Machine-Learning-Guided Typestate Analysis for Static Use-After-Free Detection

Hua Yan
School of Computer Science and Engineering
University of New South Wales, Australia
Data61, CSIRO, Australia

Yulei Sui
CAI and School of Software
University of Technology Sydney, Australia

Shiping Chen
Data61
CSIRO, Australia

Jingling Xue
School of Computer Science and Engineering
University of New South Wales, Australia

ABSTRACT

Typestate analysis relies on pointer analysis for detecting temporal memory safety errors, such as use-after-free (UAF). For large programs, scalable pointer analysis is usually imprecise in analyzing their hard "corner cases", such as infeasible paths, recursion cycles, loops, arrays, and linked lists. Due to a sound over-approximation of the points-to information, a large number of spurious aliases will be reported conservatively, causing the corresponding typestate analysis to report a large number of false alarms. Thus, the usefulness of typestate analysis for heap-intensive clients, like UAF detection, becomes rather limited, in practice.

We introduce Tac, a static UAF detector that bridges the gap between typestate and pointer analyses by machine learning. Tac learns the correlations between program features and UAF-related aliases by using a Support Vector Machine (SVM) and applies this knowledge to further disambiguate the UAF-related aliases reported imprecisely by the pointer analysis so that only the ones validated by its SVM classifier are further investigated by the typestate analysis. Despite its unsoundness, Tac represents a practical typestate analysis approach for UAF detection. We have implemented Tac in LLVM-3.8.0 and evaluated it using a set of eight open-source C/C++ programs. The results show that Tac is effective (in terms of finding 5 known CVE vulnerabilities, 1 known bug, and 8 new bugs with a low false alarm rate) and scalable (in terms of analyzing a large codebase with 2,098 KLOC in just over 4 hours).

CCS CONCEPTS

• **Security and privacy → Software and application security**; • **Theory of computation → Program analysis**; • **Computing methodologies → Machine learning**;

KEYWORDS

use-after-free; vulnerability detection; static analysis; machine learning

ACSAC 2017, December 4–8, 2017, Orlando, FL, USA

ACM ISBN 978-1-4503-5345-8/17/12…$15.00
https://doi.org/10.1145/3134600.3134620

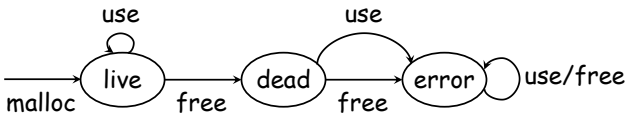

Figure 1: A finite state automation (FSA) for UAF.

ACM Reference Format:
Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. 2017. Machine-Learning-Guided Typestate Analysis for Static Use-After-Free Detection . In *Proceedings of ACSAC 2017, Orlando, FL, USA, December 4–8, 2017,* 13 pages.
https://doi.org/10.1145/3134600.3134620

1 INTRODUCTION

Use-after-free (UAF) vulnerabilities, i.e., dangling pointer dereferences (accessing objects that have already been freed) in C/C++ programs can cause data corruption [14, 70], information leaks [32, 53], denial-of-service attacks (via program crashes) [11], and control-flow hijacking attacks [9, 19, 20]. While other memory corruption bugs, such as buffer overflows, have become harder to exploit due to various mitigation techniques [14, 61, 78], UAF has recently become a significantly more important target for exploitation [32, 77].

Recent years have witnessed an increasingly large body of research on detecting or mitigating UAF vulnerabilities. Most existing approaches rely on dynamic analysis techniques by maintaining shadow memory [42, 52, 69] and performing runtime checks [9, 32, 77]. Dynamic analysis yields no or few false positives, but can incur non-negligible runtime and memory overheads, hindering their adoption in production environments. In addition, dynamic analysis often suffers from binary incompatibility issues due to code instrumentation used [61]. When used as bug detectors, dynamic approaches are often limited by test inputs used and can thus provide low code coverage and miss true bugs.

Static analysis, which approximates the program behavior at compile-time, does not suffer from the above limitations, but requires scalable yet precise pointer analysis in order to find memory errors with low false alarm rates in large programs [32]. Typestate analysis [21, 57] represents a fundamental approach for detecting statically temporal memory safety errors in C/C++ programs. For example, UAF bugs can be detected based on the finite state automaton (FSA) depicted in Figure 1. The typestates of an object o are tracked by statically analyzing all the statements (e.g., malloc

sites, free sites, and pointer dereferences at loads/stores) that affect the state transitions along all the possible program paths. A UAF warning for the object o is reported when `error` is reached. This happens when a free site $free(p)$ reaches a use site $use(q)$ (which denotes a memory access on the same object pointed by q, e.g., $*q$) along a control-flow path, where $*p$ and $*q$ are aliases, i.e., p and q point to o. In what follows, such aliases are said to be *UAF-related*. Double-free bugs are handled as a special case of UAF bugs.

ESP [17], as a representative path-sensitive typestate analysis that runs in polynomial time and space, is useful for checking properties such as "file open-close" [17] and "socket-connection" [21]. Unlike a data-flow-based path-sensitive analysis that computes execution states as its data-flow facts by finding a meet-over-all-path (MOP) solution, ESP avoids examining possibly infinite program paths by being partially path-sensitive [17, 68]. ESP uses a symbolic state as a data-flow fact, which includes an execution state and a property state of an FSA, based on the points-to information. At a control-flow joint point, ESP produces a single symbolic state, by merging the execution states whose corresponding property states are identical, thus yielding a maximal-fixed-point (MFP) solution.

Below we first discuss the challenges faced in developing a practical typestate analysis for detecting UAF bugs. We then outline the motivation behind our machine-learning-based solution.

Challenges and Insights. Unlike temporal properties such as "file open-close" and "socket-connection", UAF is much harder to handle by ESP-based typestate analysis both scalably and precisely, due to complex aliasing in the presence of a large number of free-use pairs in real-world programs. For example, `php-5.6.8` has 340 million free-use pairs with 1,391 frees and 244,917 uses.

To achieve soundness, any change to the typestate of an object must be reflected in all pointers that point to the object, i.e., all aliases of the object. In addition, the typestate transitions of an object o must be tracked efficiently and precisely from its free site $free(p)$ to all the corresponding use sites $use(q)$, where $*p$ and $*q$ are aliases (with o), along possibly many program paths spanning across possibly many functions in the program.

A typestate analysis becomes more effective if a more precise pointer analysis is used. Ideally, one may wish to combine both typestates and points-to information into the same analysis domain to form a single data-flow-based path-sensitive analysis, which will be, unfortunately, intractable due to potentially an unbounded number of paths and undecidability of aliasing [28, 48]. In order to simplify complexity, several dimensions of pointer analysis are considered to enable precision and efficiency trade-offs: *flow-sensitive* (by distinguishing the flow of control), *field-sensitive* (by distinguishing different components of an aggregate data structure), *context-sensitive* (by distinguishing calling contexts of a function), and/or *path-sensitive* (by distinguishing program paths).

In practice, these over-approximation solutions are usually imprecise, despite recent advances on sparse [23, 76, 79] and demand-driven pointer analysis [54, 56, 58], in analyzing a number of hard "corner cases" in a program, such as infeasible paths (by ignoring path sensitivity or handling it partially), recursion cycles (by merging all functions in a recursion cycle), loops (by not distinguishing different iterations of a loop), arrays (by not distinguishing array elements), and linked lists (by abstracting some of their nodes as a single one). As a result, a large number of spurious aliases will be reported, causing the corresponding typestate analysis to report a large number of spurious state transitions, i.e., false alarms. For debugging purposes, therefore, the practical usefulness of typestate analysis for UAF detection becomes limited.

Our Solution. To address the above challenges, we introduce a new UAF detection framework, Tac, to bridge the gap between typestate and pointer analyses by machine learning. Our key observation is that the spurious aliases reported by pointer analysis are alike and predictable. They share some common program features explicitly (e.g., in terms of their declaration types) or implicitly (in terms of their points-to relations). By training Tac using a Two-Class Support Vector Machine (TC-SVM), existing UAF ground truths, i.e., codebases containing labeled known false alarms and true bugs can be leveraged to enable Tac to learn the correlations between program features and the UAF-related aliases. Then its SVM classifier can be called upon to further scrutinize the UAF-related aliases reported imprecisely by the pointer analysis so that only the ones validated by the SVM classifier are further investigated by the typestate analysis. Despite its unsoundness, Tac turns out to be a practical tool for detecting UAF bugs efficiently with a low false alarm rate for large C/C++ programs.

We evaluate the effectiveness of Tac against Tac-NML (Tac without machine learning) in both its training and analysis phases. In the training phase, we exercise Tac using a large number of UAF samples, including manually identified false alarms reported by Tac-NML and true bugs (both real and injected) in a set of four C/C++ training programs. By using the standard 5-fold cross validation, Tac achieves high precision (92.6%) and recall (95.8%) while Tac-NML is imprecise (42.1%) despite a total recall (100%), measured in terms of their ability in finding the true bugs in the training samples provided.

In its analysis phase, Tac finds 109 true UAF bugs out of 266 warnings reported in a set of eight C/C++ programs including the four used in the training phase. Among the 109 bugs, there are 14 distinct ones (two UAF pairs are considered to be duplicated if they share the same free site and dereference the same pointer at the two use sites), including 5 CVE vulnerabilities, 1 known bug and 8 previously unknown ones. Compared to 19,083 warnings reported by Tac-NML, Tac reports only 266 warnings, achieving a reduction rate of 98.6%, reducing significantly the amount of manual effort needed for inspecting a vast number of false alarms.

