Iodine: Fast Dynamic Taint Tracking Using Rollback-free Optimistic Hybrid Analysis

Subarno Banerjee*, David Devecsery†, Peter M. Chen* and Satish Narayanasamy*
* University of Michigan † Georgia Institute of Technology
{subarno, pmchen, nsatish}@umich.edu ddevec@gatech.edu

***Abstract*—Dynamic information-flow tracking (DIFT) is useful for enforcing security policies, but rarely used in practice, as it can slow down a program by an order of magnitude. Static program analyses can be used to prove safe execution states and elide unnecessary DIFT monitors, but the performance improvement from these analyses is limited by their need to maintain soundness.**

In this paper, we present a novel optimistic hybrid analysis (OHA) to significantly reduce DIFT overhead while still guaranteeing sound results. It consists of a predicated whole-program static taint analysis, which assumes likely invariants gathered from profiles to dramatically improve precision. The optimized DIFT is sound for executions in which those invariants hold true, and recovers to a conservative DIFT for executions in which those invariants are false. We show how to overcome the main problem with using OHA to optimize live executions, which is the possibility of unbounded rollbacks. We eliminate the need for *any* rollback during recovery by tailoring our predicated static analysis to eliminate only *safe elisions* of `noop` monitors. Our tool, Iodine, reduces the overhead of DIFT for enforcing security policies to 9%, which is $4.4\times$ lower than that with traditional hybrid analysis, while still being able to be run on live systems.

I. Introduction

Dynamic information-flow tracking (DIFT) [1], also referred to as taint-tracking, is a powerful method for enforcing a security or privacy policy. It tags source data (e.g., sensitive user input) as tainted, propagates taints through data and/or control flow, and checks if tainted data reaches sinks (e.g., network output). DIFT can help detect a wide range of security attacks [2]–[8] such as SQL injection, cross-site scripting, overwrite attacks, etc. It is also used to enforce information-flow policies that prevent sensitive information from leaking through untrusted channels [9]–[11].

In spite of its established benefits, DIFT is rarely used in practice today, due to its prohibitive performance overhead. For most use cases, the slowdown can be up to one to two orders of magnitude [12]. The reason is that, in a pure dynamic taint-tracking [12], every instruction has to be monitored to propagate taints to the destination operand based on the source operands' taints. There have been several attempts to reduce this cost by reducing tainted sources [13], by coarsening the granularity of objects [12] and/or code-regions [4] at which taints are tracked. But these approaches can compromise accuracy, and even so the overheads remain prohibitive for production use [14]. While parallelizing DIFT can help reduce the latency overhead [15], it may increase the throughput overhead due to additional parallelization costs. Recent work [16], [17], that decouples taint tracking from the program execution by performing symbolic taint analysis in parallel and periodically resolving concrete taint values with control-flow information, introduces imprecision due to the symbolic analysis. Moreover, the application might often need to wait for the outcome of the taint analysis before it can perform security-critical operations like releasing output.

In this paper we present a new approach to significantly reduce DIFT overhead using Optimistic hybrid analysis [18] (OHA). For rigorously tested production software, execution paths that violate an information-flow policy are almost certainly either rare or impossible. For such programs, pure dynamic taint analyses fundamentally do more work than necessary. A static taint analysis can identify instructions which cannot propagate taints to a sink [19], and DIFT monitors for these instructions can be elided. We show that OHA can dramatically improve the precision and scalability of static taint analysis, and thereby reduce DIFT overhead, by assuming program properties that are almost always true but hard to prove statically (e.g., likely callee-set of a function pointer).

A fundamental problem with OHA is that, if its assumptions (*likely invariants*) fail during an execution, then the soundness of dynamic analysis for that execution is compromised. To ensure soundness, prior work [18], which used OHA for data-race detection and slicing, checked the likely invariants at runtime, and when they fail, the program execution is replayed from the *beginning* and analyzed with a conservatively optimized dynamic analysis. While this unbounded rollback-recovery strategy is acceptable for retrospective analyses, it is not feasible for online security analysis of live executions.

We present a novel OHA that enables efficient and sound DIFT for live executions. We address the availability problem by completely eliminating the need for roll-back and enabling *forward recovery* on a likely invariant failure. The fundamental cause of rollbacks in an optimistic hybrid analysis is the runtime dependence between the current monitor being elided, and any potential future invariant violations that may affect the soundness of that elision. We observe that in order to construct rollback-free OHA, we must break this dependence. In other words, any monitor elided during a program execution, before an invariant failure, has to be proven to be unnecessary to ensure soundness of the dynamic analysis for the entire execution. We refer to eliding monitors satisfying this property as *safe elisions*.

Our key idea is to constrain predicated static analysis, such that it prunes a runtime monitor only if it can prove that it is a safe elision. Given this, when a likely invariant fails at runtime, it is sufficient to simply switch to a conservatively optimized analysis, and continue forward with the execution.

Iodine is used as a dye to track blood flow in X-ray angiography. Our tool tracks information flow through program executions.

To restrict a predicated static analysis to safe elisions, we further observe that many analyses, particularly bug finding and security analyses such as DIFT, often have many monitors that do not modify any analysis' metadata state when executed. We call such monitors `noop` *monitors*. By constructing a predicated static analysis that identifies and elides only `noop` monitors, we guarantee that any elision done by our predicated static analysis will not have any effect on dynamic analysis state until an invariant failure. Consequently, the soundness of these elisions cannot depend on any potential future invariant violations, because eliding a `noop` has the same effect as executing a `noop`, making the `noop` elisions *safe elisions*, and enabling forward recovery.

We construct such a predicated static analysis for DIFT that optimizes only safe elisions as follows. We use a predicated static forward data-flow may-analysis that prunes a runtime DIFT monitor for an instruction by proving that its source operands are never tainted. In this forward data-flow analysis, all optimization decisions are induced from the invariant assumptions using forward reasoning. Therefore, so long as the likely invariants hold true in a program execution, it is guaranteed that any elided monitor for an instruction is effectively a `noop`, as that instruction's source operands are guaranteed to be untainted. This in turn guarantees that the taint set (or meta-data) at any program instance matches exactly that of an unoptimized DIFT. When a program execution violates a likely invariant, it must be detected immediately. This is trivial for all the likely invariants we use. On detecting a likely invariant violation, the program recovers forward by safely switching to a conservatively optimized analysis. This requires careful engineering to safely handle function returns after switching the analysis versions.

It is interesting to note that it is challenging to construct optimizations based on predicated backward dataflow analysis that guarantees safe elision. For example, DIFT could be further optimized by eliding monitors for instructions whose destination operands never reach a sink. This requires a backward-dataflow analysis from the source operands of all sinks to the beginning of the program. A monitor elided using a predicated version of this static analysis is not guaranteed to be a safe elision. Because, in an execution, when a likely invariant fails, it may invalidate the property assumed to elide a *past* monitor as early as the beginning of the execution, requiring an unbounded rollback. Thus, we employ only conservative backward dataflow analysis in our system.

We implemented rollback-free optimistic hybrid DIFT using whole-program context-sensitive flow-sensitive taint analysis. We evaluate our tool on security-critical applications with realistic information flow policies. We augment the Postfix mail server application with information policies to check for email integrity and privacy, and run the Nginx web server application with detection against malicious overwrite attacks. We compare the performance of our approach with conservative hybrid and state-of-the-art full dynamic taint tracking [20]. Dynamic taint tracking incurs $7\times$ overhead over native execution, and hybrid analysis-optimized taint tracking incurs 37% overhead. Our optimized taint tracking tool brings down the overhead of dynamic taint tracking to 9%.

The contributions of this paper are as follows:

- We present a novel optimistic hybrid analysis technique to realize low-overhead dynamic information-flow tracking (DIFT) for live executions.
- We solve an important unresolved problem with optimistic hybrid analysis, which prevents its use for live analysis: need for unbounded roll-back when a likely invariant fails. We prove that restricting predicated static analysis to eliding only `noop` monitors guarantees metadata equivalence between optimistic and conservative hybrid analyses. This property in turn enables forward recovery when an invariant fails.
- We present a new profiling methodology for OHA based on regression and beta-testing. We show that likely invariants profiled using regression test suites are effective in obtaining majority of the performance benefits.
- Our approach reduces the overhead of DIFT to 9%, which is $4.4\times$ lower than that with conservative hybrid analysis, and $68\times$ lower than that with pure dynamic analysis.

II. Background

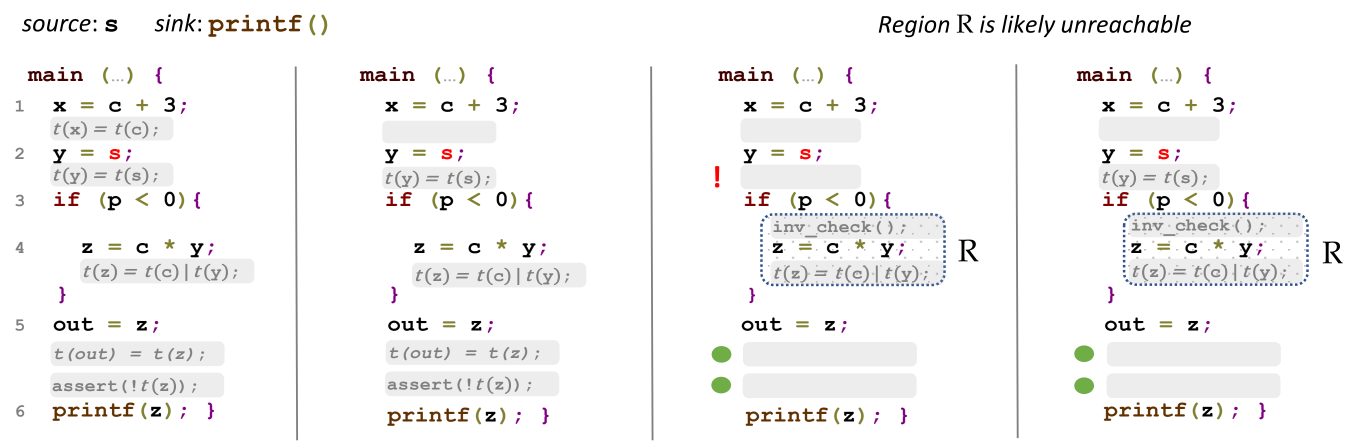
Before discussing our analysis framework, we review the necessary background related to taint tracking and how static analysis is used to improve its performance.

