A. Appendix

In the main body of the paper we explained how our anlayzers scale because of their foundation based on compositionality and crafted abstract domains. This technical appendix provides some information on our crafted abstractions, and on the analysis algorithms we use.

A.1. Infer

Infer.Classic The original version of Infer, which we dub Infer.Classic in this section, sprung out of academic research on program analysis with separation logic [5], where the program analyzer manipulates logical assertions called *symbolic heaps* describing computer memory.

Infer.Classic finds memory safety issues spanning multiple procedures/files, as it attempts to construct program proofs that stitch the results together for the constituent procedures. The summaries are precondition/postcondition specifications, where the preconditions attempt to describe the *footprint* of a procedure: the memory cells it accesses [16]. Note that there will in general be a great many pre/post specs that are true for a procedure; considering footprints instead of arbitrary preconditions greatly cuts down the possibilities.

The focus on the footprint, rather than the global state of the program, provides an approach to question a) in Section 6, which concerns concise representations of procedure meanings. Part b), the stitching together of results, is implemented using a novel logical principle, *bi-abduction*; see [5].

Let's consider the C code in Figure 7. `malloc_int_wrapper`, by mimicking the behaviour of `malloc`, gets as summary a disjunction in the post-condition:

$$\{emp\}\,\texttt{malloc\_int\_wrapper()}\,\{return == 0\} \vee \{return \mapsto -\}.$$

By knowing that an assignment to `*x` requires the memory pointed-to by `x` to be allocated, Infer computes the following summary for `set`:

$$\{x \mapsto -\}\,\texttt{set(int *x, int n)}\;\;\{x \mapsto n\}$$

The summary says that to run the procedure without crashing, x must point to an allocated cell (i.e., $x \mapsto -$) and, at the end, the value of that cell will be n.

Infer uses and composes these summaries for the analysis of the procedure `caller`. After line 6 the analysis obtains two assertions, resulting form the disjuncitons in the post of `malloc_int_wrapper`:

$$y \mapsto - \qquad \text{and} \qquad y == 0$$

If we unify `y` with the `x` parameter of `set(int *x, int n)`, then the first of these assertions is equivalent to the precondition of its summary, $x \mapsto -$. However, the second assertion after unification becomes $x == 0$ which is inconsistent with the required precondition $x \mapsto -$. We cannot safely call the `set` procedure from the symbolic heap `y == 0`, and cannot complete the proof of the overall procedure `caller`. Infer uses this failed proof information to issue an error message[3].

If we uncomment line 7 then no error is reported and the proof goes through.

The use of `malloc_int_wrapper` in this example is instructive: Accurate memory analysis at scale tends to need disjunctions for precise-enough summaries. On one hand, it is not uncommon to find `malloc` wrappers in the wild; for instance, we reported several issues to openssl which involved a much more complex `malloc` wrapper, `OPENSSL_MALLOC`[4], and both the discovered summaries need to be expressive enough to handle such examples. On the other hand, even when one does not literally have a malloc wrapper it is not uncommon for a procedure to return valid allocated pointers in some circumstances and 0 in others.

```
1   void set(int *x, int n) { *x =n; }
2
3   int* malloc_int_wrapper() { return malloc(sizeof(int)); }
4
5   int caller () {
6       int* y = malloc_int_wrapper();
7       // if (!y) {return 42;};
8       set(y, 3);
9       free(y);
10  }

Infer error message:
pointer y last assigned on line 6 could be null and is
dereferenced by call to set() at line 8, column 5
```

Figure 7: Infer.Classic example

RacerD: an Infer.AI While Infer.Classic produced impact, perhaps its greater contribution was to establish that advanced reasoning techniques would survive and even thrive in Facebook's high momentum software development environment. This emboldened us to move forward with other techniques. Sam Blackshear created an analysis framework, Infer.AI, which facilitates building *compositional abstract interpreters* as in Section 6. This section discusses a specific Infer.AI, RacerD.

RacerD detects data races in Java programs; two concurrent memory accesses, one of which is a write. The example in Figure 2 (top) illustrates some of RacerD's ideas. If we run the analyzer on this code it doesn't find a problem. The unprotected read and the protected write do not race with one another because they are known to be on the same thread. This is one way that developers try to write thread safe code while avoiding synchronization for performance reasons.

