Precise and Scalable Detection of Use-after-Compacting-Garbage-Collection Bugs

HyungSeok Han, Andrew Wesie, and Brian Pak, *Theori Inc.*

https://www.usenix.org/conference/usenixsecurity21/presentation/han-hyungseok

This paper is included in the Proceedings of the
30th USENIX Security Symposium.

August 11–13, 2021

978-1-939133-24-3

Precise and Scalable Detection of Use-after-Compacting-Garbage-Collection Bugs

HyungSeok Han
Theori Inc.

Andrew Wesie
Theori Inc.

Brian Pak
Theori Inc.

Abstract

Compacting garbage collection (*compact-gc*) is a method that improves memory utilization and reduces memory fragmentation by rearranging live objects and updating their references using an address table. A critical *use-after-free* bug may exist if an object reference that is not registered in the address table is used after *compact-gc*, as the live object may be moved but the reference will not be updated after *compact-gc*. We refer to this as a use-after-compact-gc (*use-after-cgc*) bug. Prior tools have attempted to statically detect these bugs with target-specific heuristics. However, due to their path-insensitive analysis and imprecise target-specific heuristics, they have high false-positives and false-negatives.

In this paper, we present a precise and scalable static analyzer, named CGSan, for finding *use-after-cgc* bugs. CGSan detects *use-after-cgc* bug candidates by intra-procedural static symbolic taint analysis and checks their feasibility by under-constrained directed symbolic execution. To mitigate the incompleteness of intra-procedural analysis, we employ a type-based taint policy. For scalability, we propose using directed inter-procedural control-flow graphs, which reduce search spaces by excluding paths irrelevant to checking feasibility, and directed scheduling, which prioritizes paths to quickly check feasibility. We evaluated CGSan on Google V8 and Mozilla SpiderMonkey, and we found 13 unique *use-after-cgc* bugs with only 2 false-positives while two prior tools missed 10 bugs and had 34 false-positives in total.

1 Introduction

Garbage collection [30] automatically finds and reclaims dead memory objects that are no longer used in the program execution. This feature makes garbage collection an essential part of modern memory management. Virtual machines and interpreters, such as Microsoft Common Language Runtime and JavaScript (JS) engines, apply garbage collection to determine when memory objects should be freed at runtime because the systems are too complex to specify where memory objects should be disposed in the source code.

```
void InterpretedFrame::Summarize(...) const {
  ...
  // define an unrooted pointer, `code`.
  AbstractCode code = AbstractCode::cast(GetBytecodeArray());
  // `GetParameters` triggers a GC.
  Handle<FixedArray> params = GetParameters();
  // the moved `code` is used as a function argument.
  FrameSummary::JavaScriptFrameSummary summary(
      isolate(), receiver(), function(), code,
      GetBytecodeOffset(), IsConstructor(), *params);
  ...
}
```

Figure 1: A code snippet in Google V8 triggers a *use-after-cgc* bug assigned to CVE-2019-13696 [22].

One of the most popular garbage collection variants is compacting garbage collection (*compact-gc*) [7, 11, 15] for efficient memory management. The primary cause of wasted memory is the presence of holes after memory objects are freed, known as memory fragmentation. If future memory allocations are larger than the existing holes, additional memory must be allocated and the memory holes remain unused. To resolve this issue, *compact-gc* rearranges live objects, which are still in use, and updates their references, so that the memory holes can be compacted into larger areas of free memory. For the update process, it manages an address table containing memory addresses where the references are stored. We call the references registered in the table *rooted* pointers.

Compact-gc introduces a new type of *use-after-free* bugs if an *unrooted* pointer, which is not registered in the address table, is used after *compact-gc*. It is well-known that if an *unrooted* pointer refers to a dead object, it becomes a dangling pointer after *compact-gc*. But also, if an *unrooted* pointer refers to a live object and the live object is moved after *compact-gc*, the *unrooted* pointer becomes a dangling pointer because the *unrooted* pointer is not updated. We call such bugs use-after-compact-gc (*use-after-cgc*) bugs. Figure 1 shows an example of *use-after-cgc* bugs in Google V8. First, an *unrooted* pointer, `code`, is defined by a function call. Then, `GetParameters` internally triggers *compact-gc*, which moves the object in `code` and makes `code` a dangling pointer. Finally, `code` is used as an argument of `JavaScriptFrameSummary` constructor, which leads to a *use-after-cgc* bug.

Detection of *use-after-cgc* bugs is an important problem because developers intentionally use *unrooted* pointers instead of *rooted* pointers for performance reasons. Using *rooted* pointers requires more memory operations than using *unrooted* pointers. When a *rooted* pointer is created, it is registered in the address table and it will be updated after *compact-gc*. If the pointer is used only before *compact-gc*, it does not need to be updated, and employing a *rooted* pointer is a waste of CPU cycles. Therefore, developers use *unrooted* pointers if they believe that the pointers are never used after *compact-gc*. For example, Google V8 uses *unrooted* pointers in over 5,000 locations. This highlights the need for automatic tools to guarantee they are safely used.

Although *use-after-cgc* bugs are as critical as *use-after-free* bugs, there are few *use-after-cgc* bug detectors and they have several limitations. For example, Google V8 and Mozilla SpiderMonkey have static analyzers to find *use-after-cgc* bugs, named gcmole [1] and rootAnalysis [2], respectively. They focus on finding patterns of *unrooted* pointer definition, *compact-gc*, and *unrooted* pointer use, which we refer to as *def-cgc-use* pairs. For scalability, they employ intra-procedural and path-insensitive analysis without solving path constraints. To mitigate their imprecision, they heuristically silence *def-cgc-use* pairs that developers mark as safe, which are target-specific and can be incorrect. Hence, their methods are scalable but not general and have high false-positives and false-negatives (see §2.3).

While it is possible to apply general bug finding methods, like fuzzing [19] and symbolic execution [14, 34], or previous *use-after-free* bug detection methods [5, 32, 33] to find *use-after-cgc* bugs, these cannot be both precise and scalable at the same time in practice. In particular, garbage collection is usually executed when the size of the managed heap is over a threshold. It thus may require many memory allocations to trigger the garbage collection. This makes fuzzing slow and symbolic execution not applicable because path constraints do not include any condition for the threshold. Previous *use-after-free* bug detectors employ pointer analysis to find *use-free* pairs. However, in programs using garbage collection, *free* operations are centralized in garbage collection functions and memory objects are only freed when they are dead. It requires a context-sensitive analysis, which is not scalable in practice, to figure out when the garbage collection functions free each pointer. Additionally, pointer analysis is difficult to be simultaneously precise and scalable for complex programs with garbage collection because there are too many pointers to analyze. Brown *et al.* [8] found *use-after-free* bugs caused by garbage collection in JS bindings with an incomplete source code parser, which focused on portions relevant to what they want to check. The imprecise parsing made the analysis scalable but path-insensitive and incomplete, which can lead to high false-positives and false-negatives.

In this paper, we propose a precise and scalable detection method of *use-after-cgc* bugs. At a high level, we find *def-cgc-use* pairs, which are *use-after-cgc* bug candidates, using data-flow analysis. We then automatically check their feasibilities with directed symbolic execution [4, 12, 18].

While the high-level idea is intuitive, there are two challenges to be both precise and scalable at the same time. (1) Traditional data-flow analysis [3] is scalable but not precise because it only supports data propagation through variables, not memory. (2) Directed symbolic execution is not scalable enough to be applied to large systems like JS engines.

To cover data propagation through variables and memory at scale for the systematic detection of *def-cgc-use* pairs, we employ intra-procedural static symbolic taint analysis. Symbolic taint analysis gives us precise data-flow analysis by representing taint values with symbolic expressions, but it is not scalable. Thus, we start the analysis from the function entry and do not dive into the function calls, which reduces path explosion and improves scalability. We also employ a taint policy based on value types and call-graphs to mitigate the imprecision of the intra-procedural analysis.