Contributions. This paper makes the following contributions:

- We present Tac, a new machine-learning-guided typestate analysis for detecting UAF bugs statically.
- We introduce an SVM classifier specialized for UAF detection with a set of 35 features that can effectively disambiguate the UAF-related aliases reported imprecisely by pointer analysis to help typestate analysis in finding true UAF bugs at a significantly reduced false alarm rate.
- We have implemented Tac in LLVM-3.8.0 and evaluated it using eight open-source C/C++ programs (2,098 KLOC). Tac finds 109 bugs out of 266 warnings by suppressing 19,083 warnings reported by Tac-NML. Among the 109 true bugs,

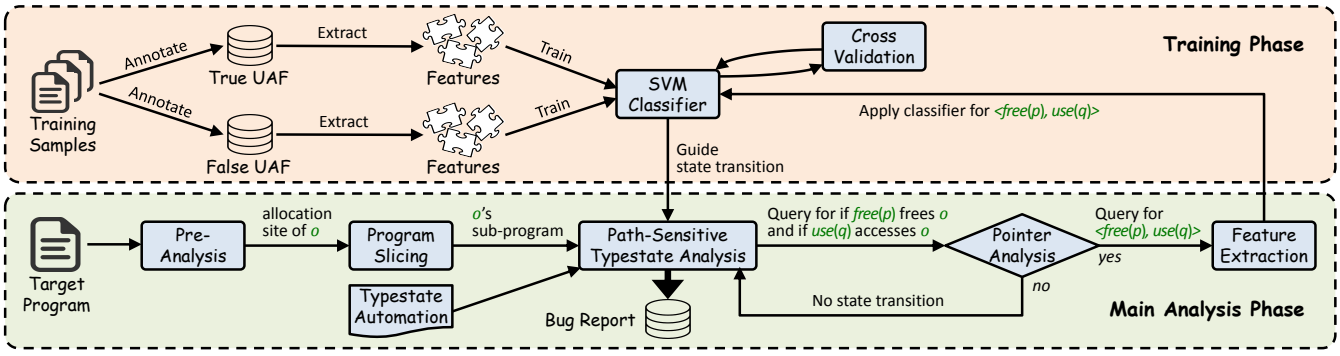

Figure 2: Tac framework.

there are 14 distinct ones, including 5 CVE vulnerabilities, 1 known bug, and 8 previously unknown bugs.

2 OVERVIEW

As shown in Figure 2, Tac has two main components. The *training phase* extracts the program features from ground truths and then uses these features to train an SVM classifier to learn harmful (benign) UAF-related aliases that cause true bugs (false alarms). A UAF pair $\langle free(p), use(q)\rangle$ is said to be *harmful* (*benign*) if $*p$ and $*q$ are regarded as aliases (non-aliases) by the SVM classifier.

The *analysis phase* filters outs many spurious UAF-related aliases reported by the pointer analysis. A pre-analysis is first performed to identify a set of candidate UAF pairs $\langle free(p), use(q)\rangle$, where $*p$ and $*q$ are found to be aliased. For every object o created at an allocation site, such that o is related to at least one candidate pair $\langle free(p), use(q)\rangle$, where $*p$ and $*q$ are aliased with o, a forward slice of the program starting from the allocation site but restricted only to the statements into which o flows (referred to as the slice of o below) is found. Then an on-demand typestate analysis is performed on the slice of o. Based on the features of $\langle free(p), use(q)\rangle$ extracted on the fly from this slice, the SVM classifier passes $\langle free(p), use(q)\rangle$, when it is harmful, to the typestate analysis for further investigation.

2.1 The Training Phase

Ground Truths. We exercise Tac using both false and true UAF samples in a set of real-world C/C++ programs as training programs. All such UAF samples are annotated for feature extraction.

Feature Extraction. We use a feature vector consisting of 35 features to describe a UAF sample. We categorize these features into the following four categories: (1) type information (e.g., global, array and struct), (2) control-flow (e.g., loop, recursion, and the distance between a free site and a use site), (3) common programming practices (e.g., pointer casting and reference counting), and (4) points-to information (e.g., the number of objects that may be used at a use site and the number of UAF pairs sharing the same free site).

Prediction Model. Our prediction model for UAF detection uses an SVM classifier. Conceptually, the SVM model used is a harmfulness predicate, which separates the input space containing all the UAF samples into two regions, marked as harmful and benign, respectively. To tune the intrinsic SVM parameters for optimal accuracy, standard grid search is applied with 5-fold cross validation by enumerating all possible combinations of the SVM parameters. Given a set of SVM parameters, 5-fold cross validation computes the accuracy of the SVM model in three steps. First, all the UAF samples are divided into 5 equal-sized subsets. Then, each subset, in turn, is used as a test set with the remaining 4 subsets combined as its training set. Finally, the averaged accuracy rate obtained is the expected accuracy of a model under the set of SVM parameters [10].

2.2 The Analysis Phase

Pre-analysis. We start conservatively with a set of candidate objects that may be unsafe (as they may induce UAF bugs). An object o (identified by its allocation site) is selected as a candidate to be further investigated by our typestate analysis if a free site $free(p)$ can reach a use site $use(q)$ via context-sensitive control-flow reachability in the program, where p and q point to o, i.e., $o \in pt(p) \cap pt(q)$. Here, $pt(v)$ denotes the points-to set of a variable v. In this case, $*p$ and $*q$ are aliased (with o). For efficiency reasons, the pre-analysis is performed in terms of Andersen's pointer analysis [5] as implemented in [59]. As is standard, context-sensitive control-flow reachability is solved as a balanced-parentheses problem by matching calls and returns to filter out unrealizable program paths on the interprocedural CFG (Control Flow Graph) of the program [50].

Slicing. For each candidate object o, the program is sliced to keep only the relevant functions that o may flow to (i.e., o's liveness scope) by using a standard mod-ref analysis, with its value-flow dependences computed by a flow-insensitive pointer analysis [23, 60]. Our typestate analysis for o will be performed on this slice.

Typestate Analysis. Our typestate analysis starts from a candidate object o created at its allocation site, with its path-sensitivity focused on the typestates of the FSA depicted in Figure 1. Following ESP [17], a data-flow fact is a symbolic state consisting of a property state, i.e., `live`, `dead` or `error`, and an execution state, which represents the values of all the variables affecting the control flow. At a two-way joint point, one symbolic state is obtained, by merging the execution states whose corresponding property states are the same. On encountering a free site $free(p)$, the FSA transits from `live` to `dead` if $o \in pt(p)$. On encountering subsequently a use/free site $use(q)/free(q)$, the FSA transits from `dead` to `error`

```
 1: void foo() {
 2:    void* p = malloc(1);//o1
 3:    void* q = p;
 4:    int flag = 0;
 5:    if (Cond) {
 6:       free(p);
 7:       p = malloc(2);//o2
 8:       flag = 1;
 9:    }
10:    if (flag == 0)
11:       use(q); // False UAF w.r.t. line 6
12:    use(q); // True UAF w.r.t line 6
13:    use(p); // False UAF w.r.t line 6
14: }
```

(a) Path-sensitivity

```
    // Data structure for linked-lists
1:  typedef struct NODE {
2:     int data;
3:     struct NODE* nxt;
4:  } NODE;

    // The 1st, 2nd, 3rd and ith nodes
5:  NODE *n1, *n2, *n3, *ni;

6:  unsigned sz = sizeof(NODE);
```

```
 7: void bar() {
 8:    ni = n1 = (NODE*)malloc(sz);//o1
 9:    for (int i = 0; i < 10; ++i) {
10:       ni->nxt = (NODE*)malloc(sz);//o2
11:       ni = ni->nxt;
12:       ni->nxt = NULL;
13:    }
14:    n2 = n1->nxt;
15:    n3 = n2->nxt;
16:    free(n2);
17:    use(n1); // False UAF w.r.t line 16
18:    use(n2); // True UAF w.r.t line 16
19:    use(n3); // False UAF w.r.t line 16
20: }
```

(b) Linked-list

Figure 3: Examples illustrating how imprecise pointer analysis leads to imprecise typestate analysis for UAF detection.

if both $o \in pt(q)$ and the aliasing relation between $*p$ and $*q$ (with respect to o) reported by the pointer analysis is also validated by our SVM classifier. In this case, a UAF/double-free warning is issued.

Given a UAF pair, $\langle free(p), use(q)\rangle$, $pt(p)$ and $pt(q)$ are computed by using a demand-driven flow-sensitive pointer analysis [58]. If $*p$ and $*q$ are found to be aliased (with o), we pass the pair to an SVM classifier for a further sanity check based on the features of $\langle free(p), use(q)\rangle$ extracted on the fly from the sliced program of o.

2.3 Examples

Figure 3 gives two examples to illustrate how the imprecision in pointer analysis leads to the imprecision in typestate analysis.

Typestate Analysis with Path-Insensitive Pointer Analysis. ESP-based typestate analysis is path-sensitive in tracking non-pointer scalar values but interprets pointer values conservatively as $\perp$, i.e., obtainable from a pointer analysis. Figure 3(a) gives an example with one true UAF bug (at line 12) and two false UAF bugs (at lines 11 and 13) with respect to the free site at line 6, by using the points-to information computed by a path-insensitive pointer analysis.

We focus on analyzing the object o_1 allocated at line 2 and freed conditionally at line 6, at which point, the property state of o_1 becomes dead. By using a flow-sensitive pointer analysis without path sensitivity, ESP can (1) prove that $use(q)$ at line 11 is not a UAF bug, (2) identify $use(q)$ at line 12 as a true UAF bug, but (3) report imprecisely $use(p)$ at line 13 as a false alarm.