A. Dynamic Taint Tracking

Dynamic information-flow tracking (DIFT) [1] or dynamic taint tracking instruments a program to monitor data-flow from certain program inputs (*sources*) to some program outputs (*sinks*). Each source type is associated with a taint identifier. Each variable in an analyzed program has an associated taint state, which is a set of taint identifiers, representing which sources the variable derives its data from. A variable's taint state is defined either by sources, or by the *propagation function* when it is the destination of an instruction. Typically, taints propagate via explicit data flows (as we assume in this paper), where a destination operand's taint is derived from the taints of source operands. We refer to analysis code that propagates taints as *track monitors*. A few DIFT systems also consider implicit flows through control flow [12]. A taint analysis policy can also define certain operations to clear or sanitize taint, which are common in hash or encryption functions. Sinks are program locations where typically taint of output values are asserted to be false. We refer to these assertions as *check monitors*.

Figure 1(a) illustrates DIFT using an example. It assumes that `s` is a source, and `printf` is a sink. Taint propagates from `s` to `y` (line 2), and then it may or may not propagate to `z` (line 4) depending on the branch outcome in line 3. If the taint does propagate to `z`, it can reach `out` (line 5), and then reach the sink (line 6), causing an assertion failure.

Taint analysis can be tailored to a specific application by adjusting the taint policy. For example, information leakage is an important concern in database and web-service applications, where taint analysis is used to track the flow of sensitive information through program execution and prevent its leakage through unsecured channels. Taint analysis [21] is widely used in security analyses of programs to detect and prevent against overwrite attacks [2]–[4], command injection attacks [5], [6],



```
main (...) {
  x = c + 3;
  t(x) = t(c);
  y = s;
  t(y) = t(s);
  if (p < 0){
    z = c * y;
    t(z) = t(c) | t(y);
  }
  out = z;
  t(out) = t(z);
  assert(!t(z));
  printf(z); }
```

```
main (...) {
  x = c + 3;

  y = s;
  t(y) = t(s);
  if (p < 0){
    z = c * y;
    t(z) = t(c) | t(y);
  }
  out = z;
  t(out) = t(z);
  assert(!t(z));
  printf(z); }
```

```
main (...) {
  x = c + 3;

  y = s;

  if (p < 0){
    inv_check();
    z = c * y;
    t(z) = t(c) | t(y);
  }
  out = z;


  printf(z); }
```

```
main (...) {
  x = c + 3;

  y = s;
  t(y) = t(s);
  if (p < 0){
    inv_check();
    z = c * y;
    t(z) = t(c) | t(y);
  }
  out = z;


  printf(z); }
```

(a) Full dynamic analysis (b) Conservative hybrid analysis (c) Optimistic hybrid analysis (d) Rollback-free OHA

Fig. 1: DIFT optimizations. Green dot indicates safe `noop` elisions, and ! indicates unsafe elision.

cross site scripting attacks in web applications [7], [8], and to enforce information flow policies [10]. It has also been applied in semantic analysis of programs for program understanding [22], testing and debugging [23], [24].

In this work, we study several custom taint policies for preventing spam, ensuring email integrity, and detecting overwrite attacks against a web server. We also consider generic taint policies to evaluate DIFT performance.

B. *Conservative Hybrid Taint Tracking*

As shown in Figure 1(a), a pure DIFT instruments virtually all the instructions to propagate taints. This can result in an order of magnitude or more overhead. However, this overhead is not fundamental to enforcing a taint analysis. Because, in a rigorously tested program, information-flow leaks are rare. As a result, many of the DIFT monitors are either not propagating taints, or even if they do, they do no reach any sink. A sound static data-flow analysis can prove these properties and prune these dynamic monitors [19].

A static analysis constructs a data-flow model of the program, using the same taint policy as the dynamic taint analysis. We assume that the analysis is done in the static-single assignment (SSA) intermediate representation [25]. From this static model, the hybrid analysis will typically optimize its dynamic taint analysis in two ways:

- **Forward Taint Analysis** reasons from taint sources forward in the program, determining if the source operands of an instruction *may* be tainted or not. If none of the source operands may be tainted for an instruction, then the static analysis can prune its track monitor. For example, in Figure 1(b), the analysis can reason that neither source operands in the instruction `x = c + 3` are tainted, and therefore `x` will not be tainted, allowing its monitor to be elided.
- **Backward Taint Analysis** reasons whether a destination operand of an instruction *may* reach a sink. If not, track monitor for that instruction is elided, even if it can be tainted. In Figure 1(b), the conservative static analysis cannot leverage this optimization to elide any monitors, because it cannot prove this property soundly for any of the instructions.

Although static analysis is helpful in reducing DIFT overhead, its effectiveness is limited in practice. The sound static analysis used in traditional hybrid analysis must often make overly-conservative assumptions to retain soundness for all possible executions of a program. This inevitably includes many infeasible program states into the analysis's search space. Furthermore, even an ideal static analysis that only explores feasible program states, would still be ineffective, as most dynamic executions cover only a small subset of common execution states. For example, an encryption function provided by a standard crypto library has many candidate algorithms, but only a few of them are ever used, as many of them are deprecated or not preferred for use in certain systems. So, there is a significant gap between the states considered by sound static analysis and those actually realized in dynamic executions. This large discrepancy between the states that sound static analysis considers and dynamic executions experience often leads to highly inaccurate static analysis, and unacceptable dynamic overheads.

III. Design

We discuss a novel hybrid analysis, Iodine, to significantly reduce DIFT overhead based on optimistic hybrid analysis (OHA) [18]. It supports live executions by eliminating the need for rollback-replay.

A. *Optimistic Hybrid Taint Analysis*

Iodine leverages the key observation of OHA: static analyses used for optimizing a dynamic analysis should ideally consider only the states that will be realized in the analyzed dynamic executions. By targeting expected executions, we can significantly improve the precision and scalability of static analysis, and consequently optimize a dynamic analysis much more efficiently than its traditional counterpart.

Figure 1(c) illustrates the untapped opportunity. If all expected executions of this program only have non-negative values for the variable `p`, the code region `R` is never executed. A sound static analysis cannot assume this behavior, because

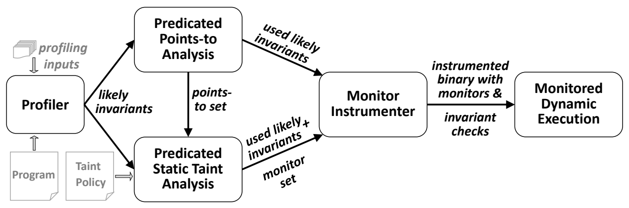


Fig. 2: Workflow of optimistic hybrid taint analysis

there are legal executions where `p < 0`. However, by constraining the static analysis to expected dynamic executions, Iodine can reason that the variable `z` does not get tainted due to `y` in line 4, and in turn proves that `out` in line 5 cannot be tainted. Therefore, it elides track monitor for line 5, and check monitor for the sink in line 6. Furthermore, backward data-flow analysis determines that taint of `y` in line 2 can never reach any sink, and elides its track monitor. None of these three monitors could be elided using conservative static analysis (Figure 1(b)).

Iodine's work-flow is illustrated in Figure 2. First, a profiler observes representative executions to gather a set of likely invariants. These *likely invariants* are common-case dynamic execution behaviors such as likely unreachable code, likely callee sets, and likely unrealized call contexts [18]. These are almost always true, but are hard to prove statically. Second, these likely invariants are used as predicates to constrain the state-space of the static taint analysis described in §II-B, resulting in a *predicated static taint analysis*. It is much more precise and scalable than a conservative sound static taint analysis, and enables Iodine to aggressively elide DIFT monitors. The program is instrumented with the remaining DIFT monitors along with invariant checks.

In most dynamic executions, the likely invariants will hold and the DIFT analysis will be sound. But, when the likely-invariants do not hold, dynamic analysis may be rendered unsound by the optimizations induced by predicated static analysis. The dynamic analysis requires a mechanism to recover from an invariant failure.

B. Problem: Rollback Recovery in OHA

When a likely invariant fails, it renders the predicated static analysis' optimizations unsound. As we use whole-program static analysis, at runtime it is non-trivial to determine the effect of a current invariant failure on the soundness of an elided monitor in the past. For example, in Figure 1 (c), if the likely unreachable code invariant (R) is violated in line 3, it would render the past elision of monitor for line 2 to be unsound.

Past work [18] conservatively addressed this problem by completely redoing the dynamic analysis by replaying the program execution from the beginning using the conservatively optimized dynamic analysis. Since invariants rarely fail, this rollback recovery is an acceptable solution for offline retrospective analyses such as debugging and forensic analyses. However, a rollback to the beginning of the program is intolerable for online security analyses on live executions, as it would severely compromise availability of the system.

Bounding rollbacks is hard for arbitrary predicated *whole-program* static analyses. Determining the latest point in a program execution up to which we need to rollback is an unsolved problem. For many analyses, especially backward data-flow analysis, it may not be possible to bound the rollback window. This unpredictable, and unbounded downtime caused by rollback creates problems guaranteeing availability for live executions.

Furthermore, support for rollback introduces significant additional overhead even for executions where the likely invariants hold true. This overhead includes the cost of logging for replay and periodic check-pointing. Therefore even when the invariants are not violated, eliminating rollbacks altogether would improve OHA by getting rid of these overheads. Plus, there is a cost for rollback-replay in case of an invariant violation. The last component is a minor cost as invariant violations can be made to be rare with sufficient profiling.

We address this problem by enabling forward recovery on any invariant failure, and completely eliminating the need for rollbacks.

C. Safe Elisions

Iodine uses a novel form of OHA called rollback-free OHA, which eliminates the need for rollback on invariant failure. Rollbacks are fundamentally caused by the dependence between the current monitor being elided and potential future invariant failures. Our idea is to distinguish safe elisions, which do not have such dependencies, from unsafe elisions.

A predicated static analysis in OHA elides a monitor as long as it can prove that it is unnecessary to guarantee soundness of dynamic analysis in an execution where the invariants hold. But an elided monitor is a *safe elision* only if it can additionally prove that an invariant violation in an execution would not affect the soundness of any preceding elisions of that monitor.

