Hybrid Pruning: Towards Precise Pointer and Taint Analysis

Dipanjan Das[1], Priyanka Bose[1], Aravind Machiry[2], Sebastiano Mariani[3], Yan Shoshitaishvili[4], Giovanni Vigna[1], and Christopher Kruegel[1]

[1] University of California, Santa Barbara, California, USA
{dipanjan,priyanka,vigna,chris}@cs.ucsb.edu
[2] Purdue University, Indiana, USA
amachiry@purdue.edu
[3] VMware, Inc.
smariani@vmware.com
[4] Arizona State University, Arizona, USA
yans@asu.edu

Abstract. Pointer and taint analyses are the building blocks for several other static analysis techniques. Unfortunately, these techniques frequently sacrifice *precision* in favor of *scalability* by over-approximating program behaviors. Scaling these analyses to real-world codebases written in memory-unsafe languages while retaining precision under the constraint of practical time and resource budgets is an open problem.
In this paper, we present a novel technique called *hybrid pruning*, where we inject the information collected from a program's dynamic trace, which is accurate by its very nature, into a static pointer or taint analysis system to enhance its precision. We also tackle the challenge of combining static and dynamic analyses, which operate in two different analysis domains, in order to make the interleaving possible. Finally, we show the usefulness of our approach by reducing the false positives emitted by a static vulnerability detector that consumes the improved points-to and taint information. On our dataset of 12 `CGC` and 8 real-world applications, our hybrid approach cuts down the warnings up to 21% over vanilla static analysis, while reporting 19 out of 20 bugs in total.

Keywords: Pointer analysis · Taint analysis · Static vulnerability detection

1 Introduction

Pointer analysis is a fundamental static program analysis technique that computes the set of abstract program objects that a pointer variable may or must point to. Pointer information is an indispensable pre-requisite for various techniques operating across a spectrum of domains, ranging from programming languages, to software engineering, to system security. One such notable client is taint analysis, which determines the set of objects in a program that are affected by external inputs. The analysis is bootstrapped by marking an initial set of

objects that can directly be influenced by an external source (*e.g.*, an attacker) as *tainted*. During taint propagation, the taint engine consults the points-to set of the destination operand of a program instruction, and propagates taint labels according to the taint policy, and the taint labels of the source operands. Therefore, an over-approximated points-to set quickly leads to taint explosion, resulting in most of the program objects getting incorrectly tainted. Many static vulnerability detection techniques employ either pointer, or taint analysis, or a combination of both [29]. In order to not miss bugs, these techniques strive to be *sound*, rather than *complete*. Consequently, such vulnerability detection clients generate numerous false positives. A precise pointer or taint analysis improves the false positive rate of a static vulnerability detector, thereby making the overall result more amenable to manual triaging.

As the size of the target program grows, precise, whole program pointer and taint analyses become prohibitively expensive. Though field, context, or flow sensitivity increases the analysis precision, such an analysis pays the price in terms of the overhead associated with the metadata management, and enumeration of individual field, context, or flow. Oftentimes, the analyses make unsound choices in order to remain scalable, *e.g.*, restricting the exploration within a specific sub-system, or making certain *soundy* assumptions [29] .

In this paper, we propose *hybrid pruning* – a novel program analysis paradigm that augments the state-of-the-art static analysis techniques with dynamic trace information. Our algorithm improves both the pointer and taint analyses at those program points where static reasoning is imprecise, and precise dynamic information is available. With the recent tide of research in fuzzing, it has become easier to generate high-quality dynamic traces with deeper program penetration. If the dynamic trace is available along a certain program path, our algorithm injects guaranteed, precise yet partial ground truth to aid the static analysis component. Although inherently *unsound* in principle, our strategy transitively improves the analysis at all those program points which were previously using the imprecise static information, thus multiplicating the advantage. However, leveraging a dynamic trace for static analysis is non-trivial, as they operate in two different analysis domains, *e.g.*, concrete instructions and run-time memory allocations *vs.* `SSA`-based `IR` and abstract memory objects. Our approach lifts the dynamic trace to the static domain to make the interleaving possible. Of course, dynamic analysis tools, such as fuzzers, will likely not succeed in exercising all possible program paths. To compensate for the lack of dynamic coverage, we fall back to the conservative static analysis for all other program paths for which a dynamic trace is absent. We demonstrate two different modes of *hybrid pruning* – *opportunistic* ($\mathtt{H_o}$) and *propagation-only* ($\mathtt{H_P}$), and show when one is better than the other depending on the quality of the dynamic trace collected. These two modes operate along a spectrum of *soundness* and *usability.* The improvement in points-to and taint analyses is positively correlated with the dynamic coverage. If the dynamic coverage is moderate, the $\mathtt{H_o}$ mode is preferred. This mode is designed to be more robust against the lack of dynamic information, because it conservatively switches to pure static mode where dynamic information is

unavailable. On the other hand, the $\mathtt{H_P}$ mode shows promise when we have high confidence in the quality of dynamic information, as just the dynamic facts are propagated using the static analysis algorithms in this mode.

Our work is motivated by the observation that the static bug detectors are oftentimes notorious in emitting warnings in a volume which far surpasses the triaging ability of the human experts. For example, as on May 7, 2022, Coverity [3], a popular static bug detector, has emitted 47,038 warnings in the Linux kernel version 5.18.0-rc4, of which 9,137 are still outstanding. We envision *hybrid pruning* as a technique to improve the state-of-the-art in the static bug detection. Therefore, to evaluate the applicability of our technique in the real world, we extended Dr.Checker [29], a purely static bug finder, to make use of *hybrid pruning.* As we anticipated, the precise points-to and taint information indeed reduced the number of false positives, while maintaining a comparable true positive rate. On our evaluation of 12 `CGC` [13] applications, the bug detectors relying on the $\mathtt{H_o}$ mode emit up to 36% less warnings, while the $\mathtt{H_P}$ mode reduces warnings up to 56%. We additionally show that, in spite of reducing significant fraction of warnings, the vulnerability detectors are still able to detect 15 ($\mathtt{H_P}$) and 19 ($\mathtt{H_o}$) out of 20 bugs in the `CGC` [13] and the real-world datasets combined.

Contributions. This paper makes the following contributions:

1. **Technique.** We propose *hybrid pruning*, a new hybrid program analysis technique that combines dynamic information with the vanilla static analysis to develop precise pointer and taint analyses.
2. **Applicability.** To demonstrate the effectiveness of our hybrid technique, we have further developed a vulnerability detection system as a client of our improved pointer and taint analyses. It exhibits significantly lower false positive rate as compared to its static counterpart.
3. **Evaluation.** We implement our approach in a practical prototype, and show its efficacy in an experimental evaluation on two different datasets, *i.e.*, `CGC` [13], and a collection of popular real-world programs.

2 Background

In this section, we equip the reader with the background information required to understand our approach.