Table 1 gives the symbolic states (including o_1's property states for the FSA shown in Figure 1 and execution states) and the points-to sets at some relevant program points. After analyzing line 9, typestate analysis combines the symbolic states from the two branches into one, resulting in $s_{line9} = s_1 \cup s_2$, where $s_1 = [\texttt{dead},\ \texttt{p = q} = \perp, \texttt{flag} = 1, Cond]$ (if-branch) and $s_2 = [\texttt{live},\ \texttt{p = q} = \perp, \texttt{flag} = 0, \neg Cond]$ (else-branch). However, after line 10, s_1 is filtered out due to path contradiction, since $flag = 1$ in s_1's execution state is inconsistent with the branch condition $flag == 0$ at line 10. As a transition from dead to error is impossible, the typestate analysis correctly proves the absence of a UAF bug for $use(q)$ at line 11.

Table 1: ESP-based Typestate analysis with path-insensitive pointer analysis for the program given in Figure 3(a).

symbolic typestates	line 4:	$[\texttt{live},\ \texttt{p = q} = \perp, \texttt{flag} = 0]$
	line 6:	$[\texttt{dead},\ \texttt{p = q} = \perp, \texttt{flag} = 0, Cond]$
	line 8:	$[\texttt{dead},\ \texttt{p = q} = \perp, \texttt{flag} = 1, Cond]$
	line 9:	$[\texttt{dead},\ \texttt{p = q} = \perp, \texttt{flag} = 1, Cond] \cup$ $[\texttt{live},\ \texttt{p = q} = \perp, \texttt{flag} = 0, \neg Cond]$
	line 10:	$[\texttt{live},\ \texttt{p = q} = \perp, \texttt{flag} = 0, \neg Cond]$
	line 11:	$[\texttt{dead},\ \texttt{p = q} = \perp, \texttt{flag} = 1, Cond] \cup$ $[\texttt{live},\ \texttt{p = q} = \perp, \texttt{flag} = 0, \neg Cond]$
points-to sets	line 6:	$pt(\texttt{p}) = \{o_1\}$
	lines 11 and 12:	$pt(\texttt{q}) = \{o_1\}$
	line 13:	$pt(\texttt{p}) = \{o_1, o_2\}$

At line 12, the pointer analysis finds precisely that p and q point to o_1 allocated at line 2. Therefore, a state transition from dead to error occurs in s_1, so that a true UAF bug for $use(q)$ at line 12 is reported. However, at line 13, due to the lack of path-sensitivity, p is found to point to both $o1$ (allocated at line 2) and $o2$ (allocated at line 7), resulting in a spurious alias relation between $*p$ and $*q$. Therefore, $free(p)$ at line 6 and $use(q)$ at line 13 are considered to access o_1, triggering a spurious state transition from dead to error in s_1. Thus, a false alarm is raised for $use(p)$ at line 13.

Typestate Analysis with Imprecise Handling of Lists. Figure 3(b) gives a linked-list example to demonstrate that field-sensitivity is not powerful enough to enable pointer analysis to distinguish the internal structure of an aggregate object.

With field-sensitivity, we can distinguish the head node (represented by o_1) from the remaining 10 nodes (abstracted by o_2) in the linked-list, created at the two allocation sites at lines 8 and 10, respectively. Thus, the typestate analysis can correctly prove the absence of a UAF bug for $use(n1)$ at line 17 and report $use(n2)$ at line 18 as a true UAF bug. However, the pointer analysis cannot distinguish the accesses to the second and third elements of the linked-list, since $pt(n2) = pt(n3) = \{o_2\}$, resulting in a spurious alias relation between $*n2$ and $*n3$. Therefore, a false alarm for $use(n3)$ at line 19 is reported.

2.4 Discussion

There are many spurious aliases introduced by pointer analysis. We propose to apply machine learning to significantly reduce their presence in order to improve the precision of typestate analysis.

3 TAC APPROACH

We introduce Tac, including its training phase (Section 3.1) and machine-learning-guided typestate analysis phase (Section 3.2).

3.1 Training

The aim of our SVM classifier is to further disambiguate the UAF-related aliases imprecisely reported by pointer analysis.

Building an SVM Classifier. We use $x \in X$ to denote a UAF sample representing a pair of free and use sites $\langle free(p), use(q)\rangle_x \in X$. A feature F_i is either a syntactic or semantic property of a program, mapping x to either a boolean or numeric value $F_i : X \rightarrow \mathbb{N}$. Following the standard normalization [10] to achieve accuracy in the training process, we adjust the values of the samples in X in order to map a feature to a real number between 0 and 1 inclusive, by using function $\mathscr{F}_i : X \rightarrow [0,1]^n$. Specifically, given a sample $x \in X$, this is done as $\mathscr{F}_i(x) = (F_i(x) - \min(F_i(X)))/(\max(F_i(X)) - \min(F_i(X)))$, where min (max) returns the minimum (maximum) value for F_i among all the samples in X. Finally, a feature vector of length n is defined as $\mathscr{F} = (\mathscr{F}_1, \ldots, \mathscr{F}_n)$ containing a set of n features to capture the properties of every sample.

During the training process, we build an SVM classifier $\mathscr{C} : [0,1]^n \rightarrow \{0,1\}$ that takes a feature vector $\mathscr{F}$ of a sample $\langle free(p), use(q)\rangle_x$ as input and returns whether $*p$ and $*q$ are aliases (1) or not (0). The typestate analysis phase will make use of the classifier to reduce the number of UAF-related spurious aliases. For a program, let X_{all} be the set of all UAF pairs $\langle free(p), use(q)\rangle_x$ causing the FSA in Figure 1 to transit into `error`, where $*p$ and $*q$ are found to be aliased by a pointer analysis used. Only the following subset X_{ML} will be further investigated by the typestate analysis:

$$X_{ML} = \{\langle free(p), use(q)\rangle_x \in X_{all} \mid \mathscr{C}(\mathscr{F}(x)) = 1 \wedge pt(p) \cap pt(q) \neq \emptyset\}$$

In other words, the UAF pairs $\langle free(p), use(q)\rangle_x$ in $X_{all} \setminus X_{ML}$ are ignored, since $*p$ and $*q$ are not aliases by the SVM classifier.

Extracting Program Features. Table 2 gives a set of 35 features, which are divided into four categories below, to represent a UAF sample. Note that this set of features can be extended by considering other program characteristics or reused by other program analyses.

- **Type Information (Features 1 – 9)**. Type information is used to identify arrays (F_1), structs (F_2), C++ containers (F_3), different kinds of use sites (F_4, F_5 and F_6), global variable accesses for free and use sites (F_7 and F_8), and type compatibility for the pointers p and q at a free site $free(p)$ and a use site $use(q)$ (F_9).
- **Control Flow (Features 10 – 17)**. We consider the following control-flow properties, including whether a pair of free and use sites resides in the same loop or recursion cycle (F_{10} and F_{11}), the distance between a free site and a use site in the program's call graph (F_{12}), control-flow reachability from a free site to a use site via a loop back-edge (F_{13}), control-flow dominance and post-dominance between a free site and a use site (F_{14} and F_{15}), the number of indirect calls along the shortest path from a free site to a use site in the program's call graph (F_{16}), and control-flow reachability from a use site to a free site for a UAF pair (F_{17}).
- **Common Programming Practices (Features 18 – 25)**. We consider a number of programming practices for memory management, including setting p to null immediately after $free(p)$ (F_{18}), returning an integer or a boolean value from a wrapper for $free(p)$ to signify the success or failure for $free(p)$ (F_{19} and F_{20}), pointer casting (F_{21}), setting p to point to a newly allocated object after $free(p)$ (F_{22}), reference counting for an object (F_{23}), and null checking before a pointer is freed (F_{24}) or used (F_{25}).
- **Points-to Information (Features 26 – 35)**. We take advantage of the points-to information computed by the pointer analysis used, including the sizes of the points-to sets at free and use sites (F_{26} and F_{27}), the number of UAF pairs sharing the same free (F_{28}) or use (F_{29}) site, and the number of aliased pointers pointing to a candidate UAF object (F_{30}). In addition, we also consider whether a candidate object is allocated in loops (F_{31}), recursion cycles (F_{32}) or as a node of a linked-list (F_{33}) participating in a points-to cycle (causing the object to abstract many concrete nodes in the list). Finally, we consider whether p and q at a UAF pair, $\langle free(p), use(q)\rangle$, are the same variable (F_{34}) and whether q at $use(q)$ is defined just before the free site (F_{35}).

Let us revisit the two examples in Figure 3. In Figure 3(a), $F_9, F_{14}, F_{15}, F_{22}, F_{26}, F_{27}, F_{34}$ and F_{35} for line 13 are useful to predict that the UAF pair at lines 6 and 13 is not a bug. In Figure 3(b), F_{31}, F_{33} and F_{35} for line 19 can help avoid a false alarm that would otherwise be raised for the UAF pair at lines 16 and 19.

Support Vector Machine (SVM). Given a set of labeled samples represented by their feature vectors, an SVM [15] can separate them by computing an underlying mathematical function, called a kernel function. There are four commonly used kernels: linear, polynomial, radial basis function (RBF) and sigmoid kernels. Following [26, 36, 65], a RBF kernel is used. The kernel function maps each feature vector to a high dimensional space, where the SVM computes a hyperplane that best separates the labeled samples into two sides. Once trained, an SVM classifier can be used to classify a given UAF pair according to simply which side of the hyperplane it falls in.

3.2 Typestate Analysis

We describe our ESP-based typestate analysis for UAF detection. The basic idea is to achieve improved precision (compared to the prior work) by applying an SMV classifier to further validate the UAF-related aliases found imprecisely by the pointer analysis.

Our typestate analysis is a whole-program analysis. We describe how it works, first intraprocedurally and then interprocedurally.

3.2.1 Intraprocedural Analysis. Intraprocedurally, our typestate analysis is performed on the CFG of a function, $CFG = (N, E)$, where N is a set of nodes representing program statements and $E \subseteq N \times N$ is a set of edges corresponding to the flow of control between nodes. For a given edge e, $src(e)/dst(e)$ denotes its source/destination node.