Rollback-free OHA is realized by restricting its predicated static analysis to only using safe elisions, and switching to a conservatively optimized analysis on invariant violation.

D. `Noop` Monitor Elisions are Safe Elisions

Statically proving safe elisions is non-trivial for many analyses. To make such an analysis practical and simple to construct, we further observe that `noop` monitors are safe elisions. A `noop` track monitor is one that does not change the analysis metadata state. A `noop` check monitor is one that always succeeds. For example, in Figure 1(c), monitors for lines 5 and 6 are `noop` monitors, if we assume `R` is unreachable. Monitor for line 2, however, is not a `noop` monitor, as its execution can modify the taint set even if invariants hold true.

Eliding `noop` monitors is safe for the following reasons. By construction, OHA instruments invariant checks such that they detect any invariant violation *before* an execution violates the invariant. For example, in Figure 1(c), OHA detects an invariant violation before entering `R`. Given this, when a `noop` monitor is elided before an invariant violation, it is guaranteed that it would be a `noop` monitor even in the conservatively optimized analysis, and therefore its elision is sound even when there is a later invariant violation. Thus, `noop` monitor elisions are safe elisions.

E. Elisions in Predicated Forward Analysis are Safe

In §II-B, we discussed forward and backward static taint analysis. Forward static data-flow taint analysis elides a monitor for an instruction in an SSA intermediate representation by proving that its source operands must not be tainted. The taint for the destination operand of such an instruction remains unchanged. Thus, all the monitors elided by predicated forward taint analysis are `noop` monitors, and therefore safe elisions.

Figure 1(d) shows rollback-free OHA. Its predicated static taint analysis is limited to forward taint analysis. Therefore, it elides only the monitors for lines 5 and 6, which are both `noop` safe elisions.

F. Elisions in Predicated Backward Analysis may not be Safe

Monitors elided by a predicated backward taint analysis are not guaranteed to be safe elisions. A backward taint analysis seeks to prove that an instruction's destination taint does not reach a sink, and if so it elides its monitor. Monitors elided by this analysis are not guaranteed to be `noop`s. For example, the monitor for line 2 in Figure 1(d) is not a `noop`, because it changes the taint of `y`. But a predicated backward analysis can elide it by assuming `R` is unreachable. However, during an execution, if that invariant fails, recovery must somehow produce the correct taint state of `y`, before proceeding forward. Given that we use a whole-program analysis, it is unclear how far the execution needs to be rolled-back and re-executed.

A more fundamental reason why elisions in backward-analysis may not be safe is their dependence on invariants holding true in the future. It may still be possible to construct safe elisions through sophisticated optimizations. For example, if we can somehow determine the set of all monitors elided due to a particular invariant (`R` is unreachable), then hoisting the invariant check before those elisions can make them safe elisions. Such a transformation is non-trivial for a predicated whole program analysis, and therefore we did not pursue this avenue. Fortunately, we found the predicated forward taint analysis to be quite effective by itself. Also, backward analysis is not useful for certain information-flow policies such as one that monitors taints from sources to all possible locations in a program.

G. Rollback-Free Optimistic Hybrid Taint Analysis

Iodine uses a predicated forward taint analysis along with a conservative backward taint analysis. Optimized dynamic analysis (*fast-path*) is executed until an invariant fails. As the analysis only elides `noop` monitors, it tracks exactly the same meta-data as a conservatively optimized analysis at all program points. We instrument a conditional branch for every invariant check, which switches the control to a conservatively optimized analysis (*slow-path*) when any of those checks fail. The execution then continues forward in the slow-path. Care is taken to ensure a safe switch. At the time of the switch, the return addresses on the stack would be pointing to fast-path return sites. We address this problem by checking every return site, and transferring control to either the fast or slow path based on the current mode of execution.

Iodine conservatively disables all optimistic optimizations upon an invariant violation. Given adequate profiles, which is a reasonable assumption for a rigorously tested production software, invariant failures are very rare. If there is indeed an invariant failure in production, the program can be re-optimized offline after removing the offending invariant from the likely-invariant set. Thus, in the steady-state, invariant violations would be extremely rare. Also, since it is common for live systems to be periodically restarted [26], the execution can switch back to the fast-path on a restart. Alternatively, only the optimizations induced by the violated invariant could be selectively disabled. However, there is no known method that can be easily applied to realize this. Also, this approach would require numerous variants of slow-paths.

IV. Proof Sketch

In this section, we formalize the notion of two analyses being state-identical, and then prove the soundness of rollback-free optimistic hybrid analysis by showing that it's state-identical to a conservative hybrid analysis.

A. Notations and Notions

An analysis A is a transformation of a program P that only generates additional metadata state σ_A and has no side-effect on P's program state μ_P. We define out_A to be the outcome of all dynamically failed check monitors.

We will use the following notations to refer to analyses instances:
UNOP is the unoptimized dynamic analysis that does not elide any monitors.
CONS is the dynamic analysis optimized by conservative static analysis.
OPTI$_I$ is the dynamic analysis optimized by predicated static analysis assuming the set of invariants I.
RFOPTI$_I$ is the rollback-free dynamic analysis optimized by forward-only predicated static analysis assuming the set of invariants I.

$\sigma_A(l)$ denotes the metadata state of dynamic analysis A at the program location l. I-FAIL(i) denotes the point(s) in program execution where the invariant assumption i dynamically fails. I-CHECK(i) denotes the program location(s) where the invariant validation checks are instrumented. A `noop` monitor is either a track monitor that does not modify σ_A, or a check monitor that succeeds.

Definition 1. *Analysis equivalence* : We say that dynamic analysis A' is equivalent to dynamic analysis A, denoted by $A' \equiv A$, if for all executions, their analysis outcomes are the same, i.e., $out_{A'} = out_A$.

Definition 2. *State-identical* : We say that dynamic analysis A' is state-identical to dynamic analysis A, denoted by $A' = A$, if for all executions, their terminating metadata states σ_A and $\sigma_{A'}$ are identical, i.e., $\sigma_{A'} = \sigma_A$.

B. Axioms

Axiom 1. CONS *is sound* [19], *i.e.,* CONS $\equiv$ UNOP.

CONS only elides those monitors which can be proven to not change the analysis outcome in all executions. $\therefore$ CONS $\equiv$ UNOP.

Axiom 2. OPTI$_I$ *is sound when the invariants hold* [18]*, i.e.,* $I \models$ OPTI$_I \equiv$ CONS.

In addition to those elided by CONS, OPTI$_I$ elides only those monitors that can be proven to not change the analysis outcome in dynamic executions that satisfy I. $\therefore$ $I \models out_{\text{OPTI}_I} = out_{\text{CONS}} \rightarrow I \models \text{OPTI}_I \equiv \text{CONS}$.

Axiom 3. *Invariant violation is detected before a program execution reaches a state that fails an invariant, i.e.,* I-CHECK$(i) <$ I-FAIL(i).

By construction, our invariant checks are instrumented such that this property holds.

Axiom 4. RFOPTI$_I$ *only elides monitors that are* `noop`*s.*

By construction in §III-C, RFOPTI$_I$ uses forward predicated static data-flow analysis to elide only those monitors that it can prove are `noop`s.

C. Soundness of Rollback-free OHA

We first show that RFOPTI$_I$ is state-identical to a sound conservative hybrid analysis for executions where the invariants hold. Next, we provide a simple program transformation that makes the RFOPTI$_I$ state-identical to CONS even at the point of a dynamic invariant failure. Finally, we show that the above property allows a forward recovery of RFOPTI$_I$ upon an invariant failure, and makes the whole dynamic analysis sound for all executions.

Lemma 5. RFOPTI$_I$ *is state-identical to* CONS *when the invariants hold, i.e.,* $I \models$ RFOPTI$_I =$ CONS.

Proof: By Axiom 4, RFOPTI$_I$ elides only those monitors that can be proven to be `noop`s in dynamic executions that satisfy I. $\therefore$ $I \models \sigma_{\text{RFOPTI}_I} = \sigma_{\text{CONS}} \rightarrow I \models \text{RFOPTI}_I =$ CONS. ■

Lemma 6. RFOPTI$_I$ *is sound until an invariant fails, i.e.,* $\sigma_{\text{RFOPTI}_I}(\text{I-FAIL}(i)) = \sigma_{\text{CONS}}(\text{I-FAIL}(i))$.

Proof: Consider the analysis RFOPTI$_{\{i\}}$ with a single invariant i. $\neg\{i\} \not\models \text{RFOPTI}_{\{i\}} =$ CONS, i.e., we cannot guarantee soundness for the entire program P if the invariant fails in a dynamic execution.
Let I-FAIL(i) be the first instance of an invariant failure in the dynamic execution of P. Now, consider the program P' obtained by the following transformation (shown in Figure 3): immediately after the location of each invariant check, we instrument a HALT instruction conditional on the invariant i having failed. The elided monitors are shown as equivalent `noop`s.

By Axiom 3, the invariant check preceding I-FAIL(i) will detect the invariant failure before the program execution reaches a state that fails the invariant. Therefore, the modified program P' will HALT after the failed I-CHECK(i), and before I-FAIL(i). This is equivalent to a program executing without an invariant failure.

By Lemma 5, RFOPTI$_I =$ CONS for P'. Since, P and P' only differ in their termination behavior and P' HALTs at I-FAIL(i), we have that:
$\sigma_{\text{RFOPTI}_I}(\text{I-FAIL}(i)) = \sigma_{\text{CONS}}(\text{I-FAIL}(i))$ for P. ■

```
                 ...
          l1:    noop
                 ...
 I-CHECK(i):     if (¬i)
                   HALT
  I-FAIL(i):     ...
          l2:    noop
                 ...
```

Fig. 3: Transformed program P'

Theorem 7. RFOPTI$_I$ *with forward-recovery is sound.*

Proof: By the soundness of RFOPTI$_I$ on the HALT-transformed program P' in Lemma 6, we have that the metadata state $\sigma_{\text{RFOPTI}_I}(\text{I-FAIL}(i))$ at the location of invariant failure is state-identical to that in CONS. Therefore, the forward-recovery mechanism can simply switch to CONS on an invariant failure, and that analysis as a whole is analysis-equivalent to CONS. $\therefore$ by Axiom 1, RFOPTI$_I$ with forward-recovery is sound. ■

D. Insight Summary

Contrasting Axiom 2 and Lemma 5, the key difference is that when I holds, OPTI$_I \equiv$ CONS but RFOPTI$_I =$ CONS. While the generic OPTI$_I$ aggressively elides monitors to only preserve analysis-equivalence, RFOPTI$_I$ only elides `noop` monitors, thus being state-identical to CONS. This allows the analysis to simply switch to conservative analysis CONS upon invariant violation.

