**Theorem 4.1 (Under-Approximation Soundness).** For all \(p, C, T, \epsilon, m, q:\)

\[(\epsilon, m, q) \in \text{Eval}(p, C, T) \quad \text{implies} \quad T \models [p \ast m] C [\epsilon : q]\]

**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 \((C_1; C_2)\) 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 \(m_1 \ast m_2\). 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 BiabDiscover\((p, q)\) function. This function is the biabduction notion from [Calcagno et al. 2011].

BiabDiscover\((p, q)\) returns a set of pairs of the form \((m, f)\) such that \(p \ast m \models q \ast f\). 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 abducting the frame $f$ and anti-abducting the missing resource $m$ via 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 CONS 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 (FRAME) 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 $\text{PRUNE}(y, Y, q)$ 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 $\text{PRUNE}$ is that for some $Y'$, the entailment $y \rightarrow Y' \ast \text{PRUNE}(y, Y, q) \land Y = \text{nil} \models q$ holds. $\text{PRUNE}$ can be implemented by simply stripping $y \rightarrow \ast$ 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, $\text{assert}(\text{noLeaks})$ 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 equalities and other pure facts into account. This has been a standard operation on symbolic heaps in separation logic program analyses from the beginning [Distefano et al. 2006].

To be sound for bug catching, this calculation should be sound for concluding that there are leaks $(p \not\models \text{noLeaks})$ – 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 $(p \models \text{noLeaks})$.

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 $(\epsilon, m, q)$ 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 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 $\rightarrow$ and $\not\rightarrow$ 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 POPL’09 and a worklist presentation from JACM’11 – 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 Eval(emp, C, T) to $T(f(\ ))$. 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 $y \mapsto Y$, where $\vec{y}$ denote the procedure parameters and $Y$ denote the logical variables recording their initial values. We illustrate the analysis via the following example.

Example 4.2. Consider the $\text{foo}(\ )$ summary in Fig. 6, a simplified version of the code in Listing 4. In this example, we show how our analysis finds a null-pointer-dereference (NPE) error. Specifically, $\text{set}(y, v)$ dereferences the heap cell whose address is the content of the pointer $y$ and updates its
void foo(){
    [emp ∧ true]
    local x;
    [x ← X ∧ X = nil]
    x := malloc();
    σ_3 ≡ [ok: ∃L. x ← L * L ← V ∧ X = nil ∧ L ≠ nil]
    assume(x != NULL);
    σ_3 ≡ [ok: ∃L. x ← L * L ← V ∧ X = nil ∧ L ≠ nil]
    [x] := NULL;
    σ_4 ≡ [ok: ∃L. x ← L * L ← nil ∧ X = nil ∧ L ≠ nil]
    set(x, 1);
    σ_5 ≡ [er: ∃L. x ← L * L ← nil ∧ X = nil ∧ L ≠ nil]
}

void set(y,v){
    [y ← Y * v ← V *_Y ← W ∧ W = nil]
    local z;
    σ_0 ≡ [y ← Y * v ← V * z ← Z * Y ← W ∧ Z = nil ∧ W = nil]
    [y] := [y];
    σ_1 ≡ [ok: y ← Y * v ← V * z ← W * Y ← W ∧ Z = nil ∧ W = nil]
    [z] := v;
    σ_2 ≡ [er: y ← Y * v ← V * z ← W * Y ← W ∧ Z = nil ∧ W = nil]
}

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

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 5, assigning nil to its content; and (3) set(x, 1)
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 ← Y and v ← V,
respectively. Initially, the Pulse-X algorithm starts with the pre-state y ← Y * v ← 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. Recall that on encountering
the memory read on line 10, Pulse-X uses its predefined summary in Fig. 3 via Eval (Fig. 5). This
in turn calls the biadductive procedure, BiaddDiscover(., .), 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 ← Z * y ← Y * Y ← W] z := [y] [ok: z ← T * y ← Y * Y ← W]

and poses the biadductive query BiaddDiscover(y ← Y * v ← V * z ← Z ∧ Z = nil, z ← Z * y ← Y * Y ← W),
which yields {(m, f)} with m ≡ Y ← W and f ≡ v ← V ∧ Z = nil. Note that m is used to compute the
pre-assertion and it is sent back to the beginning of the procedure, as denoted by the highlighted
which yields \{ (locs(3), (1)) \}. Its precondition is shown in Fig. 6. The other \(e\) specifications for \(z := [y]\) are applied similarly and result in two additional \(e\) disjuncts being analysed.

