UFO: Predictive Concurrency Use-After-Free Detection

Jeff Huang
Parasol Laboratory
Texas A&M University
jeff@cse.tamu.edu

ABSTRACT

Use-After-Free (UAF) vulnerabilities are caused by the program operating on a dangling pointer and can be exploited to compromise critical software systems. While there have been many tools to mitigate UAF vulnerabilities, UAF remains one of the most common attack vectors. UAF is particularly difficult to detect in concurrent programs, in which a UAF may only occur with rare thread schedules. In this paper, we present a novel technique, UFO, that can precisely *predict* UAFs based on a single observed execution trace with a provably higher detection capability than existing techniques with no false positives. The key technical advancement of UFO is an extended maximal thread causality model that captures the largest possible set of feasible traces that can be inferred from a given multithreaded execution trace. By formulating UAF detection as a constraint solving problem atop this model, we can explore a much larger thread scheduling space than classical happens-before based techniques. We have evaluated UFO on several real-world large complex C/C++ programs including Chromium and FireFox. UFO scales to real-world systems with hundreds of millions of events in their execution and has detected a large number of real concurrency UAFs.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Security and privacy** → **Browser security**;

KEYWORDS

UAF, Concurrency, Vulnerabilities, UFO

ACM Reference Format:
Jeff Huang. 2018. UFO: Predictive Concurrency Use-After-Free Detection. In *ICSE '18: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3180155.3180225

1 INTRODUCTION

Programs written in unsafe languages such as C/C++ are prone to memory errors. Among them, Use-After-Free (UAF) vulnerabilities are a severe threat to software security. Just a few years back [1], UAF vulnerabilities have become the most exploited memory errors in both Windows Vista and Windows 7, and many zero-day cyber attacks targeting web browsers and network servers have been launched through UAF vulnerabilities [2].

A UAF occurs when the program accesses data via a dangling pointer, a pointer which points to an invalid object, *i.e.*, the memory that stored the object has been deallocated and/or reallocated for storing another object. UAFs are particularly difficult to detect in concurrent programs, in which multiple threads may interleave when accessing memory via pointers. Due to different thread schedules, a program can execute in many different ways even with exactly the same program input. When the *use* and *free* of a certain pointer happen in different threads and are not properly synchronized, they may constitute a UAF that can rarely manifest in the testing phase, because the particular thread schedule that triggers the UAF is difficult to occur. However, the UAF has a much larger chance to happen in production runs, where there is a large user base to exercise different schedules, and worse, the schedules may be controlled by attackers [3].

Existing techniques [4–6] are very limited in detecting such *concurrency UAFs*. Most dynamic tools, *e.g.*, AddressSanitizer [5] and KASAN [7], are evidence-based, such that they can only detect a UAF when it actually happened. Other recent techniques such as DangSan [13], DangNull [2] and FreeSentry [1] are effective in preventing UAF vulnerabilities by tracking pointers at runtime and nullifying them before the pointed objects are destroyed, but they cannot defend UAF vulnerabilities that are not observed.

Web browsers are one of the hardest-hit areas of concurrency UAF vulnerabilities. Browsers are often built from components using different memory management methods. For example, in Google Chrome and Firefox, JavaScript objects are garbage-collected, XPCOM objects are reference-counted, and the layout engine uses manual management. This mixture makes it extremely difficult to reason about objects shared between codes using different memory management methods. Figure 1 shows a Priority-0 (*most urgent*) bug that affects the recent Chrome browser on Linux and OS X as well as Chrome OS. In this bug, `client` is a `MidiManagerClient` object, and it is managed by a renderer process; `send_thread` is owned by the browser process, which is independent of any renderers. It is possible that the `client` is freed when the corresponding renderer process

```
MidiManagerAlsa::DispatchSendMidiData(){
    ...
  send_thread_.message_loop()->PostTask(
      FROM_HERE,
      base::Bind(
        &MidiManagerClient::AccumulateMidiBytesSent,
        base::Unretained(client),
        data.size()));
    ...
```

Figure 1: A real-world concurrency UAF in the Chrome browser (http://crbug.com/564501).

dies, but the tasks are still in the message loop. When the task starts, it calls `MidiManagerClient::AccumulateMidiBytesSent` on the `client`, which is already freed. This bug is a typical concurrency UAF and can be exploited to escape the sandbox [14].

In this paper, we present a novel technique called UFO (*UAF Finder Optimal*) for detecting concurrency UAFs. A salient feature of UFO is that it can *predict* UAF vulnerabilities that are *unseen* from the observed execution, but guarantee to happen in an alternative run of the same program with a different schedule. Moreover, every reported UAF by UFO must be real and reproducible with a corresponding schedule to manifest it. Such an ability, which we call *sound predictability*, is unique for concurrent programs, because alternative thread schedules can be inferred from an observed execution, based on a sound thread causality model such as the classical happens-before model [19] or the more recent maximal causality model (MaxModel) [16].

UFO is underpinned by the MaxModel (discussed in more detail in Section 3). A key strength of MaxModel is that given an observed execution trace (*i.e.*, the *seed* trace), it can capture the *maximal* set of feasible traces that can be inferred from the seed trace. However, MaxModel is blind to semantic properties such as UAFs. By extending MaxModel with additional *use* and *free* events to support UAF detection, UFO is empowered with the maximal predictive power: UFO detects all possible and real concurrency UAFs based on an observed seed trace and it is impossible to precisely detect more concurrency UAFs than UFO based on the same seed trace[1]. In other words, any new UAF that is not detected by UFO but detected by another technique on the same observed execution could be a false positive.

Moreover, we introduce a novel constraint encoding for UAFs. This is challenging because of the UAF semantics and memory reuse (Section 4.2.2). By encoding both the UAF violations and the extended MaxModel as first order logical constraints over the observed events in the execution and solving the constraints with a high performance SMT solver [22], UFO detects a maximal number of real UAFs from the observed traces.

[1]To clarify, UFO may still miss real UAFs under a different seed trace, *e.g.*, one generated from a different program input.

We have implemented UFO for Pthread-based multithreaded C/C++ programs based on ThreadSanitizer [17], and evaluated it on several large real-world concurrent systems including the Chromium and the FireFox web browsers. Our experimental results show that UFO is powerful in detecting UAFs and can scale to real-world program executions with hundreds of millions of critical events. UFO detected many UAF vulnerabilities that industrial-strength tools such as AddressSanitizer [5], Dr.Memory [18] and Valgrind [21] all fail to detect. At the time of writing, UFO has detected 79 UAFs in Chromium and 105 in Firefox.