V. Implementation

We built the Iodine tool, an instance of our rollback-free optimistic hybrid analysis discussed in §III for taint tracking. The profiler, the profile-driven predicated static analysis and the dynamic analysis instrumenter are implemented in the LLVM 3.9 compiler infrastructure [27], and we run our analysis tool after all other compiler optimization passes. We support programs written in the C language. To track taint flows through external libraries, we compile and statically link all dependency libraries into a single object, except `libc` for which we write stubs, and thereafter analyze them together. Our analysis marks policy-determined inputs as tainted, follows how taints propagate through program execution, and asserts checks on the usage of tainted data. To instrument the final optimized taint analysis code, we build upon LLVM's Data Flow Sanitizer[20] as our instrumentation backend. We discuss the implementation details below.

A. Specifying Information-Flow Policies

To evaluate the effectiveness of optimistic analysis on taint tracking, we design a configurable taint policy that treats all types of external inputs to the program as potential taint sources. This includes most program interfaces such as terminal, file, socket input functions, and command-line arguments. Source functions can taint data in different ways, e.g. the return value of `getchar()` becomes tainted, while the buffer operand of `read()` becomes tainted. We allow the policy to mark standard output interfaces, such as terminal, file and socket outputs as taint sinks and assert that the appropriate arguments to these functions should not be tainted.

We allow specifying configurable taint policies to identify taint sources, sink locations, and untaint functions via a flexible interface of source-level annotations. The annotation language can specify custom taint markings, e.g. `taint(password) = secret` would attach a new taint marking `secret` to the variable `password`; and specify custom taint checks. Additionally, the source annotations specify *untaint* functions to indicate when a particular taint marking should be removed. This makes Iodine easily adaptable to employ useful information-flow policies, as we demonstrate in §VI-B.

B. Static Taint and Pointer Analysis

Static taint analysis computes how the taints of data propagate through the program under a given selection of taint sources, sinks, and propagation policies. We perform static taint analysis using a whole-program context-sensitive flow-sensitive data-flow may-analysis [2]. We do so by building a definition-use graph (DUG) [28], each node representing an instruction that defines the resultant data value, and an edge connecting an instruction to those that use its definition. Our taint analysis is a context-sensitive analysis, making it more precise by distinguishing between different invocations of the same function. The analysis first creates local DUGs separately for each function in the program which captures the intra-procedural data flows within each function. Then the analysis traverses the call graph of the program, beginning at the `main` function. For each function call, it creates a clone of the local DUG of the callee function and connects the arguments and return values of that replica to the call-site. If the function call is recursive, the call is connected to the existing nodes in the DUG representing that callee, otherwise a new set of nodes is added for the call. It then recursively processes the callee function until all calls are resolved, resulting in a complete context-sensitive DUG of the program.

Once the DUG is constructed, the analysis can induce the following two optimizations:
Forward optimizations: Taints are propagated through the whole-program DUG using forward data-flow until a transitive closure is reached. Since our dataflow analysis is a *may* analysis, the absence of taint flow is a sound *must not* assertion. Therefore, any instruction that does not have tainted source operands can elide dynamic monitors for taint tracking. This optimization is induced only using a forward data-flow analysis.
Backward optimizations: Taint flows that do not eventually reach a sink can be pruned out using a backward co-reachability analysis on the DUG. We enable these optimizations only in the conservative static analysis. This optimization is induced using a backward data-flow analysis on the result of forward taint-flows.

Pointer Analysis: To track taint flows via indirect memory operations to aliased locations, we need to propagate taints to all aliases of a pointer accessing a tainted value. To this end, we perform a pointer analysis to compute the points-to set of each pointer location and then use this information during the static taint analysis to add taint-flow edges to the DUG from pointer definition to its aliasing uses. We use Andersen's-based [29] whole-program context-sensitive flow-insensitive inclusion-based pointer-analysis with heap cloning [30] and well-known optimizations including HVN/HRU [31], online-cycle-detection [32], and BDDs for points-to sets [33].

C. Predicated Static Taint Analysis

We improve the precision of the forward-only static data-flow taint analysis by assuming profiled likely invariants to increase its accuracy and efficiency. We use the following types of invariants:

Likely unreachable code is a code region that is unlikely to be executed. These can significantly reduce the static analysis' search space by pruning away all nodes defined by and edges incident upon them in the analyses DUGs. This invariant is profiled by instrumenting every basic block.

Likely callee set is the set of functions that are likely to be invoked by an indirect function call. These invariants effectively convert all indirect function calls in the DUG to direct calls to the likely callee functions, thereby removing all the imprecision due to unresolved indirect call destinations. These are easily profiled by monitoring values of function pointers in the program.

Likely unrealized call contexts is the set of calling contexts that are unlikely to be realized for a function call. These invariants reduce unnecessary replication of the local DUGs for call contexts that are unlikely to occur, thereby making context sensitive data-flow analysis more precise and scalable. These can be profiled by monitoring the state of call stack at the time of entering a function call.

We consider only those likely invariants which are not violated in any of our profiled executions. These invariant assumptions are used to prune the static analysis DUGs in our profile-driven pointer analysis as well as the predicated static data-flow analysis. The set of used invariants is recorded, and later used by the monitor instrumenter to add invariant checks.

We implement predicated versions of pointer analysis and forward data-flow analysis. The backward data-flow analysis is not predicated as required in §III-C, so Iodine uses a conservative backward data-flow analysis along with the predicated forward data-flow analysis.

By assuming these likely invariants, the DUG constructed for static analysis is unsound, and therefore much smaller than a DUG constructed for a traditional sound program analysis. This smaller DUG improves scalability and accuracy of our optimistic pointer and taint analyses, significantly reducing the aliasing-rate of pointers thereby improving accuracy and scalability [18]. The improvement in scalability is so significant that it allows us to scalably apply context-sensitive points-to and taint analysis for all our test programs, even further improving accuracy.

D. Optimistic Hybrid Taint Analysis

The predicated static taint analysis identifies the set of instructions that need to be monitored. We then instrument these monitors using LLVM DFSan [20] after the compiler has finished all its optimization passes. DFSan is a purely dynamic DIFT tool that provides a flexible interface to specify data labels and enforce custom information-flow policies. We modify DFSan's instrumentation framework to selectively process only those instructions which contribute to taint flows in our predicated static analysis, thereby effectively eliding the `noop` monitors. We chose DFSan as our state-of-the-art dynamic baseline due to its ease of integration with our LLVM

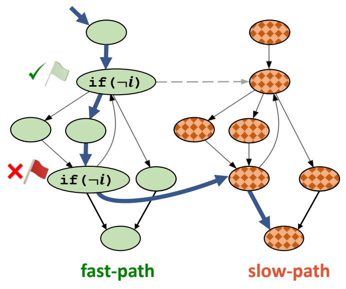


Fig. 4: Forward recovery switching mechanism: Each function implements fast-path and slow-path in separate control flow domains, and execution switches from fast-path to slow-path upon detecting an invariant violation.

static analysis framework. Iodine's optimizations are largely orthogonal to those of traditional dynamic DIFT tools, and will also likely proportionally improve the overheads of other competing dynamic tools such as libdft [34] and Minemu [14].

Metadata tracking: The runtime monitors track metadata for each program variable and memory locations at the byte-granularity in separate taint data structures in a shadow memory, and we only consider explicit taint flows [12]. We provide two options for tracking taints: (1) where each value can be assigned a single taint type with taint propagations being computed as logical `or` operations, and (2) where we track multiple taint types per location and compute taint propagations using the union mapping function [12] as bitwise logical operations on bit fields representing the taint sets.

Invariant checks: We also instrument invariant checks to detect invariant violations. Likely unreachable code is checked by instrumenting a special invariant violation function call at the entry of the code block. Likely callee sets are verified using a set inclusion check upon a function pointer update. Likely unrealized call contexts are verified by checking call-stack set inclusion at the call-sites with reasonable overheads [18].

E. *Forward Recovery Mechanism*

To manage switching between the slow-path and fast-path versions of our code, each function implements both the fast-path and slow-path code in separate control-flow domains, as shown in Figure 4. These two analyses paths are created statically for each function as follows. First the control flow graph for a function is replicated, and the fast-path version is instrumented with monitors resulting from the predicated static analysis, while the slow-path instrumentation uses the conservative static analysis. Next, immediately after every invariant check in the fast-path, we insert a conditional jump to the slow-path that is taken whenever the guarding invariant check fails. Note, that the slow-path has no invariant checks, as it uses a conservative static analysis.

Another key issue is to handle the slow-path switching correctly for function calls. When an invariant fails while executing a function that is deep in the function call-graph, that particular function can switch using the above mechanism. Additionally, all functions in the call-stack up to the `main` function must switch to the slow-path upon a return from the slow-path domain. To achieve this, we instrument a conditional switch to the slow-path after every call-site that checks a global flag upon function return and switches to the slow-path if that flag is raised by the invariant violation. We found the dynamic overhead of this simple recovery mechanism to be negligible. If this overhead were unacceptable, we could rewrite the return addresses on stack with those in the slow-path domain. As the observed overheads for our simple solution were low, we did not implement this more complicated strategy.

Execution begins in the fast-path domain and continues, as shown by the bold path in Figure 4, until it encounters an invariant violation, at which time it immediately switches to the slow-path domain and continues forward. This switch is safe due to two reasons: (1) the two domains only differ in analysis logic and maintain the same program state, and (2) safe elision guarantees equal analysis metadata state at invariant violation. Subsequent returns through the call-stack switches to the slow-path through the second mechanism.