Similarly, on encountering the memory read \([z] := v\) on line 11, Pulse-X uses the following \(e\) summary of memory store in Fig. 3:

\[
[z \mapsto W \land W = \text{nil}] [z] := v \ [e_r : z \mapsto W \land W = \text{nil}]
\]  

and poses the query \texttt{BiabDiscover} \(y \mapsto Y \ast v \mapsto V \ast z \mapsto W \ast Y \mapsto W \land Z = \text{nil}, \ z \mapsto W \land W = \text{nil}\), which yields \{(m, f)\} with \(m \equiv \text{emp} \land W = \text{nil}\) and \(f \equiv y \mapsto Y \ast v \mapsto V \ast Y \mapsto W \land Z = \text{nil}\). \(m\) is used to compute the pre-condition while \(f\) is combined with the post-condition of \([z] := v\) to obtain \(\sigma_2\) (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 \(\sigma_2\).

\[
[y \mapsto Y \ast v \mapsto V \ast y \mapsto W \land W = \text{nil}] \text{ set}(y, v) \ [e_r : y \mapsto Y \ast v \mapsto V \ast y \mapsto W \land W = \text{nil}]
\]  

Similarly, it generates four other summaries for \(\text{set}(y, v)\) as follows.

\[
[y \mapsto Y \ast v \mapsto V \land y = \text{nil}] \text{ set}(y, v) \ [e_r : y \mapsto Y \ast v \mapsto V \land y = \text{nil}]
\]  

\[
[y \mapsto Y \ast v \mapsto V \ast Y \mapsto y \mapsto W \land W = \text{nil}] \text{ set}(y, v) \ [e_r : y \mapsto Y \ast v \mapsto V \ast Y \mapsto y \mapsto W \land W = \text{nil}]
\]  

\[
[y \mapsto Y \ast v \mapsto W \ast y \mapsto W \land W = \text{nil}] \text{ set}(y, v) \ [e_r : y \mapsto Y \ast v \mapsto V \ast y \mapsto W \land W = \text{nil}]
\]  

\(2\) and \(3\) correspond to the two disjuncts generated after line 10 while \(4\) and \(5\) correspond to those inferred after line 11.

Pulse-X processes the statements of \texttt{foo()} similarly, where it uses the following summary of \(x := \text{malloc}()\) in Fig. 6.

\[
x \mapsto X \ x := \text{malloc}() \ [\text{ok} : \exists L. x \mapsto L \ast L \mapsto V \land L \neq \text{nil}]
\]  

Note that, upon calling \texttt{set}(\(x, 1\)) on line 6 from state \(\sigma_3\), of the above five summaries, only \(1\) is applicable, for which the biabductive procedure returns the output \(\sigma_3\), as shown in Fig. 6. As such, \(\sigma_3\) is returned as the error post-condition, in accordance with the post-state of \(1\). In particular, Pulse-X infers the following \(e\) specification:

\[
[\text{emp}] \text{ foo}() \ [e_r : \exists L. L \mapsto \text{nil} \land L \neq \text{nil}]
\]  

Note that the above triple denotes a manifest error as it satisfies the four conditions of \textbf{Theorem 3.5}:

\(1\) its precondition is \(\text{emp}\); \(2\) its postcondition is satisfiable i.e., sat(\(\exists L. L \mapsto \text{nil} \land L \neq \text{nil}\)) holds;

\(3\) \text{locs}(L \mapsto \text{nil}) = \{L\} and \(L\) is existentially quantified; and \(4\) sat(\(L \neq \text{nil}\)) holds. In the implementation, Pulse-X also generates an error trace that is based on the chain of function calls (e.g., \texttt{foo();set(x, 1)}) and their corresponding triples. If the trace, like the one in this example, includes a null-dereference \(e\) 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 \(\sigma_4\). For instance, the biabductive query generated from the precondition of \(2\) (after renaming variables) and \(\sigma_4\) is as follows:

\[
\text{BiabDiscover}(\exists L. x \mapsto L \ast z \mapsto Z \ast L \mapsto \text{nil} \ast X \mapsto \land L \neq \text{nil}, \ x \mapsto X \ast v \mapsto 1 \ast z \mapsto Z \land X = \text{nil})
\]  

As the underlined sections show that the \(x \mapsto X \land X = \text{nil}\) in the precondition of \(2\) is incompatible with \(x \mapsto L \ast L \mapsto \text{nil} \land L \neq \text{nil}\) in \(\sigma_2\) (they cannot be reconciled), this query returns the empty set.
4.2 Implementation Notes