In summary, this paper makes the following contributions:

- We present UFO, a powerful predictive detection technique for concurrency UAF vulnerabilities. To our best knowledge, UFO is the first concurrency UAF detection technique with sound predictability, *i.e.*, it can precisely predict a maximal set of UAFs from a multithreaded program execution trace while the predicted UAFs are all real and can be reproduced.
- We implement and evaluate UFO on both popular benchmarks and real applications. Experiments show that UFO cannot only detect known UAF violations, but also new UAFs that cannot be detected by industrial-strength tools. Moreover, with numerous performance optimizations, UFO scales to large complex programs such as Chromium and Firefox.
- UFO is open source: https://github.com/parasol-aser/UFO.

2 MOTIVATION

In this section, we show how UFO can predict UAFs from an observed execution, why it has a higher detection capability than other UAF detectors, and why it is precise, *i.e.*, every reported UFO is real.

```
initially: x = 0
p = malloc(10); q = malloc(10);
Thread1:
1. start(t2);              Thread2:
2. lock(l)                 10. lock(l)
3. free(q);                11. *q = 0;
4. unlock(l)               12. if(x == 0)
5. lock(l)                 13.    *p = 0;
6. x = 1;                  14. unlock(l)
7. unlock(l)
8. free(p);
9. join(t2);
```

Figure 2: Motivating example.

Figure 2 shows a simple example illustrating the UAF problems in a multithreaded program. The program contains two possible UAFs: ❶ the pointer dereference at line 11 executed after *free(q)* at line 3, which is a real UAF that can occur under certain thread interleavings; ❷ a potential UAF between lines 8 and 13, which in fact can never occur.

Suppose that the program is executed following the schedule `S0` denoted by lines:

`S0: 1-10-11-12-13-14-2-3-4-5-6-7-8-9`,

the evidence-based UAF detectors [5, 7, 18, 21] cannot find the real UAF ❶, because it does not occur in the observed schedule. The classical happen-before based data-race detectors [17] can neither find this UAF, because at the time when the *free(q)* operation is executed, threads 1 and 2 are happens-before synchronized by the *unlock(l)* operation at line 14 and the *lock(l)* operation at line 2. In other words, there is a determined happen-before edge from the event at line 11 to the event at line 3. However, it is possible that thread 1 acquires the lock l first and frees the memory pointed by q at line 3 before the pointer access at line 11, following the schedule `S1`:

`S1: 1-2-3-4-10-11-12-13-14-5-6-7-8-9`,

which manifests the UAF ❶.

How about the UAF ❷ between lines 8 and 13? It should be noticed that in the example above, the read x at line 12 can read different values depending on the execution of line 6. In other words, the pointer dereference at line 13 is control-dependent on line 6. If the *free(p)* operation at line 8 is executed before line 13, then line 12 will read value 1 on x and hence line 13 will not be executed. Thus the UAF ❷ is a false positive.

UFO can detect the real UAF ❶ and at the same time precisely determine that the UAF ❷ is not possible (hence prevents from reporting any false positives). The basic idea of UFO is to infer other feasible schedules based on an observed execution trace and to detect UAFs in all the inferred schedules, in addition to the observed schedule. UFO is inspired by early work on predictive race detection [16], which proposes a MaxModel to capture a maximal set of feasible schedules from a trace. By encoding all potential pointer UAFs as additional constraints to the MaxModel, UFO is able to verify all feasible UAFs that can be inferred from an execution.

The key technical novelty of UFO is extending the MaxModel to support UAF detection, which requires both extending the original trace model and efficiently encoding UAF constraints while ensuring every detected UAF is real and no UAF is missed. Moreover, we develop two practical optimizations to improve the runtime performance of UFO.

In the rest of this paper, we first introduce the MaxModel in Section 3 and then present the design and technical details of UFO in Section 4. We present the implementation of UFO in Section 5 and evaluation results of UFO in Section 6.

3 MAXIMAL CAUSAL MODEL

Taking as input an observed execution trace of a multithreaded program, the MaxModel captures the largest set of feasible traces that can be inferred from the observed trace. This allows us to detect all potential UAF vulnerabilities captured by the MaxModel that are also *real*: for every detected UAF there exists a valid execution that manifests it (possibly via an alternative schedule).

A trace is a full sequence of events, which are operations performed by threads on concurrent objects. For example, a write event to memory location **x** with value **v** by thread **t** is denoted as **write(t,x,v)**. A full sequence means that all events on concurrent objects are included, *i.e.*, no critical event is missing. The following common types of critical events (by a thread t) are considered in the original MaxModel:

- ***read(t,x,v)/write(t,x,v)***: read/write x with value v;
- ***lock(t,l)/unlock(t,l)***: acquire/release lock l;
- ***start(t,t')***: fork a new thread t';
- ***join(t,t')***: block until thread t' terminates.
- ***begin(t)/end(t)***: the first/last event of thread t; these are extra events introduced to capture the happens-before relation between threads.

When a multithreaded program P is executed, a set of finite traces of events will be produced, called P-feasible traces. There are two important properties held by the sets of $\mathcal{P}$-feasible traces: *prefix closedness* and *local determinism*. The former says that the prefixes of a $\mathcal{P}$-feasible trace are also $\mathcal{P}$-feasible. The latter means the execution of a concurrent operation is only determined by the previous events in the same thread. More specifically,

Prefix Closedness: $\mathcal{P}$-feasible is prefix closed: if $\tau_1\tau_2 \in$ $\mathcal{P}$-feasible, then $\tau_1 \in \mathcal{P}$-feasible. Prefix closedness ensures that the events are generated in execution order, with the possibility of interleaving in-between any of them.

Local Determinism: Each event is determined only by the previous events in the same thread (and not other events of other threads) and can occur at any consistent moment after them.

In addition to the two basic axioms defined above, the sets of P-feasible traces must also obey a memory consistency model, *i.e.*, which value a read can return. Note that the event *value* is also a part of the definition. For example, if the value returned by a read is changed, it becomes a different read event, such that a conditional after the event may produce a different trace. The original MaxModel focuses on sequential consistency and has also been extended to relax memory models such as TSO and PSO [20]. In this paper, we follow the original MaxModel for sequential consistency only.

Together, MaxModel enables associating a maximal set of traces MaxModel(τ) to any consistent trace τ, which comprises precisely the traces that can be generated by any program that can generate τ. It is shown in [16] that MaxModel(τ) is both sound and maximal: any program which can generate τ can also generate all traces in MaxModel(τ), and for any trace τ' not in MaxModel(τ) there exists a program generating τ which cannot generate τ'. In contrast, traditional happens-before causal models [19] consisting of all the legal interleavings of τ are not maximal. In other words, there are feasible traces that are captured by MaxModel but not by the happen-before model.