VI. Evaluation

Our evaluation shows the following:

- Iodine enables production use of taint tracking by dramatically reducing the overhead of taint tracking compared to conservative hybrid analysis and pure dynamic analysis.
- Iodine efficiently implements real-world information-flow policies for security-critical applications.
- Iodine requires reasonable profiling efforts. We show regression tests are adequate to get majority of the performance benefits.
- Iodine improves the precision and scalability of static taint analysis.

A. *Experimental Setup*

We evaluate Iodine over several security-sensitive real-world applications. Our benchmark suite consists of the following:

- Postfix mail server [35] test generators–
  - **`qmqp-source`**, **`smtp-source`**: mail servers.
  - **`qmqp-sink`**, **`smtp-sink`**: mail clients.
  - **`sendmail`**: Postfix to Sendmail interface.
- **`nginx`**, **`thttpd`**: serving static webpages [36], [37].
- **`redis`**: database server [38] performing key-value store, list operations, and geographic search [39].
- **`vim`**: pattern search and text processing [40].
- **`gzip`**: (de-)compressing large media files [41].

We test Iodine in a manner that parallels how we envision it will be used in practice. We first profile a set of profiling executions to gather likely invariants. Then, we use these profiled invariants in a predicated static analysis to construct our final optimized dynamic taint analysis for a given information-flow policy. We generate a set of 500 diverse profile inputs by sweeping the programs' parameter space (e.g., data size, #clients, #requests, compression factor, etc.; excluding standardized parameters, e.g., TCP/SMTP port). We run the `postfix` stress tests; `nginx`, `thttpd` serving pydoc3 documentation and loading several webpages; `redis` benchmarking application and performing geo-search

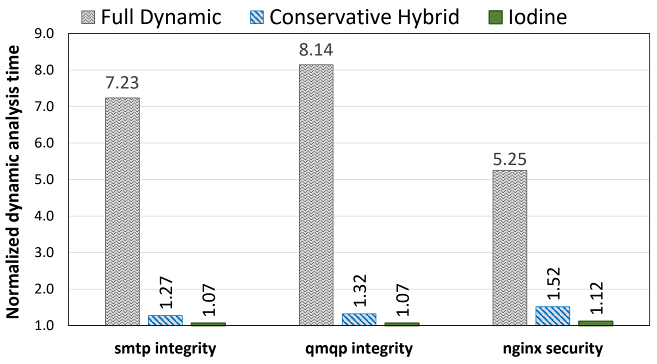


Fig. 5: Dynamic information-flow tracking applications

[39]; `vim` challenge solutions from [40]; and `gzip` with SPEC's `bzip2` and `sphinx` reference inputs. We randomly partition these inputs into two disjoint sets- a *profile set* consisting 400 executions, and a *performance test set* of 100 executions. We note that in an actual production environment the profiling overhead of Iodine would be amortized over all future executions of the program, not just the 100 we test.

To evaluate the benefits of Iodine, we compare it against a conservative hybrid information flow tracking (IFT) tool, and a naive dynamic IFT tool that uses no static analysis. Our conservative hybrid IFT runs sound variants of the same static algorithms Iodine uses, however as Iodine's predicated static analysis only analyzes a subset of program states, it can often run more scalable context-sensitive predicated static analyses, where sound static analysis must use context-insensitive analysis for scalability. For our native dynamic baseline we compare against our backend tool, DFSan[20] ($4.84\times$ avg. for SPECint). This overhead is comparable to other purely dynamic taint analysis systems, such as libdft [34] ($5.08\times$). For our conservative hybrid baseline we use DFSan with sound static optimizations. The overhead of our conservative hybrid system ($2.83\times$) is also comparable to prior conservative hybrid systems, such as TaintPipe [42] ($2.67\times$).

All experiments are run on a single core of an Intel Xeon E5-2620 processor with 16GB RAM running Linux 4.4.

B. IFT Security Policies

We demonstrate the effectiveness of Iodine using real taint policies by applying it to a set of commonly used applications with realistic taint policies adapted from Dytan [12] and Google desktop's privacy policy [43]. The policies we study are:

Email integrity and privacy: We add security checks to the Postfix mail server, following the policies outlined in [13], [43]. These policies ensure: receiver addresses are entirely determined by user input and message dates are only determined by the `time` system call (email integrity), and message bodies are passed through sanitizing functions that perform encryption, and check for unmatched HTML tags or scripting tags (privacy + security).

Overwrite attacks on web server: We enforce a taint policy on the Nginx web server that taints all network inputs, and asserts that tainted values are not used as function pointers, return addresses, or format strings. This policy detects a malicious overwrite attack [12].

Results: Iodine shows a $4.4\times$ reduction in runtime overhead for these realistic case studies, incurring only 7% to 12% overhead, compared to 27% to 52% obtained with conservative hybrid analysis. These results are shown in Figure 5, as well as those of a naive dynamic IFT analysis. With these significant runtime improvements Iodine enables taint tracking in many production systems where performance concerns often preclude security.

C. Generic Information-Flow Policies

As we only have a limited set of realistic taint policies, we further test Iodine's effectiveness in reducing taint overhead over additional benchmarks by using synthetic taint policies. We implement two different synthetic variants of taint analysis to evaluate the effectiveness of our framework in a forward-only analysis versus a forward-backward analysis.

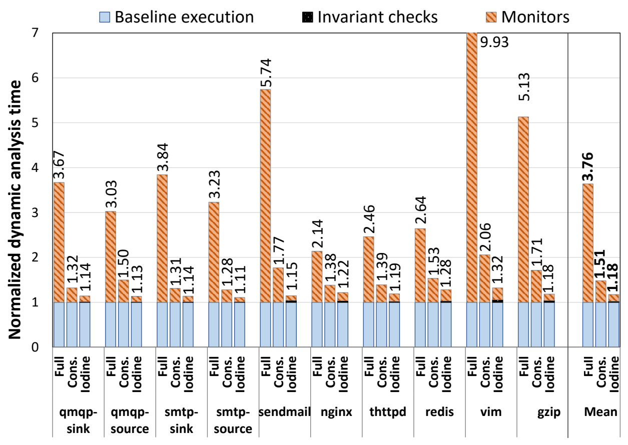
Some-to-some: Propagates taints from a randomly sampled fraction of the taint sources to the set of all sink instructions. Both forward and backward static taint analyses are used to implement this policy.

Some-to-all: Treats all instructions as potential sinks and propagates taints from the sampled taint sources. Only forward static taint optimizations are used to optimize this analysis.

Some-to-all taint policies are useful in many non-security contexts such as database provenance and lineage queries, information flow in debugging and software testing. This optimization also isolates the forward optimizations of our hybrid IFT framework, showing directly how effective predicated static analysis is at optimizing taint checks versus a sound static analysis. We treat all input interfaces from console/file/network as potential taint sources and elect to randomly sample $\frac{1}{3}$ of them for these taint policies. All output interfaces to console/file/network are taint sinks. We find this sampling gives us overhead numbers similar to the realistic policies evaluated previously. All subsequent results are based on the above generic policies. We consistently use the $\frac{1}{3}$ sampling fraction and the same set of sampled taint sources for a program in all subsequent experiments, except in §VI-H where we change this parameter.

Results: Similar to our taint policy tests, Iodine significantly reduces the runtime of dynamic taint tracking in our synthetic tests, as shown in Figure 6. When applied to some-to-some taint tracking (Figure 6a), Iodine reduces the dynamic overhead of conservative hybrid taint analysis by $2.8\times$, bringing the overhead of taint tracking from 51% with conservative hybrid analysis down to 18% over native unmonitored execution. We also apply Iodine to some-to-all taint tracking (Figure 6b). Iodine sees similar reductions in overhead, reducing overhead of taint tracking to 24%, versus 92% for conservative hybrid analysis, and 276% for a pure dynamic analysis. Once again, Iodine brings overheads down significantly, further showing its capability to reduce overheads and enable taint-tracking on production systems.

SPEC benchmarks: To further evaluate Iodine's performance on compute-intensive programs, we run it with the


(a) some-to-some taint analysis

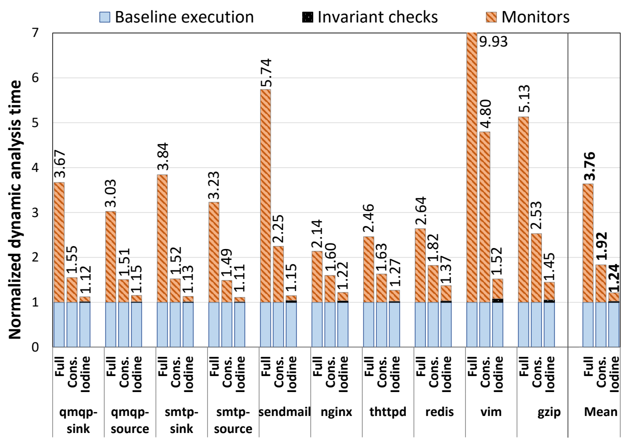

(b) some-to-all taint analysis

Fig. 6: Iodine compared to pure (full) dynamic DIFT and conservative hybrid DIFT.