Pulse-X is a fork of Pulse, which is itself a program analyzer built on top of the Infer platform (www.fbinfer.com). Note that there is a distinction between Infer-the-platform (which includes several static analysis tools including RacerD and Pulse) and Infer-the-original which is the biabduction engine of the first version of Infer. The platform shares a frontend and some code with the original that doesn’t involve biabduction or separation logic, but is otherwise separate. Pulse-X does not use any of the original Infer biabduction code: we use a general compositional abstract interpretation framework implemented inside Infer that is also used for RacerD (but not biabduction). Pulse-X is not yet in the public Infer github repo, which is why we do not share it yet. But we intend to share it in due course.

Our implementation of Eval does not use proof search per se. Rather, it uses a worklist algorithm which maintains a set of \((\epsilon, m, q)\) 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 \(\text{assert(noLeaks)}\) at the end.

- 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 \(\text{main()}\), 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. As mentioned above, 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 \(\text{skip}\)), 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 \(\text{assert()}\) known internally to the analysis and that of C. Programmers sometimes use \(\text{assert()}\) or \(\text{abort()}\) to indicate when they do not want to be warned, and we treat \(\text{abort()}\) this way for OpenSSL, as in §2.1.
Table 1. Experimental results on compositional bug reporting.

<table>
<thead>
<tr>
<th>Project</th>
<th>#files</th>
<th>LoC(k)</th>
<th>#procs</th>
<th>BoC(m)</th>
<th>Infer</th>
<th>Pulse-X-10</th>
<th>Pulse-X-100</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenSSL-1.0.1h</td>
<td>1536</td>
<td>444</td>
<td>8324</td>
<td>3</td>
<td>80</td>
<td>26</td>
<td>63</td>
</tr>
<tr>
<td>OpenSSL-3.0.3</td>
<td>2452</td>
<td>754</td>
<td>21639</td>
<td>8.5</td>
<td>116</td>
<td>30</td>
<td>38</td>
</tr>
<tr>
<td>wdt</td>
<td>194</td>
<td>25.4</td>
<td>6679</td>
<td>8.5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>bistro</td>
<td>424</td>
<td>37.6</td>
<td>7290</td>
<td>9.7</td>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>SQuangLe</td>
<td>36</td>
<td>8.3</td>
<td>12938</td>
<td>17.9</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>RocksDB</td>
<td>1291</td>
<td>411.7</td>
<td>14669</td>
<td>18</td>
<td>32</td>
<td>8</td>
<td>35</td>
</tr>
<tr>
<td>FbThrift</td>
<td>5639</td>
<td>937.7</td>
<td>21753</td>
<td>29</td>
<td>8</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>OpenR</td>
<td>341</td>
<td>78.3</td>
<td>124461</td>
<td>195.7</td>
<td>60</td>
<td>50</td>
<td>137</td>
</tr>
<tr>
<td>Treadmill</td>
<td>409</td>
<td>25.3</td>
<td>236676</td>
<td>393.7</td>
<td>60</td>
<td>48</td>
<td>132</td>
</tr>
<tr>
<td>Watchman</td>
<td>557</td>
<td>63.2</td>
<td>245661</td>
<td>407.3</td>
<td>70</td>
<td>48</td>
<td>131</td>
</tr>
</tbody>
</table>

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 goals of our experiments are 1) to demonstrate that our approach scales to large codebases, and 2) to study the effectiveness of our bug reporting criterion. For the first experiment we ran Pulse-X on a variety of publicly-available open-source C and C++ projects, and compared the runtimes to those of Infer. This produces many more bug reports than is feasible to triage manually, so for the second experiment we decided to zoom in on OpenSSL. Infer had previously reported bugs on OpenSSL back in 2015, so this provided an additional point of comparison, making OpenSSL a natural choice for this part. We conducted all our experiments on a Linux machine with 24 cores.

Of course, 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).

5.1 Scale

The purpose of the work in this section is to test the hypothesis

**Hypothesis H1.** Pulse-X is broadly comparable with Infer in terms of performance, while reporting a comparable number of bugs.

Note that this hypothesis says nothing about the quality of the bugs reported.