Example: Consider the example in Figure 2 again. The trace τ_0 as shown in Figure 3 will be generated following the schedule `S0`. Note that the original MaxModel does not model pointer *use* and $free$ events. For illustration purpose,

```
Trace τ0:                   Trace τ1:
1. start(t1,t2)             1.  start(t1,t2)
    begin(t2)               2:  lock(t1,l)
10: lock(t2,l)              3:  read(t1,q,*q)
11: read(t2,q,*q)           3:  write(t1,*q,0)
11: write(t2,*q,0)          4.  unlock(t1,l)
12: read(t2,x,0)                begin(t2)
13: read(t2,p,*p)           10: lock(t2,l)
13: write(t2,*p,0)          11: read(t2,q,*q)
14: unlock(t2,l)            11: write(t2,*q,0)
    end(t2)                 12: read(t2,x,0)
2:  lock(t1,l)              13: read(t2,p,*p)
3:  read(t1,q,*q)           13: write(t2,*p,0)
3:  write(t1,*q,0)          14: unlock(t2,l)
4.  unlock(t1,l)                end(t2)
5.  lock(t1,l)              5.  lock(t1,l)
6.  write(t1,x,1)           6.  write(t1,x,1)
7.  unlock(t1,l)            7.  unlock(t1,l)
8.  read(t1,p,*p)           8.  read(t1,p,*p)
8.  write(t1,*p,0)          8.  write(t1,*p,0)
9.  join(t1,t2)             9.  join(t1,t2)
```

Figure 3: Traces for the motivating example.

we use two events (a read and a write) to model each pointer use and free. For example, $*p = 0$ at line 11 corresponds to `read(t2,q,*q)` and `write(t2,*q,0)`, in which `*q` denotes the left value of the pointer `q` (*i.e.*, the address of memory location storing the pointed value). In the next section, we will extend MaxModel to support UAF detection.

From τ_0, the trace τ_1 (which manifests the UAF ❶) can be inferred by the MaxModel. However, this trace is not allowed by the traditional happens-before model, because there is a happens-before edge from the *unlock*(t_2,l) event at line 14 to the *lock*(t_1,l) event at line 2. On the other hand, there is no way to generate a trace in MaxModel that can put the event at line 13 after the event at line 8 (which could manifest the UAF ❷), because that would violate the local determinism and memory consistency model of MaxModel. More specifically, if line 13 is after line 8, then the value of x will be set to 1 by the write event at line 6 first, and the read event at line 12 will return the value 1 (instead of 0 as that in τ_0). Because the read event at line 12 is now a different event, it is possible that the event at line 13 becomes infeasible (*e.g.*, due to a `if` branch as in the example program).

4 PREDICTIVE UAF DETECTION

The MaxModel allows us to define a maximal notion of UAFs: a trace τ has a UAF iff there is some $\tau' \in$ MaxModel(τ), which contains a read or a write to a memory region that has already been deallocated. In this section, we first extend MaxModel to support concurrency UAF detection by introducing new types of events. Then we present our algorithm for detecting UAFs by encoding both the extended MaxModel and the UAFs as first-order logical constraints.

4.1 Extended MaxModel

We introduce three *additional* types of events into the model:

- ***malloc(t, addr, size)***: allocate a new memory region from the address *addr* to $addr + size$;
- ***use(t, addr, range)***: read/write the memory region from address *addr* to $addr + range$;
- ***free(t, addr, size)***: deallocate the memory region from address *addr* to $addr + size$, such that it can be reused.

These events are relevant to the UAF detection and all their attributes can be obtained at runtime. A `malloc` event corresponds to allocating a dynamic memory on the heap, *e.g.*, creating a new object. The *addr* and *size* correspond to the return value and the calling parameter, respectively, of the `*malloc(size_t size)` function (or `calloc`, `new`). When memory allocation is done, the actual heap space allocated is one word larger than the requested memory. The extra word is used to store the size of the allocation and is later used by `free`. A `use` event is an ordinary read or write event that accesses a memory region. The *addr* and *range* correspond to the beginning address of the accessed memory and the bit-length of the accessed data. A `free` event corresponds to a memory deallocation function (`free`, `delete`, or `realloc`) and the attributes *addr* and *size* correspond to the beginning address and the size, respectively, of the deallocated memory.

4.2 The UFO Algorithm

Given an observed trace τ (with all the events in the extended MaxModel included), our basic idea for detecting UAFs is to generate MaxModel(τ) and the UAF conditions among the events in τ, both are encoded as constraints. By solving the conjuncted constraints with an SMT solver, we can determine if there exists any trace $\tau' \in$ MaxModel(τ), in which a *free* event is executed before a *use* of the same memory region, and the memory address of the use is originated from the same *malloc* corresponding to the *free*.

We introduce an order variable O_i for each event e_i in τ, and construct a formula Φ over these variables, in which $O_i < O_j$ means that the event e_i happens before the event e_j. Φ is a conjunction of two sub-formulas: $\Phi = \Phi_{max} \wedge \Phi_{uaf}$, where Φ_{max} encodes the MaxModel and Φ_{uaf} encodes each potential UAF. In the rest of this section, we present Φ_{max} and Φ_{uaf} in detail.

4.2.1 MaxModel Constraints. Φ_{max} is constructed by a conjunction of two sub-formulas: $\Phi_{sync} \wedge \Phi_{rw}$, where Φ_{sync} denotes the inter-thread order constraints determined by synchronization events, and Φ_{rw} the data-validity constraints over read and write events. Φ_{sync} can be further decomposed as $\Phi_{mhb} \wedge \Phi_{lock}$, the conjunction of the must-happen-before constraints Φ_{mhb} and the lock-mutual-exclusion constraints Φ_{lock}.

Must-happen-before constraints (Φ_{mhb}). The must-happen-before (MHB) constraints reflect a *subset* of the classical happens-before relation, ensuring a minimal set of ordering relations that events in any feasible interleaving must obey. Specifically, MHB requires that (1) the total order of the events in each thread is always the same; (2) a *begin* event can happen only after the thread is started by another thread; (3) a *join* event can happen only after the *end* event of the joined thread. Clearly MHB yields a partial order over the events of τ which must be respected by any trace in MaxModel(τ). We denote MHB by $\prec$, which will be used later. We can specify $\prec$ easily as constraints Φ_{mhb} over the O variables: we start with $\Phi_{mhb} \equiv true$ and conjunct it with a constraint $O_{e_1} < O_{e_2}$ whenever e_1 and e_2 are events by the same thread and e_1 occurs before e_2, or when e_1 is an event of the form $start(t, t')$ and e_2 of the form $begin(t')$, etc.