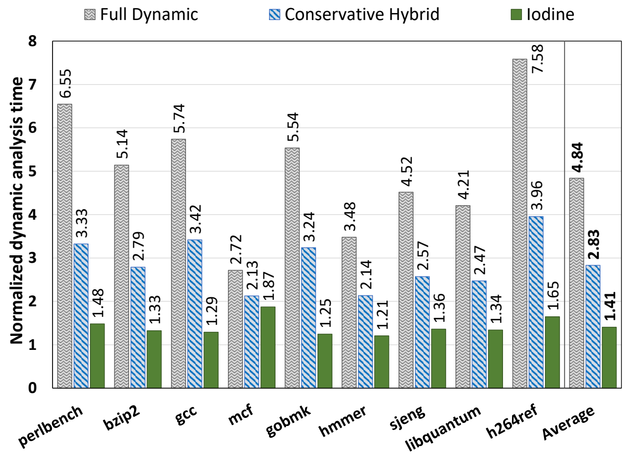


Fig. 7: Taint tracking performance on SPECint C benchmarks

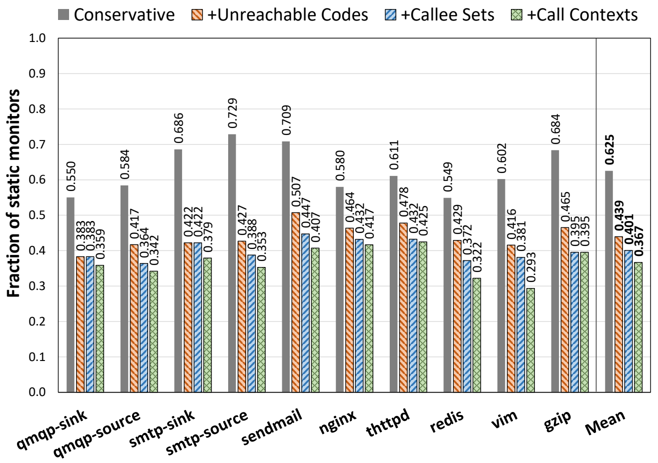


Fig. 8: Improved static analysis precision by assuming different invariants for some-to-some analysis. Conservative analysis uses a context-insensitive pointer-analysis, while the predicated analysis can scalably apply a context-sensitive pointer analysis.

same randomized some-to-some analysis setup on the SPECint benchmarks that are written in C with reference inputs. The results of these experiments are shown in Figure 7. The SPECint benchmarks are tuned to be CPU bound, and therefore exhibit higher DIFT overheads compared to our other case studies. Iodine improves the dynamic overhead of taint analysis by $4.5\times$, bringing the overhead of taint tracking over unmonitored execution from 183% with conservative hybrid analysis down to 41%. This speedup is in fact much higher than the $2.8\times$ speedup for our other programs.

Comparison to ideal analysis: We show that Iodine's optimization mechanism is so efficient it approaches optimal by comparing Iodine's some-to-all results with a dynamically gathered optimal analysis. Our optimal analysis only monitors instructions that are dynamically found to propagate taint, the very minimum set of instructions a some-to-all analysis could gather. We measure the average dynamic overhead of this ideal some-to-all taint analysis to be 13%. This shows that at 24% overhead, Iodine is 86% closer to optimal than traditional hybrid's 92%, and beginning to truly approach the realm of optimal dynamic taint analysis.

D. Memory Overheads

Iodine maintains the exact metadata state as a conservative analysis. Therefore, the memory space overhead of metadata tracking remains unchanged. Iodine does increase code-size by generating two versions of the code: the fast-path and slow-path. However, as only one version of the code is executed at a time, this has little impact on the caching behavior or performance of the program. On average, the code footprint of a program instrumented by Iodine increases by $2.1\times$, compared to $1.4\times$ with conservative hybrid taint analysis, and $1.8\times$ with pure dynamic taint analysis.

E. Iodine's Framework Overheads

Invariant Check Cost: Figure 6 also isolates the invariant checking costs. Invariant checks are only required in Iodine's optimistic analysis framework and are absent from the full

TABLE I: Static analysis time breakups for some-to-some taint analysis

Benchmark	Conservative Static Analysis			Predicated Static Analysis			
	Points-to	Taint	**Total**	Profiling	Points-to†	Taint	**Total**
`qmqp-sink`	8s	4m 28s	4m 36s	1m 19s	12s	36s	2m 07s
`qmqp-source`	7s	14m 18s	14m 25s	1m 45s	5s	1m 12s	3m 02s
`smtp-sink`	9s	6m 12s	6m 21s	2m 00s	16s	44s	3m 39s
`smtp-source`	11s	11m 44s	11m 55s	2m 19s	9s	1m 08s	3m 35s
`sendmail`	15s	16m 53s	17m 08s	2m 02s	13s	1m 37s	4m 32s
`nginx`	19s	20m 04s	20m 24s	1m 12s	12s	1m 30s	2m 54s
`thttpd`	18s	17m 54s	18m 12s	59s	16s	1m 14s	2m 29s
`redis`	1m 18s	19m 43s	21m 01s	2m 01s	10s	1m 25s	3m 35s
`vim`	32s	61m 22s	61m 54s	5m 12s	88s	2m 54s	9m 35s
`gzip`	8s	8m 49s	8m 58s	7m 03s	17s	1m 22s	8m 42s

†Our optimistic framework enables us to scalably apply more accurate context-sensitive points-to analysis during the predicated static analysis

dynamic and conservative hybrid analysis. Overall we observe that invariant checks have nearly no effect on end runtime, incurring only 2% of overall execution time.

Checking likely-unreachable-code incurs almost no overhead. Checking likely-callee-sets requires small set inclusion checks upon function pointer updates, which are fairly rare. Likely-unused-call-contexts involve checking if current call-context is among those assumed during static-analysis, and we optimize it by hashing the call-stack to lookup a Bloom-filter thereby avoiding majority of expensive checks [18].

Invariant Violations and Switching Overhead: Overall Iodine observes largely inconsequential rates of invariant violations, with only `sendmail`, `redis` and `vim` violating an invariant during some-to-all analysis in 3, 2, and 5 (out of 100) executions respectively. This indicates that our profiling methodology captures the common-case dynamic execution behavior effectively, sgnfiicantly optimizing the dynamic analysis. The amortized overhead of the slow path analysis resulting from these violations is less than 0.5%. Note that the slow-path overhead can be no worse than that of conservative hybrid analysis.

We also find that the runtime overhead of the switching mechanism at function call return sites, discussed in §V-E, is negligible.

F. Precise and Scalable Static Analysis

Figure 8 shows how assuming different types of invariants successively reduces the number of required static monitors for a some-to-some taint analysis. While the conservative static analysis requires instrumenting 63% instructions on average, our predicated static taint analysis nearly halves this value at 37%, providing the foundation for Iodine's impressive performance results. This translates to eliding 54%(`nginx`)−86%(`vim`) of the dynamic taint checks from a conservatively optimized analysis.

Table I summarizes the breakdown of static analysis times for both the conservative static and our predicated static versions. Applying the invariant assumptions to constrain the static analysis search space enables us to scalably apply a context-sensitive pointer analysis. This further improves the precision of our predicated static analysis. We see that a reasonable effort spent in profiling significantly reduces the overall static taint analysis time. In fact, the total static analysis time including the profiling time is lower than that of conservative static analysis for all our test programs. This makes Iodine suitable for deployment in production where the applications are constantly evolving thereby requiring re-analyzing them statically for hybrid analysis.

G. Profiling During Regression Testing Is Effective

An important concern with profile-based optimizations is the time and effort spent in profiling as well as the system's sensitivity to the profile set. Unlike profile-based optimizations that try to capture frequently executed program states, we propose a conservative but effective methodology that attempts to capture all feasible program states. We observe that software regression tests seek to maximize code and path coverage, and are therefore good candidates for conservative profiling.

We evaluate this approach by profiling three programs-`nginx`, `redis` on their packaged regression test suites, and `vim` on open-source test suites [44], [45]. The results in Figure 9 show that profiling on regression test suites alone is very effective. It reduces the runtime overhead to 31% compared to 55% with conservative hybrid analysis. We however observe invariants being violated dynamically after this profiling, and so recommend further profiling on beta tests. We use our original set of 400 profile inputs (in §VI-A) as a proxy for beta testing. Profiling on the beta tests (shaded right halves) reduces the invariant violation rate significantly and brings down the analysis overhead to 23%.

Thus, we leverage the existing software testing suites to perform Iodine's initial profiling, and recommend reasonable beta testing for learning invariants to optimize Iodine. Mature software systems have well designed regression test suites that attain high code coverage. As a result, we find that our likely unreachable code invariants were learnt accurately after profiling regression tests. Most invariant violations after profiling on regression test suites were due to likely callee sets and likely unrealized call contexts, mainly because test suites often invoke many functions only in some mock context. Profiling on a few actual executions during beta tests easily overcomes this. Alternatively, improving test suites that attain high coverage for calling contexts [46] can be effective, as can learning invariants during production runs.

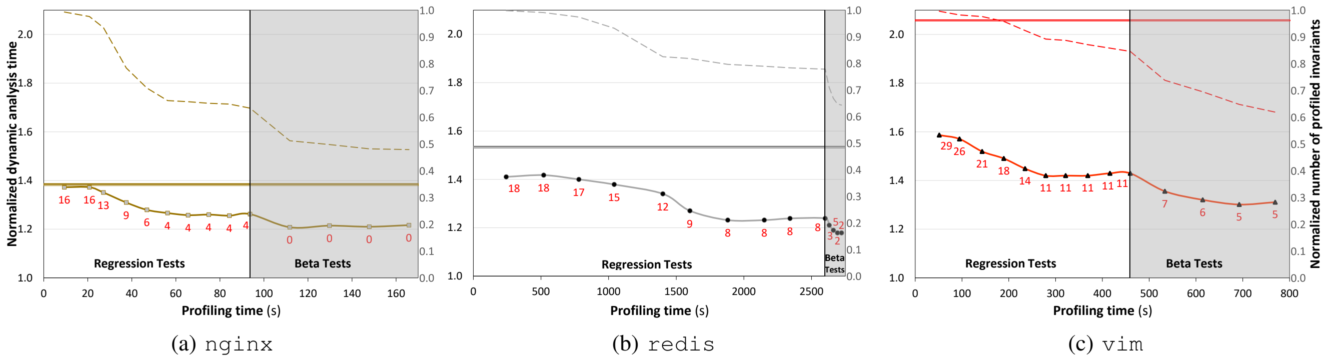


Fig. 9: Profiling invariants while software testing: Profiling is done in two phases- first on regression test suites (left unshaded), and then on beta tests (right shaded). The solid marked lines plot analysis overheads with Iodine using invariants gathered at different stages of profiling. The numbers labeled on the plot indicate the number of dynamic invariant violations. The horizontal solid lines representing conservative hybrid analysis are an upper bound to Iodine's runtime overheads. The dashed lines against the secondary (right) y-axis plot the number of invariants used normalized to that after profiling a single execution.