Technically, RacerD works by first computing a summary for each method in a class, which records potential accesses together with information such as whether they are protected by a lock or confined to a thread. Next, RacerD looks through all the summaries for the class in question checking for potential conflicts. In this example, the summaries record an approximation of the information that the accesses to `mCount` are both protected by being on the same thread:

```
protectedWriteOnMainThread_OK()
Thread: true, Lock: true, Write to this.RaceWithMainThread.mCount:6
unprotectedReadOnMainThread_OK()
Thread: true, Lock: false, Read of this.RaceWithMainThread.mCount:10
protectedReadOffMainThread_OK()
Thread: false, Lock: true,  Read of this.RaceWithMainThread.mCount:13
```

RacerD concludes that a race is not possible because the threading information indicate mutual exclusion.

On the other hand, if we include additional methods that do conflict, then RacerD will report potential races, as in Figure 2, bottom. The summaries for the additional methods record information about the accesses as follows:

```
protectedWriteOffMainThread_BAD()
Thread: false, Lock: true, Write to this.RaceWithMainThread.mCount:17
unprotectedReadOffMainThread_BAD()
Thread: false, Lock: false, Read of this.RaceWithMainThread.mCount:20
```

[3] In fact, Infer records somewhat more than the bare pre/post information in summaries, to be able to report the line of the dereference in `set` when showing the user an error trace

[4] `rt.openssl.org/Ticket/Display.html?id=3403&user=guest&pass=guest`

The summary for `protectedWriteOffMainThread_BAD` shows the potential conflict with the unprotected read on the main thread. One is protected by a lock, and the other is protected by knowledge that it is on the main thread, but these are not sufficient to provide mutual exclusion: RacerD reports a race between this access and the one at line 8. Similarly, the access at line 20 in `unprotectedReadOffMainThread_BAD` is protected by neither a lock nor a thread. RacerD reports races with both the access at line 6 and the access at line 17.

RacerD employs a crafted abstraction oriented to finding races rather than memory safety (as with Infer.Classic). (i) it uses sets of accesses rather than pre/post specs as the summaries; (ii) it does not maintain any disjunctions, taking the union of accesses in branches of if statements. A consequence of using this crafted abstraction is that RacerD is blazingly fast: e.g., it can analyze 10k LOC in under 2 seconds [2].

The choice to avoid disjunctions entails a precision loss, and leads to false positives. Programs that race or not depending on boolean conditions can trip up the analysis. A typical example is the implementation of ownership transfer, where (say) you set a value to indicate that you are going to access an object outside of synchronization, code in other synchronization blocks then avoids to access the object: RacerD can easily report false positive races in such cases. In Facebook's Android code, fine-grained idioms like this are present in infrastructure code, but much less so in product code. For instance, Facebook's Litho UI library has fine-grained examples that lead to false positives of this variety in Infer, but the Litho authors advised us not to concentrate on the fine-grained idioms, to in a sense go against our first instincts as analysis experts to pursue subtle examples: they advised to do a better job with the coarser uses of concurrency typical in our product code (the majority), for which the precision loss was found to be acceptable. See [2, 15] for further discussion.

A.2. Zoncolan

Zoncolan performs a full program analysis of 100 million lines of Hack code in less than 30 minutes. To scale up to such a code base, Zoncolan uses a parallel compositional analysis in conjunction with a non-uniform abstract domain to approximate the flow of dangerous information. The code in Fig. 3 is an example of a vulnerability prevented by Zoncolan. Zoncolan needs to follow the interprocedural flow of untrusted data (*e.g.*, user-input) to sensitive parts of the code base, approximating virtual calls. Zoncolan utilizes a dependency graph to resolve the virtual calls and to schedule the analysis of individual functions, that will be analyzed in parallel.

For each function, Zoncolan performs a *forward* analysis, to propagate tainted data to the exit point of the method, and a *backward* analysis to propagate sinks to the entry arguments.