We studied the scalability of Pulse-X and Infer on the projects in Table 1, counting the number of issues reported by Infer and Pulse-X. We evaluated Pulse-X in two configurations concerning the maximum number of disjuncts involved: 10 and 100. As Pulse-X is based on under-approximation, setting a limit on the maximum number of disjuncts in abstract states is sound for proving the presence of bugs. We say “Infer” to refer to the “biabduction” analysis of Infer. As Infer was aimed originally at over-approximation, it does not include a disjunct limit. Nonetheless, by default Infer sets a limit on the wall-clock time and the number of symbolic execution steps that are allowed for the analysis of each procedure, which we have left at their defaults of 1100 steps and 10 seconds per procedure.

The first column in Table 1 shows the project names. In the next two columns, we present the number of files and the number of source lines of code excluding blank lines and comments for each project (LoC, and k for thousands); these two numbers are computed by using cloc, and they
do not include the code in libraries used by the projects. The next two columns show the number of procedures (#procs) and number of bytes of IR code (BoC, and m for millions) actually analysed by Pulse-X, including those in libraries used by each project. Note that the size of IR code analysed can, because of libraries, be considerably larger for a project with a smaller number of source lines excluding libraries. The translation of C code into IR was done starting from Make build and the translation of C++ code from Buck build. Pulse-X-10 and Pulse-X-100 refer to Pulse-X with the indicated disjunct limit. The columns for Pulse-X and Infer show the number of null pointer and memory leak issues found by each.

We plot the running times of the tools in Fig. 7. All told, we find that Pulse-X-100 and Infer’s performance are comparable, as are the number of issues both tools report. This finding says nothing about the quality of the issues found; to study this, we focus on OpenSSL.

**Note on Incremental Performance.** The compositional nature of the Pulse-X analysis enables us not only to perform wholesale analyses of large projects efficiently, but also to get fast results on code changes. For example, fixing the bug we described in Listing 1 and then rerunning Pulse-X in incremental mode will re-analyse only the changed file to validate the fix. This takes less than ten seconds, compared to the tens of minutes needed to analyse the whole of OpenSSL. This is not a new feature of the analysis compared to other compositional techniques such as Infer, but it bears repeating here as our compositional criterion for bug catching gives us the same benefits as before in this respect. Even though Pulse-X is a scientific tool that is not deployed in production within an industrial workflow, this incrementality, together with the scaling results above, suggest that it has performance characteristics compatible with being run in CI at code review time.
5.2 Accuracy

We evaluate Infer and Pulse-X on OpenSSL, a widely used implementation of the SSL/TSL protocols as well as a general-purpose cryptography library.

**Old Bugs.** 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 (fallable) human judgement involved.

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 this section we consider OpenSSL-1.0.1h, a legacy version of OpenSSL from 2015 which had previously been analysed by Infer. The evaluation in this section is testing the following hypothesis.

**Hypothesis H2:** On legacy OpenSSL (from 2015) Pulse-X has a superior fix rate to the present-day Infer.

OpenSSL uses procedures CRYPTO_malloc, CRYPTO_free, and CRYPTO_realloc in place of the usual malloc, free, and realloc from the C standard library. We model these CRYPTO wrapper functions by providing their triples to Pulse-X, as well as Infer.

The results of these experiments are as follows:

- Pulse-X-10 found 26 bugs, 19 of which had been fixed, including 12 null pointer dereferences 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 (https://github.com/openssl/openssl, commit hash 147ed5f9def8640c9f6ba512e63a890d58ac1d6).

- Pulse-X-100 found 63 bugs, 53 of which had been fixed, including 11 null pointer dereferences and 42 memory leaks, fix rate is 84% (= 53/63). Similar to the case of Pulse-X-10, the unfixed issues concern either calls to unknown functions or buggy procedures that no longer exist in the current revision.

- The present-day Infer discovered 39 fixed bugs and 41 unfixed issues, Infer’s is 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 as such were not reported by Pulse-X.

These results confirm hypothesis H2.

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 over 300 of these latent null dereferences, more than

9https://mailing.openssl.dev.narkive.com/2DbkkYzD/openssl-org-3403-null-dereference-and-memory-leak-reports-for-openssl-1-0-1h-from-facebook-s-infer