For scalable directed symbolic execution, we perform under-constrained directed symbolic execution, which starts from the function entry, skipping the paths from the program entry to the function entries, similar to UC-KLEE [26]. We also guide the directed symbolic execution based on the directed inter-procedural control-flow graphs (ICFGs) to avoid traversing paths that are irrelevant to check the feasibility of *def-cgc-use* pairs. Additionally, we prioritize the traversal of paths using the directed scheduling.

We implement a precise and scalable detection of *use-after-cgc* bugs in a static analyzer, named CGSan, and apply it to Google V8 and Mozilla SpiderMonkey. As a result, CGSan found 13 *use-after-cgc* bugs, including 10 bugs that the prior tools could not detect. And CGSan had only 2 false-positives while prior tools had 34 false-positives in total. We also show that the directed ICFGs and the directed scheduling improve the scalability of the directed symbolic execution. Lastly, we present three kinds of patches for *use-after-cgc* bugs based on our study and various patches by the developers.

In summary, our main contributions are as follows:

- We propose a precise and scalable static analyzer for *use-after-cgc* bug detection, called CGSan, based on intra-procedural static symbolic taint analysis and under-constrained directed symbolic execution.
- We present novel techniques to boost up the scalability of the directed symbolic execution, with the directed ICFGs and the directed scheduling.
- We evaluate CGSan on Google V8 and Mozilla SpiderMonkey and found 13 unique bugs with only 2 false-positives.
- We make our source code public to support open-science: https://github.com/DaramG/CGSan.

2 Background

In this section, we review how a compacting garbage collection manages memory objects. We then introduce a definition of a *use-after-cgc* bug with simple examples and explain how prior tools detect *use-after-cgc* bugs and show their limitations. Lastly, we describe symbolic taint analysis and directed symbolic execution, which are the basis of our techniques for finding *use-after-cgc* bugs.

2.1 Compacting Garbage Collection

Compacting garbage collection (*compact-gc*) [7, 11, 15] is an optimized garbage collection that solves the memory fragmentation problem. Due to its memory efficiency, many systems, including Java Virtual Machine, Microsoft Common Language Runtime, and some JS engines employ *compact-gc*. Thus, these systems may have potential *use-after-cgc* bugs.

Compact-gc is usually located within the memory allocator and is explicitly triggered when the allocated heap size is over a threshold. It starts with tracing memory to find objects that are no longer in use. If the objects are reachable from the root objects, which developers assert as still in use, they can be accessed in the rest of the program. It determines as garbage those objects that are not reachable from the roots, reclaims these garbage objects, and compacts memory by relocating live objects into contiguous memory.

Programs have diverse kinds of memory objects managed by *compact-gc*. There is a concept of a memory cell that is a basic type of memory object, which each different kind of memory object is derived from. For example, Google V8 and Mozilla SpiderMonkey have abstract super-classes representing the memory cell named `Object` and `Cell`, respectively.

When *compact-gc* relocates live objects, it moves their data and updates their references. Figure 2 shows memory layouts before and after *compact-gc*. It manages an address table that has addresses where the references are stored and, when needed, recursively updates the references in the address table. We call pointers in the table *rooted* pointers and pointers not in the table *unrooted* pointers. In general, object types derived from the cell type are *unrooted* pointers.

The address table should be kept updated by the lifetimes of *rooted* pointers. If *rooted* pointers are not saved into the table after they are initialized, they will not be updated during *compact-gc* and become dangling pointers. If they are not removed from the table after they are disposed, *compact-gc* will overwrite values at invalid addresses. To make sure that *rooted* pointers are registered and deregistered by their lifetime, developers employ a custom smart pointer in C++ whose constructor and destructor perform registration and deregistration. For instance, V8 and SpiderMonkey employ custom smart pointers, `Handle<T>` and `Rooted<T>` to represent *rooted* pointers for memory object type `T`.

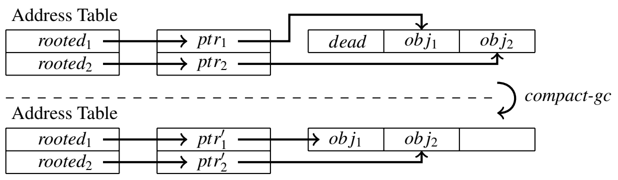


Figure 2: Memory layouts before and after *compact-gc*.

2.2 Use-after-Compacting-GC Bugs

In this subsection, we define *use-after-cgc* bugs with the terminologies and introduce their simple examples.

2.2.1 Definitions

Before diving into sample examples of *use-after-cgc* bugs, we first introduce several terminologies related to a *use-after-cgc* bug and define what a *use-after-cgc* bug is.

***Rooted* pointer.** As described in §2.1, a *rooted* pointer is a reference that is registered in the address table of *compact-gc* for the relocation.

***Unrooted* pointer.** An *unrooted* pointer is a reference that refers to an object managed by *compact-gc* but not in the address table of *compact-gc*. Therefore, it becomes dangling pointers after *compact-gc*. It is straightforward that an *unrooted* pointer to a dead object becomes a dangling pointer after *compact-gc*. For an *unrooted* pointer to a live object, it may become a dangling pointer after *compact-gc* if *compact-gc* moves the live object since the *unrooted* pointer is not in the address table and is not updated.

***Compact-gc* function.** A *compact-gc* function is a function that triggers *compact-gc*. There are two kinds of *compact-gc* functions: an explicit *compact-gc* function and an implicit *compact-gc* function. An explicit *compact-gc* function is a basic function that performs *compact-gc*. And an implicit *compact-gc* function is a function that inter-procedurally triggers an explicit *compact-gc* function. Note that an implicit *compact-gc* function may not trigger *compact-gc* depending on its calling context.

***Def-cgc-use* pair.** A *def-cgc-use* pair is a sequence of an *unrooted* pointer definition, *compact-gc* invocation, and the *unrooted* pointer use in a function. We denote a *def-cgc-use* pair as $< f, l_{def}, l_{cgc}, l_{use} >$ where function f defines an *unrooted* pointer, invokes a *compact-gc* function, and uses the *unrooted* pointer, at l_{def}, l_{cgc}, and l_{use}, respectively.

***Use-after-cgc* bug.** A *use-after-cgc* bug is a kind of *use-after-free* bug caused by the use of an *unrooted* pointer that becomes a dangling pointer after *compact-gc*. In this paper, we conclude as a *use-after-cgc* bug when a *def-cgc-use* pair has a feasible path, which is from the function entry of f through l_{def}, l_{cgc} to l_{use}.

```
void mayGC (Context* cx, int x) {
  if (x == 42) GC (cx);   // trigger GC if `x` == 42.
}

Object* buggy (Context* cx) {
  Object* unrooted;
  unrooted = defObj (cx); // define `unrooted` by a function call.
  GC (cx);                // trigger GC directly.
  return unrooted;        // BUG: use `unrooted` as a return value.
}

void Object::buggyMethod (Context* cx) {
                          // implicitly define `this`.
  mayGC (cx, 42);         // trigger GC.
  method ();              // BUG: implicitly use `this`.
}

void safe1 (Context* cx, Rooted<Object*> rooted) {
  mayGC (cx, 42);         // trigger GC and update ptr in `rooted`.
  use (*rooted);          // NOT A BUG: use updated ptr in `rooted`.
}

void safe2 (Context* cx, Object* unrooted) {
                          // explicitly define `unrooted`.
  mayGC (cx, 43);         // do not trigger GC.
  use (unrooted);         // NOT A BUG: use `unrooted` as argument
}                         //            but it is not moved.

void safe3 (Context* cx, int y) {
  Object* unrooted;
  unrooted = defObj (cx); // define `unrooted` by a function call.
  mayGC (cx, y);          // trigger GC if `y` == 42.
  if (y != 42)
    unrooted->field = 42; // NOT A BUG: use `unrooted` if `y` != 42.
}
```

Figure 3: Example code snippets depict two *use-after-cgc* bug cases and three safe cases.