Following ESP [17], we assume three types of nodes on a CFG, i.e., N = `JointNode` $\cup$ `BranchNode` $\cup$ `StmtNode`: (1) a joint node (i.e., ϕ-node) $n \in$ `JointNode` has two incoming edges $InEdge_0(n)$ and $InEdge_1(n)$, and a single outgoing edge; (2) a branch node

Table 2: 35 Features used by the SVM classifier for a UAF sample $\langle free(p), use(q)\rangle$, where $o \in pt(p) \cap pt(q)$.

Group	ID	Feature	Type	Description
Type Information	1	Array	Boolean	o is an array or an element of an array
	2	Struct	Boolean	o is a struct or an element of a struct
	3	Container	Boolean	o is a container (e.g., vector or map) or an element of a container
	4	IsLoad	Boolean	$use(q)$ is a load instruction
	5	IsStore	Boolean	$use(q)$ is a store instruction
	6	IsExtCall	Boolean	$use(q)$ is an external call
	7	GlobalFree	Boolean	$free(p)$, where p is a global pointer
	8	GlobalUse	Boolean	$use(q)$, where q is a global pointer
	9	CompatibleType	Boolean	p and q are type-compatible at $free(p)$ and $use(q)$
Control Flow	10	InSameLoop	Boolean	$free(p)$ and $use(q)$ are in the same loop
	11	InSameRecursion	Boolean	$free(p)$ and $use(q)$ are in the same recursion cycle
	12	#FunctionInBetween	Integer	number of functions in the shortest path from $free(p)$ to $use(q)$ in the program's call graph
	13	DiffIteration	Boolean	$use(q)$ appears after $free(p)$ via a loop back-edge
	14	Dom	Boolean	$free(p)$ dominates $use(q)$
	15	PostDom	Boolean	$use(q)$ post-dominates $free(p)$
	16	#IndCalls	Integer	number of indirect calls in the shortest path from $free(p)$ to $use(q)$ in the program's call graph
	17	UseBeforeFree	Boolean	a UAF pair, $free(p)$ and $use(q)$, is also a use-before-free
Common Programming Practices	18	NullifyAfterFree	Boolean	p is set to null immediately after $free(p)$
	19	ReturnConstInt	Boolean	a const integer is returned after $free(p)$
	20	ReturnBoolean	Boolean	a Boolean value is returned after $free(p)$
	21	Casting	Boolean	pointer casting is applied to q at $use(q)$
	22	ReAllocAfterFree	Boolean	p is redefined to point to a newly allocated object immediately after $free(p)$
	23	RefCounting	Boolean	o is an reference-counted object
	24	ValidatedFreePtr	Boolean	null checking for p before $free(p)$
	25	ValidatedUsePtr	Boolean	null checking for q before $use(q)$
Points-to Information	26	SizeOfPointsToSetAtFree	Integer	number of objects pointed to by p at $free(p)$
	27	SizeOfPointsToSetAtUse	Integer	number of objects pointed to by q at $use(q)$
	28	#UAFSharingSameFree	Integer	number of UAF pairs sharing the same $free(p)$
	29	#UAFSharingSameUse	Integer	number of UAF pairs sharing the same $use(q)$
	30	#Aliases	Integer	number of pointers pointing to o
	31	AllocInLoop	Boolean	o is allocated in a loop
	32	AllocInRecursion	Boolean	o is allocated in recursion
	33	LinkedList	Boolean	o is in a points-to cycle (signifying its presence in a linked-list)
	34	SamePointer	Boolean	p and q at $free(p)$ and $use(q)$ are the same pointer variable
	35	DefinedBeforeFree	Boolean	q at $use(q)$ is defined before $free(p)$

Table 3: Program statements in the LLVM-like SSA form.

$p, q, i \in \mathcal{T}$ (Top-level Vars), $a, o \in \mathcal{A}$ (Address-taken Objs), $c, fld \in C$ (Consts)			
uop	$\in$	$\{++, --, !\}$	unary operators
bop	$\in$	$\{+, -, \times, /, \&\&, \|\|, ==, \neq, <, >, \leq, \geq\}$	binary operators
$\mathcal{E}$	::=	$p, q \mid c \mid \mathcal{E}_1$ bop $\mathcal{E}_2 \mid$ uop $\mathcal{E}$	scalar expressions
$use(p)$	::=	$*p=q \mid q=*p \mid p[i]=q \mid q=p[i]$ $\mid p\rightarrow fld=q \mid q=p\rightarrow fld$	memory access
StmtNode	::=	$p = \mathcal{E}$	scalar statement
		$\mid p=\&a \mid p=malloc_o \mid use(p) \mid free(p)$	memory statements

$n_{\mathcal{E}} \in$ BranchNode with a branch condition expression $\mathcal{E}$ has a single incoming edge $InEdge_0(n)$ and two outgoing edges, $OutEdge_1(n)$ if $\mathcal{E}$ evaluates to true and $OutEdge_0(n)$ otherwise; and (3) a statement node $n \in$ StmtNode has a single incoming edge and a single outgoing edge.

Program Representation. Table 3 gives all the statements put on an LLVM-like SSA form for a function [23, 30, 34, 58].

The set of variables is separated into two subsets: $\mathcal{T}$ containing all *top-level variables*, including pointers and non-pointer scalars, and $\mathcal{A}$ containing all possible targets, i.e., *address-taken objects* of a pointer. C is a set of all constants. We use $use(p)$ to denote a memory access via a pointer p, including a pointer dereference $*p$, a field access $p \rightarrow fld$, and an array access $p[i]$. Complex statements like $*p = *q$ are simplified to $t = *q$ and $*p = t$ by introducing a top-level pointer t. Accessing a multi-dimensional array as in $q = p[i][j]$ is transformed into $q = p[k]$, where $k = i * n + j$ and n represents the size of the second dimension of the array. We consider UAF only for the objects o in the heap allocated via $p = malloc_o$ (but not for the objects a on the stack allocated via $p = \&a$).

Given a $CFG = (N, E)$, our typestate analysis computes and maintains the data-flow facts in $DF(e)$ for every edge $e \in E$, where $DF(e)$ maps e to a set of symbolic states $\mathcal{S}$ with each element $s = \langle \rho, \sigma \rangle$ consisting of a property state $\rho \in Properties = \{$live, dead, error$\}$ and an execution state σ. The notation $\sigma(\mathcal{E})$ is used to evaluate the expression $\mathcal{E}$ in σ. As is standard, $\sigma[v \leftarrow v']$ denotes the state obtained by updating the value of v in σ with v' and leaving the values of all other variables in σ unchanged.

$$F_{jnt}(n, S_1, S_2) = \alpha(S_1 \cup S_2)$$
$$F_{br}(n, S, \mathcal{E}) = \alpha(\{\langle \rho, \sigma' \rangle \mid \sigma' = \sigma \cup \{\mathcal{E}\} \wedge Feasible(\sigma') \wedge \langle \rho, \sigma \rangle \in S\})$$
$$F_{stmt}(n, S, o) = \alpha(\{TF(n, \langle \rho, \sigma \rangle, o) \mid \langle \rho, \sigma \rangle \in S\})$$

(a) Flow functions for three types of CFG nodes

$$\alpha(S) = \{\langle d, \bigsqcup_{\langle \rho, \sigma \rangle \in S[d]} \sigma \rangle \mid d \in Properties \wedge S[d] \neq \emptyset\}$$
where $S[d] = \{\langle \rho, \sigma \rangle \in S \mid d = \rho\}$

(b) Grouping function for merging symbolic states

$$TF\ (n,\ \langle \texttt{live},\ \sigma \rangle,\ o) = \begin{cases} \langle \texttt{live},\ \sigma[q \leftarrow \sigma(\mathcal{E})] \rangle & \text{if } n \text{ is } q = \mathcal{E} \\ \langle \texttt{dead}, \sigma \rangle & \text{else if } n \text{ is } \texttt{free}(p) \wedge o \in pt(p) \\ \langle \texttt{live},\ \Gamma(\sigma, n) \rangle & \text{otherwise} \end{cases}$$

$$TF\ (n,\ \langle \texttt{dead},\ \sigma \rangle,\ o) = \begin{cases} \langle \texttt{dead},\ \sigma[q \leftarrow \sigma(\mathcal{E})] \rangle & \text{if } n \text{ is } q = \mathcal{E} \\ \langle \texttt{error},\ \Gamma(\sigma, n) \rangle & \text{else if } n \text{ is } \texttt{use}(q) \wedge o \in pt(q) \wedge predict(n) \\ \langle \texttt{error},\ \sigma \rangle & \text{else if } n \text{ is } \texttt{free}(q) \wedge o \in pt(q) \wedge predict(n) \\ \langle \texttt{dead},\ \Gamma(\sigma, n) \rangle & \text{otherwise} \end{cases}$$

where $predict(n) = \begin{cases} True & \text{if } \exists\, m \in \mathcal{F}_o :\ \langle m, n \rangle \in X_{ML} \\ False & \text{otherwise} \end{cases}$

(c) Transfer function for program statement (with $\mathcal{F}_o$ defined in Figure 5)

Statement n	if $pt(p) = \{o'\}$ and $o' \in Singleton$	otherwise
$q = *p$	$\sigma[q \leftarrow \sigma(o')]$	$\sigma[q \leftarrow \bot]$
$q = p \rightarrow fld$	$\sigma[q \leftarrow \sigma(o'.fld)]$	$\sigma[q \leftarrow \bot]$
$q = p[i]$	$\sigma[q \leftarrow \sigma(o'[\sigma(i)])]$	$\sigma[q \leftarrow \bot]$
$*p = q$	$\sigma[o' \leftarrow \sigma(q)]$	$\sigma[\forall o' \in pt(p) : o' \leftarrow \bot]$
$p \rightarrow fld = q$	$\sigma[o'.fld \leftarrow \sigma(q)]$	$\sigma[\forall o' \in pt(p) : o'.fld \leftarrow \bot]$
$p[i] = q$	$\sigma[o'[\sigma(i)] \leftarrow \sigma(q)]$	$\sigma[\forall o' \in pt(p) : o'[\sigma(i)] \leftarrow \bot]$
$p = \&a$ $p = malloc_o$	$\sigma[p \leftarrow \bot]$	