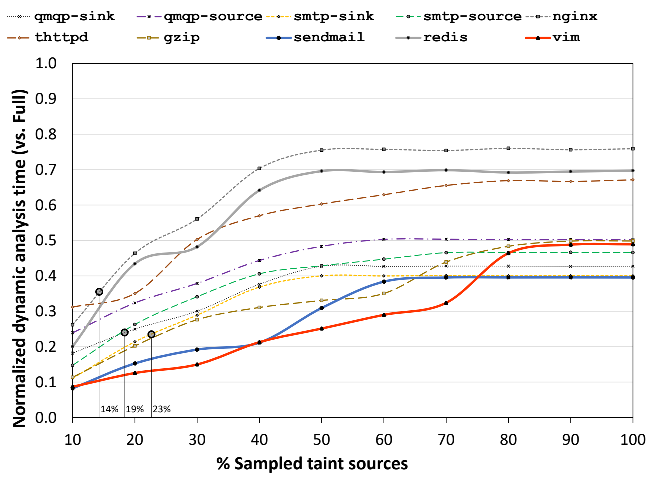


Fig. 10: Iodine's performance benefits reduce with larger fractions of the program's data space being tainted. Fraction of taints observed for realistic taint policies (§VI-B) are annotated.

Moreover, Iodine is resilient to weak profiling. Our analysis needs no guarantees that all states are profiled; and even if the invariants fail dynamically, the constructed optimized analysis is still sound. Failing invariants can be learned over time and the optimized analysis can be adaptively re-constructed to exclude those without requiring analysis rollbacks (§VII). Unlike a carefully crafted 'bootstrapping' process that ultimately determines its effectiveness, Iodine requires test suites with reasonable coverage for profiling, and is moreover resilient to profiling inaccuracies.

H. Sensitivity to Fraction of Tainted Data

Hybrid analyses (both traditional and Iodine) elide instrumentation that cannot propagate taint. As a growing set of inputs carry taints, the taints spread faster to nearly the program's entire data space. If nearly all data is tainted, there is no optimization opportunity and Iodine fails to effectively elide taint checks. To investigate this behavior, in Figure 10, we look at how Iodine's normalized runtime varies with increasing the taint sampling fraction in our some-to-all taint analysis in §VI-C. We statically identify all viable taint sources (input interfaces from console/file/network) and randomly sample the stipulated fraction of them to be active. Since selected sources might vary in their dynamic execution frequencies, we run on 100 different samples for a given sampling fraction (except for 100%). As expected, we observe that Iodine's performance degrades in general when dealing with larger fraction of tainted inputs, although Iodine shows significant benefits for many realistic levels of tainted input. This behavior is fundamental to hybrid analysis, and is no worse in Iodine than in a conservative hybrid analysis.

Iodine is effective when the target program and the taint policy induce a low fraction of tainted data. We observe that this property indeed holds for the IFT security policies studied in §VI-B; the static fraction of active taint sources therein are between 14-23% (circled in Figure 10).

VII. Discussion

Limitations: Our implementation is for programs written in the C language, and we currently do not support native programs written in assembly. OHA's principles are however generally applicable to static analyses of x86 binaries [47], and has been shown to benefit other analyses for Java programs as well [18].

Support for multi-threaded programs require a concurrency analysis on top of Iodine's information-flow analysis, and prior work [18] has shown that OHA can benefit there as well.

Rarity and severity of invariant violations: For well-tested software, invariants should rarely fail as profiles would have captured the common-case program states. However for moderately large software with diverse features, optimistically gathered invariants may eventually fail when the program encounters unprofiled behavior. If this happens, Iodine switches to a conservative hybrid analysis. Thus, even in the worst case, Iodine is still as fast as the best available conservative hybrid technique.

Currently, owing to the minuscule invariant violation rate, Iodine conservatively continues on the slow-path until the next system reboot when it switches back to the fast-path. While this is no worse than conservative hybrid analysis, we envision the following strategies to preserve most of Iodine's performance benefits even after an invariant violation.

Background re-analysis: Upon an invariant-failure, we can 'learn' this new behavior and re-analyze the program without the offending invariant. For many useful static analyses, this can be done incrementally rather than redoing from scratch [48]. For a dataflow analysis like Iodine's, this boils down to adding new nodes and edges to the programs' definition-use graph, and recomputing the transitive closure. The re-compilation process can continue in the background while the monitored program runs slowly. Upon completion of the re-compilation process, the program can switch to the newly optimized analysis at a pre-determined safe program point.

Graceful degradation: Upon an invariant-failure, instead of switching to the most conservatively optimized analysis, we can switch to a less aggressive optimistic analysis that excludes the failing invariant. Even better, if we can compute and succinctly encode the mapping between assumed invariants and the set of induced optimizations, we can selectively disable only those optimizations induced by the violated invariant, essentially re-instrumenting the monitors that were elided by assuming that invariant. Dynamically re-instrumenting the new analysis on-demand also eliminates the memory overhead of maintaining multiple analysis versions.

Iodine can be implemented at the runtime system layer wherein the invariant violation handler can invoke a dynamic instrumentation framework to re-instrument the modified analysis, thereby opening opportunities to further benefit from JIT compilation techniques [49]. Such a setup further opens the possibility to actively learn new invariants and re-optimize the analysis.

VIII. Related Work

Iodine builds on the prior optimistic hybrid analysis work [18] in two major ways- (1) it constructs a rollback-free OHA by limiting to only *safe elision* optimizations thereby solving the recovery problem in OHA, and (2) applies this novel technique to realize a low overhead DIFT solution for live executions. Below, we discuss relevant prior work on DIFT, hybrid program analyses, and profile-based optimizations.

Dynamic Analysis: There has been significant work on dynamic taint tracking systems [9], [12], [50]. Past work has developed many optimized dynamic techniques, such as creating highly specific information-flow policies [5], [6], [11], reducing its scope to only apply to related processes [51], optimizing low-level taint operations [34], writing minimal emulators targeted for taint tracking [14], or even providing custom hardware support [10], [52]–[54]. All of these optimizations operate purely on the dynamic state of the program, attempting to make existing set of taint operations faster. Iodine elides taint operations through static analysis, reducing the set of instructions monitored, making its optimization complementary to these prior approaches.

Taint tracking has also been parallelized either by partitioning the execution into epochs to perform local analysis and then aggregating results [15], [55], or by decoupling taint analysis from the program execution [16], [17], [56], [57], wherein the dynamic instrumentation only performs lightweight logging followed by an offline analysis. These efforts reduce latency of taint tracking through parallelization, but not overall work, like Iodine does. They too are complimentary to Iodine's optimizations.

Static Analysis: Several systems have attempted to solve taint tracking using language features to enforce a taint policy at compile-time, sometimes with limited dynamic checks [58], [59]. These systems achieve low runtime overhead, but place the burden on the programmer to specify and guarantee taint policy using an unfamiliar restrictive language. Iodine optimizes dynamic analysis, and does not require source code changes, other than trivial annotations specifying taint sources, sinks, and untaint functions.

Hybrid Analysis: Hybrid analysis has been explored in the past [60] for accelerating DIFT. Moore et al. provide the soundness conditions for static analysis to determine when it is safe to stop tracking certain variables dynamically [19]. In addition to removing unnecessary monitors using static analysis, Chang et al. statically transform untrusted programs into policy-enforcing programs to further reduce the amount of data to be tracked dynamically [61]. Jee et al. statically separate the taint tracking logic from the program logic and then optimize it using abstract taint flow algebra [16]. Hybrid systems have also coalesced taint checks through static analysis [4], [42]. While these traditional hybrid analyses use sound static analysis to conservatively reduce dynamic overheads, Iodine further improves runtime overheads with use of unsound, predicated static analysis. Iodine's use of optimistic hybrid analysis with forward recovery could likely be combined with these systems for further taint optimizations.

Blended analysis [62] uses dynamic information to improve the accuracy of a best-effort static taint checking tool for JavaScript applications [63]. While they utilize dynamic information to make static analysis tractable for corner-case dynamic language features (e.g., `eval`), our likely invariants captures common program behaviors to improve whole-program static analysis. Moreover, their end goal is just to improve static analysis, and stop short of optimizing dynamic analysis. They also do not provide soundness or completeness guarantees for any results produced. Iodine produces sound and complete dynamic analysis for live executions.

Profile-guided Compiler Optimizations: Profile-guided optimizations [64], [65] learn invariants through profiling and use them for local optimizations. In particular, work on JIT optimizing compilers such as those that speculatively inline functions [66], or speculatively convert indirect function calls to direct function calls [67], speculatively optimize execution, as done in Iodine. Our work differs in two key ways. First, while compiler optimizations focus on optimizing program logic, Iodine aims at eliding unnecessary runtime DIFT monitors. A more fundamental difference is that Iodine uses invariants to improve precision and scalability of whole-program static analysis. In contrast, profile-guided optimizations do not typically consider whole-program static analysis, and therefore the methods for checking invariants and recovery are simpler and cheaper than optimistic hybrid analysis.

IX. Conclusion

We presented a novel optimistic hybrid analysis (OHA) technique to optimize DIFT. We solve a key challenge that limits applying OHA to online analyses on live executions — rollback recovery. We eliminated the need for rollbacks by restricting our predicated static analysis optimizations to `noop` *safe elisions*. Iodine significantly improves the precision of static data-flow and pointer analysis, thereby drastically reducing DIFT overhead for important security policies to 9%.

Acknowledgment

We thank the anonymous reviewers and our shepherd, Deian Stefan, for their thoughtful comments. This work was supported by the National Science Foundation under grants SHF-1527301 and SHF-1703931. The views and conclusions contained in this paper are solely those of the authors.

References

[1] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2004, Boston, MA, USA, October 7-13, 2004*, 2004, pp. 85–96.

[2] J. Newsome and D. X. Song, "Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2005, San Diego, California, USA*, 2005.

[3] J. Kong, C. C. Zou, and H. Zhou, "Improving software security via runtime instruction-level taint checking," in *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability, ASID 2006, San Jose, California, USA, October 21, 2006*, 2006, pp. 18–24.

[4] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu, "LIFT: A low-overhead practical information flow tracking system for detecting security attacks," in *39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-39 2006), 9-13 December 2006, Orlando, Florida, USA*, 2006, pp. 135–148.

[5] W. G. J. Halfond, A. Orso, and P. Manolios, "Using positive tainting and syntax-aware evaluation to counter SQL injection attacks," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2006, Portland, Oregon, USA, November 5-11, 2006*, 2006, pp. 175–185.