2.2.2 Simple Examples

We describe two *use-after-cgc* bug cases and three safe cases in Figure 3 to illustrate the concept. We focus on intra-procedurally depicting examples for each step of *def-cgc-use* pairs including why each case is a bug or not. We assume that GC is an explicit *compact-gc* function, Context* represents a memory state, defObj returns a new object that will be moved after *compact-gc*, and use internally accesses the first argument. Additionally, Object* is a basic memory cell type of *compact-gc*, which is an *unrooted* pointer type, while Rooted<Object*> is a *rooted* pointer type. Note that * operation of Rooted<Object*> returns the corresponding *unrooted* pointer, whose type is Object*.

***Unrooted* pointer definition.** Memory objects managed by *compact-gc* are created by their constructors with a given Context. At an intra-procedural level, *unrooted* pointers can be passed in as function arguments or defined by memory reads or function calls. For example, buggy and safe3 define *unrooted* pointers named unrooted by a function call to defObj. And safe2 explicitly defines unrooted as a function argument while the this pointer in buggyMethod is implicitly defined by the C++ language. By contrast, safe1 gets a *rooted* pointer as a function argument and uses an *unrooted* pointer obtained from the *rooted* pointer after *compact-gc* updates it, which is safe.

***Compact-gc*.** Intra-procedurally, there are two kinds of methods to trigger *compact-gc*: call explicit *compact-gc* functions or implicit *compact-gc* functions. For instance, buggy triggers an explicit *compact-gc* function, GC, while buggyMethod, safe1, and safe3 call implicit *compact-gc* functions, mayGC. However, mayGC in safe2 cannot trigger GC because mayGC calls GC only if the second argument is 42.

***Unrooted* pointer use.** The use of *unrooted* pointers has the same definition as the use of general values. In the example, unrooted is used as a return value in buggy, an argument in buggyMethod, which is implicitly passed by the compiler, an explicit argument in safe2, and dereferenced in safe3. We assume that return values will be used in the callers and function arguments will be used in the callee functions.

Feasibility. To be a *use-after-cgc* bug, there must exist a feasible path of *def-cgc-use*. It is straightforward to recognize that buggy and buggyMethod have feasible *def-cgc-use* paths. However, safe1 is lacking an *unrooted* pointer definition and safe2 is lacking *compact-gc*. safe3 does not have a *cgc-use* path because *compact-gc* and *unrooted* pointer use are mutually exclusive: it triggers GC if y == 42 but unrooted is used only if y != 42.

2.3 Existing Detection Tools

To find *use-after-cgc* bugs, Google V8 and Mozilla SpiderMonkey have their source-code-based static analyzers, named gcmole [1] and rootAnalysis [2], respectively. They first calculate *compact-gc* functions based on call-graphs. They then perform intra-procedural data-flow analysis for *unrooted* pointers to find *def-cgc-use* pairs. Finally, they conclude *def-cgc-use* pairs as *use-after-cgc* bugs without checking their path constraints, for scalability, which leads to false-positives.

To mitigate false-positives due to path-insensitive analysis, they employ two heuristics: annotations from their developers and a list of known non *compact-gc* functions. In V8 and SpiderMonkey, there are source code annotations that assert there is no *compact-gc* in the scope. This allows gcmole and rootAnalysis to skip analysis on code with those annotations. However, the no_gc annotation in V8 is only advisory and does not ensure that *compact-gc* cannot be triggered. In addition, analyzers have a list of functions that will not trigger *compact-gc* and are exempted from the list of *compact-gc* functions. Developers should maintain the list when they change the implementation of functions in the list. This means that the list also can be wrong by mistake and lead to false-negatives (see §7.5).

Limitation. Although these tools are used in the development process, their methods are not precise due to inaccurate heuristics, path-insensitive and incomplete analysis. For example, gcmole could not detect the bug in Figure 7a due to its inaccurate heuristics, and rootAnalysis has false-positives because it is based on C++ Abstract Syntax Tree (AST) instead of compiled output (see §7.5).

Table 1: A table of *use-after-cgc* bugs each tool found in the example of Figure 3. ✓ denotes a bug detected by each tool while ✗ denotes a case each tool did not detect.

Functions	Truth	gcmole	rootAnalysis	CGSan
`mayGC`	✗	✗	✗	✗
`buggy`	✓	✓	✓	✓
`buggyMethod`	✓	✗	✗	✓
`safe1`	✗	✗	✗	✗
`safe2`	✗	✓	✓	✗
`safe3`	✗	✓	✓	✗

To show the imprecision of gcmole and rootAnalysis, we evaluate them on the examples in Figure 3 and the results are in Table 1. The prior tools detect three *use-after-cgc* bugs in `buggy`, `safe2`, and `safe3`. They are unable to distinguish safe cases due to their path-insensitive analysis. They incorrectly determine that `mayGC` function always triggers *compact-gc* because it contains a call to `GC`, which makes them detect `safe2` as a bug. In addition, they do not consider path constraints at line 33 in `safe3` and conclude that `safe3` has a feasible *def-cgc-use* pair. Surprisingly, they cannot detect `buggyMethod` as a bug. The fact that they are based on C++ AST enforces to handle many kinds of expressions and statements individually, causing them to miss some cases like the implicit `this` pointer. This highlights the imprecision of prior tools and motivates the development of our tool, CGSan.

2.4 Symbolic Taint Analysis

Taint analysis [27] is a program analysis technique that tracks information flows by predefined taint sources, taint propagations, and taint sinks. Conventional taint analysis treats values as tainted or untainted without exact value representation. Thus, conventional taint analysis is imprecise and may improperly under-taint or over-taint. For example, XORing two equivalent operands always returns zero, so the result should be untainted. However, conventional taint analysis marks the result as tainted if the operand value is tainted.

One method that overcomes these limitations is dynamic symbolic taint analysis, such as TaintPipe [21] and StraightTaint [20]. They propagate taints based on program execution traces. They first calculate symbolic summaries for the code segments, then update the taint state by applying symbolic summaries to the latest taint state while following the given program execution traces. They can be free from under-tainting and over-tainting problems because symbolic summaries express values with exact symbolic representation.

Our Analysis. Our goal is to find paths that trigger *use-after-cgc* bugs, which is a harder problem than to determine whether a given path triggers *use-after-cgc* bugs. We thus derive a variant of symbolic taint analysis that statically traverses all possible paths without relying on program execution traces, and does not dive into function calls for scalability. We refer to this as intra-procedural static symbolic taint analysis. Notably, intra-procedural analysis loses the precision, but we mitigate it with a type-based approach and call-graphs.

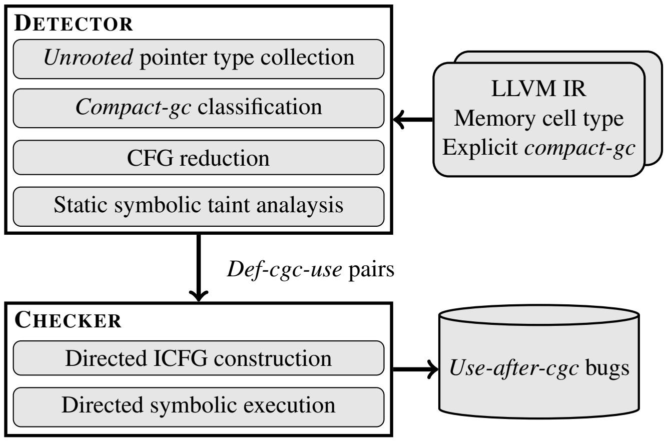

DETECTOR
Unrooted pointer type collection
Compact-gc classification
CFG reduction
Static symbolic taint analysis
LLVM IR
Memory cell type
Explicit *compact-gc*
Def-cgc-use pairs
CHECKER
Directed ICFG construction
Directed symbolic execution
Use-after-cgc bugs

Figure 4: CGSan Architecture.