(d) $\Gamma(\sigma, n)$: Updating execution states for memory statements

Figure 4: The data-flow functions for TAC's machine-learning-guided intraprocedural typestate analysis.

```
1:  procedure Solve (n_malloc_o, CFG = (N, E))
2:    foreach e ∈ E    DF(e) := ∅
3:    DF(OutEdge(n_malloc_o)) := {[live, ⊤]}
4:    Worklist := {dst(OutEdge(n_malloc_o))}
5:    while(Worklist ≠ ∅)
6:        Remove a node n from Worklist
7:        switch (n)
8:        case: n ∈ JointNode
9:            S := F_jnt (n, DF(InEdge_0(n)), DF(InEdge_1(n)))
10:           Add (OutEdge_0(n), S)
11:       case: n_E ∈ BranchNode
12:           S_T := F_br (n, DF(InEdge_0(n)), E)
13:           S_F := F_br (n, DF(InEdge_0(n)), ¬E)
14:           Add (OutEdge_1(n), S_T)
15:           Add (OutEdge_0(n), S_F)
16:       case: n ∈ StmtNode
17:           S := F_stmt (n, DF(InEdge_0(n)))
18:           Add (OutEdge_0(n), S)
19:           if(n is free(p)) F_o := F_o ∪ {free(p)}
20: end procedure
21: procedure Add (e, S)
22:   if(DF(e) ≠ S)
23:       DF(e) := S
24:       Worklist := Worklist ∪ {dst (e)}
25: end procedure
```

Figure 5: TAC's intraprocedural typestate analysis.

Machine-Learning-Guided Typestate Analysis. Figure 5 gives a standard worklist algorithm for the typestate analysis that computes and updates the data-flow facts on the CFG of a function for a given UAF candidate object o (determined by pre-analysis) until a fixed point. Unlike ESP [17], which starts its path-sensitive analysis from the entry of a CFG, our analysis starts from the allocation statement n_{malloc_o} (line 3) of o to trade precision for efficiency.

Our analysis handles three types of CFG nodes, `JointNode` (lines 8 – 10), `BranchNode` (lines 11 – 15) and `StmtNode` (lines 16 – 19) using the flow functions given in Figure 4(a), mapping an input state to an output state for every node. At line 19, we record the current free sites for object o in $\mathcal{F}_o$ found during the control-flow traversal so that we can pair them with uses of o in order to validate their associated aliases using our SVM classifier. Figure 4(b) gives the typestate grouping function $\alpha(S)$ that reorganizes a set of symbolic states S by merging the execution states of two symbolic states s_1 and $s_2 \in S$ if s_1 and s_2 have the same property state.

For a joint node, the flow function F_{jnt} unions the data-flow facts on its incoming edges. For a branch node, F_{br} updates its input execution state σ with $\sigma' = \sigma \cup \{\mathcal{E}\}$ if $Feasible(\sigma')$ holds, i.e., the branch predicate $\mathcal{E}$ is not ruled out due to path contradiction, decided by a satisfiability solver, which is `Z3` [18] in our evaluation. For a statement node, F_{stmt} maps its input state to a new output state by using the transfer function TF defined in Figure 4(c). Note that a free statement is handled at line 19 as a special case.

$TF(n, \langle \texttt{live}, \sigma \rangle, o)$ and $TF(n, \langle \texttt{dead}, \sigma \rangle, o)$ handle the state transitions of o when the current property states are `live` and `dead`,

```
1:int main() {
2:   int* p = malloc(1);//o
3:   int flg = userInput();
4:   int* r = &flg;
5:   int i = 1, j = 0;
6:   if (flg < 0) {
7:      free(p);
8:      *r = i;
9:   }
10:  else {
11:     *p = i;
12:     *r = j;
13:  }
   . . . //q is defined here
14:  if (flg == 1)
15:     *q = i;
16:}
```

Line	Symbolic State
2:	[live, p=⊥]
4:	[live, r=p=flg=⊥]
5:	[live, r=p=flg=⊥, i=1, j=0]
6:	[live, r=p=⊥, flg<0, i=1, j=0]
7:	[dead, r=p=⊥, flg<0, i=1, j=0]
8:	[dead, r=p=⊥, flg=1, i=1, j=0]
11:	[live, r=p=⊥, flg≥0, i=1, j=0]
12:	[live, r=p=⊥, flg=0, i=1, j=0]
13:	[dead, r=p=⊥, flg=1, i=1, j=0] ∪ [live, r=p=⊥, flg=0, i=1, j=0]
14:	[dead, r=p=⊥, flg=1, i=1, j=0]

Figure 6: An example for illustrating Tac.

respectively. The former is handled in the usual way. So let us focus on the latter. Tac reports a UAF bug if $TF(n, \langle \texttt{dead}, \sigma \rangle, o)$ signifies a state transition of o from `dead` to `error` (Figure 1), when n is a $use(q)$ or a $free(q)$. Thus, there are two cases. If n is a $use(q)$, then a UAF warning is issued when both (1) $o \in pt(q)$, implying that $*p$ in a free site $free(p)$ seen earlier and $*q$ are found to be aliased with o by the pointer analysis [58], and (2) $predict(n)$ returns true, implying that this alias is also validated by our SMV classifier. If n is a $free(q)$, then a double-free warning is issued, instead.

Finally, Figure 4(d) gives the rules for performing strong updates on address-taken objects in order to improve the precision of $Feasible(\sigma')$ in Figure 4(a) for top-level variables. We can distinguish two cases when tracking the execution states of statements. For a scalar statement $p \leftarrow \mathcal{E}$, σ simply evolves into $\sigma[p \leftarrow \sigma(\mathcal{E})]$. However, updating σ for a memory-related statement is more complex, as shown in Figure 4(d). Strong updates are performed when p points to exactly one (runtime) singleton object o' in $Singleton$, which contains all objects in $\mathcal{A}$ except for the locals in recursion cycles and all the heap objects [34, 58]. Otherwise, the variables on the left-hand side of an assignment are updated to be $\perp$ conservatively. Note that dynamically (statically) allocated arrays are treated as heap objects (locals or globals). For an array access $p[i]$, $o'[\sigma(i)]$ represents any element in o' if i is statically unknown.

3.2.2 Interprocedural Analysis. Given a whole program, our typestate analysis proceeds context-sensitively on its interprocedural CFG [29] with indirect calls resolved by Andersen's pointer analysis [5]. Every function has a unique entry node and a unique exit node, with each callsite being split into a call node and a return node. Context-sensitivity is achieved by solving a *balanced-parentheses problem* [50] with an additional abstract call stack (a sequence of callsites) maintained in every symbolic state to filter out unrealizable inter-procedural paths by matching calls and returns. Following ESP [17], we apply a mod-ref analysis to avoid analyzing a function invoked at a callsite if it may not access the candidate UAF object being analyzed by using value-flow slicing [17, 60]. Unlike ESP [17], which starts its from the entry of the program, our analysis starts from an allocation statement, as discussed above.

3.2.3 Example. We use an example in Figure 6 to illustrate how Tac correctly reports the true UAF bug (at lines 7 and 15). At line 2, a memory object o is allocated and pointed by p. In the if-branch (lines 6 – 9), o is freed, indicated with flg set as 1. In the else-branch (lines 10 – 13), o is updated, indicated with flg set as 0. Lines 14 – 15 are the buggy code that mistakenly check $flg == 1$ instead of $flg == 0$ before dereferencing p, causing a UAF bug. Figure 6 gives the symbolic states obtained by Tac at some program points.

Tac starts from o's allocation site at line 2, where the property state of o is initialized as `live` and the symbolic state of p is set as $\perp$. At line 3 (not shown), flg is assumed to be initialized to $\perp$ (returned by `userInput()`). Let us see how the if-branch (lines 6 – 9) is analyzed. When analyzing line 6, Tac records its branch condition $flg < 0$ in the resulting symbolic state. At line 7, o is freed, causing the property state of o to transit from `live` to `dead`. At line 8, Tac makes a strong update to get $flg = 1$, since r points to flg, where $flg \in Singleton$.

Let us now move to the else-branch (lines 10 – 13). When analyzing line 10, Tac records $flg \geq 0$ in the resulting symbolic state. At line 11, the property state of o remains unchanged according to Γ. At line 12, Tac makes a strong update to get $flg = 0$.

At line 13, F_{br} is applied to merge the two symbolic states from the two branches. The `if` branch at lines 14-15 filters out the states that do not satisfy $flg == 1$. Thus, [`dead, r=p=⊥, flg=1, i=1, j=0`] is kept but [`live, r=p=⊥, flg=0, i=1, j=0`]] dropped.

Finally, there are two cases when line 15 ($*q = i$) is analyzed. If $*q$ is found not to be aliased with $*p$ according to the pointer analysis, then no UAF bug exists. Otherwise, $predict(*q = i)$ comes into play. The FSA for o will transit from `dead` to `error` if $\langle$`free(p), *q=i`$\rangle \in X_{ML}$ and remains in the `dead` state otherwise.

Table 4: Open-source benchmarks.