10See the supplementary material, available at https://www.dropbox.com/s/ltpwcmxtc6aihp/list-bugs-openssl-1.0.1h.txt?dl=0, for the description of these 53 fixed bugs.
ten times the number of manifest issues reported. Upon inspection, most of these seem to be false positives, in that 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 300+ 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 criterion is purposely extremely conservative, reporting null dereferences only when they can occur in all contexts. It might be that some of the suppressed latent issues can be worth fixing, and we do not claim our reporting criterion to be the ultimate one (perhaps there is no ultimate). Indeed, Listing 7 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 pointer arg->data. As the allocation might fail, arg->data could point to NULL. Subsequently, the dereference on line 9 might cause a null-pointer-error. Pulse-X classified this issue as latent because this error occurs only when the condition on line 4 (arg->count == 0) holds. Interestingly, that is the case in a call site to this function in main, as shown in Listing 8, 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.

```c
int chopup_args(ARG *arg, char *buf, int *argc, char **argv[]) {
    int num,i;
    ...
    if (arg->count == 0) {
        arg->count=20;
        arg->data= (char **)OPENSSL_malloc(sizeof(char *)*arg->count);
    }
    for (i=0; i<arg->count; i++)
        arg->data[i]=NULL;
    ...
}
```

Listing 7. Latent error in chopup_args.

```c
int main(int Argc, char *ARGV[]){
    ARG *arg;
    ...
    arg->count=0;
    ...
    if (!chopup_args(&arg, buf, &argc, &argv)) break;
    ...
}
```

Listing 8. Manifest error in main of openssl.c.

Comparing with the 15 bugs found by Infer and fixed in 2015, Pulse-X-10 reported 4 overlapping fixed bugs, Pulse-X-100 reported 5, and present Infer discovered 4. Of the 10 fixed bugs not reported by Pulse-X-100, 5 were null dereferences found by Pulse-X-100 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. Note that the 15 fixed from 2015 were reported by the last author during a training
phase of Infer, when there were many more false positives (the exact number of false positives from then is not known). 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.

**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 commit hash 147ed5f9def86840c9f6ba512e63a890d38ac1d6, available at [https://github.com/openssl/openssl](https://github.com/openssl/openssl)), in case it found new bugs that would be worth fixing.

**Hypothesis H3.** Pulse-X finds new bugs worth fixing on latest OpenSSL.

Running with a 10-disjunct limit, Pulse-X discovered 30 issues on 3.0.0. Manual inspection by us revealed what we thought were 15 real bugs (8 null-pointer-dereferences 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](https://github.com/openssl/openssl/pull/15834) and [https://github.com/openssl/openssl/pull/15910](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 supplementary materials mentioned in the introduction). We take the information that the OpenSSL maintainers judged these issues worth fixing as confirmation of hypothesis H3.

When we ran Pulse-X with 100 rather than 10 disjuncts, it found a greater number of issues, 38, including a non-zero overlap with fixed bugs from the 10-disjunct case. We did not classify all those 38 because doing so is time-consuming, and the non-zero overlap means that hypothesis H3 is also confirmed for the 100-disjunct case: to classify those issues would furnish no new information as to our hypothesis. Curiously, though, the 100-disjunct case did not find a superset of the bugs of the 10 case: several real bugs (4th, 6th, 13th and 14th) were not found in the 100-disjunct case. The reason is that when the disjunct limit increases, the number of specs inferred for each function might increase; consequently, if a function contains a buggy statement after several function calls, the limit might be reached before the analysis reaches the buggy statement.

We discuss one false positive reported by Pulse-X in the function `ossl_do_ex_data_init` (Listing 9). 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_once`, and `pthread_rwlock_init` 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 5.

```c
int ossl_do_ex_data_init(OSSL_LIB_CTX *ctx) {
    OSSL_EX_DATA_GLOBAL *global = ossl_lib_ctx_get_ex_data_global(ctx);
    if (global == NULL)
        return 0;
    global->ex_data_lock = CRYPTO_THREAD_lock_new();
    return global->ex_data_lock != NULL;
}
```

Listing 9. 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\(^\text{11}\). This would stamp out all the false positives observed with OpenSSL, and

\(^{11}\)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.
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 H3 only mentions Pulse-X – and these numbers are given just for information.

We also remark that Coverity scan runs regularly on OpenSSL and does result in bug fixes\textsuperscript{12}; 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.

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. [2020] also presented an intra-procedural analysis. Our work goes beyond [Raad et al. 2020] with inter-procedural reasoning, using Eval (our predicate transformer-style semantics of Pulse-X) for summary inference, with an implemented inter-procedural analysis algorithm, and an experimental evaluation. Most significantly, we proposed a rigorous approach to the problem of compositional bug reporting, without a global program context.