2.5 Directed Symbolic Execution

Directed symbolic execution [4, 18] aims to verify whether the target point can be reached from the program entry point, which is known as the reachability problem. It is a widely used technique for reducing false-positives of static analysis. To guide symbolic execution to the target point, inter-procedural control-flow graphs are built, prioritizing the shortest distance path from the program entry to the target. For example, Ma *et al.* [18] employ *call-chain-backward symbolic execution* that starts from the target and checks reachability while traversing call-graphs reversely to quickly strip unreachable paths. And WOODPECKER [12] speeds up directed symbolic execution by skipping instructions irrelevant to the target point.

Our Analysis. There are two key differences between previous directed symbolic execution and ours. (1) We have multiple ordered targets, which means that we aim to check the feasibility of paths from the first target through several targets to the last target. (2) We start execution from the function entry, like UC-KLEE [26], not the program entry. This avoids exhausting the analysis time finding paths from the program entry to the first target in large programs. We also speed up our directed symbolic execution with the directed ICFGs and the directed scheduling, which we propose.

3 Overview

The main goal of CGSan is to automatically detect *use-after-cgc* bugs in a scalable and precise manner. In this section, we outline the overall architecture of CGSan and describe how CGSan finds *use-after-cgc* bugs on running examples.

Table 2: A table of *def-cgc-use* pairs that the DETECTOR module found in the example of Figure 3. l_{def}, l_{cgc}, and l_{use} denote where function f defines an *unrooted* pointer, invokes a *compact-gc*, and uses the *unrooted* pointer, respectively.

Function (f)	l_{def}	l_{cgc}	l_{use}
`buggy`	line 7	`GC` in line 8	line 9
`buggyMethod`	line 12	`mayGC` in line 14	line 15
`safe2`	line 23	`mayGC` in line 25	line 26
`safe3`	line 31	`mayGC` in line 32	line 34

3.1 Architecture

Figure 4 depicts the architecture of CGSan. At a high-level, CGSan takes LLVM IR of the target program, explicit *compact-gc* functions, and the memory cell type, and outputs a set of *use-after-cgc* bugs. CGSan consists of two major modules: DETECTOR, and CHECKER.

DETECTOR. CGSan takes LLVM IR of the target program as an input instead of an AST of the source code because the LLVM IR is a more accurate and already in single static assignment (SSA) form, which simplifies the analyzer implementation. This module takes as inputs the LLVM IR, explicit *compact-gc* functions, and the memory cell type, and finds *def-cgc-use* pairs. First, it computes the type hierarchy and collects *unrooted* pointer types, which are derived from the memory cell type. It then obtains a set of *compact-gc* functions, which internally or explicitly trigger *compact-gc*, from call-graphs. Next, an intra-procedural control-flow graph (CFG) is constructed while removing nodes and edges irrelevant to *def-cgc-use* pairs. Lastly, this module traverses the reduced CFGs and detects *def-cgc-use* pairs by intra-procedural static symbolic taint analysis with the taint policy based on *unrooted* pointer types and call-graphs (see §4).

CHECKER. This module confirms whether the detected *def-cgc-use* pairs are feasible by under-constrained directed symbolic execution. For each *def-cgc-use* pairs, it first constructs the directed ICFGs, which do not have nodes and edges irrelevant to checking *def-cgc-use* pairs and are optimized by the constant constraint propagation. It then performs under-constrained directed symbolic execution based on the directed ICFGs to determine feasible *use-after-cgc* bugs. To be more scalable, we employ the directed scheduling that determines which branches should be taken first (see §5).

3.2 Running Examples

We now depict the procedure of CGSan on the program of Figure 3 introduced in §2.2.2. As shown in Table 1, CGSan correctly found that only `buggy` and `buggyMethod` have *use-after-cgc* bugs. In the rest of this subsection, we describe how CGSan works using source code instead of LLVM IR for the sake of simplicity.

First, the DETECTOR module finds *unrooted* pointer types and *compact-gc* functions based on the fact that `Object*` is a basic memory cell type and `GC` is an explicit *compact-gc* function, respectively. It concludes that there is only one *unrooted* pointer type, `Object*`, because this example does not have any type derived from `Object*`. It then calculates a call-graph and collects *compact-gc* functions by reversely traversing the call-graph from `GC`. As a result, it outputs `GC`, `mayGC`, `buggy`, `buggyMethod`, `safe1`, `safe2`, and `safe3` as *compact-gc* functions.

Before performing static taint analysis, the DETECTOR module constructs and reduces CFGs of *compact-gc* functions. It removes nodes and edges that are irrelevant to *def-cgc-use* pairs from the CFGs by finding which nodes contain *unrooted* pointer definition, *compact-gc*, and *unrooted* pointer use based on *unrooted* pointer types. For example, it removes all nodes and edges from CFGs of `mayGC` and `safe1` because they do not define any *unrooted* pointer before calling the *compact-gc* functions. It also deletes the `else` edge of line 33 from the CFG of `safe3` because an *unrooted* pointer will be never used after taking that edge. CFGs of other functions will be preserved because each has a node including an *unrooted* pointer definition, *compact-gc*, and *unrooted* pointer use.

The DETECTOR module performs intra-procedural static symbolic taint analysis to collect *def-cgc-use* pairs in *compact-gc* functions. This module follows the reduced CFGs to cut down the search space of the analysis. It introduces values whose types are *unrooted* pointer types, `Object*`, as tainted when they are defined, and it marks all tainted variables as freed when a *compact-gc* function is called. During the analysis, it continues checking for use of freed tainted variables and concludes that there are *def-cgc-use* pairs if freed tainted variables are used. Finally, as shown in Table 2, it finds *def-cgc-use* pairs in `buggy`, `buggyMethod`, `safe2`, and `safe3`.

Lastly, the CHECKER module classifies feasible *def-cgc-use* pairs. It first builds the directed ICFGs for the detected *def-cgc-use* pairs in two steps. (1) It removes nodes and edges of ICFGs disconnected from nodes in *def-cgc-use* pairs, similar to CFG reduction in the DETECTOR module. For example, `else` edges in `mayGC` of line 2 and `safe3` of line 33 will be removed. (2) It then strips the reduced ICFGs by the constant constraint propagation. It is straightforward to delete the edge in calling `mayGC(cx, 43)` of `safe2` by traditional constant-propagation so that `safe2` never calls `GC`. In `safe3`, variable `y` must not be 42 to reach `use` node in line 34. Therefore, the constant constraint propagation spreads the constraint, `y != 42`, to other nodes. It concludes that `mayGC` in `safe3` cannot call `GC` if we must reach `use` node in line 34. And the CHECKER module finds feasible paths for *def-cgc-use* pairs in `buggy` and `buggyMethod` by directed symbolic execution with the directed ICFGs and the directed scheduling. Finally, CGSan concludes *use-after-cgc* bugs exist in `buggy` and `buggyMethod`, which is the correct answer as shown in Table 1.

4 Detector

Recall from §3.1, the Detector module performs four steps. We describe how we collect *unrooted* pointer types from the memory cell type, and classify *compact-gc* functions from the given explicit *compact-gc* functions. We then explain CFG reduction for skipping paths irrelevant to *def-cgc-use* pairs. Lastly, we depict static symbolic taint analysis for the systematic detection of *def-cgc-use* pairs.