The forward analysis of `getIDs` (Fig. 8) infers that the user input at lines 5 and 6 flows to the first and second component of the pair returned by that method.

The backward analysis of `getConfirmationForm` (Fig. 9) states that a tainted value for the argument `$member_id` will reach the `action` field of `<form>` at line 20 after a string concatenation, and the argument `$group_id` since values reaching the `input::value` field do not pose any security threat.

The summary also illustrates the non-uniform abstraction being used: The exact literal string being concatenated with `$member_id` is abstracted to a single bit "via stringconcat" and unlike what we do for sources and sinks, Zoncolan does not retain the program location where the concatenation happens (*i.e.*, line 16). Finally, in the function `render` Zoncolan utilizes compositional reasoning to stich together the pieces of the flow from the input to the form action, obtaining the trace in Fig. 10. The summaries computed by Zoncolan contain just enough call-edge information to be able to reproduce full traces when displaying the alarms.

A.3. Recursion, Fixpoints, etc.

The early versions of Infer, which was derived from [5], worked by constructing a call graph, which which was used to schedule a bottom-up analysis algorithm where called are analyzed before callers. Cycles in the graph were broken arbitrarily to find a starting point. Summaries were stored in cache, which was consulted when analyzing a code modification in incremental fashion. This version of Infer implemented a shape analysis, one of the more expensive forms of analysis even intra-procedurally, and timeouts were used to ensure that the local analyses terminated (in that case, delivering a $\top$ result in the jargon).

In 2015, Cristiano Calcagno replaced the bottom-up Infer backend with one that works "on demand" instead of bottom up, and which does not require prior computation of a call graph. The analysis has a "begin anywhere" property, where can start anywhere in the codebase, irrespective of caching. In case the analysis needs a procedure summary and it is not in cache, the analyzer is called recursively to produce the summary. The on demand mode allowed for additional parallelism, and led to non-trivial performance gains over the bottom-up implementation. Infer.Classic and Infer.AI both use on-demand, presently.

Infer has used a bounded approach to (possibly mutual) recursion. We have experimented with different amounts of recursive unwinding, and at present Infer stops after one iteration. This choice is consistent with Infer's use to prevent regressions (like a testing tool), but not full proof. (For cognoscenti, mathematically this is like saying to calculate $F^i(\bot)$ for a given i and take that as the summary whether or not it is a fixpoint of F, with a slight modification for mutual recursion. This can be seen as a version of bounded symbolic model checking, which calculates an over-approximation of a finite unwinding of a program.)

As we indicated in the main body of the paper, in the case of mutually recursive functions Zoncolan iterates the analysis until the function summaries reach a fixpoint. To enforce convergence, Zoncolan uses a widening operator [7].

Infer and Zoncolan build on basic ideas from the research literature on compositional program analysis [8, 5], but there appears to be much valuable work to be done in both the theory and practice of algorithms in this area, especially when it comes to the scaling properties of compositional algorithms.

$$\begin{tikzcd}
\langle 5, \mathsf{user-input}\rangle \arrow[dr, "{[1]}"] & & \langle 6, \mathsf{user-input}\rangle \arrow[dl, "{[2]}"'] \\
& \mathsf{getIDs} &
\end{tikzcd}$$

Figure 8: Summary for `GenericGroupForm::getIDs`

$$\begin{tikzcd}
\$\texttt{member\_id} \arrow[d] & \\
\langle 20, \mathsf{form :: action}\rangle & \mathsf{, via\ stringconcat}
\end{tikzcd}$$

Figure 9: Summary for `getConfirmationForm`

$$\begin{tikzcd}
\langle 5, \mathsf{user-input}\rangle \arrow[d] \\
\langle 10, \mathsf{GenericGroupForm :: getIDs}\rangle \arrow[d] \\
\$\mathsf{member\_id} \arrow[d] \\
\langle 11, \mathsf{AddMemberToGroup :: getConfirmationForm}\rangle, 2 \arrow[d, "\mathsf{via\ stringconcat}"] \\
\langle 20, \mathsf{form :: action}\rangle
\end{tikzcd}$$

Figure 10: Trace to the vulnerability