2.1 Flow-sensitive, static points-to analysis

We provide a brief overview of an Andersen-style, flow-sensitive, static points-to (SPT) analysis technique, which we will use later on to demonstrate our hybrid approach. The goal of any static points-to analysis is to determine the set of objects that a given pointer can point to, at any point in the program. Specifically, a points-to analysis answers a membership query *IsPtsTo(p, x)*, which indicates whether a memory object x is in the points-to set of the pointer

p. A flow-sensitive points-to analysis computes the points-to set of the pointers according to the control-flow of the program. Given a program, the analysis starts by generating constraints for every pointer according to their usage in the program. The solution to the generated constraints gives the points-to results for all the pointers. A points-to analysis either categorizes, or transforms any program statement into one or more of the statements in Figure 1.

$$\frac{\texttt{p = \&x}}{l_x \in PtsTo(p)}\ \textsc{Address-Of} \qquad \frac{\texttt{p = q}}{PtsTo(p) \supseteq PtsTo(q)}\ \textsc{Copy} \qquad \frac{\texttt{p = *q}}{PtsTo(p) \supseteq PtsTo(*q)}\ \textsc{Dereference}$$

$$\frac{\texttt{*p = q}}{PtsTo(*p) \supseteq PtsTo(q)}\ \textsc{Assign}$$

Fig. 1: The premise (highlighted) of an inference rule represents the type of the statement encountered in a program, and the conclusion corresponds to the constraints in SPT.

$$\frac{PtsTo(p) \supseteq PtsTo(q) \quad l_x \in PtsTo(q)}{l_x \in PtsTo(p)}\ \textsc{Copy} \qquad \frac{PtsTo(p) \supseteq PtsTo(*q) \quad l_r \in PtsTo(q) \quad l_x \in PtsTo(r)}{l_x \in PtsTo(p)}\ \textsc{Dereference}$$

$$\frac{PtsTo(*p) \supseteq PtsTo(q) \quad l_r \in PtsTo(p) \quad l_x \in PtsTo(q)}{l_x \in PtsTo(r)}\ \textsc{Assign}$$

Fig. 2: Rules to solve the SPT constraint graph.

Constraint generation. The analysis iterates over the statements in a program, and collects the constraints according to the rules in Figure 1, where l_x and *PtsTo(p)* denote the location of the variable x, and the points-to set of the pointer p respectively. The constraints are usually managed by creating a *constraint graph*, where the nodes represent pointers or memory objects, and edges represent the constraints.

Constraint solving. Once the constraints are generated, each of the constraints will be solved until a fixed point is reached, *i.e.*, no changes occur to the points-to set of all the pointers. The rules in Figure 2 are used to solve the generated constraints.

2.2 Static taint tracking

Static Taint Tracking (STT) [36] is a data-flow tracking technique used to track the flow of the tainted data within a program. STT is most commonly used in vulnerability detection, where the program input is tainted, and vulnerabilities are modeled as an usage of unsanitized data in the sensitive operations. For example, a usage of tainted data in an arithmetic operation can cause an integer overflow or an underflow bug. Similarly, an out-of-bounds access bug can occur when tainted data is used as the index in an array. STT consists of the following components:

Taint source. Functions that read an input from the user, or the environment, *e.g.*, `read`, `scanf` are considered as taint sources. The variables into which the data is read are labeled as *tainted*.

Taint propagation. Typically, the result (destination) of an operation is labeled *tainted* if any one of its operands (source) is already tainted, *e.g.*, for a binary operation $r \leftarrow f(a, b)$, taint propagates to r if either a or b is tainted.

An STT requires points-to information to track the flow of tainted data through pointers. To inject taint, the STT must know to which objects the source pointer can point, so that it can taint all those objects. Note that an imprecise pointer analysis could result in over-tainting, resulting in many data elements being incorrectly considered as tainted [41]. In this paper, we use the taint propagation rules similar to the ones proposed in Dr. Checker [29].

3 Motivation

3.1 Running example

We use the code in Listing 1.1 to explain various aspects of our technique. To generate execution traces, we exercise the program with a test suite. However, the part of the code highlighted in *red* is ***not*** executed in any of the dynamic runs.

Points-to. The `process_buf()` function returns either of its `char` pointer arguments (`res` at Line 10, or `req` at Line 13) depending on the value of `r` (Lines 8 and 11). `c_buff` gets assigned the pointer returned by the `process_buf()` call once at Line 28 (`res_buff`), then again at Line 35 (`greq`), and lastly at Line 47 (`q`).

Taint. At Line 32, the program reads user data into the buffer pointed to by `c_buff`, which, in turn, points to `res_buff`.

Bugs. There are five array indexing operations (Lines $37-41$, 50). However, the operation at the Line 39 could lead to an out-of-bounds write of `buff`, because `res_buff` gets *tainted* at Line 32 (via `c_buff`). In turn, the index `res_buff[0]` can contain a value greater than the size of `buff` (*i.e.*, 16). Likewise, the `write` at Line 50 can lead to an out-of-bounds write of the buffer pointed by `c_buff` (*i.e.*, `q` from Line 47). The remaining three indexing operations are safe.

3.2 Imprecision in vanilla static analysis

Consider an STT technique based on the SPT analysis that we presented in Section 2 on our example in Listing 1.1. The call to `read_user_data` taints object ids $\{3, 1, 4, 2\}$ because of the points-to set of `c_buff@28`. At Lines 37, 39, 40, 41, and 50, we are using data from the tainted objects (*i.e.*, $\{3, 1, 4, 2\}$) as indices to write to arrays. Consequently, any static vulnerability detection technique that relies only on the STT information, and checks for the use of tainted data as an array index (unsafe operation) will raise a potential out-of-bounds alert. However, as described in Section 3.1, only the buffer pointed to by `res_buff`

contains tainted data. Therefore, only the warnings raised at Lines 39 and 50 are true positives. Next, we show how we use dynamic information to improve the precision of static analysis techniques to eliminate these false positives.

```
#define BSIZE 512
// global object, ID: 1
char greq[BSIZE];
// global object, ID: 2
char gres[BSIZE];
char *process_buf(IOLevel r, char *res, char *req) {
  switch(r) {
    case IORECV:
      ...
      return res;
    case IOSEND:
      ...
      return req;
  }
  return NULL;
}

int main(...) {
  // stack object, ID: 3
  char req_buff[BSIZE];
  // stack object, ID: 4
  char res_buff[BSIZE];
  // stack object, ID: 5
  char buff[16];
  char *c_buff, *t_buff;
  ...
  // The return value will be res_buff
  c_buff = process_buf(IORECV, res_buff, req_buff);
  ...
  // Read user (tainted) data into the buffer
  // pointed to by c_buff, i.e., res_buff
  read_user_data(c_buff, BSIZE);
  ...
  // The return value will be greq
  c_buff = process_buf(IOSEND, gres, greq);
  ...
  buff[req_buff[0]] = 'R';
  // BUG: Potential out of bounds write
  buff[res_buff[0]] = 'S';
  buff[greq[0]] = 'r';
  buff[gres[0]] = 's';
  ...
  if (...) {
    // heap object, ID: 6
    char *q = getenv(...);
    t_buff = c_buff;    // c_buff points to greq here
    c_buff = q;
    ...
    // BUG: Potential out of bounds write
    c_buff[res_buff[0]] = 'I';
    ...
  }
  ...
  return 0;
}
```

Listing 1.1: Example program to demonstrate the effectiveness of *hybrid pruning*. The region highlighted in red is ***never*** executed in any of the dynamic runs.