4.1 Unrooted Pointer Type Collection

According to §2.1, *unrooted* pointer types are derived from the memory cell type. We thus traverse the type hierarchy in reverse from the given memory cell type and collect *unrooted* pointer types. However, the type information in LLVM IR is not rich enough to build the type hierarchy for collecting *unrooted* pointer types because they are optimized by compilers. For instance, `AbstractCode` in Figure 1, which is an *unrooted* pointer type, becomes `i64` type in LLVM IR. Therefore, we obtain type hierarchy from source-level LLVM metadata that preserves source-level types and collect *unrooted* pointer types.

4.2 Compact-GC Classification

There are two kinds of *compact-gc* functions: an explicit *compact-gc* function, and an implicit *compact-gc* function, which inter-procedurally triggers an explicit *compact-gc* function. We first calculate call-graphs while resolving indirect call targets by previous type-based approaches [16,24] to construct more precise call-graphs. We then reversely traverse call-graphs from the given explicit *compact-gc* functions and gather a set of *compact-gc* functions.

4.3 CFG Reduction.

As described in §2.2.1, finding *use-after-cgc* bugs is the same as finding *def-cgc-use* pairs. In other words, a *use-after-cgc* bug requires a path from l_{def} through l_{cgc} to l_{use}. We thus remove nodes and edges that are irrelevant to *def-cgc-use* pairs from CFGs of every function. We first recognize which CFG nodes have l_{def} or l_{use} by finding instructions that define or use *unrooted* pointer typed values, and identify CFG nodes having l_{cgc} by finding *compact-gc* function calls. We then delete edges from the CFG that are not in paths from the function entry to l_{def}, from l_{def} to l_{cgc}, and from l_{cgc} to l_{use}. Lastly, we remove nodes from the CFG that are not reachable from the function entry and get the reduced CFG. Note that this CFG reduction allows the Detector module to be more scalable without any additional false-negatives. In §7.2, we will show how effectively this CFG reduction cuts down the search space of the Detector module without any additional false-negatives.

4.4 Static Symbolic Taint Analysis

To systematically detect *def-cgc-use* pairs, we employ intra-procedural static symbolic taint analysis that tracks data-flows of *unrooted* pointers by symbolic evaluation while following the reduced CFG. For scalability, we do not solve path constraints and perform an intra-procedural analysis, while mitigating its incompleteness with a taint policy based on *unrooted* pointer types and *compact-gc* functions. The path constraints will be considered in the Checker module. We also assumed that each symbolic variable, if it is a pointer, refers to a unique object, i.e., no aliasing, similar to UC-KLEE [26].

Taint Introduction. Intra-procedural analysis requires the initialization of values such as function arguments, return values of function calls, and values in unseen memory. We initialize them as new unconstrained symbolic values because intra-procedural analysis assumes that they can be any value. We introduce them as new symbolic taint values if they are *unrooted* pointer types, which allows us to get taint sources without recursing into function calls. If they are not *unrooted* pointer types, we introduce them as normal symbolic values. This policy covers all *unrooted* pointer definitions depicted in §2.2.2: we mark the return value of `defObj` and function arguments that are `Object*` type as symbolic taint values. Note that we easily cover the implicitly added arguments like `this` in `buggyMethod` because we are based on LLVM IR.

Taint Propagation. Symbolic taint analysis presents values as symbolic expressions and propagates them by symbolic evaluation. We define a different policy for function calls to detect *def-cgc-use* pairs. For *compact-gc* functions, we assume that they always trigger *compact-gc* and *compact-gc* always moves live objects without updating *unrooted* pointers. Thus, we mark all symbolic taint values, which refer to *unrooted* pointers, as freed. For other functions, we make them return symbolic taint values if their return values are *unrooted* pointer types. If not, we make them return normal symbolic values. Notably, when a pointer is passed to a function call, we assume that the corresponding memory will not be modified because the Detector module is based on the intra-procedural analysis.

Taint Checking. We monitor four operations to capture *def-cgc-use* pairs: a freed taint value is used in 1) a memory address of memory load, 2) a memory address or a value of memory store, 3) a function call argument or 4) a return value. This policy is based on the assumption that arguments will be used in callee functions and the return value will be used in caller functions. Finally, we conclude that there is a *def-cgc-use* pair if we find that a freed taint value, i.e. a freed *unrooted* pointer, is used. Notably, when a *compact-gc* function call takes a non-freed taint value as an argument, we do not conclude whether the current analyzed function has a *def-cgc-use* pair because the taint value is not freed. Instead, we check whether the *compact-gc* function has a *def-cgc-use* pair when we perform the analysis from its function entry.

5 Checker

Prior tools [1, 2] are path-insensitive and employ target-specific heuristics to filter out infeasible *def-cgc-use* pairs, which is scalable but not precise as we discussed in §2.3. In contrast, we apply under-constrained directed symbolic execution to check feasible *def-cgc-use* pairs. For scalability, we reduce paths to explore for checking *def-cgc-use* pairs by constructing their directed ICFGs. Also, while exploring the directed ICFGs, we prioritize which branches should be taken first using the directed scheduling.

5.1 Directed ICFG Construction

As shown in §3.2, we do not need to traverse all paths to check feasible *def-cgc-use* pairs. We thus remove irrelevant nodes and edges from their ICFGs and optimize the reduced ICFGs to the directed ICFGs by the constant constraint propagation.

Irrelevant ICFG Reduction. The pseudo-code of the irrelevant ICFG reduction is shown in Algorithm 1. The `Reduce` function takes in the *def-cgc-use* pair denoted as $< f, l_{def}, l_{cgc}, l_{use} >$ and the ICFG of function f denoted as G, and outputs the reduced ICFG. First, the `ReduceForPair` function intra-procedurally removes nodes and edges from G that are not in paths from the function entry through l_{def} and l_{cgc} to l_{use} as the CFG reduction in §4.3.

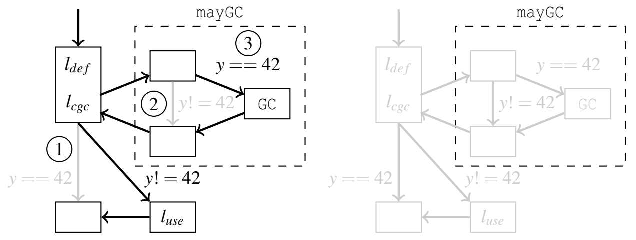
Next, the `ReduceForGc` function inter-procedurally reduces the ICFG in l_{cgc}. `GetSubICFG`(G, l_{cgc}) returns G_{cgc}, the sub ICFG of the function call target at l_{cgc} of G. And `GetCgcCalls`(G_{cgc}) collects L_{gc}, a set of locations where *compact-gc* functions are invoked in G_{cgc}. Then, `Trim` removes nodes and edges disconnected with nodes in L_{gc}. In the `for`-loop of line 8-10, we recursively reduce ICFGs of *compact-gc* functions in L_{gc} if they are not connected with others in L_{gc}. If so, we skip the reduction because triggering *compact-gc* once in the connected nodes is enough to be a feasible *def-cgc-use* pair. Lastly, `SetSubICFG`(G, l_{cgc}, G_{cgc}) updates the sub ICFG at l_{cgc} of G with G_{cgc}.

To help illustrate this irrelevant ICFG reduction, we provide an example on `safe3` of Figure 3, and its reduced ICFG is shown in Figure 5a. First, `ReduceForPair` removes ① because we cannot reach l_{use} if we take that edge. In `ReduceForGc`, `GetSubICFG` returns an ICFG of `mayGC`, which is boxed in Figure 5a. `GetCgcCalls` then outputs `GC` node and `Trim` removes ② because that edge is not connected with `GC` node. Finally, `SetSubICFG` updates the sub ICFG and we get the reduced ICFG in Figure 5a.