Pulse is an analyzer for memory issues being developed at Facebook, which has evolved alongside the incorrectness logic theory. It implemented ideas about under-approximate compositional analysis before the formal foundation was in place, and this then influenced IL and ISL. As we mentioned in the introduction, Pulse-X is itself an experimental fork of Pulse. We made this fork to distinguish goals in tool engineering. The goal of Pulse-X is scientific, to probe the effect of trading Infer’s over-approximate separation logic for the under-approximate ISL. In doing so, our aim was to minimize the conceptual distance from Infer’s original foundation. In particular, our summaries look much like Infer’s, except that they are based on under-approximate rather than over-approximate triples. On the other hand, the goal of Pulse is to help engineers, and minimizing this conceptual difference is not as important for that purpose. For example, Pulse has a more compact notion of summary which is less obviously similar to that of Infer, and there is no requirement for Pulse to evolve in a way that eases scientific comparison.

Still, the distance between Pulse-X and Pulse has been narrowing rather than growing, and the Pulse-X work has even repaid some of IL’s debt to Pulse: the reporting criterion based on manifest bugs has been implemented in Pulse as well as Pulse-X, it is switched on in production, and this has resulted in a significant reduction in observed false positives. We expect that a fuller account of the Pulse story will be forthcoming after there has been extended industrial experience with it.

Long before the work on IL and Pulse 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\textsuperscript{-} transition [Ball et al. 2005]. Summaries were used in these works in an effort to speed

\textsuperscript{12}https://scan.coverity.com/projects/tpm2-openssl
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] and JPF-Star [Pham et al. 2019b]. 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 may get stuck and not reach many of the functions. As an example of (rough) comparison with this work, 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). By contrast, when run on OpenSSL (circa 2015), Pulse-X found 40 leak bugs which were reported and fixed (so we consider them true positives) with 12 false positives. Although the two tools were not run on exactly the same OpenSSL version (and thus this is not a systematic study), the difference is striking. There are numerous technical differences between Pulse-X and UC-KLEE related to path explosion and areas of unsoundness, but the greatest difference concerns what we call compositional reporting. Generated preconditions can lead to bugs but that would be deemed infeasible or not useful by programmers. This is managed in UC-KLEE using heuristics and programmer annotations. Their best-performing heuristic 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-10 and 84% for Pulse-X-100).

JPF-Star performs under-constrained symbolic execution by using separation logic to pre-constrain the test cases generated. For high path coverage, it uses inductive definitions in preconditions to represent an expressive fragment of heap inputs and utilises a separation logic solver to generate the test inputs that drive the symbolic execution engine to different paths of a given program. It was also combined with random heap-based input generation to support unknown functions e.g., Java native methods [Pham et al. 2019a]. However, as with JPF, it does not use summaries. Additionally, its symbolic states might over-approximate, and it does not report bugs compositionally.

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., [Ispoglou et al. 2020]). Fuzzing a complete program poses a similar problem to symbolic execution: when starting from `main()`, many parts of the code can be difficult to reach. Harnesses are “fake main programs” which set up the environment and then call a collection of functions repeatedly. Constructing them can be time-consuming and error-prone, hence the desire for automatic generation. 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 automatic 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. But, with these caveats, our results do provide some corroborative evidence of potential effectiveness.

Although we did not do a detailed comparison, we also noted in the previous section that our experimental results compare favourably with UC-KLEE [Ramos and Engler 2015] on some dimensions. UC-KLEE is a leading symbolic execution tool that targets running on individual functions instead of complete programs.

These remarks on Infer and UC-KLEE are not meant as criticisms of them. We stress that we build on the insights and reported experience of Infer, and we acknowledge that UC-KLEE has convincingly made the case that symbolic execution can more easily be applied to real code, finding real bugs, if it is applied on a per-function basis. (This experience with UC-KLEE, incidentally, agrees with the perspective of the Google and Facebook CACM papers [Sadowski et al. 2018; Distefano et al. 2019], and with that of the present paper.) Stepping back from comparisons of

13Of course this is almost always the case with static analyses, which are tackling undecidable problems.
14It received the best paper award at the 2014 USENIX Security Symposium.
specific tools showing that one has certain apparent advantages over another, the more general point we wish to make is the positive potential of using sound under-approximate abstractions when reasoning compositionally. We believe the results of this paper underline this point, but add that the general point does not depend crucially on our specific abstractions. For example, it might be possible to define a rigorous compositional reporting criterion for a tool like UC-KLEE, maybe similarly to how we have done so here; if successful, then symbolic execution tools might be deployed automatically in CI to identify regressions on diffs, without the human in the loop, which would likely considerably boost their impact. We certainly don’t claim to have discovered the ultimate compositional reporting criterion in this paper but rather an example with associated experimental backing, an example criterion which we hope others will be motivated to improve, extend, or even replace.

REFERENCES