Objects ID	Objects Name	Tainted data? (Ground Truth)	Static Taint Tracking (STT) Flow-Sens PT	H_P-PT	H_o-PT
1	`greq`	✗	✓	✗	✗
2	`gres`	✗	✓	✗	✗
3	`req_buff`	✗	✓	✗	✗
4	`res_buff`	✓	✓	✓	✓
5	`buff`	✗	✗	✗	✗
6	`q`	✗	✗	✗	✗

Table 1: Tainted objects (✓: Tainted, ✗: not Tainted) when different points-to analysis techniques are used. The colors **green** and **red** represent true positives, and false positives respectively.

Vulnerability Warnings (Ground Truth)	Static Taint Tracking (STT) Flow-Sens PT	H_P-PT	H_o-PT
Out-of-bounds write on Line 39	1	1	1
Out-of-bounds write on Line 50	1	0	1
False positives	3	0	0
Total warnings	5	1	2

Table 2: Vulnerability warnings of static taint tracking when different points-to analysis techniques are used. The color **green** represents true positives, and **red** represents false positives and false negatives respectively.

Pointer	Dynamic points-to
`q`	N/A
`buff`	{5}
`req_buff`	{3}
`greq`	{1}
`res_buff`	{4}
`gres`	{2}
`req`	{1,3}
`res`	{2,4}
`return`	{3,1,4,2}
`c_buff@28`	{4}
`c_buff@35`	{2}
`c_buff@47`	N/A

Table 3: Dynamic points-to information collected for the example in Listing 1.1. N/A indicates that the code corresponding to the pointer is not executed in any of the dynamic runs.

3.3 Precision gain due to *hybrid pruning*

First, we exercise the program either using tests, or by fuzzing, to collect dynamic points-to and taint facts. Then, we augment the static pointer and taint analysis techniques with the recorded dynamic facts in either of the following two ways – *propagation-only* (H_P), or *opportunistic* (H_o).

Propagation-only (H_P). In this mode, the static analysis is first initialized with the recorded dynamic facts. Then, the static pointer and taint analysis algorithms propagate those dynamic facts even to those program points that are not executed dynamically. In other words, the information generated at any program point is derived ***only*** from the dynamic information, but propagated by the static analysis rules. The benefit of the H_P over dynamic-only analysis is that the former compensates for the lack of dynamic information by static propagation of dynamic facts, at the program points where the dynamic information is absent. Greater the dynamic coverage is, more effective the H_P mode will be in eliminating the spurious points-to and taint sets. The H_P *hybrid pruning* strategy, when applied to static points-to (SPT) and static taint-tracking (STT) analyses, yields H_P-PT (propagation-only points-to) and H_P-TT (propagation-only taint-tracking) analyses, respectively.

In Listing 1.1, H_P-PT prunes the over-approximated SPT set of `c_buff@28` from $\{3, 1, 4, 2\}$ to $\{4\}$. Consequently, an STT that relies on H_P-PT correctly taints only the object with id 4 (`res_buff`), thus improving the precision of the taint analysis, as shown in Table 1 (Column H_P-PT). Furthermore, as shown in Table 2 (Column H_P-PT), a static vulnerability detection technique that uses this hybrid taint-tracking emits no false warnings. However, for cases where dynamic information is inadequate, *e.g.*, the points-to information of `c_buff@47` is

absent, the H_P mode might fail to compute certain information. The missing information might introduce false negatives, as shown in Table 2 (Column H_P-PT), where using H_P-PT resulted in missing the vulnerability in Line 50 of Listing 1.1.

► *Difference between H_P and classic dynamic analysis.* Since H_P mode propagates dynamic facts using static algorithms, it essentially compensates for the 'lost' information at certain program points. In Listing 1.1, a purely dynamic approach would compute an empty points-to set for `t_buff@46`, because Line 46 was never executed in any of the dynamic runs. However, Line 35 was dynamically executed, which made `c_buff@35` point to `greq`. That information will be propagated in H_P mode, resulting in `t_buff@46` correctly pointing to the `greq` buffer.

Opportunistic (H_o). As explained above, the points-to and taint information that the H_P mode propagates might be incomplete due to lack of dynamic coverage at certain program points – resulting in false negatives. To alleviate this issue, we use the dynamic information in the H_o mode only at those program points that are executed dynamically. For all other program points, we use the static information. Opportunistic use of the dynamic facts conservatively preserves the static points-to and taint sets at those program points where the dynamic information is not available. The only difference between the H_o and the H_P modes is that the H_o allows static information to be generated, while the H_P does not. The H_o *hybrid pruning* strategy, when applied to static points-to (SPT) and static taint-tracking (STT) analyses, yields H_o-PT (opportunistic points-to) and H_o-TT (opportunistic taint-tracking) analyses, respectively.

In Listing 1.1, though the code highlighted in red is not dynamically executed, H_o-PT infers the points-to relation between `c_buff@47` and object with id 6. Consequently, an STT that relies on H_o-PT correctly taints the relevant buffer, as shown in Table 1 (Column H_o-PT). Furthermore, as shown in Table 2 (Column H_o-PT), a static vulnerability detection technique that uses this hybrid taint-tracking emits no false warnings, yet discovers both the vulnerabilities.

4 Hybrid Pruning

Our technique works in three steps. First, we generate the dynamic facts (Section 4.1), *e.g.*, points-to and taint sets, by exercising the program with a test suite, or using fuzzing. In the next phase, which we call *domain re-mapping* (Section 4.2), we lift the dynamic facts to the same domain as that of the static ones, so that a unified analysis becomes possible. Finally, we run the static analysis, and inject (Section 4.3) the dynamic facts, wherever available, thus eliminating potentially spurious points-to and taint sets at those program points. Note that the precision improvement is **not only** local to the point of injection, but also carried forward to the downstream analysis sites by the static algorithms. For example, "fixing" an over-approximated points-to set progressively taints fewer objects further down the analysis. Finally, we run a number of vulnerability detectors (Section 4.4), which uses the hybrid facts to eliminate spurious warnings.

4.1 Generation of dynamic facts

During a program's execution, we record **(i)** the allocation and deallocation of program objects, **(ii)** the read and write accesses on those objects, **(iii)** the callsite-based program context of the instructions involved in (i) and (ii), and **(iv)** the arguments of the input API, *e.g.*, `read`. Once this information is collected, we compute the dynamic points-to and taint information corresponding to those memory objects from the recorded trace. Next, we describe how we recover the dynamic facts from the collected trace in detail.

Dynamic context. We keep track of a function's call-stack c at run-time, by emulating a parallel stack updated at every `call` and `ret` instruction. For every instruction I, we compute its dynamic context $\Delta(I) = (c, \tau)$, where c is the call-stack with which I is executed, and τ is an unique identifier for each I.