Constant Constraint Propagation. Traditional constant-propagation [3] substitutes values with known constant values while following the program execution. This is enough for code optimization because the instructions to be executed do not affect the current execution state. However, the reduced ICFG enforces taking specific branches and we must satisfy their branch conditions before taking them. Branch conditions

Algorithm 1: Irrelevant ICFG Reduction

```
Input  : The def-cgc-use pair $(f, l_{def}, l_{cgc}, l_{use})$,
         The ICFG of $f$ $(G)$
Output : The reduced ICFG
1  function Reduce($G$, $l_{def}$, $l_{cgc}$, $l_{use}$)
2      $G \leftarrow$ ReduceForPair($G$, $l_{def}$, $l_{cgc}$, $l_{use}$)
3      return ReduceForGc($G$, $l_{cgc}$)
4  function ReduceForGc($G$, $l_{cgc}$)
5      $G_{cgc} \leftarrow$ GetSubICFG($G$, $l_{cgc}$)
6      $L_{gc} \leftarrow$ GetCgcCalls($G_{cgc}$)
7      $G_{cgc} \leftarrow$ Trim($G_{cgc}$, $L_{gc}$)
8      for $l \in L_{gc}$ do
9          if IsNotConnected($L_{gc}$, $l$) then
10             $G_{cgc} \leftarrow$ ReduceForGc($G_{cgc}$, $l$)
11     return SetSubICFG($G$, $l_{cgc}$, $G_{cgc}$)
```



(a) The reduced ICFG. (b) The directed ICFG.

Figure 5: The reduced ICFG and the directed ICFG of `safe3`. The ICFG in the dotted box represents the ICFG of `mayGC`. Black and grey mean remaining and removed, respectively.

sometimes check whether symbolic values are specific constant values or not, which we call constant constraints. For example, the reduced ICFG in Figure 5a does not have ①, which condition is $y == 42$, and enforces y not to be 42 even before taking that branch. Therefore, we propagate constant constraints instead of constant values in both the forward and backward direction of the program execution. Note that we only perform propagation for variables stored in LLVM IR registers, not memory.

Algorithm 2 depicts the overview of the constant constraint propagation. The `Propagate` function gets the reduced ICFG and the context as inputs, and returns the directed ICFG where the context includes constant constraints of function arguments and the return value. We start invoking the `Propagate` function with the reduced ICFG from the irrelevant ICFG reduction stage and the empty context.

In the beginning, the `Prepare` function updates the context with the reduced ICFG. If the reduced ICFG makes `phi` instruction return a constant, we add that constant constraint to the context. And if branch conditions for following the reduced ICFG are that values must be some constants or must not, we add them to the context.

Algorithm 2: Constant Constraint Propagation

```
Input  : The reduced ICFG ($G$),
         The context of constant constraints ($C$)
Output : The directed ICFG
1 function Propagate($G$, $C$)
2     $C' \leftarrow$ Prepare($G$, $C$)
3     $G'$,$C' \leftarrow$ Backward($G$, $C'$)
4     $G' \leftarrow$ Forward($G'$, $C'$)
5     if IsSameICFG($G$, $G'$) then
6         return $G'$
7     return Propagate($G'$, $C$)
```

`Backward` function propagates constant constraints and deletes edges that are unreachable in the given context while intra-procedurally traversing the reduced ICFG from the function end to the function entry. For example, if the context has a constant constraint of the value defined by a `phi` instruction, we filter out values that do not satisfy the constant constraint and remove the corresponding edges from the ICFG. In particular, if the reduced ICFG enforces a return value from a function call to satisfy a constant constraint, we update a constant constraint of the return value in the context of the function call with the constant constraint. The updated context will be used to trim paths that return values violating the constant constraint in the `Forward` function.

Next, the `Forward` function inter-procedurally and recursively performs forward propagation of constant constraints based on the given context and removes edges unsatisfiable with constant constraints. It then deletes unreachable edges due to the previous removal. Finally, we return the directed ICFG if the directed ICFG and the given ICFG are the same. If not, we invoke `Propagate` with the directed ICFG and the given context until they are the same as described in line 5-7. This is because the directed ICFG can introduce a new constant constraint if some edges are deleted, so that we calculate the fixed point of the `Propagate` function.

To help understand the constant constraint propagation, we provide an example on `safe3`. Figure 5b shows the directed ICFG of `safe3`. Based on its reduced ICFG in Figure 5a, the `Prepare` function appends the constant constraint, `y != 42`, to the context. The `Backward` function performs the backward propagation with the constant constraint and updates the context with the constraint that the second argument of `mayGC` function call, y, must not be 42. Next, the `Forward` function inter-procedurally performs forward propagation and removes the edge denoted as ③ because `y != 42`. This removes all paths for *def-cgc-use* and the `Forward` function makes all edges unreachable. And we recursively call `Propagate` because the given ICFG, which is the reduced ICFG, and the obtained ICFG have different edges. But it is straightforward to end because the obtained ICFG is empty. Finally, we get the directed ICFG, which is empty, as shown in Figure 5b.

5.2 Directed Scheduling

Although the directed ICFG reduces the search space, scheduling which paths will be executed first affects the scalability of directed symbolic execution. We thus considered shortest-distance symbolic execution (SDSE) [18], prioritizing the path traversals with the shortest distance to the target. However, SDSE can be stuck because the shortest distance path often includes an error-handling path, e.g, functions immediately return if arguments are invalid, and its path constraints are often unsatisfiable. Therefore, we prioritize paths by loop scheduling, stateful scheduling, and retrievable scheduling.

We orchestrate a task queue, using depth-first-search to schedule paths, where a task is an execution state and an ICFG node to be executed. We start with a task queue that has the initial execution state and the entry node of a function. We pop a task from the front of the task queue and evaluate its node with its execution state. Then, we collect the successor nodes from the directed ICFG whose branch conditions are satisfiable with path constraints of the execution state. We prepend the nodes to the front of the task queue, and we repeat until the queue becomes empty or timeout.

Loop Scheduling. To mitigate cases that are stuck in traversing loops, many previous works [17,29,31,35] limit the number of iterations of each loop, also known as loop unrolling. However, there is a trade-off between too few iterations and too many iterations, potentially resulting in an incomplete analysis. We employ loop scheduling based on how many times the execution state already took the branch from the current node to the successor node, which is denoted as n. Instead of ignoring a successor node once n reaches the limit, as loop unrolling does, we push the task to the back of the queue. This scheduling has the same impacts as loop unrolling but preserves the completeness of the analysis.

The loop scheduling also prepends successor nodes to the queue in the order of n, so that the successor node with the smallest n will be first in the queue unless n is over the limit in which case the node is appended to the end of the queue. We empirically chose the limit as 100 for our analysis. This guides us to take branches that are previously less taken and avoid cases that are stuck in traversing loops.

Stateful Scheduling. We have three ordered targets to check *def-cgc-use* pairs, which start from l_{def} through one of the *compact-gc* function calls in l_{cgc} to l_{use}. It is straightforward to guide to l_{def} and l_{use} because they are unique in the directed ICFG. However, the directed ICFG can contain multiple *compact-gc* function calls in l_{cgc} and allow paths that do not trigger *compact-gc* in l_{cgc} when there are connected nodes containing *compact-gc* in l_{cgc}. This is because the directed ICFG cannot determine which node in the connected nodes having *compact-gc* will be touched. To avoid skipping *compact-gc*, we do not append the task that already passed all *compact-gc* nodes in l_{cgc} without touching any of them. In addition, we prioritize traversing nodes containing *compact-gc* before touching *compact-gc*, and prioritize traversing nodes that do not contain *compact-gc* after touching *compact-gc*.