Lock-mutual-exclusion constraints (Φ_{lock}). The locking semantics requires that any two code regions protected by the same lock are mutually exclusive, *i.e.*, they should not interleave. Φ_{lock} captures the ordering constraints over *lock* and *unlock* events. For each lock l, we extract the set S_l of all the corresponding pairs, (e_a, e_b), of *lock*/*unlock* events on l, following the program order locking semantics: the *unlock* is paired with the most recent *lock* on the same lock by the same thread. Then we conjunct $\Phi_{lock} \equiv true$ with the formula

$$\bigwedge_{(e_a,e_b),(e_c,e_d)\in S_l} (O_{e_b} < O_{e_c} \vee O_{e_d} < O_{e_a})$$

Data-validity constraints (Φ_{rw}). The data-validity constraints ensure that every event in the *considered* trace is feasible. Note that in constructing MCM for an input trace τ, the considered trace does not necessarily contain all the events in τ but may contain a subset of them, so that all the trace prefixes corresponding to partial executions of the program are considered as well. For an event to be feasible, all the events that must-happen-before it should also be feasible. Moreover, every read event that must-happen-before it should read the *same value* as that in the input trace; otherwise the event might become infeasible due to a different value read by an event that it depends on. Each read, however, may read a value written by any write, as long as all the other constraints are satisfied.

Let $\prec_e$ denote the set of events that must-happen-before an event e. Consider a read event r in $\prec_e$, $read(t, x, v)$, we let W^x be the set of $write(\_, x, \_)$ events in τ ('_' denotes any value), and W^x_v the set of $write(\_, x, v)$ events in τ. The data-validity constraint of an event e, $\Phi_{rw}(e)$, is defined as $\bigwedge_{r\in\prec_e} \Phi_{value}(r)$, where $\Phi_{value}(r) \equiv$

$$\bigvee_{w\in W^x_v} (\Phi_{rw}(w) \wedge O_w < O_r \bigwedge_{w\neq w'\in W^x} (O_{w'} < O_w \vee O_r < O_{w'}))$$

The constraint $\Phi_{rw}(e)$ enforces that every read that must-happen-before e should read the same value as that in the input trace. The constraint $\Phi_{value}(r, v)$ enforces the read event $r = read(t, x, v)$ to read the value v on x (written by any $write$ event $w = write(\_, x, v)$ in W^r_v), subject to the condition that the order of w is smaller than that of r and there is no interfering $write(\_, x, \_)$ in between. In addition, w itself must be feasible, which is ensured by $\Phi_{rw}(w)$.

Since the MaxModel models all the partial traces as well, the data-validity constraint Φ_{rw} is thus satisfiable if any event in the input trace τ is feasible, written as a disjunction of the feasibility constraints of all events in τ:

$$\Phi_{rw} \equiv \bigvee_{e\in\tau} \Phi_{rw}(e)$$

It is worth noting that each solution of the order variables to Φ_{max} corresponds to the schedule of a trace in MaxModel(τ). The size of MaxModel(τ) may be huge as the number of unique solutions to Φ can be exponential. In practice, however, we do not need to directly solve Φ_{max} to produce all the schedules in MaxModel(τ). For example, when used for checking UAFs, it suffices to find one schedule that satisfies the UAF condition.

4.2.2 UAF Constraints. In large programs, memory allocation and deallocation requests from function call such as `malloc` and `free` can be very frequent because the application memory is heavily reused. It is common that the memory region returned for one malloc request overlaps with a previous *malloc* request (which has been freed). When a *use* event accesses a memory region that overlaps with another memory region that has been *freed* before, it does not necessarily mean a UAF, because the memory region may have been reused. To illustrate this problem, consider the example in Figure 4:

```
                 Thread1:
                 1. p1 = malloc(100)
                   //address:[100, 200]
                 2. free(p1)
                 3. p2 = malloc(100)
                   //address:[100, 200]
                 ...

                 4. *p4 = 0
Thread2:                             Thread3:
...
                                     ...
5. free(p2)
6. p3 = malloc(150)
  //address:[150, 300]               7. free(p3)
```

Figure 4: A memory access event matched with multiple *malloc* and *free* pairs.

There are three *malloc* and *free* pairs: `pair1` with *malloc* at line 1 and *free* at line 2, pointed by `p1`; `pair2` with *malloc* at line 3 and *free* at line 5, pointed by `p2`; `pair3` with *malloc* at line 6 and *free* at line 7, pointed by `p3`. The corresponding memory regions are 100–200, 100–200, and 150–300. The write event at line 4 writes to the address *150*, and it happened in parallel with `pair2` and `pair3`. Because it is unclear from the generated event trace where p4 is derived from, it is hard to determine whether the write should be matched with `pair1`, `pair2`, or `pair3`.

To address this problem, we propose three solutions: 1) no memory reusing, 2) pointer origin tracking, and 3) purely constraint solving. The first solution is simple: do not really perform the free. We modify the memory allocator such that upon a free, we only record the free operation in the trace but do not free the actual memory. This works well for test runs with small inputs as long as the memory is not exhausted, but may not work for production with long running executions.

The idea of origin tracking is to associate each *use* and *free* event with a corresponding *malloc* which the used memory address propagates from. This solution produces simple constraints because each *use* can be match with a unique *free* according to their associated *malloc* event. The downside is that it requires data-flow tracking on pointers at runtime, which incurs extra performance overhead.

The third solution does not require pointer tracking, but generates more constraints. The basic idea is that when there are multiple malloc-free pairs that a *use* may match with, although it is unknown which pair the *use* should be matched with, there must be a UAF if the *use can* be matched with more than one pair. The reason is that a safe *use* can be matched with one and only one malloc-free pair. Otherwise, it means that the memory address of the *use* originated from one malloc is used to access a memory region allocated by another malloc.

We next present these two solutions in more detail.

Pointer Origin Tracking. This approach tracks the origin of the pointer address in the *use* event, *i.e.*, which *malloc* event it corresponds to. Similarly, for each *free* event, we track its corresponding *malloc* event, which allocates the memory region that the *free* event deallocates. The difference is that a *malloc* event is expected to have a unique corresponding *free* event. Otherwise, if there is more than one, the program has a double-free error, and if there is none, the program has a memory leak. Since we focus on detecting UAFs only, we assume there is a one-to-one mapping from *malloc* to *free* events.

We hence perform an online data-flow analysis on pointer variables to record the *malloc* event of the pointer address that is used by each *use* event and each *free* event. More specifically, for each `p = malloc(size)` operation, we call the corresponding *malloc* event as the origin of the pointer `p`. If the value of `p` flows to another pointer `p'`, then the *malloc* event of `p` is also the origin of `p'`. When `p'` is used in a *use* or *free* event, we can connect the *use* or *free* event with the origin, *i.e.*, the *malloc*.