Memory objects. We maintain the tuple $(sz, rt, \Delta(I))$ for each memory object o allocated, or deallocated by an instruction I, where sz is the size of the object (in bytes), rt is its run-time address, and $\Delta(I)$ being its dynamic context. We extract the size sz of the local and global memory objects from their types. The size of the heap object is extracted from the *size* argument passed to the allocation routines, *e.g.*, `malloc`. Note that, different instances of an object o with the same context $\Delta(I)$ might get created at different points in time in an execution, or even across different executions. We merge the dynamic facts associated with all those instances of an object o, by its context $\Delta(I)$, at the end of trace collection. For each object o created by the same instruction I with the same context $\Delta(I)$, we compute its id $\pi(o) = \{hash(\Delta(I), \tau(I))\}$, which uniquely identifies the object for a given context.

Points-to facts. To compute the points-to sets, we track all the write operations to the program objects. Assume, a memory `write` instruction writes to the address w_d of a memory object $o_d = (sz_d, rt_d, _)$. If the value being written to is a memory address w_s of an object $o_s = (sz_s, rt_s, _)$, then we make the offset $(w_d - rt_d)$ of the object o_d point to the offset $(w_s - rt_s)$ of the object o_s. Formally, the updated points-to set $\rho(o_d, w_d - rt_d) = \rho(o_d, w_d - rt_d) \cup (\pi(o_s), w_s - rt_s)$.

Taint facts. We use the same taint sources as that of static taint analysis. However, different from the static case, dynamically we taint the exact number of bytes read by an input API, *e.g.*, a `read(fd, buf, count)` call taints `count` bytes of the buffer `buf`.

4.2 Domain re-mapping

Hybrid pruning seeds static analysis algorithms with the dynamic information. Static analysis operates on an intermediate representation (`IR`), and models program memory in terms of abstract objects. However, dynamic analysis executes native CPU instructions, and objects are created at run-time on the program stack, or heap. We use the following two-fold approach to bridge this gap. First, we assign a unique instruction id τ to each `IR` instruction. Additionally, to represent a memory object, we use a unique object id π as discussed earlier. We include both the τ and π in the dynamic events, and the generated dynamic

facts. During the static analysis, we re-use the same τ as that of the dynamic analysis, and use identical definition of static context as that of dynamic context $\Delta(I)$. Hence, the static and dynamic object id π evaluates to be the same, for the same object, created in the same context. The *hybrid pruning* leverages this fact to establish the equivalence between a dynamic memory object, and its static counterpart.

4.3 Injection of dynamic facts

We augment both the static pointer and taint analyses with the dynamic information to achieve *hybrid pruning*. For the pointer analysis, we leverage the flow-sensitive analysis from `SVF` [45]. Our static taint analysis engine is flow-, context-, and field-sensitive. In addition to the family of input `API`s, *e.g.*, `scanf`, `gets`, *etc.*, we consider the command-line arguments of the program as the taint sources. The taint analysis is parameterized by the underlying pointer analysis, *i.e.*, while propagating the taint labels, it queries the pointer analysis for the points-to sets of the destination operand of an instruction. Taint sinks are determined by the taint policies of the respective vulnerability detectors. Depending on how we inject dynamic facts during static analysis, we develop two modes of *hybrid pruning* – *propagation-only* and *opportunistic*.

$$\frac{PtsTo(p) \supseteq PtsTo(q) \quad dynV(q) \quad l_x \in DynPtsTo(q)}{l_x \in PtsTo(p)}\ \textsc{DynCopy} \qquad \frac{PtsTo(p) \supseteq PtsTo(q) \quad \neg dynV(q) \quad l_x \in PtsTo(q)}{l_x \in PtsTo(p)}\ \textsc{ICopy}$$

$$\frac{PtsTo(p) \supseteq PtsTo(*q) \quad l_r \in PtsTo(q) \quad dynV(r) \quad l_x \in DynPtsTo(r)}{l_x \in PtsTo(p)}\ \textsc{DynDereference}$$

$$\frac{PtsTo(p) \supseteq PtsTo(*q) \quad l_r \in PtsTo(q) \quad \neg dynV(r) \quad l_x \in PtsTo(r)}{l_x \in PtsTo(p)}\ \textsc{IDereference}$$

Fig. 3: Rules to solve the hybrid constraint graph.

Propagation-only (`HP`). In this case, we propagate just the dynamic facts using static analysis rules. For the SPT we presented in Section 2, we can achieve `HP` *hybrid pruning* by **(i)** ***not*** generating any ADDRESS-OF constraints, and **(ii)** modifying the COPY and DEREFERENCE constraints to consider only the dynamic information. While **(i)** prevents generation of any new static fact, **(ii)** ensures propagation of dynamic facts following the SPT rules. We split the constraint-solving rules in Figure 2 depending on the availability of the dynamic information. Specifically, we follow the DYNCOPY and DYNDEREFERENCE rules as shown in Figure 3, to process the COPY and DEREFERENCE instructions in the `HP` mode. The $dynV(p)$ predicate checks whether the program point corresponding to the pointer p has been dynamically executed. If so, we consider the dynamic points-to set returned by the $DynPtsTo(p)$ predicate. In the `HP` mode of STT, we ignore all the static taint sources. We use just the dynamic taint information for all the dynamically executed instructions. In effect, we consider only those instructions that have been both dynamically executed, and found to be tainted. Due to the space constraint, we refrain from presenting the modified transfer functions for the taint propagation.

Opportunistic ($\mathtt{H_o}$). In this case, we generate new static facts, if dynamic information is unavailable. If the later is available at a program point, it is given priority over its static counterpart. For the SPT we presented in Section 2, we can achieve the $\mathtt{H_o}$ *hybrid pruning* by **(i)** generating the ADDRESS-OF constraints, and **(ii)** modifying the COPY and DEREFERENCE constraints to give preference to dynamic information, if available. Otherwise, the constraint solving rules are made to use static information. Policy **(i)** ensures the generation of new static facts, which compensates for the lack of dynamic coverage. In fact, if the dynamic information is available, we use the same constraint-solving rules as in the case of the $\mathtt{H_P}$ mode, while processing the COPY and DEREFERENCE instructions. However, we also introduce two new rules, *viz.*, ICOPY and IDEREFERENCE as shown in Figure 3, to deal with those cases when dynamic information is absent. The $\mathtt{H_o}$ strategy falls back to SPT in that case. In the $\mathtt{H_o}$ mode of STT, we enable the static taint sources. Also, we propagate the static taint except when the dynamic information is available at an instruction, it is given priority. In other words, the taint engine never taints an instruction that has been dynamically executed, yet was never tainted.

4.4 Vulnerability detection

The vulnerability detectors use the taint information to detect potentially buggy program points. Since, the taint analysis itself is a client of the pointer analysis, the checkers run when both the pointer and taint analyses are over. In our research prototype, we only use detectors capable of finding spatial vulnerabilities, *e.g.*, buffer overflow, out of bounds, *etc.* Temporal bugs, *e.g.*, use after free, double free, *etc.*, are considered out of scope. Specifically, we use the taint-based bug detectors, *i.e.*, *Improper Tainted-Data Use Detector* (`ITDUD`), and *Tainted Loop Bound Detector* (`TLBD`) from the DR. CHECKER [29] project. `ITDUD` monitors whether tainted data is used in risky functions *e.g.*, `strcpy`, `memcpy`, *etc.* Where as, `TLBD` checks if the loop bound can possibly be tainted.