Retrievable Scheduling. The directed ICFG enforces taking specific branches, which means that the corresponding branch conditions must be satisfied. When path constraints of the current task are unsatisfiable with the condition, the next task in the queue is likely unsatisfiable also because they will have almost the same path constraints due to the depth-first-search. We thus find the task that has path constraints satisfiable with the branch condition from the front of the queue, and process that task first. We refer this to as retrievable scheduling. Retrievable scheduling is inspired by the path kneading of ShellSwap [6], but, as far as we know, this has not been applied to the directed symbolic execution.

6 Implementation

We have implemented CGSan with 0.9K lines of C++ code and 9.5K lines of F# code. We use C++ for loading LLVM IR with LLVM version 11.0 and use F# for the rest of the system. Specifically, it takes 6.1K lines of F# code to load LLVM into F#. And we employ Z3 [23] version 4.8.8 for solving path constraints. We now describe relevant implementation details.

Compiling Source to LLVM IR. V8 and SpiderMonkey are compilable under LLVM, so it is straightforward to transform the source code to LLVM IR. We disable function inlining to avoid duplicate analysis of the same source code, and we enable source-level debugging to get rich LLVM metadata, such as type information used in the DETECTOR module.

Configurable Symbolic Execution. The DETECTOR module and the CHECKER module both need symbolic evaluation, though their requirements differ. We built a configurable and reusable symbolic execution library to reduce implementation efforts. It has options to control the symbolic execution, such as guiding paths to explore, function modeling, and memory operation hooks. In the case of the DETECTOR module, we turn off solving path constraints, add function models, and register callbacks for return instructions, memory loads, and stores as addressed in §4. For the CHECKER module, we configure guiding paths with the directed ICFGs and the directed scheduling, while solving path constraints as described in §5.

7 Evaluation

We now evaluate CGSan to answer the following questions:

1. Can the DETECTOR module find *def-cgc-use* pairs and does CFG reduction affect the scalability? (§7.2)

2. Can the CHECKER module effectively figure out feasible *def-cgc-use* pairs? (§7.3)

3. Can CGSan find real-world *use-after-cgc* bugs? (§7.4)

4. How does CGSan perform against prior tools? (§7.5)

7.1 Experimental Setup

We evaluate CGSan on LLVM IR compiled from the latest versions of two major JS engines: Google V8 8.1 and Mozilla SpiderMonkey 74. The evaluation was performed on a machine with an AMD Ryzen 3900X (12 cores) and 64GB RAM, running 64-bit Ubuntu 18.04 LTS. We did not evaluate JavaScriptCore because it did not perform *compact-gc*.

Recall that CGSan requires the explicit *compact-gc* functions and the memory cell type as inputs. We thus set the memory cell types as `Object` for V8 and `Cell` for SpiderMonkey, and we pass `CollectGarbage` and `gc` as the explicit *compact-gc* functions of V8 and SpiderMonkey. We also set timeouts of 10 seconds for analyzing a function in the DETECTOR module and 10 minutes for checking the feasibility of a detected *def-cgc-use* pair in the CHECKER module.

7.2 DETECTOR Statistics

Table 3 presents the overall statistics of the DETECTOR module. From the given memory cell type, the DETECTOR module collected 549 and 244 *unrooted* pointer types in V8 and SpiderMonkey, respectively. It also identified 8,224 and 14,680 *compact-gc* functions in V8 and SpiderMonkey. Finally, it found 20 and 1,464 *def-cgc-use* pairs in V8 and SpiderMonkey. Note that SpiderMonkey had a lot of *def-cgc-use* pairs but they did not have any feasible path, which means that they were not *use-after-cgc* bugs (see §7.3). In total, the DETECTOR module finished in a half-hour for each target, however, even with the help of intra-procedural analysis and CFG reduction, there were timeout cases in 112 functions of V8 and 118 functions of SpiderMonkey. Timeouts were due to the path explosion problem in large functions and may lead to false-negatives.

Effectiveness of CFG Reduction. Recall from §4.3, CFG reduction improves the scalability of the DETECTOR module without any additional false-negatives. To show the effectiveness of CFG reduction, we evaluated the DETECTOR module with and without CFG reduction, and Table 3 shows the results. After applying CFG reduction, the detection time decreased by 40% and the number of timeout cases decreased by 48% on average of both targets. This highlights that CFG reduction makes the DETECTOR module more scalable and more complete because fewer timeout cases mean that we covered more functions. We also verified *def-cgc-use* pairs found by the DETECTOR module with CFG reduction includes all *def-cgc-use* pairs found by the DETECTOR module without CFG reduction, i.e. there is no additional false-negative.

7.3 CHECKER Statistics

Before evaluating the CHECKER module, we manually analyzed the detected *def-cgc-use* pairs. We first checked whether *compact-gc* functions of *def-cgc-use* pairs trigger *compact-gc*. If so, we verified whether *def-cgc-use* pairs have feasible

Table 3: The detection result of the DETECTOR module with and without CFG redcution.

Target	Unrooted Type	Compact-gc	DETECTOR w/ CFG reduction			DETECTOR w/o CFG reduction		
			def-cgc-use	**Timeout**	**Time**	***def-cgc-use***	**Timeout**	**Time**
V8	549	8,224	20	112	36m	20	263	67m
SpiderMonkey	244	14,680	1,464	118	35m	1,426	178	51m

Table 4: The check result of manual analysis, the CHECKER module with and without the directed ICFG.

Target	Manual Analysis		CHECKER w/ Directed ICFG				CHECKER w/o Directed ICFG			
	Feasible	**Infeasible**	**Feasible**	**Infeasible**	**Timeout**	**Avg. Time**	**Feasible**	**Infeasible**	**Timeout**	**Avg. Time**
V8	19	1	18	0	2	98s	8	0	12	364s
SpiderMonkey	0	1,464	0	1,309	155	64s	0	8	1,456	597s

paths. We could reduce the manual effort as some *def-cgc-use* pairs shared the same *compact-gc* functions. In addition, we used the fact that developers of SpiderMonkey manually add `AutoSuppressGC` to temporally disable *compact-gc*. We easily filtered out the *compact-gc* functions that do not trigger *compact-gc* by checking whether they are within an `AutoSuppressGC` scope. For example, `AutoEnterAnalysis` in Figure 6, which internally calls `AutoSuppressGC`, temporally disables GC until the end of `AddTypePropertyId` function, which is the scope of `enter`. This ensures that *compact-gc* in `addType` of line 10 does not do anything.

As described in Table 4, we concluded that there were 19 feasible pairs in V8 and no feasible pairs in SpiderMonkey by the manual analysis. And the CHECKER module found 18 feasible pairs and 0 infeasible pairs in V8 while detecting 0 feasible pairs and 1,309 infeasible pairs in SpiderMonkey. Compared to the manual analysis, CGSan only missed a feasible case in V8, which CGSan could not verify within the given timeout. Also, while the timeout was 10 minutes, it took 98 seconds and 64 seconds on average to check the feasibility of a pair in V8 and SpiderMonkey. This highlights that the CHECKER module is precise and scalable.

The crux of the CHECKER module is that the directed ICFG and the directed scheduling improve the scalability while preserving the precision. To show their effectiveness, we compare the CHECKER module with and without them.

Effectiveness of the Directed ICFG. As shown in Table 4, after using the directed ICFG, the CHECKER module detected 2.25× more feasible *def-cgc-use* pairs in V8 and 163× more infeasible *def-cgc-use* pairs in SpiderMonkey while having fewer timeouts. And the CHECKER module with the directed ICFG found a superset of feasible *def-cgc-use* pairs and infeasible *def-cgc-use* pairs compared to the CHECKER module without the directed ICFG. This means that the directed ICFG preserves the precision. In addition, the directed ICFG effectively reduced irrelevant search space, so that the average time to check a *def-cgc-use* pair decreased by 83%.

```
1  void js::AddTypePropertyId(
2    JSContext* cx, ObjectGroup* group, JSObject* obj, jsid id,
3    TypeSet::Type type) {
4    ...
5    // Internally trigger `AutoSuppressGC` and
6    // temporally disable GC in this function scope.
7    AutoEnterAnalysis enter(cx);
8    ...
9    // This invokes GC but GC does not do anything.
10   types->addType(sweep, cx, type);
11   ...
12 }
```

Figure 6: *Compact-gc* suppression of SpiderMonkey.