After matching each *use* and *free* with the *malloc* event, we can encode the UAF constraint as follows: for each *free* event, e_{free}, and for each *use* event, e_{use}, that has the same origin as e_{free}, we add the constraint $\Phi_{uaf} = e_{use} > e_{free}$. If $\Phi_{uaf} \wedge \Phi_{max}$ is satisfiable, it means that there exists a schedule of the program (which produced τ) in which e_{use} happens after e_{free}. In other words, the *use* of a pointer happens after the *free* of the memory pointed by the pointer.

Purely Constraint Construction. Consider a *use* event e_{use} and suppose it has three potentially matching malloc-free pairs: P1(e_{m1},e_{f1}), P2(e_{m2},e_{f2}) and P3(e_{m3},e_{f3}) . Ideally, we would construct three constraints $O_{m1} < O_{use} < O_{f1}$, $O_{m2} < O_{use} < O_{f2}$ and $O_{m3} < O_{use} < O_{f3}$, and count the number of solutions. If at least two of these three constraints can be satisfied (conjuncted with the constraints generated from MaxModel), then e_{use} with one of the two *free* events (whose constraints are satisfied) constitutes a real UAF.

However, solving such constraints requires computing the number of solutions for individual constraint clauses, which is not well supported by existing high performance SMT solvers such as Z3 [22]. We hence develop a sub-optimal solution for this problem, but generates simple constraints. It is sub-optimal in that it does not encode all possible UAFs, but only a subset of them. The key idea is that all malloc-free pairs with overlapping memory regions must be globally ordered, *i.e.*, all these *malloc* and *free* events must be synchronized. If a *use* can happen outside of all its potentially matching malloc-free pairs, then it must be a UAF. In other words, suppose the three malloc-free pairs P1-P2-P3 are globally ordered as e_{m1}-e_{f1}-e_{m2}-e_{f2}-e_{m3}-e_{f3} in the observed trace. Then if in a certain schedule the *use* event e_{use} can be executed in the gap between any *free* and its next *malloc*, then the e_{use} must be involved in a UAF. The constructed constraints can be written as $(O_{f1} < O_{use} < O_{m2}) \vee (O_{f2} < O_{use} < O_{m3}) \vee (O_{f3} < O_{use})$.

Φ_{mhb}:	$O_1 < O_2 < \ldots < O_9$ $O_{10} < O_{11} < \ldots < O_{14}$ $O_1 < O_{10} \wedge O_{14} < O_9$
Φ_{lock}:	$O_4 < O_{10} \vee O_{14} < O_2$ $\vee\ (O_4 < O_{10} \wedge O_{14} < O_7)$
Φ_{rw}:	$O_{12} < O_6$
Φ_{uaf1}:	$O_3 < O_{11}$
Φ_{uaf2}:	$O_8 < O_{13}$

Figure 5: The encoded constraints for the motivating example in Figure 2.

4.3 Example

Figure 5 shows the constructed constraints for the two possible UAFs ❶ and ❷ in our motivation example in Figure 2. For ❶, the UAF constraint is written as $O_3 < O_{11}$, because the *use* event at line 11 and the *free* event at line 3 have the same origin, which is the *malloc* event corresponding to the statement `q = malloc(10)`. Similarly, the UAF constraint for ❷ is written as $O_8 < O_{13}$.

Figure 5 also shows the constructed constraints for Φ_{max}. For example, $O_1 < O_2$ is included in Φ_{mhb} because the event at line 1 should always happen before the event at line 2, and $O_1 < O_{10}$ because the first event of Thread2 should happen after its starting event by Thread1 at line 1. Φ_{mhb} is written as $O_4 < O_{10} \vee O_{14} < O_2 \vee (O_4 < O_{10} \wedge O_{14} < O_7)$, because the three lock regions protected by lock/unlock event pairs at lines (2,4), (5,7) and (10,14) cannot overlap. Φ_{rm} is written as $O_{12} < O_6$, because the read event on x at line 12 returns the value 1 in the observed trace τ, and to ensure the consistency of this read event (and the feasibility of the other events following it), the write event at line 6 which writes 1 to x must happen before it.

Next, we employ an SMT solver to solve each formula $\Phi_{max} \wedge \Phi_{uaf1}$ and $\Phi_{max} \wedge \Phi_{uaf2}$. For ❶, the solver returns a solution, in which $O_1 = 1, O_2 = 2, O_3 = 3, O_4 = 4, O_5 = 10, O_6 = 11, O_7 = 12, O_8 = 13, O_9 = 14, O_{10} = 5, O_{11} = 6, O_{12} = 7, O_{13} = 8, O_{14} = 9$, corresponding to the schedule: 1-2-3-4-**10-11-12-13-14**-5-6-7-8-9.

Hence, we find a real UAF vulnerability: *free(q)* (line 3) can be executed before **q = 0* (line 11) following the schedule above. For ❷, however, the formula is not satisfiable, so it is not a real UAF.

5 UFO IMPLEMENTATION

We have implemented UFO based on ThreadSanitizer [17] and the Z3 SMT solver [22] for Pthread-based multithreaded C/C++ programs, with around 12K additional code. UFO consists of two main phases: online program tracing and offline predictive trace analysis.

5.1 Trace Generation

We modified ThreadSanitizer to generate the trace containing events defined in our extended MaxModel. To generate memory access events, we first instrument the target program with an LLVM pass, and record each load or store instructions on a non-local variable as a tuple $\langle tid, PC, addr, isWrite, value, pc \rangle$. For implicit events through function calls, *e.g.*, `free()`, `delete()`, `memcpy()` and `strcmp()`, we wrap these functions during linking. When these functions are invoked at runtime, we intercept the call and generate an event from the function type and parameters. For example, *memcpy(a,b,100)* by thread *t* will generate two events: `range_read(t,b,100)` and `range_write(t,a,100)`.

To map the events back to the source code when reporting UAF violations, besides the memory access and thread interaction information, we store in the events the PC, which is the code address of the load/store instruction or the called function, as well as the address and offset of each module loaded by the target program.

All events are encoded in a compact format, and are written to files for offline predictive analysis. At runtime, a huge number of events will be generated in a very fast way (around twenty million events per second), and the normal execution of an application may be blocked for a relatively long time when flushing events to disk. To reduce the runtime perturbation from the tracing library, we allocate a thread-local buffer for each live thread, such that events are buffered first. When a thread-local buffer is full, it is flushed to a global buffer queue, such that the application thread can continue executing without being blocked. A worker thread running the background is invoked once a local buffer is flushed, and compresses the data in the queue using Snappy [23] and flushes it to disk asynchronously.