4.5 Implementation

To generate the dynamic facts, we instrument the program using `LLVM` 7.0 [5]. The static and hybrid analysis engines are based on `SVF` 7.0 [45]. We extended `SVF` to add support for taint analysis, while using its pointer analysis (`fspta`) out-of-the-box. We use the `DataFlowSanitizer` [6] (`DFSan`), a generic dynamic data flow analysis `LLVM` pass, which instruments the program to perform dynamic taint tracking. The `DFSan` also handles memory taint by maintaining a *shadow* memory [37]. Our analysis injects and propagates taint automatically, and collects all the tainted instructions and memory objects. The vulnerability checkers are adapted from the DR. CHECKER [29].

5 Evaluation

We evaluate the effectiveness of our approach in a downstream security application, *e.g.*, vulnerability detection. We show that using our hybrid points-to and taint analysis we generate fewer false-positives (warnings that are *not* real bugs), while still detecting ***real*** bugs.

5.1 Evaluation setup

Dataset Our approach was evaluated on the following two datasets.

CGC. The corpus of 246 programs [13] used by DARPA in the Cyber Grand Challenge (CGC) [4]. We chose to use `cb-multios` [33], a port of the CGC challenge set to Linux x86 by Trail of Bits. `cb-multios` project failed to port five programs to Linux. Moreover, the programs are meant to be compiled in 32-bit, while our DFSan based implementation generates only 64-bit binaries owing to the limitation imposed by the shadow memory mechanism. Due to unsupported architecture, 89 programs aborted with an early memory corruption inside the custom heap allocator. From the remaining ones, we randomly sampled 12 programs containing spatial vulnerabilities to include in our dataset.

Real-world. Though CGC programs mimic real-world applications both in terms of complexity and functionality, we collected 8 vulnerable versions (Table 4) of 4 distinct real-world GNU applications containing only *spatial* vulnerabilities from the CVE database [2]. We used the test suites available with the respective versions of those utilities to exercise those programs.

Instrumentation. This step was carried out on an Ubuntu 18.04.3 LTS, 64-bit system equipped with an Intel Core i7-4770 (3.40GHz) CPU, and 32GB of memory, under moderate workload.

Trace collection. We re-used the same setup from the previous phase. The programs were exercised by their respective test suite. Real-world applications were let to run until they gracefully exit. However, many CGC programs being interactive, and menu-driven in nature, they run in a waiting loop until a specific program option is chosen, *e.g.*, sending a QUIT command. It is not guaranteed that the test cases will drive the programs to completion. To ensure the convergence of the experiment, we imposed a hard time-limit of 15 seconds per program execution by sending a SIGTERM signal, and installed signal handlers to record traces at termination.

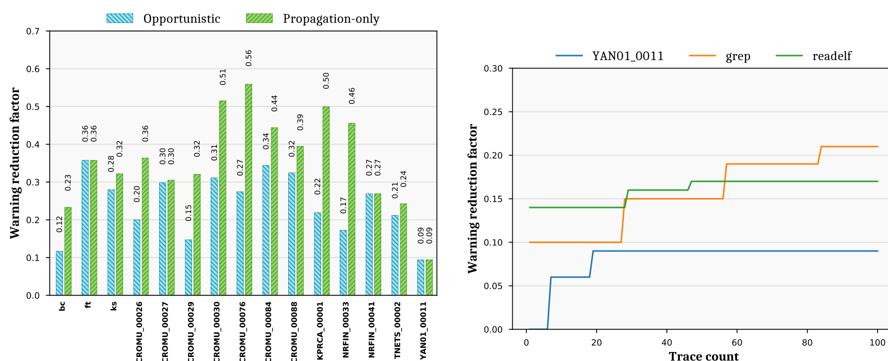
Hybrid analysis. We deployed this analysis to a `Celery` [1] cluster consisting of 8 servers with an analysis time-limit of 6 hours per program, per configuration. Each server was equipped with an Intel Xeon E5645 2.40GHz CPU, and 96GB of memory, running Ubuntu 16.04.6 LTS, 64-bit. Despite the time-limit in place, ***none*** of the analyses was observed to hit the limit.

5.2 Vulnerability detection

We measured the effectiveness of our pruning strategy in terms of the reduction of warnings due to the following two reasons—**(i)** We were interested to understand

if our technique is able to significantly bring down the number of warnings emitted by a static bug detector such that those alarms can be verified by the analysts manually. **(ii)** We had the partial knowledge of the vulnerabilities (ground truth) present in our dataset. In other words, we did not have the knowledge of all the bugs contained in our subjects. Both the sources of building the ground truth—the bugs documented with the `CGC` dataset, and the CVE database records for the vulnerable real-world programs—were incomplete. Therefore, we could only confidently determine the true positives for bugs by associating the warnings to our known bugs. However, a similar strategy would incorrectly flag a warning, which is indeed a bug, as a false positive just because the associated bug report is not present in our (incomplete) ground truth. Establishing a complete ground truth would not only require the involvement of human experts, but also would be hard to scale to all the programs included in our dataset.

Warning Reduction Factor (`WRF`). To measure the effectiveness of *hybrid pruning*, we introduce the notion of *warning reduction factor* (`WRF`), a metric that captures the effect of *hybrid pruning* on emitted warnings, *w.r.t.* the baseline static vulnerability detection technique. An `WRF` $= 0\%$, the worst-case scenario for our technique, corresponds to no improvement due to *hybrid pruning* over the static analysis. A non-zero `WRF` quantifies the improvement in performance induced by *hybrid pruning*. We define `WRF` as the fraction of warnings that are **not** raised by our hybrid ($\mathtt{H_o}/\mathtt{H_P}$) analysis. Formally, $\mathtt{WRF} = (|\omega_{\mathcal{B}}| - |\omega_{\mathcal{H}}|)/|\omega_{\mathcal{B}}|$, where $|\omega_{\mathcal{B}}|$ and $|\omega_{\mathcal{H}}|$ denote the number of warnings reported by the baseline and the hybrid analyses respectively.