Program	Version	Language	LOC	#Frees	#Uses
rtorrent	0.96	C++	13,036	118	3,039
less	451	C	27,134	86	7,902
bitlbee	4.2	C	68,413	201	5,897
nghttp2	1.6.0	C++	71,387	29	7,566
mupdf	1.2.337	C++	122,481	253	105,911
h2o	1.7.2	C++	517,731	896	150,887
xserver	1.14.3	C	568,964	1,675	90,841
php	5.6.7	C	709,356	1,391	244,917
Total	—	—	2,098,502	4,649	616,960

4 EVALUATION

Our evaluation aims to demonstrate the effectiveness of our machine-learning-guided approach in detecting UAF bugs with a low false alarm rate in real-world programs. We evaluate Tac using eight popular open-source C/C++ programs described in Table 4: `rtorrent`, a fast text-based BitTorrent client; `less`, a text file viewer; `bitlbee`, a cross-platform IRC instant messaging gateway; `nghttp2`, an implementation of hypertext transfer protocol; `mupdf`, an E-book viewer; `h2o`, an optimized HTTP server, `xserver`, a windowing system for bitmap displays on UNIX-like OS; and `php`, a general-purpose scripting language for web development.

Tac is implemented in the LLVM compiler (version 3.8.0) [30]. The source files of each C/C++ program are compiled under -O0 into LLVM bit-code files by `Clang` and then merged using the `LLVM Gold Plugin` at link time to produce a whole program bc file.

In the training phase, Tac uses the widely-used SVM classifier `libSVM` [10]. In the analysis phase, Tac's pre-analysis is implemented on top of SVF [59]. For the flow-sensitive demand-driven pointer analysis [58] deployed in the analysis phase, the budget of a points-to query is set as 50,000 (the maximum number of def-use chains traversable) in the underlying pointer analysis to enable early termination and returning conservative may-alias results.

Our experiments were conducted on a 3.0 GHZ Intel Core2 Duo processor with 128 GB memory, running RedHat Enterprise Linux 5 (2.6.18). As listed in Table 4, the eight programs combined exhibit a total of 2,098,502 LOC, containing 4,649 free sites and 616,960 use sites. As shown in Table 5, these programs contain 6 known UAF bugs, with 5 registered in the CVE database and 1 unregistered.

Table 5: 14 (distinct) UAF bugs detected by Tac, including 5 known CVE vulnerabilities and 1 known bug given in Column 2 and 8 new bugs given in Column 3.

	Known bugs		New bugs
Program	Identifier	Detected	#Detected
rtorrent	—	—	0
less	—	—	1
bitlbee	CVE-2016-10188	✓	0
nghttp2	CVE-2015-8659	✓	0
mupdf	BugID-694382	✓	0
h2o	CVE-2016-4817	✓	5
xserver	CVE-2013-4396	✓	0
php	CVE-2015-1351	✓	2

4.1 The Training Phase

We train the SVM classifier for Tac using both false and true UAF samples in real-world programs, as illustrated in Table 6. To generate false alarm samples, we run Tac-NML, an ESP-based typestate analysis without machine learning, to analyze four relatively small ones in the set of eight programs evaluated (Table 4), `rtorrent`, `less`, `bitlbee`, and `nghttp2`. Then, we manually inspect 30% (a limit set for the manual labor invested) of all the warnings reported by Tac-NML for each program. To generate true UAF bugs, we use all the 138 C programs and 322 C++ programs (which are small programs extracted from real-world applications) in the CWE-416-Use-After-Free category of Juliet Test Suite (JTS) [1], with each program containing one single UAF vulnerability. In addition, we also make use of synthetic UAF bugs automatically introduced into the training programs, inspired by the bug insertion technique [47]. To do so, we first find all *use-before-free* pairs $\langle use(p), free(q)\rangle$ statically by conducting a control-flow reachability analysis from a $use(p)$ to a $free(q)$, where $*p$ and $*q$ are aliases identified by a flow-sensitive pointer analysis [58]. Next, we swap $use(p)$ and $free(q)$ for each pair and run Valgrind [43] to detect dynamically if the thus injected UAF bug manifests itself as a true bug under the default test inputs in every program. Finally, all UAF samples, including 623 false and 858 true bugs as shown in Columns 2 and 3 of Table 6, are annotated for feature extraction.

The training phase applies the standard 5-fold cross validation to find optimal intrinsic SVM parameters that yield the best classification accuracy. We consider three standard metrics: accuracy, precision and recall. *Accuracy* is the percentage of correctly classified samples out of all the samples. *Precision* is the percentage of correctly classified true positive samples out of the samples that are classified as true positives. *Recall* is the percentage of correctly classified true positive samples out of all the true positive samples.

Table 6: Results of training. #True and #False are the numbers of true and false UAF samples, respectively.

	Samples		Results		
Program	#True	#False	Accuracy	Precision	Recall
rtorrent	46	69	88.6%	81.0%	93.4%
less	22	237	96.9%	77.0%	91.0%
bitlbee	52	31	90.4%	86.7%	100.0%
nghttp2	43	61	82.7%	75.5%	86.0%
JTS-C	138	138	96.4%	97.8%	94.9%
JTS-C++	322	322	97.4%	97.2%	97.5%
Total	623	858	95.0%	92.6%	95.8%

Due to 5-fold cross validation, Tac is highly effective for the training programs, with its the accuracy, precision and recall results given in Columns 4 – 6 of Table 6. For all the training programs combined, Tac's accuracy, precision and recall are 95.0%, 92.6% and 95.8%, respectively. These results indicate that the SVM classifier trained by using the 35 features (Table 2) and the RBF kernel (Section 3.1) is effective in classifying true and false UAF samples.

4.2 The Analysis Phase

Our results are summarized in Table 7. Column 2 gives the number of candidate UAF pairs computed by Tac's pre-analysis, which selects a candidate $\langle free(p), use(q)\rangle$ if $free(p)$ can reach $use(q)$ context-sensitively via control-flow, where $*p$ and $*q$ are found to be aliases by Andersen's pointer analysis [5] implemented in [59].

In Columns 3 – 4, we give the results produced by Tac-NML (i.e. Tac without machine learning). For each program, Column 3 gives the number of warnings reported and Column 4 gives the reduction rate with respect to the number of warnings produced by the pre-analysis. On average (across the eight programs), Tac-NML achieves a reduction rate of 81.2%. This indicates that path-sensitive typestate analysis alone is quite effective in improving the precision of a coarse-grained pre-analysis. However, a total of 19,803 UAF warnings are still reported, making Tac-NML impractical.

In Columns 5 – 6, we give the results produced by Tac when machine learning is enabled. For each program, Tac has improved Tac-NML significantly by reducing the number of warnings further (Column 5) and thus achieving an impressive reduction rate (with respect to Tac-NML) (Column 6). On average, Tac achieves a reduction rate of 96.5%, resulting in only 266 warnings. This shows its effectiveness in suppressing warnings raised.

Tac is also efficient, as shown in Column 7 (with the analysis time of a program averaged over the five runs). Tac spends a total of 4.2 hours on analyzing all the eight programs (consisting of 2,098 KLOC in total), with 90 seconds for the smallest program (`less`) and 5,942 seconds for the largest program (`php`).

In the last three columns, with Column 8 giving the number of true bugs (confirmed by manual inspection), Column 9 the false positive rate (FPR), and Column 10 the true positive rate (TPR) for

Table 7: Analysis results. #Candidates is the number of candidate UAF pairs found by pre-analysis. #Warns$^{\text{Tac}-NML}$ and #Warns$^{\text{Tac}}$ denote the number of warnings reported by Tac-NML and Tac, respectively. Reduction1 is computed as (#Candidates - #Warns$^{\text{Tac}-NML}$) / #Candidates. Reduction 2 is computed as (#Warns$^{\text{Tac}-NML}$ - #Warns$^{\text{Tac}}$) / #Warns$^{\text{Tac}-NML}$. #True is the number of true bugs (confirmed by manual inspection), FPR is the false positive rate and TPR is the true positive rate.

Program	#Candidates	#Warns$^{\text{Tac}-NML}$	Reduction1	#Warns$^{\text{Tac}}$	Reduction2	Time (secs)	#True	FPR	TPR
rtorrent	803	229	71.5%	0	100.0%	90	0	—	—
less	4,628	790	82.9%	3	99.6%	316	1	66.7%	33.3%
bitlbee	529	113	78.6%	16	85.8%	151	9	43.8%	56.3%
nghttp2	975	210	78.5%	16	92.4%	83	7	56.3%	43.8%
mupdf	21,701	1,658	92.4%	50	97.0%	197	19	62.0%	38.0%
h2o	18,143	3,559	80.4%	23	99.4%	6,205	9	60.9%	39.1%
xserver	53,258	6,706	87.4%	102	98.5%	2,053	40	60.8%	39.2%
php	26,306	5,818	77.9%	56	99.0%	5,942	24	57.1%	42.9%
Total	126,343	19,083	—	266	—	15,037	109	—	—

each program, we see that Tac is capable of finding UAF bugs at low false positive rates. Out of the total 266 warnings reported, 109 are true bugs, yielding an FPR of 58.2% (or a TPR of 41.8%). Thus, our machine-learning-guided approach is effective in locating UAF bugs (with reasonable manual inspection effort required).

Among the 109 bugs found, 14 bugs are distinct (with the UAF pairs sharing the same free site and dereferencing the same pointer at their use sites being counted as one), as listed in Table 5. These include 6 known ones (5 known CVE vulnerabilities and 1 known bug) and 8 new ones (1 in `less`, 5 in `h2o` and and 2 in `php`).

For `less`, the 1 new bug is found in a while loop in function `ch_delbufs` in `ch.c` (illustrated in Figure 7). For `h2o`, the 5 new bugs are all interprocedural due to premature connection close operations, including one in function `do_emit_writereq` in `connection.c` (illustrated in Figure 8), one in function `h2o_timeout_unlink` in `timeout.c`, one in function `h2o_http2_scheduler_run` in `scheduler.c`, one in function `h2o_linklist_unlink` and one in function `h2o_linklist_islink` in `linklist.c`. For `php`, the 2 new bugs are interprocedural, found in `zend_persist.c` (illustrated in Figure 9).