For multiple-process programs (*e.g.*, `Chromium` and `Firefox`), we store the trace of each process in separate directories and invoke the analyzer for different processes in parallel to improve the performance. In addition, because UAFs cannot happen in a single thread, we do not start tracing until the first child thread has been forked by the main thread.

5.2 Trace Analysis

In the trace analysis phase, we build the MaxModel constraints from the thread local traces. Then, we search for *malloc* and *free* pairs and the conflicting memory accesses. In our implementation, we only consider concurrency UAFs, *i.e.*, the *use* and *free* events are from different threads. For each pair of *use* and *free* events that have overlapping memory region, we first run a fast happens-before algorithm to identify candidate UAFs. If the *use* and *free* do not happen before each other by inter-thread synchronization, we consider them as a candidate UAF. For each candidate UAF, we proceed to build the constraints $\Phi_{uaf} \wedge \Phi_{max}$ and invoke Z3 to solve them. The default constraint solving time is set to two minutes for each invocation. If the solver returns a solution, we report the UAF as well as the schedule that can manifest it.

To report the detected UAFs, we first retrieve the PCs of the call stacks from the trace and calculate the offset address of each PC from the stored module information. We then invoke a symbolizer, *e.g.*, `llvm-symbolizer` or `atos` to map the UAF to the source code.

For long running programs, the size of the generated trace could be very large, and it is difficult to even load and process the whole trace. In addition, long traces lead to a huge number of constraints, which are difficult to solve in a reasonable time. Previous predictive techniques [16] propose a windowing strategy, which splits the whole trace into pieces and analyzes only a limited number of events each time. In UFO, we design an adaptive windowing algorithm that loads events based on the number of threads alive in that specific window: the fewer threads, the more events we load. For example, in a period of execution, if there are more threads interacting with each other, the number of events loaded from each thread is decreased. One limitation of splitting the trace is that if the conflicting *use* and *free* are loaded in different windows, some real UAFs may be missed.

6 EVALUATION

In this section, we aim to answer the following two questions:

(1) *UAF detection effectiveness* - Can UFO detect real UAFs in real-world applications? If so, how effective is it?
(2) *Performance* - What is the runtime overhead of UFO? What is the trace size and how efficient is the offline analyzer? Can it scale to read-world large programs?

Table 1: Experimental Results on Pbzip2 and HTTrack.

Program	Trace								#UAF		Time
	#threads	#events	#read	#write	#sync	#malloc	#use	#free	Candidate	Real	
Pbzip2	4	1590	957	249	263	77	1206	44	39	23	14s
HTTrack	2	27.7M	23.6M	4M	2696	38.9K	27.6M	32.8K	33	4	9s

6.1 Methodology

We evaluated UFO on four real-world open source programs, including two popular benchmarks with known UAFs (`Pbzip2`) or concurrency bugs (`HTTrack`) collected from [15], and two popular web browsers: `Chromium` and `Firefox`. For `Chromium` and `Firefox`, we conducted a set of real user interactions and generated a collection of traces. We set the timeout period for analyzing each trace to two hours and the adaptive window size to 100K.

For performance evaluation, we collected three popular web benchmarks Octane [26], SunSpider [27], and Dromaeo [28], and measured the runtime overhead and trace file size by running the UFO instrumented browsers with these benchmarks.

For encoding the UAF constraints, we evaluated all the three solutions (recall Section 4.2.2). We found that the simple "no memory reusing" solution works well for our UAF detection tests, but it does not work for the performance tests (because the memory can be quickly exhausted). Both the Pointer Origin Tracking and the Purely Constraint Construction solutions work for performance tests, but Purely Constraint Construction works better: it detects the same number of unique UAFs as that by the Pointer Origin Tracking solution. However, it is more efficient at runtime because tracking the dynamic flow of pointers is expensive, *e.g.*, it incurs around 4X additional runtime overhead. The UAF detection data reported in Section 6.2 corresponds to the simple solution, and the performance data in Section 6.3 corresponds to the Purely Constraint Construction solution.

All experiments were conducted on an 8-core 2.60GHz Intel i7 machine with 24GB memory running Ubuntu 14.04.

6.2 UAF Detection Effectiveness

6.2.1 Pbzip2 and HTTrack. Table 1 summarizes the results on `Pbzip2` and `HTTrack`. Overall, UFO detected 23 real UAFs in `Pbzip2` and 4 in `HTTrack`. We note that every report UAF by UFO has a unique signature (we remove those dynamic UAFs from the same program source locations). We also tested these two benchmarks with AddressSanitizer [5], Dr.Memory [18] and Valgrind [21]. None of the other tools reported any UAF violations in normal executions of these programs, because no real UAF actually happened in the observed trace.

`Pbzip2` is a frequently studied parallel compression tool containing known UAFs [8]. To decompress a file, it first loads the file into memory and stores the chunks in a FIFO list, and then starts several new threads to operate on the FIFO list. However, due to the lack of proper synchronization, it is possible that the FIFO list is deleted by the main thread before the other parallel tasks finish. The observed trace contains four thread with a total of 1590 events, including 957 reads (or range reads), 249 writes (or range writes), and 263 synchronization events (thread fork, join, mutex lock, unlock, etc.). There are 77 *malloc*, 44 *free*, and 1206 *use* events. In total, UFO detected 39 candidate UAFs (*i.e.*, pairs of *use* and *free* events accessing overlapping memory regions, but are from different threads). Among them, 23 are verified to be real (*i.e.*, the solver returns solutions to the generated constraints). For example, UFO detected six real UAFs in `Pbzip2` for each file operation: there are six accesses to the FIFO list in the `void* consumer(void*)` function, which are not properly synchronized with the *free* call from the main thread. The total offline analysis time is 14s.

`HTTrack` is an offline browser utility that can download a web site to local directory. It has a known order violation concurrency bug [9] between creation of a global data structure `global_opt` and the use of it, which may result in null-pointer dereference and hence program crashes. However, it was unknown if this program has UAF vulnerabilities or not. UFO confirmed that `HTTrack` does have UAF vulnerabilities. The observed trace contains two threads performing more than 27M events in total, including 23.6M reads, 4M writes, 2696 synchronization events, 39K malloc, 33 *free*, and 27.6M *use* events. In total, UFO detected 33 candidate UAFs, among which 4 UAFs are verified to be real. The total offline analysis time is 9s.