(a) Warning reduction by *hybrid pruning* compared to baseline analysis (b) Warning reduction with dynamic trace

Fig. 4: Analysis of warning reductions

Results and analysis. In this experiment, we ran the bug detectors (Section 4.4) in three different configurations: **(A) static-*only*:** flow-sensitive static points-to + static taint **(B)** $\mathtt{H_o}$**-*only*:** flow-sensitive $\mathtt{H_o}$ points-to + $\mathtt{H_o}$ taint, and **(C)** $\mathtt{H_P}$**-*only*:** flow-sensitive $\mathtt{H_P}$ points-to + $\mathtt{H_P}$ taint. To demonstrate the reduction in the warnings, we evaluated our approach on the `CGC` [13] dataset. The

Subject	Warnings				
	Static	H_o	Bug Found?	H_P	Bug Found?
CROMU_00026	249	199	✓	158	✗
CROMU_00027	141	99	✓	98	✓
CROMU_00029	261	223	✓	177	✗
CROMU_00030	305	210	✓	148	✓
CROMU_00076	321	233	✓	141	✗
CROMU_00084	700	459	✓	389	✓
CROMU_00088	528	357	✓	320	✓
KPRCA_00001	209	163	✓	105	✓
NRFIN_00033	93	77	✓	51	✓
NRFIN_00041	268	196	✓	196	✓
TNETS_00002	33	26	✓	25	✓
YAN01_00011	32	29	✓	29	✓
readelf-2.28 (CVE-2017-6969)	2255	1872	✓	999	✗
readelf-2.28 (CVE-2017-8398)	2255	1872	✓	999	✓
readelf-2.30 (CVE-2018-10372)	3038	2582	✓	1231	✓
readelf-2.32 (CVE-2019-14444)	3061	2663	✓	1176	✓
readelf-c0e331c (CVE-2017-15996)	2369	2037	✗	996	✗
date-15fca2a (CVE-2014-9471)	581	238	✓	192	✓
locate-4.2.30 (CVE-2007-2452)	1038	571	✓	343	✓
grep-235aad7 (CVE-2012-5667)	539	426	✓	270	✓

Table 4: Warnings emitted, and corresponding bugs (true positives) discovered by bug finders based on pure static, H_o, and H_P modes of points-to and taint analyses. ✓ and ✗ denote if the bug has been found or missed by an analysis.

static-*only* configuration, which emits the *most* number of warnings, serves as the baseline for this experiment. Figure 4a shows the reduction in the number of warnings when H_o-*only* and H_P-*only* analyses are used. Further we observe that the WRF increases as the size (lines of code), and the complexity (*e.g.*, pointer-heavy programs), the number of taint sources, or the dynamic coverage increases. Intuitively, the first three factors make the analysis harder for a static bug detector, thus generating larger number of spurious warnings. The fourth factor, *i.e.*, the dynamic coverage, indeed benefits the hybrid analysis, as we show in Section 5.3. The H_o-*only* configuration reduces the warnings up to 36% (WRF=0.36), while the reduction in the H_P-*only* configuration is higher, up to 56% (WRF=0.56). We argue that this is no worse than any dynamic-*only* analysis system (*e.g.*, fuzzing) which suffers from insufficient coverage. H_P-*only* mode is helpful only when we have high confidence in the completeness of the dynamic information, *e.g.*, the test suite is exhaustive, providing good coverage. If the dynamic coverage is lacking, H_o-*only* mode is preferred.

This study reinforces the trade-off [51] between *usability* and *soundness*. We envision our bug detection system to be used in practice in either of these two modes: **(a) Conservative:** When an analyst chooses to minimize the likelihood of missing bugs, but is ready to tolerate a reduced reduction in the warnings; H_o-*only* mode is helpful. **(b) Priority:** When an analyst prioritizes finding the *most*

number of bugs in a small time window, thus requiring a significant reduction in spurious warnings; $\texttt{H}_\texttt{P}$-*only* mode is a perfect fit.

While cutting down the number of warnings is desirable, it is not sufficient because of the potential risk of missing the true bugs. To evaluate the impact of *hybrid pruning* on the bug detection capability of the static bug detectors, we ran the same on both the CGC [13] and the real-world datasets. Table 4 summarizes the bugs discovered by the bug detectors while running in the $\texttt{H}_\texttt{P}$-*only* and $\texttt{H}_\texttt{o}$-*only* configurations. While the former is found to miss five bugs, the later misses just one bug. Intuitively, though insufficient dynamic coverage exhibits greater warning reduction in the $\texttt{H}_\texttt{P}$-mode, it misses more bugs than the $\texttt{H}_\texttt{o}$-mode, which compensates for the lack of dynamic coverage, by design. Please note that, even $\texttt{H}_\texttt{o}$-mode can also miss true bugs in some cases. We discuss that in Section 6.

Hence, we show that *hybrid pruning* enable scalable and efficient bug triaging by cutting down on false alarms while retaining comparable true-positive rate.

5.3 Effect of dynamic trace

An important aspect to consider is how the quantity of dynamic information available affects the overall performance of *hybrid pruning*. We conducted this experiment on three subjects, *i.e.*, YAN01_00011, grep and readelf in $\texttt{H}_\texttt{o}$mode; where we gradually inject more dynamic traces into our analysis system. We use fuzzing as an inexpensive way of trace generation, and randomly pick 100 traces. Every time a new trace is introduced, we continuously monitor the performance of our analysis system in terms of warning (WRF) reductions. With more traces being made available, pointer analysis improves, as additional dynamic information yields new points-to sets not discovered before. Moreover, taint analysis improves due to the combined improvement in the points-to sets, as well as the reduction in the spurious static taint sets. Since the bug detectors consume both the pointer and the taint information, the number of warnings reduces over time. Initially, the WRF increases, and then becomes stable at the point when the dynamic coverage saturates. We observe that the performance of *hybrid pruning* is positively correlated with the amount and the quality (coverage) of the dynamic trace. We present in Figure 4b. Specifically, for every subject, the corresponding line gradient in Figure 4b represents the correlation of WRF with the trace count, *e.g.*, gradient increases mean WRF increases as we add more traces.

Gradient increases. Points-to result improves when additional dynamic information yields new points-to sets that has dynamically never been seen before by the analysis. Also, taint can improve either due to more precise points-to sets, or additional dynamic taint information overriding its static counterpart at newer program points. Warning improves as it is positively correlated to the improvement of either or both the factors.

Gradient unchanged. Neither points-to, nor taint improves. Typically, it is the case when multiple traces exercise the same program path.

Gradient decreases. Increased dynamic information can discover more target objects pointed to by the same pointer; thereby increasing the size of its points-

to set. Similarly, extensive dynamic information available at the same program point can newly *taint* an object which was found not to be *tainted* in prior runs.

6 Limitations and Discussion

Potential false negatives. Our pruning strategy is context-insensitive, meaning that the different call contexts of the same callee method are indistinguishable from each other.

```
1  void square(int* p) {
2      *p = (*p) * (*p);    // Unsafe binary operation
3  }
4
5  int main() {
6      int n, c = 50, i;
7      scanf("%d", &i);
8      if (i < 100) {
9          scanf("%d", &n);     // Tainted input
10         square(&n);
11     } else
12         square(&c);
13     return 0;
14 }
```

Listing 1.2: False negative of *hybrid pruning*. Instructions in **green** are dynamically executed while the **red** ones are not.