It is important to emphasize that for the 14 distinct UAF bugs found, as listed in Table 5, only 3 bugs appear in the four training programs. This demonstrates again the effectiveness of our approach in applying machine learning to static UAF detection.

4.3 Case study

Let us take a look at some representative UAF bugs (both previously known and unknown) found by Tac in three programs.

`less`. Figure 7 shows a new UAF bug found in `less` (version 451) by Tac. At line 782, the program frees an object pointed to by `bn` and then starts possibly the next iteration of the while-loop at line 778. At line 780, `bn` is made to point to the same freed object and then dereferenced four times at line 781, causing one distinct UAF bug. This bug occurs since the programmer forgot to update `ch_bufhead` in the while loop after `bn` has been freed.

`h2o`. Figure 8 shows a known UAF bug (CVE-2016-4817) in `h2o` (version 1.7.2) detected by Tac. The program frees `conn` at line 261 in `close_connection_now` through a nested call chain via lines 834 and 861. Then `conn` is used in the function `timeout_unlink` called

```
          //ch.c
           774   static void ch_delbufs()
           775   {
           776         register struct bufnode *bn;
           777
step2      778         while (ch_bufhead != END_OF_CHAIN)
           779         {
step3      780               bn = ch_bufhead;
step4      781               (bn)->next->prev = (bn)->prev;
                             (bn)->prev->next = (bn)->next;
step1      782               free(((struct buf *) bn));
           783         }
           784         ch_nbufs = 0;
           785         init_hashtbl();
           786   }
```

Figure 7: A UAF bug found in `less`, with the free, use and their aliasing highlighted in pink, blue and red, respectively.

```
         //lib/http2/connection.c
          228 void close_connection_now(http2_conn_t *conn) {
step1     261     free(conn);
          262 }

          811 static void parse_input(http2_conn_t *conn) {
          829     if (ret < 0) {
step2     834         close_connection_now(conn);
          836     }
          848 }

          850 static void on_read(socket_t *sock, int stat) {
          852     http2_conn_t *conn = sock->data;
step3     861     parse_input(conn);
step4     865     timeout_unlink(&conn->_write.timeout_entry);
step5     866     do_emit_writereq(conn);
          868 }

step6     994 int do_emit_writereq(http2_conn_t *conn) {
step7     1006    buf = {conn->_write.bbytes, conn->_write.bsz};
step8     1007    socket_write(conn->sock, &buf, 1, on_w_compl);
          1012 }
```

Figure 8: CVE-2016-4817 and two new bugs in `h2o`.

at line 865. Tac has succeeded in finding this CVE vulnerability and also two new bugs on dereferencing `conn` at lines 1006 – 1007 in the function `do_emit_writereq` called at line 866. These two new bugs are counted as one distinct bug.

`php`. Figure 9 shows a known UAF bug (CVE-2015-1351) and two new ones in `php` (version 5.6.7) detected by Tac. These bugs are found in two files. CVE-2015-1351 is in `zend_shared_alloc.c`,

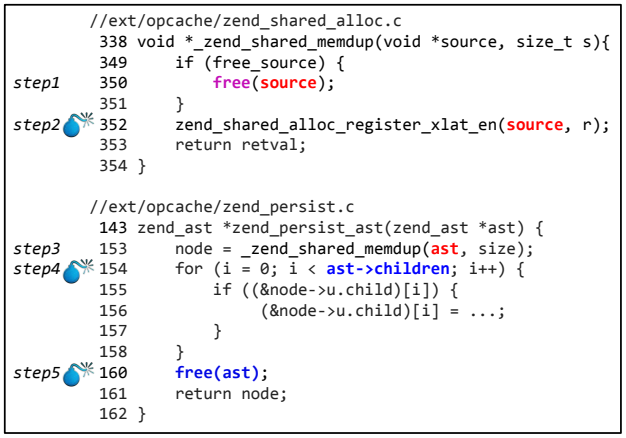

Figure 9: CVE-2015-1351 and two new bugs in php.

where the object pointed by `source` is freed at line 350 and then used inside a function called at line 352. In addition, Tac also finds two new UAF bugs in `zend_persist.c`. One is a UAF, where the object pointed to by `source` (and also by `ast`) is freed at line 350 and then accessed at line 154. The other is a double-free bug, as the same object (pointed by `source` and `ast`) is freed at line 350 and then again at line 160. CVE-2015-1351 was fixed in the latest version by simply moving line 352 to just before line 349. However, the two new ones remain unfixed.

5 RELATED WORK

UAF Detection. Most of the existing UAF detection techniques rely on dynamic analysis. CETS [42] enforces full memory safety by inserting metadata-manipulation instrumentations to perform runtime checking at pointer dereferences for detecting temporal memory errors, such as UAF. Undangle [9] applies dynamic taint analysis on binary code to track and detect UAF bugs based on the staleness of a pointer. Valgrind [43], as a memory-error debugging tool, can detect UAF bugs in binary code, at high time and space overheads. AddresSanitizer [52] performs a lightweight source level instrumentation by leveraging compiler optimizations, but may miss UAF bugs due to memory reallocation and unavailable (third-party) library code during instrumentation.

Static UAF detectors exist but are rare. Model checking (as in, e.g., coccinelle [46]) and abstract interpretation (as in, e.g., Clang [4] and Frama-C [16]) can be configured for UAF detection with user specified checking rules. However, they suffer from either the scalability issue or high false negative rates due to the lack of interprocedural analysis [4] and/or imprecision in handling aliases [46].

UAF Mitigation. Instead of detecting UAF bugs, some efforts are made on protecting against their exploitation. Cling [3] and Diehard [6] represent safe memory allocators that restrict memory reallocation by checking type consistency or approximating infinite heap. In these cases, dereferenced dangling pointers cannot access memory reallocated to other objects. Thus, UAF exploits are made harder. Alternatively, FreeSentry [32] and DangNull [32] track pointer propagation to invalidate all aliased pointers immediately their pointed-to object is freed, at the expense of high runtime and memory overheads. Control-flow integrity [2, 20, 44, 62, 64, 80] restricts the program execution to follow a precomputed CFG even if there are memory corruption bugs (e.g., UAF). Garbage collection for C/C++ [7] can mitigate some UAF bugs based on its automatic memory management. However, this requires every call to malloc() to be replaced by a call to a special allocator, and is thus hardly useful for legacy code and code using customized allocators.

Static Analysis for Memory Error Detection. Static analysis has been used for detecting a wide range of memory errors, such as buffer overflows [31, 35, 74], memory leaks [13, 60], uninitialized variables [41, 75], information leaks [12, 22, 38], SQL injection and XSS errors [25, 27, 63, 67], and format string vulnerabilities [55], on top of various program representations, such as inter-procedural SSA form [37, 60] abstract syntax tree [72], code property graph [71, 73], and value-flow graph [17, 59], to capture the syntax and/or semantic properties of a program. Tac, developed on top of SVF [59], inherits the strengths of traditional static analysis but also addresses its limitations (e.g., imprecision in handling path-sensitivity, loops, recursion cycles, arrays and lists) by learning and predicting the UAF-related aliases using machine learning techniques.

Machine Learning for Bug Detection. In recent years, machine learning techniques have been shown to be effective in guiding program analysis for bug detection, such as fault invariant classification [8] for reflecting important aspects of fault-revealing properties in a program, dynamic memory leak detection by classifying staleness values of objects [33], defect prediction (e.g., [66]), detection of malicious Java applets [51], source and sink classification for information flow analysis for Android apps [49], automatic program repair [39, 40], and abstract interpretation [24, 45]. This paper introduces machine learning techniques to typestate analysis for detecting temporal memory safety errors, such as UAF.

6 CONCLUSION

We present Tac, a machine-learning-guided static UAF detection framework that bridges the gap between typestate and pointer analyses by capturing the correlations between program features and UAF-related aliases that are often imprecisely answered by the state-of-the-art pointer analysis. Tac is effective (in terms of finding 5 known CVE vulnerabilities, 1 known bug, and 8 new bugs with a low false alarm rate) and scalable (in terms of analyzing a large real-world codebase with 2,098 KLOC in just over 4 hours).

Tac relies on pointer analysis and machine learning. Its accuracy can be further improved in several ways. First, path-sensitivity can be strengthened by solving path feasibility more soundly and precisely. Currently, non-singleton objects are over-approximated to contain $\perp$ and path conditions are interpreted as non-satisfiable when the underlying satisfiability solver returns unknown results (causing infeasible paths to be considered conservatively as feasible). Second, a more advanced pointer analysis can itself enable more UAF pairs to be ruled out as UAF bugs. Finally, a better SVM classifier can be developed by adding more UAF training samples in real-world programs and extending the set of features introduced.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable comments. The work is supported by ARC Grants (DP150102109 and DE170101081) and a CSIRO scholarship.

REFERENCES

[1] Juliet Test Suite 1.2. https://samate.nist.gov/SRD/testsuite.php.

[2] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *CCS'05*. 340–353.

[3] Periklis Akritidis. 2010. Cling: a memory allocator to mitigate dangling pointers. In *Security'10*. 177–192.

[4] Clang Static Analyzer. http://clang-analyzer.llvm.org/.

[5] Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language*. Ph.D. Dissertation. DIKU, University of Copenhagen.

[6] Emery D. Berger and Benjamin G. Zorn. 2006. DieHard: probabilistic memory safety for unsafe languages. In *PLDI'06*. 158–168.

[7] Hans-Juergen Boehm. 1993. Space efficient conservative garbage collection. In *PLDI'93*. 197–206.

[8] Yuriy Brun and Michael D. Ernst. 2004. Finding latent code errors via machine learning over program executions. In *ICSE'04*. 480–490.

[9] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. 2012. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *ISSTA'12*. 133–143.