6.2.2 Chromium and Firefox. We evaluated UFO on `Chromium` and `Firefox` both built from trunk as of July 2017. To generate traces that contain risky *use* and *free* events, we manually checked their issue trackers, and analyzed the current unresolved issues. We found that the PDF module and the printing facilities in Chromium are more prone to bugs than other components. Several unresolved issues are related to actions related to PDF and printing. In our experiments, we conducted a number of browser actions related to PDF and printing: open five PDF files, including one that triggered a UAF vulnerability in MuPDF [29], print these PDF files, cancel printing jobs, close files before printing finishes, etc. In addition, we opened several rich web pages such as *Youtube* and *Facebook*, and performed actions related to multimedia and printing.

In total, `Chromium` forked 16 processes with 334 threads together, `Firefox` forked 15 processes with over 460 threads. UFO took 11 hours to analyze the 162GB compressed traces and detected 184 UAFs (79 in `Chromium` and 105 in `Firefox`) in four processes. The results are reported in Table 2.

Table 2: Experimental Results on Chromium and Firefox.

Program	Pid	Trace								#UAF		Time
		#threads	#events	#read	#write	#sync	#malloc	#use	#free	Candidate	Real	
Chromium	#1	16	222M	140M	80M	446K	655K	220M	545K	113	65	469s
	#2	15	268M	170M	90M	340K	650K	266M	530K	40	14	108s
Firefox	#1	88	520M	370M	148M	740K	965K	518M	660K	629	70	1323s
	#2	41	145M	110M	34M	180K	225K	144M	147K	148	35	83s

Table 3: Performance evaluation on Chromium and Firefox.

Benchmark	Chromium				Firefox			
	Original	UFO			Original	UFO		
	Score	score	#event	trace size	Score	score	#event	trace size
SunSpider	205	535	3.1G	11 GB	225	605	10.4G	47.4GB
Octane	36215	23389	1.9G	8.5GB	31828	575	10.8G	48.0GB
Dromaeo JS	1640	768	7.4G	22.5GB	1382	31	10.1G	30.7GB
Dromaeo DOM	2451	1512	5.1G	16.8GB	3460	67	5.8G	22.7GB

For instance, for a Chromium process (pid #1), the trace contains 16 threads performing 220M events in total, including 140M reads, 80M writes, 450K synchronization events, 660K *malloc*, 550K *free*, and 220M *use* events. UFO detected 113 candidate UAFs, among which 65 UAFs are verified to be real by Z3. The total offline analysis time is 469s. For the Firefox process (pid #2), the trace contains 88 threads performing around 520M events in total, including 370M reads, 148M writes, 740K synchronization events, around 1M malloc, 660K *free*, and 518M *use* events. In 1323s, UFO identified 629 candidate UAFs and verified that 70 UAFs are real. Many UAFs share the same free site, but with different use sites.

We have also manually inspected these UAFs and reported them to the developers. However, due to complexity of these two projects, developers are still in the process of confirming them. Many UAFs found in Chromium are under the base/message_loop package where a pointer freed by a child thread (we believe it is a render thread) in the incoming_task_queue.cc can be used by the main browser thread. Most UAFs in Firefox were found in the JavaScript Engine. For example, a pointer freed in js::LifoAlloc::freeAll() can be referenced by several methods in js::jit::BacktrackingAllocator. We have filed bug reports in the Chromium bug database [10] and Mozilla Bugzilla [11]. More UAFs will be disclosed at our open source repository [12] once they are confirmed and fixed.

6.3 UFO Performance

Table 3 reports the runtime performance results of UFO on the two browsers. For SunSpider and Octane, the scores are the time in ms to perform the benchmark computation. The lower, the better. For Dromaeo, the scores are the number of runs per second. The higher, the better. The average runtime overhead of UFO is 11X without pointer origin tracking, and

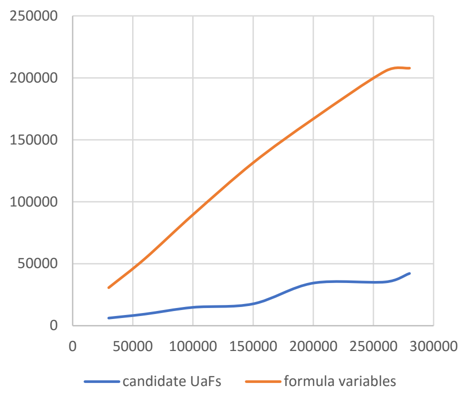

Figure 6: Tradeoff between offline analysis performance and the detection ability by the window size.

it does not impose significant slowdown for normal interaction between application and users, thus indicating that UFO can be used for in-house testing for applications as large as desktop web browsers. For the JavaScript engine benchmarks, UFO introduced 207% overhead on average. This is because JavaScript is compiled by the JIT-compiler, and is executed without UFO instrumentation. For HTML rendering benchmarks, the average slowdown of UFO is 26X.

The runtime performance of UFO is determined by the amount of traced events, and the offline analysis time of UFO is highly influenced by the window size. Figure 6 shows the correlation between window size and the number of constraint variables and the number of candidate UAFs for analyzing a trace in Chromium. As the window size become larger, the number of constraint variables grows linearly and the number of candidate UAFs also increases. However, there exists a tradeoff between UFO offline analysis performance and the detection ability, which can be tuned by the window size.

7 RELATED WORK

Researchers have proposed several types of approaches for detecting UAF vulnerabilities, including dynamic detectors [4, 5], static detectors [30, 31], as well as hybrid techniques [6]. For example, Undangle [6] is a hybrid detection tool that combines runtime tracing and offline analysis. In general, static detectors are imprecise, while existing dynamic techniques suffer from limited code coverage or inability to handle concurrent programs. UFO is distinguished by the ability to precisely and maximally *predict* UAFs that are unseen from the observed multithreaded dynamic execution.

GUEB [30] is a static tool that detects heap UAF vulnerabilities on binary code. The key idea is that for a program path with UAF vulnerabilities, three events will happen orderly: allocating heap memory, free memory, and accessing the heap memory. GUEB tracks heap operations and address transfers (pointer assignments), and discovers which program location allocates or frees which heap memory. Then it analyzes the resulting graph to identify UAF vulnerabilities. GUEB provides better coverage of UAF detection than dynamic tools. However, due to the imprecision of points-to analysis, GUEB is neither sound or precise. Also, the path analysis and value set analysis performed by GUEB are expensive and difficult to scale to large programs.

UAFChecker [31] is a static UAF detector. It applies the inter-procedure analysis to make the result more accurate. It uses function inlining and function summary to perform the inter-procedure analysis. UAFChecker builds a finite state machine to model the UAF vulnerabilities. This step may report many candidate UAF vulnerabilities. UAFChecker then uses symbolic execution to check the satisfiability of path constraints of the candidate vulnerabilities, and eliminate the false positives.