In Listing 1.2, `square` is being called from two different contexts at Line 10 and Line 12, making p point to $\{n, c\}$. The tainted input n can flow to the multiplication operation at Line 2, if and only if i is less than 100. However, assume that the program is exercised ***only*** with test cases having i greater than 100. Therefore, in all the dynamic runs, the constant c is passed to the `square` call at Line 12, which establishes the dynamic points-to relation $p \to c$. During *hybrid pruning*, when the algorithm evaluates the points-to set of p due to the call at Line 10, it will find that the instructions of `square` have already been dynamically visited, albeit from a different context (Line 12). The context-insensitive pruning strategy disregards the difference in call-sites. At this point, the dynamic points-to set will be given preference, and consequently the static points-to relation $p \to n$ gets killed. Due to the missing points-to relation, the taint engine will no longer propagate the taint to the multiplication operation at Line 2. In turn, the `ITDUD` vulnerability detector will fail to detect the potentially unsafe binary operation. To summarize, the context-insensitive pruning strategy can lead to false negatives in both the pointer and taint analyses, as well the vulnerability detection. As we show in Section 5, the performance of *hybrid pruning* is positively correlated with the quantity, and the quality (coverage) of the available dynamic trace.

Theoretical limitation. To detect temporal bugs, *e.g.*, use-after-free (UAF), double free, *etc.*, a bug detector needs to have both the *reachability* (if the attacker can trigger the events), and the *timing* (if the attacker can control the sequence of events) information. Therefore, the taint information alone is not enough in order to detect this kind of bugs. However, such a bug detector could still benefit from the precise pointer information to infer if different events, *e.g.*, *use*, *free*, *etc.*, are operating on the same program objects. Hence, how the hybrid

points-to information improves the discovery of the temporal bugs could be an interesting research direction to explore.

Applicability to other analyses. Since *hybrid pruning* is inherently unsound, it is the best fit for applications where *soundness* is not a strict necessity, for example, in static vulnerability detection, limited cases of call-graph and control-flow graph construction, dynamic symbolic execution, *etc.* Indirect call resolution is a challenging problem—a purely static pointer analysis is likely to miss potential targets unless it is configured to be 'overly' conservative, in which case, it may become unusable. Hybrid pruning can indeed be effective, because it can restrict such a pointer to a smaller set of interesting targets.

7 Related Work

In this section we will discuss state-of-the-art techniques related to our work.

Pointer analysis: Pointer analysis is a fundamental program analysis technique with a very rich literature [20,42–44], and wide applications [23,27]. Steensgaard *et. al.* [44] provides a linear time algorithm based on type inference techniques for pointer analysis. Anderson inclusion-based pointer analysis is another important milestone for pointer analysis which provides good precision compared to Steensgaard *et. al.* with an acceptable performance overhead [43]. Yulei *et. al.*perform value-flow, and pointer analysis in an iterative manner to improve the precision of both [45]. Pointer analysis techniques are designed to be sound as they are mostly used in compiler optimization. However, there are other clients of pointer analysis that does not have this requirement. Vulnerability detection is one such client where less false positives [8], and more precision is required. There are few unsound pointer analysis techniques tailored for bug detection [9,11,12]. Similarly, speculative execution is one such client where the occasional lost of soundness is acceptable [24]. In order to achieve precision, one can also use dynamic analysis which is precise, but can never be sound. Marcus *et. al.*proposes a technique to compute pointer analysis results dynamically, which are called dynamic points-to results [19,32]. They also show that the static pointer analysis results are an order of magnitude imprecise than dynamic points-to results. Another work shows how the dynamic points-to results can be used for program slicing [31]. Additionally, David *et. al.*integrates pointer analysis with Dynamic symbolic execution to increase the precision of pointer analysis [46]. However, dynamic information heavily relies on the tests, and can never be sound. In this work, we explore the possibility of augmenting the static pointer analysis—which is imprecise but sound with dynamic points-to—which are precise but unsound. We then show how this can be used to increase the precision of vulnerability detection techniques.

Taint analysis: Taint tracking is a data flow tracking technique to track the effect of user data at various program points [36]. Static taint tracking [28] requires a precise pointer analysis, else it usually ends up with Taint explosion [41], tainting all program data. Consequently, almost all the static taint tracking techniques are developed for Java [28] and other strongly typed languages where the pointer analysis results are relatively precise. Dynamic taint

tracking(DTT) [25, 36] is usually performed by instrumenting program instructions [25], resulting in memory and run-time overhead. Though several techniques have been developed to improve the run-time overhead; it still suffers from the lack of dynamic coverage [21, 22, 30, 47].
Vulnerability detection. Nevertheless imprecise, the importance of static analysis in vulnerability detection is undeniable. A large body of work on the static detection of vulnerabilities in C/C++ programs has evolved over the last two decades. Engler et al.first explored this domain using various static analysis techniques [17,50,52]. Other techniques target only specific classes of vulnerabilities, such as, buffer overflows [16,18,53], memory leaks [49], integer anomalies [35,48], and format string errors [38]. However, as the complexity of software grows, these techniques either do not scale, or incur a large number of false positives.

The key motivation behind this work is to bring the best of both the worlds together,*i.e.*, the *scalability* offered by the static analysis, and the *precision* guaranteed by the dynamic analysis. We attempt to combine both in a novel way, such that, we can draw on the strengths of each. There exists previous attempts that combine static and dynamic analysis for various applications [7, 10, 14, 15, 26, 39, 40]. Tapti *et. al.*combines static analysis with dynamic data flow tracking (DFT) to increase the the precision of pointer analysis [34]. They used this precise pointer analysis to protect memory disclosure, and transient execution attacks. Other techniques have aimed to improve vulnerability detection as a downstream client, *e.g.*, using dynamic analysis to verify the results of static analysis, guiding fuzzing through static program analysis, using static analysis to localize program faults in untested code from fuzzer generated crash, *etc.* [14,26,39,40]. However, none of them combine the static and dynamic analysis in an interleaved way to improve the points-to, and taint analysis—which is further used in vulnerability detection to reduce the false warnings. To our knowledge, we are the first to explore this direction.

8 Conclusion

In this paper, we introduce *hybrid pruning* where we improve the precision of static points-to and taint analyses by combining dynamic information collected from program's run-time trace in a novel way. We propose two different modes of operation, *viz.*, $\mathtt{H_o}$ and $\mathtt{H_P}$, whose applicability is decided by the amount of dynamic information available. Our in-depth evaluation demonstrates both significant improvement in the precision of the points-to sets, and the reduction of the taint sets. When static vulnerability detection is used as a client of the improved pointer and taint analyses, the former is able to find 19 out 20 bugs in CGC and real-world software, where as cutting down 21% of the false warnings – making the analysis outcome more amenable to manual triaging.