[10] Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: a library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)* 2, 3 (2011), 27.

[11] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M Frans Kaashoek. 2011. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *APSYS'11*. Article No.5.

[12] Qi Alfred Chen, Zhiyun Qian, Yunhan Jack Jia, Yuru Shao, and Zhuoqing Morley Mao. 2015. Static detection of packet injection vulnerabilities: a case for identifying attacker-controlled implicit information leaks. In *CCS'15*. 388–400.

[13] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. 2007. Practical memory leak detection using guarded value-flow analysis. In *PLDI'07*. 480–491.

[14] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. 2015. Losing control: on the effectiveness of control-flow integrity under stack attacks. In *CCS'15*. 952–963.

[15] Corinna Cortes and Vladimir Vapnik. 1995. Support-vector networks. *Machine learning* 20, 3 (1995), 273–297.

[16] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C. In *SEFM'12*. 233–247.

[17] Manuvir Das, Sorin Lerner, and Mark Seigle. 2002. ESP: path-sensitive program verification in polynomial time. In *PLDI'02*. 57–68.

[18] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *TACAS'08*. 337–340.

[19] Nelson Elhage. 2011. Virtunoid: a KVM guest → host privilege, escalation exploit. *Black Hat USA* 2011 (2011).

[20] Xiaokang Fan, Yulei Sui, Xiangke Liao, and Jingling Xue. 2017. Boosting the Precision of Virtual Call Integrity Protection with Partial Pointer Analysis for C++. In *ISSTA'17*. 329–340.

[21] Stephen J Fink, Eran Yahav, Nurit Dor, G Ramalingam, and Emmanuel Geay. 2008. Effective typestate verification in the presence of aliasing. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 17, 2 (2008), 9.

[22] Michael Gordon, Deokhwan Kim, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard. 2015. Information Flow Analysis of Android Applications in DroidSafe. In *NDSS'15*.

[23] Ben Hardekopf and Calvin Lin. 2011. Flow-sensitive pointer analysis for millions of lines of code. In *CGO'11*. 289–298.

[24] Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. 2017. Machine-Learning-Guided Selectively Unsound Static Analysis. In *ICSE'17*. 519–529.

[25] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. 2006. Pixy: a static analysis tool for detecting web application vulnerabilities. In *SP'06*. 258–263.

[26] S Sathiya Keerthi and Chih-Jen Lin. 2003. Asymptotic behaviors of support vector machines with Gaussian kernel. *Neural computation* 15, 7 (2003), 1667–1689.

[27] Adam Kieyzun, Philip J Guo, Karthick Jayaraman, and Michael D Ernst. 2009. Automatic creation of SQL injection and cross-site scripting attacks. In *ICSE'09*. 199–209.

[28] William Landi. 1992. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)* 1, 4 (1992), 323–337.

[29] William Landi and Barbara G Ryder. 1992. A safe approximate algorithm for interprocedural aliasing. In *PLDI'92*. 235–248.

[30] Chris Lattner and Vikram Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *CGO'04*.

[31] Wei Le and Mary Lou Soffa. 2008. Marple: a demand-driven path-sensitive buffer overflow detector. In *FSE'08*. 272–282.

[32] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. 2015. Preventing use-after-free with dangling pointers nullification.. In *NDSS'15*.

[33] Sangho Lee, Changhee Jung, and Santosh Pande. 2014. Detecting memory leaks through introspective dynamic behavior modelling using machine learning. In *ICSE'14*. 814–824.

[34] Ondrej Lhoták and Kwok-Chiang Andrew Chung. 2011. Points-to analysis with efficient strong updates. In *POPL'11*. 3–16.

[35] Lian Li, Cristina Cifuentes, and Nathan Keynes. 2010. Practical and effective symbolic analysis for buffer overflow detection. In *FSE'10*. 317–326.

[36] Hsuan-Tien Lin and Chih-Jen Lin. 2003. *A study on sigmoid kernels for SVM and the training of non-PSD kernels by SMO-type methods*. Technical Report. Department of Computer Science, National Taiwan University.

[37] Benjamin Livshits and Monica S Lam. 2003. Tracking pointers with path and context sensitivity for bug detection in C programs. In *FSE'03*. 317–326.

[38] Benjamin Livshits, Aditya V Nori, Sriram K Rajamani, and Anindya Banerjee. 2009. Merlin: specification inference for explicit information flow problems. In *PLDI'09*. 75–86.

[39] Fan Long and Martin Rinard. 2016. An analysis of the search spaces for generate and validate patch generation systems. In *ICSE'16*. 702–713.

[40] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *POPL'16*. 298–312.

[41] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. 2016. UniSan: Proactive kernel memory initialization to eliminate data leakages. In *CCS'16*. 920–932.

[42] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. 2010. CETS: compiler enforced temporal safety for C. In *ISMM'10*. 31–40.

[43] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavy-weight dynamic binary instrumentation. In *PLDI'07*. 89–100.

[44] Ben Niu and Gang Tan. 2015. Per-input control-flow integrity. In *CCS'15*. 914–926.

[45] Hakjoo Oh, Hongseok Yang, and Kwangkeun Yi. 2015. Learning a strategy for adapting a program analysis via bayesian optimisation. In *OOPSLA'15*. 572–588.

[46] Mads Chr Olesen, René Rydhof Hansen, Julia L Lawall, and Nicolas Palix. 2014. Coccinelle: tool support for automated CERT C secure coding standard certification. *Science of Computer Programming* 91 (2014), 141–160.

[47] Jannik Pewny and Thorsten Holz. 2016. EvilCoder: automated bug insertion. In *ACSAC'16*. 214–225.

[48] Ganesan Ramalingam. 1994. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16, 5 (1994), 1467–1471.

[49] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. 2014. A machine-learning approach for classifying and categorizing android sources and sinks.. In *NDSS'14*.

[50] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *POPL'95*. 49–61.

[51] Johannes Schlumberger, Christopher Kruegel, and Giovanni Vigna. 2012. Jarhead: analysis and detection of malicious java applets. In *ACSAC'12*. 249–257.

[52] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: a fast address sanity checker. In *ATC'12*. 309–318.

[53] Fermin J Serna. 2012. The info leak era on software exploitation. *Black Hat USA* (2012).

[54] Lei Shang, Xinwei Xie, and Jingling Xue. 2012. On-demand dynamic summary-based points-to analysis. In *CGO'12*. 264–274.

[55] Umesh Shankar, Kunal Talwar, Jeffrey S Foster, and David Wagner. 2001. Detecting format string vulnerabilities with type qualifiers.. In *Security'01*. 201–220.

[56] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. 2016. Boomerang: demand-driven flow-and context-sensitive pointer analysis for Java. In *ECOOP'16*. 22:1–22:26.

[57] Robert E Strom and Shaula Yemini. 1986. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering (TSE)* 1 (1986), 157–171.

[58] Yulei Sui and Jingling Xue. 2016. On-Demand Strong Update Analysis Via Value-Flow Refinement. In *FSE'16*. 460–473.

[59] Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural Static Value-Flow Analysis in LLVM. https://github.com/unsw-corg/SVF. In *CC'16*. 265–266.

[60] Yulei Sui, Ding Ye, and Jingling Xue. 2012. Static memory leak detection using full-sparse value-flow analysis. In *ISSTA'12*. 254–264.

[61] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. Sok: eternal war in memory. In *SP'13*. 48–62.

[62] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *Security'14*. 941–955.

[63] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: effective taint analysis of web applications. In *PLDI'09*. 87–97.

[64] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. 2015. Practical context-sensitive CFI. In *CCS'15*. 927–940.

[65] Vladimir Vapnik. 2013. *The nature of statistical learning theory*. Springer science & business media.

[66] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically learning semantic features for defect prediction. In *ICSE'16*. 297–308.

[67] Gary Wassermann and Zhendong Su. 2008. Static detection of cross-site scripting vulnerabilities. In *ICSE'08*. 171–180.

[68] Westley Weimer and George C Necula. 2005. Mining temporal specifications for error detection. In *TACAS'05*. 461–476.

[69] Wei Xu, Daniel C DuVarney, and R Sekar. 2004. An efficient and backwards-compatible transformation to ensure memory safety of C programs. In *FSE'12*. 117–126.
[70] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. 2015. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In *CCS'15*. 414–425.
[71] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *SP'14*. 590–604.
[72] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. 2012. Generalized vulnerability extrapolation using abstract syntax trees. In *ACSAC'12*. 359–368.
[73] Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. 2015. Automatic inference of search patterns for taint-style vulnerabilities. In *SP'15*. 797–812.
[74] D. Ye, Y. Su, Y. Sui, and J. Xue. 2014. WPBOUND: Enforcing Spatial Memory Safety Efficiently at Runtime with Weakest Preconditions. In *ISSRE'14*. 88–99.
[75] Ding Ye, Yulei Sui, and Jingling Xue. 2014. Accelerating dynamic detection of uses of undefined values with static value-flow analysis. In *CGO'14*. 154–164.
[76] Sen Ye, Yulei Sui, and Jingling Xue. 2014. Region-based selective flow-sensitive pointer analysis. In *SAS'14*. 319–336.
[77] Yves Younan. 2015. FreeSentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *NDSS'15*.
[78] Yves Younan, Wouter Joosen, and Frank Piessens. 2012. Runtime countermeasures for code injection attacks against C and C++ programs. *ACM Computing Surveys (CSUR)* 44, 3 (2012), 17.
[79] Hongtao Yu, Jingling Xue, Wei Huo, Xiaobing Feng, and Zhaoqing Zhang. 2010. Level by level: making flow-and context-sensitive pointer analysis scalable for millions of lines of code. In *CGO'10*. 218–229.
[80] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dong Song, and Wei Zou. 2013. Practical control flow integrity and randomization for binary executables. In *SP'13*. 559–573.