DOUBLETAKE [4] detects three types of memory errors: UAF, heap buffer overflow and memory leak. At runtime, the program execution is divided into epochs, each ends with an irrevocable system call. At the beginning of an epoch, DOUBLETAKE checks program state (registers and all writable memory). At the end of an epoch, DOUBLETAKE checks program state again to see if any memory errors have occurred. If so, it rollbacks current epoch and re-execute the epoch with additional instrumentation to pinpoint the exact locations of the error. DOUBLETAKE replaces the system memory allocator with a deterministic allocator: given the same sequence of malloc and free requests, it must provide the same addresses for allocated objects. During re-execution, DOUBLETAKE utilizes the hardware watchpoints to pause the program when the memory error happens.

AddressSanitizer [5] is a popular dynamic memory error detector that detects a wide range of runtime memory errors including UAFs and memory leaks. To find UAF vulnerabilities, AddressSanitizer uses shadow memory to record whether each byte of application is accessible, and instruments the program to check the shadow memory on each store or load. To compact the shadow memory, AddressSanitizer encodes the state (accessibility) of every 8-byte sequence of heap memory into one byte, and uses a direct mapping to translate an application address to its shadow memory. The average overhead is 2.3X. AddressSanitizer may miss UAF if a larger memory (*e.g.*, 1 MB) has been allocated and deallocated between the *free* and the following *use*.

Similar to AddressSanitizer, KASAN [7] is a dynamic memory error detector for finding use-after-free and out-of-bounds bugs in the Linux kernel.

Many UAF vulnerabilities are explored by crafting bogus virtual table of the C++ code. T-VIP [32] tackles such problems by instrumenting binary code and adding runtime policy enforcements to prevent UAFs. Ironclad C++ [33] tries to address this problem by augmenting existing C++ programs with additional pointer library in a semi-automatic way.

DANGULL [2] and FreeSentry [1] try to prevent UAFs from happening by tracking object pointers at runtime, and nullifying them when the object is destroyed. DangSan [13] is a more recent system that significantly improves the runtime performance of UAF detection by optimizing the detection workloads. It scales to programs with large numbers of pointer writes and many concurrent threads. However, although it can efficiently detect UAFs in concurrent programs, it does not have the predictive power.

8 CONCLUSION

Concurrency Use-After-Free (UAF) vulnerabilities are a rising threat to software security. We have presented UFO, a new technique that can effectively and precisely predict concurrency UAFs based on a single execution trace, even though the UAFs do not happen in the observed execution. The foundation of UFO is the maximal thread causality model (MaxModel), which we have extended to encode UAF vulnerabilities as first-order logical constraints. By formulating UAF detection as a constraint-solving problem and leveraging the power of the MaxModel, UFO enables maximally verifying all real UAFs that can be inferred from an execution. Experiments on real-world applications demonstrate that UFO can detect new UAF vulnerabilities that cannot be found by existing industrial-strength UAF detection tools. Moreover, UFO scales to large complex applications such as Chromium and Firefox and has detected many concurrency UAFs.

9 ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their helpful feedback on earlier versions of this paper. This work was supported by a Google Faculty Research Award and NSF awards CCF-1552935 and CNS-1617985.

REFERENCES

[1] Yves Younan. Freesentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *NDSS*, 2015.
[2] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *NDSS*, 2015.
[3] Lei Xu, Jeff Huang, Sungmin Hong, Jialong Zhang, and Guofei Gu. Attacking the Brain: Races in the SDN Control Plane. In *USENIX SECURITY*, 2017.

[4] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Doubletake: Fast and precise error detection via evidence-based dynamic analysis. In *ICSE*, 2016.

[5] AddressSanitizer. https://github.com/google/sanitizers/wiki/AddressSanitizer.

[6] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *ISSTA*, 2012.

[7] The Kernel Address Sanitizer. https://www.kernel.org/doc/html/latest/dev-tools/kasan.html.

[8] Concurrency bugs in pbzip2. https://github.com/jieyu/concurrency-bugs/blob/master/pbzip2-0.9.4/.

[9] Concurrency bugs in HTTrack. https://github.com/jieyu/concurrency-bugs/tree/master/httrack-3.43.9/.

[10] UAF bug report in Chromium. https://bugs.chromium.org/p/chromium/issues/detail?id=759205.

[11] UAF bug report in Firefox. https://bugzilla.mozilla.org/show_bug.cgi?id=1392831.

[12] UFO open source git repository. https://github.com/parasol-aser/UFO

[13] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. Dangsan: Scalable use-after-free detection. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 405–419, 2017.

[14] Chromium use-after-free bug. http://crbug.com/564501.

[15] Jie Yu, and Satish Narayanasamy. A Case for an Interleaving Constrained Shared-memory Multi-processor. In *International Symposium on Computer Architecture*, pages 325-336, 2009.

[16] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. Maximal sound predictive race detection with control flow abstraction. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 337–348, 2014.

[17] ThreadSanitizer. http://clang.llvm.org/docs/ThreadSanitizer.html.

[18] Derek Bruening and Qin Zhao. Practical memory checking with dr. memory. In *CGO*, 2011.

[19] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[20] Shiyou Huang and Jeff Huang. Maximal causality reduction for TSO and PSO. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 447–461, 2016.

[21] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 89–100, 2007.

[22] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pages = 337–340, 2008.

[23] Google Snappy. https://google.github.io/snappy.

[24] Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. Sound predictive race detection in polynomial time. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 387–400, 2012.

[25] The PARSEC Benchmarks. http://parsec.cs.princeton.edu/.

[26] Octane Benchmark Suite. https://developers.google.com/octane.

[27] SunSpider JavaScript Benchmark. https://webkit.org/perf/sunspider/sunspider.html.

[28] Dromaeo JavaScript Performance Test Suite. http://dromaeo.com.

[29] Cve-2016-6265: Mupdf library use after free. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-6265.

[30] Josselin Feist, Laurent Mounier, and Marie-Laure Potet. Statically detecting use after free on binary code. *J. Computer Virology and Hacking Techniques*, 10(3):211–217, 2014.

[31] Jiayi Ye, Chao Zhang, and Xinhui Han. Poster: Uafchecker: Scalable static detection of use-after-free vulnerabilities. In *CCS*, 2014.

[32] Robert Gawlik and Thorsten Holz. Towards automated integrity protection of c++ virtual function tables in binary programs. In *ACSAC*, 2014.

[33] Christian DeLozier, Richard Eisenberg, Santosh Nagarakatte, Peter-Michael Osera, Milo M.K. Martin, and Steve Zdancewic. Ironclad C++: A library-augmented type-safe subset of C++. In *OOPSLA*, 2013.