References

1. Celery: Distributed task queue. http://www.celeryproject.org.

2. Common vulnerabilities and exposures. `https://cve.mitre.org`.
3. Coverity linux scan. `https://scan.coverity.com/projects/linux`.
4. Darpa cyber grand challenge. `https://www.darpa.mil/program/cyber-grand-challenge`.
5. The llvm compiler infrastructure. `https://llvm.org`.
6. Llvm dataflowsanitizer pass. `https://clang.llvm.org/docs/DataFlowSanitizer.html`.
7. Subarno Banerjee, David Devecsery, Peter Chen, and Satish Narayanasamy. Iodine: Fast dynamic taint tracking using rollback-free optimistic hybrid analysis. 2019.
8. Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM*, pages 66–75, 2010.
9. Sebastian Biallas, Mads Chr Olesen, Franck Cassez, and Ralf Huuck. Ptrtracker: Pragmatic pointer analysis. In *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*, pages 69–73. IEEE, 2013.
10. Pietro Borrello, Daniele Cono D'Elia, Leonardo Querzoni, and Cristiano Giuffrida. Constantine: Automatic side-channel resistance using efficient control and data flow linearization. In *CCS' 21*, 2021.
11. Marcio Buss, Daniel Brand, Vugranam Sreedhar, and Stephen A. Edwards. A novel analysis space for pointer analysis and its application for bug finding. *Sci. Comput. Program.*, pages 921–942, 2010.
12. Marcio Buss, Stephen A Edwards, Bin Yao, and Daniel Waddington. Pointer analysis for c programs through ast traversal. 2005.
13. Brian Caswell. Cyber grand challenge corpus.
14. Christoph Csallner, Yannis Smaragdakis, and Tao Xie. Dsd-crasher: A hybrid analysis tool for bug finding. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(2):8, 2008.
15. David Devecsery, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Optimistic hybrid analysis: Accelerating dynamic analysis through predicated static analysis. 2018.
16. Nurit Dor, Michael Rodeh, and Mooly Sagiv. Cssv: Towards a realistic tool for statically detecting all buffer overflows in c. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 155–167, New York, NY, USA, 2003. ACM.
17. Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, 2001.
18. Vinod Ganapathy, Somesh Jha, David Chandler, David Melski, and David Vitek. Buffer overrun detection using linear programming and static analysis. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, pages 345–354, New York, NY, USA, 2003. ACM.
19. Axel Gross. Evaluation of dynamic points-to analysis. 2004.
20. Ben Hardekopf, Ben Wiedermann, William R Cook, and Calvin Lin. A formal specification of pointer analysis approximations. *In submission to Programming Language Design and Implementation (PLDI)*, 2009.
21. Alex Ho, Michael Fetterman, Christopher Clark, Andrew Warfield, and Steven Hand. Practical taint-based protection using demand emulation. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 29–41. ACM, 2006.

22. Kangkook Jee, Vasileios P Kemerlis, Angelos D Keromytis, and Georgios Portokalidis. Shadowreplica: efficient parallelization of dynamic data flow tracking. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 235–246. ACM, 2013.
23. Vineet Kahlon. Bootstrapping: A technique for scalable flow and context-sensitive pointer alias analysis. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 249–259, 2008.
24. Kirk Kelsey, Tongxin Bai, Chen Ding, and Chengliang Zhang. Fast track: A software system for speculative program optimization. In *Code Generation and Optimization, 2009. CGO 2009. International Symposium on*, 2009.
25. Vasileios P Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D Keromytis. libdft: Practical dynamic data flow tracking for commodity systems. In *Acm Sigplan Notices*, volume 47, pages 121–132. ACM, 2012.
26. Seokmo Kim, R. Young Chul Kim, and Young B. Park. Software vulnerability detection methodology combined with static and dynamic analysis. *Wireless Personal Communications*, pages 777–793, 2016.
27. Ondrej Lhoták and Kwok-Chiang Andrew Chung. Points-to analysis with efficient strong updates. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 3–16, 2011.
28. Aravind Machiry. The need for extensible and configurable static taint tracking for c/c++, 2017. `https://machiry.github.io/blog/2017/05/31/static-taint-tracking`.
29. Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. DR. CHECKER: A soundy analysis for linux kernel drivers. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1007–1024, Vancouver, BC, 2017. USENIX Association.
30. Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. Taintpipe: Pipelined symbolic taint analysis. In *USENIX Security Symposium*, 2015.
31. Markus Mock, Darren C. Atkinson, Craig Chambers, and Susan J. Eggers. Improving program slicing with dynamic points-to data. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '02/FSE-10, pages 71–80, 2002.
32. Markus Mock, Manuvir Das, Craig Chambers, and Susan 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, pages 66–72, 2001.
33. Trail of Bits. Darpa challenge binaries on linux, osx, and windows. `https://github.com/trailofbits/cb-multios`, 2016.
34. Tapti Palit, Jarin Firose Moon, Fabian Monrose, and Michalis Polychronakis. Dynpta: Combining static and dynamic analysis for practical selective data protection. In *2021 IEEE Symposium on Security and Privacy (SP)*, 2021.
35. Dipanwita Sarkar, Muthu Jagannathan, Jay Thiagarajan, and Ramanathan Venkatapathy. Flow-insensitive static analysis for detecting integer anomalies in programs. In *Proceedings of the 25th conference on IASTED International Multi-Conference: Software Engineering*, pages 334–340. ACTA Press, 2007.
36. Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, 2010.

37. Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. USENIX ATC, 2012.
38. Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, SSYM'01, Berkeley, CA, USA, 2001. USENIX Association.
39. Bhargava Shastry, Markus Leutner, Tobias Fiebig, Kashyap Thimmaraju, Fabian Yamaguchi, Konrad Rieck, Stefan Schmid, Jean-Pierre Seifert, and Anja Feldmann. Static program analysis as a fuzzing aid.
40. Bhargava Shastry, Federico Maggi, Fabian Yamaguchi, Konrad Rieck, and Jean-Pierre Seifert. Static exploration of taint-style vulnerabilities found by fuzzing. In *11th USENIX Workshop on Offensive Technologies*. USENIX Association, 2017.
41. Asia Slowinska and Herbert Bos. Pointless tainting?: evaluating the practicality of pointer tainting. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 61–74. ACM, 2009.
42. Yannis Smaragdakis, George Balatsouras, et al. Pointer analysis. *Foundations and Trends in Programming Languages*, 2(1):1–69, 2015.
43. Manu Sridharan and Stephen J. Fink. The complexity of andersen's analysis in practice. In *Proceedings of the 16th International Symposium on Static Analysis*, 2009.
44. Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1996.
45. Yulei Sui and Jingling Xue. Svf: Interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th International Conference on Compiler Construction*, 2016.
46. David Trabish, Timotej Kapus, Noam Rinetzky, and Cristian Cadar. Past-sensitive pointer analysis for symbolic execution. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, page 197–208, 2020.
47. Guru Venkataramani, Ioannis Doudalis, Yan Solihin, and Milos Prvulovic. Flexitaint: A programmable accelerator for dynamic taint propagation. In *High Performance Computer Architecture*, 2008.
48. Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M Frans Kaashoek. Improving integer security for systems with kint. In *OSDI*, 2012.
49. Yichen Xie and Alex Aiken. Context-and path-sensitive memory leak detection. In *ACM SIGSOFT Software Engineering Notes*. ACM, 2005.
50. Yichen Xie, Andy Chou, and Dawson Engler. Archer: Using symbolic, path-sensitive analysis to detect memory access errors. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-11, 2003.
51. Yichen Xie, Mayur Naik, Brian Hackett, and Alex Aiken. Soundness and its role in bug detection systems. In *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.
52. Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. *ACM Transactions on Computer Systems (TOCS)*, 24(4):393–423, 2006.
53. Misha Zitser, Richard Lippmann, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 97–106. ACM, 2004.