[6] T. Pietraszek and C. V. Berghe, "Defending against injection attacks through context-sensitive string evaluation," in *Recent Advances in Intrusion Detection, 8th International Symposium, RAID 2005, Seattle, WA, USA, September 7-9, 2005, Revised Papers*, 2005, pp. 124–145.

[7] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans, "Automatically hardening web applications using precise tainting," in *Security and Privacy in the Age of Ubiquitous Computing, IFIP TC11 20th International Conference on Information Security (SEC 2005), May 30 - June 1, 2005, Chiba, Japan*, 2005, pp. 295–308.

[8] V. Haldar, D. Chandra, and M. Franz, "Dynamic taint propagation for java," in *21st Annual Computer Security Applications Conference (ACSAC 2005), 5-9 December 2005, Tucson, AZ, USA*, 2005, pp. 303–311.

[9] W. Enck, P. Gilbert, B. Chun, L. P. Cox, J. Jung, P. D. McDaniel, and A. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, 2010, pp. 393–407.

[10] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August, "RIFLE: an architectural framework for user-centric information-flow security," in *37th Annual International Symposium on Microarchitecture (MICRO-37 2004), 4-8 December 2004, Portland, OR, USA*, 2004, pp. 243–254.

[11] D. Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall, "Tainteraser: protecting sensitive data leaks using application-level taint tracking," *Operating Systems Review*, vol. 45, no. 1, pp. 142–154, 2011.

[12] J. A. Clause, W. Li, and A. Orso, "Dytan: a generic dynamic taint analysis framework," in *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2007, London, UK, July 9-12, 2007*, 2007, pp. 196–206.

[13] H. Yin, D. X. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: capturing system-wide information flow for malware detection and analysis," in *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*, 2007, pp. 116–127.

[14] E. Bosman, A. Slowinska, and H. Bos, "Minemu: The world's fastest taint tracker," in *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection RAID 2011*, 2011.

[15] O. Ruwase, P. B. Gibbons, T. C. Mowry, V. Ramachandran, S. Chen, M. Kozuch, and M. P. Ryan, "Parallelizing dynamic information flow tracking," in *SPAA 2008: Proceedings of the 20th Annual ACM Symposium on Parallelism in Algorithms and Architectures, Munich, Germany, June 14-16, 2008*, 2008, pp. 35–45.

[16] K. Jee, V. P. Kemerlis, A. D. Keromytis, and G. Portokalidis, "Shadowreplica: efficient parallelization of dynamic data flow tracking," in *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, 2013, pp. 235–246.

[17] J. Ming, D. Wu, J. Wang, G. Xiao, and P. Liu, "Straighttaint: decoupled offline symbolic taint analysis," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, 2016, pp. 308–319.

[18] D. Devecsery, P. M. Chen, J. Flinn, and S. Narayanasamy, "Optimistic hybrid analysis: Accelerating dynamic analysis through predicated static analysis," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '18, March 2428, 2018, Williamsburg, VA, USA*, 2018.

[19] S. Moore and S. Chong, "Static analysis for efficient hybrid information-flow control," in *Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011, Cernay-la-Ville, France, 27-29 June, 2011*, 2011, pp. 146–160.

[20] "DFSan. Clang DataFlowSanitizer," http://clang.llvm.org/docs/DataFlowSanitizer.html.

[21] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, 2010, pp. 317–331.

[22] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum, "Understanding data lifetime via whole system simulation (awarded best paper!)," in *Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA*, 2004, pp. 321–336.

[23] T. Leek, G. Baker, R. Brown, M. Zhivich, and R. Lippmann, "Coverage maximization using dynamic taint tracing," MIT Lincoln Laboratory, Tech. Rep. TR-1112, 2007.

[24] W. Masri, A. Podgurski, and D. Leon, "Detecting and debugging insecure information flows," in *15th International Symposium on Software Reliability Engineering (ISSRE 2004), 2-5 November 2004, Saint-Malo, Bretagne, France*, 2004, pp. 198–209.

[25] B. Alpern, M. N. Wegman, and F. K. Zadeck, "Detecting equality of variables in programs," in *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, 1988, pp. 1–11.

[26] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, "Microreboot - A technique for cheap recovery," in *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, 2004, pp. 31–44.

[27] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, 2004, pp. 75–88.

[28] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of program analysis*. Springer, 1999. [Online]. Available: https://doi.org/10.1007/978-3-662-03811-6

[29] L. O. Andersen, "Program analysis and specialization for the c programming laguage," in *PhD thesis, DIKU, University of Copenhagen*, 1994.

[30] C. Lattner, A. Lenharth, and V. S. Adve, "Making context-sensitive points-to analysis with heap cloning practical for the real world," in *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, 2007, pp. 278–289.

[31] B. Hardekopf and C. Lin, "Exploiting pointer and location equivalence to optimize pointer analysis," in *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings*, 2007, pp. 265–280.

[32] ——, "The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code," in *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, 2007, pp. 290–299.

[33] M. Berndl, O. Lhoták, F. Qian, L. J. Hendren, and N. Umanee, "Points-to analysis using bdds," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, 2003, pp. 103–114.

[34] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, "libdft: practical dynamic data flow tracking for commodity systems," in *Proceedings of the 8th International Conference on Virtual Execution Environments, VEE 2012, London, UK, March 3-4, 2012 (co-located with ASPLOS 2012)*, 2012, pp. 121–132.

[35] "The Postfix mail server," http://www.postfix.org.

[36] "The NGINX web server," https://www.nginx.com.

[37] "The Tiny HTTP server," https://acme.com/software/thttpd.

[38] "The Redis database server," https://redis.io.

[39] "The Tile38 geolocation information systems," http://tile38.com.

[40] "VimGolf," https://vimgolf.com.

[41] "The GNU Gzip data compression utility," https://www.gnu.org/software/gzip.

[42] J. Ming, D. Wu, G. Xiao, J. Wang, and P. Liu, "Taintpipe: Pipelined symbolic taint analysis," in *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, 2015, pp. 65–80.

[43] "Google desktop - privacy policy," http://desktop.google.com/en/privacypolicy.html.

[44] "VRoom," https://github.com/google/vroom.

[45] "Run Vim Tests," https://github.com/inkarkat/runVimTests.

[46] A. M. Smith, J. Geiger, G. M. Kapfhammer, and M. L. Soffa, "Test suite reduction and prioritization with call trees," in *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*, 2007, pp. 539–540.

[47] G. Balakrishnan and T. W. Reps, "WYSINWYX: what you see is not what you execute," *ACM Trans. Program. Lang. Syst.*, vol. 32, no. 6, pp. 23:1–23:84, 2010.

[48] M. G. Burke, "An interval-based approach to exhaustive and incremental interprocedural data-flow analysis," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 341–395, 1990.

[49] C. Kerschbaumer, E. Hennigan, P. Larsen, S. Brunthaler, and M. Franz, "Information flow tracking meets just-in-time compilation," *TACO*, vol. 10, no. 4, pp. 38:1–38:25, 2013.

[50] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige, "Tainttrace: Efficient flow tracing with dynamic binary rewriting," in *Proceedings of the 11th IEEE Symposium on Computers and Communications (ISCC 2006), 26-29 June 2006, Cagliari, Sardinia, Italy*, 2006, pp. 749–754.

[51] Y. Ji, S. Lee, E. Downing, W. Wang, M. Fazzini, T. Kim, A. Orso, and W. Lee, "RAIN: refinable attack investigation with on-demand inter-process information flow tracking," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, 2017, pp. 377–390.

[52] V. Nagarajan, H.-S. Kim, Y. Wu, and R. Gupta, "Dynamic information flow tracking on multicores," in *Proceedings of the 2008 Workshop on Interaction between Compilers and Computer Architectures*, 2008.

[53] K. Jee, G. Portokalidis, V. P. Kemerlis, S. Ghosh, D. I. August, and A. D. Keromytis, "A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware," in *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*, 2012.

[54] J. Lee, I. Heo, Y. Lee, and Y. Paek, "Efficient dynamic information flow tracking on a processor with core debug interface," in *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*, 2015, pp. 79:1–79:6.

[55] A. Quinn, D. Devecsery, P. M. Chen, and J. Flinn, "Jetstream: Cluster-scale parallelization of information flow queries," in *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, 2016, pp. 451–466.

[56] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn, "Parallelizing security checks on commodity hardware," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008*, 2008, pp. 308–318.

[57] S. Ma, X. Zhang, and D. Xu, "ProTracer: Towards practical provenance tracing by alternating between logging and tainting," in *23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, 2016.

[58] A. C. Myers, "Jflow: Practical mostly-static information flow control," in *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, 1999, pp. 228–241.

[59] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, 2003.

[60] M. D. Ernst, "Static and dynamic analysis: synergy and duality," in *ICSE WORKSHOP ON DYNAMIC ANALYSIS (WODA 2003)*, 2003, pp. 24–27.

[61] W. Chang, B. Streiff, and C. Lin, "Efficient and extensible security enforcement using dynamic data flow analysis," in *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, 2008, pp. 39–50.

[62] B. Dufour, B. G. Ryder, and G. Sevitsky, "Blended analysis for performance understanding of framework-based applications," in *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2007, London, UK, July 9-12, 2007*, 2007, pp. 118–128.

[63] S. Wei and B. G. Ryder, "Practical blended taint analysis for javascript," in *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013*, 2013, pp. 336–346.

[64] B. Calder, P. Feller, and A. Eustace, "Value profiling," in *Proceedings of the Thirtieth Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 30, Research Triangle Park, North Carolina, USA, December 1-3, 1997*, 1997, pp. 259–269.

[65] M. Mock, M. Das, C. Chambers, and S. J. Eggers, "Dynamic points-to sets: a comparison with static analyses and potential applications in program understanding and optimization," in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'01, Snowbird, Utah, USA, June 18-19, 2001*, 2001, pp. 66–72.

[66] M. G. Burke, J. Choi, S. J. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley, "The jalapeño dynamic optimizing compiler for java," in *Java Grande*, 1999, pp. 129–141.

[67] C. Chambers and D. Ungar, "Customization: Optimizing compiler technology for self, A dynamically-typed object-oriented programming language," in *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation (PLDI), Portland, Oregon, USA, June 21-23, 1989*, 1989, pp. 146–160.