Table 5: The comparison of the directed scheduling on V8.

Directed Scheduling	Feasible	Timeout	Avg. Time
w/o Loop Scheduling	1	19	581s
w/o Stateful Scheduling	13	7	221s
w/o Retrievable Scheduling	16	4	157s
CGSan	18	2	98s

Effectiveness of Directed Scheduling. To demonstrate the effectiveness of three methods of the directed scheduling, we evaluated the CHECKER modules without each scheduling, and Table 5 shows the results. We evaluated only on V8 because the CHECKER module did not find any feasible *def-cgc-use* pair in SpiderMonkey.

The CHECKER module without loop scheduling found only one feasible case and had 19 timeout cases while taking 5.9× more time to check a *def-cgc-use* pair on average. Note that it pushed a task to the back of the task queue if it already took the next branch over the limit, even though it did not have loop scheduling. The results highlight that loop scheduling effectively prioritizes branches. Also, the CHECKER without stateful scheduling and retrievable scheduling found fewer feasible cases and took more time than CGSan, thus these two scheduling methods are effective in terms of scalability.

7.4 Bug Findings

We manually investigated 18 feasible *def-cgc-use* pairs from CGSan and categorized them into 15 unique *use-after-cgc* cases by functions of *def-cgc-use* pairs. Table 6 shows unique *use-after-cgc* cases and how they defined *unrooted* pointers and used them after *compact-gc*. They defined *unrooted* pointers by function arguments, and function calls, and they used *unrooted* pointers as function arguments, return values, and values of memory store. Most *unrooted* pointer definitions and uses in Table 6 were done by function calls because the DETECTOR module employs the intra-procedural analysis. After we reported all bugs to the vendor, developers fixed 12 cases and marked 2 cases as "Won't Fix".

"Fixed" Case. To describe fixed cases, we choose case 1 and case 13 in Table 6, which have two common *def-use* patterns. Figure 7a shows the buggy implementation of case 1. This function defines an *unrooted* pointer, `raw_dictionary`, by the function call, which is the override operator $*$, at line 7. It then triggers the *compact-gc* function named `AddShadowingKey` at line 14. After iterating the loop once, `raw_dictionary` is used as a function argument at line 11, which leads to a *use-after-cgc* bug. Notably, there is a `no_gc` mark in the function, even though `AddShadowingKey` internally triggers *compact-gc*. In Figure 7b, there is the buggy implementation of case 13, `AddAsyncParentModule`. This takes an *unrooted* pointer, `this`, as an argument, which the C++ compiler implicitly appends. Then, `ArrayList::Add` internally triggers *compact-gc*, and `this` is used as an argument of `set_async_parent_modules` method, which the C++ compiler implicitly passes. Therefore, this leads to a *use-after-cgc* bug.

"Won't Fix" Case. Developers claimed that they will not fix case 4 and case 10 in Table 6 because these cases were not buggy even though they had feasible *def-cgc-use* pairs. We assumed that all *unrooted* pointers will be freed after *compact-gc*. However, `AllocateRawWithImmortalMap` of case 4 only took permanent *unrooted* pointers as an argument, which are never freed even after *compact-gc*, e.g, built-in constant JS objects like `undefined`. This means that it is not buggy for now, but can be buggy if developers invoke it with movable *unrooted* pointers. Therefore, instead of fixing the *use-after-cgc* bug pattern, developers enforce passing permanent *unrooted* pointers when calling `AllocateRawWithImmortalMap` by adding a debug check. `CheckStackGuardState` of case 10 intentionally accessed an *unrooted* pointer after *compact-gc* to calculate the memory address difference before and after *compact-gc*. Therefore, developers claimed that they are not buggy and marked them as "Won't Fix".

Impact of Discovered Bugs. After we reported the found bugs, Google confirmed their impacts and gave rewards for case 7 in Table 6 and the case where the DETECTOR module found but the CHECKER module could not verify within the timeout. As we described in Table 6, developers did not fix

```
1  template <typename Derived, typename Shape> ExceptionStatus
2    BaseNameDictionary<Derived, Shape>::CollectKeysTo(
3      Handle<Derived> dictionary, KeyAccumulator* keys) {
4    ...
5      DisallowHeapAllocation no_gc;
6      // define an unrooted pointer, `raw_dictionary`.
7      Derived raw_dictionary = *dictionary;
8      for (InternalIndex i : dictionary->IterateEntries()) {
9        ...
10       // use the unrooted pointer, `raw_dictionary`.
11       PropertyDetails details = raw_dictionary.DetailsAt(i);
12       if ((details.attributes() & filter) != 0) {
13         // trigger GC.
14         keys->AddShadowingKey(k);
15         ...
```

(a) A *use-after-cgc* bug in `CollectKeysTo`.

```
// implicitly define an unrooted pointer, `this`.
void SourceTextModule::AddAsyncParentModule(
  Isolate* isolate, Handle<SourceTextModule> module) {
  // trigger GC.
  Handle<ArrayList> new_array_list =
      ArrayList::Add(
        isolate, handle(async_parent_modules(), isolate), module
      );
  // implicitly use the unrooted pointer, `this`.
  set_async_parent_modules(*new_array_list);
}
```

(b) A *use-after-cgc* bug in `AddAsyncParentModule`.

Figure 7: Two *use-after-cgc* bugs found by CGSan on V8.

case 4 and 10 because they are not buggy. For other cases, developers fixed but Google did not reward because: case 1, 2, 3, 11, and 12 affected other users of V8, but not Chrome, and may require user interaction; case 13 was in the experimental features; case 14 was a bug but not a serious security issue; case 5, 6, 9, and 15 were already found by gcmole.

7.5 Comparison against Prior Tools

To show the performance of CGSan relative to prior tools, we compare CGSan against gcmole and rootAnalysis. We evaluated the number of bugs each tool found in the examples of Figure 3 and two JS engines, V8 and SpiderMonkey.

Comparison on the examples. In §2.3, we showed that the two prior tools, gcmole and rootAnalysis, correctly concluded `buggy` as a *use-after-cgc* bug, but incorrectly concluded `safe2` and `safe3` as *use-after-cgc* bugs and `buggyMethod` as not a bug, in the examples of Figure 3. Whereas, according to Table 1, CGSan correctly determined *use-after-cgc* bugs as only in two functions, `buggy` and `buggyMethod`. This result highlights that CGSan is more precise than prior tools, gcmole and rootAnalysis.

Comparison on the JS engines. We also check that CGSan is effective in finding *use-after-cgc* bugs in real-world JS engines. To demonstrate this, we ran gcmole and rootAnalysis on V8 and SpiderMonkey, respectively. We could not apply each tool to both JS engines because they are tightly coupled with their targets, e.g. target-specific heuristics, and would require significant modifications to apply to others.

Table 6: A list of unique *use-after-cgc* cases CGSan found. `Def` and `Use` represent patterns of *unrooted* pointer definition and use, respectively. In the `Prev.` column, ✓ indicates the case detected by gcmole and ✗ is the case not detected before.

Idx	Function	Def	Use	Status	Patch Strategy	Prev.
1	BaseNameDictionary<Derived, Shape>::CollectKeysTo	Call	Call	Fixed	Relocate *compact-gc*	✗
2	Deserializer::DeserializeDeferredObjects	Call	Call	Fixed	Remove *compact-gc*	✗
3	Deserializer::ReadObject	Call	Return	Fixed	Remove *compact-gc*	✗
4	Factory::AllocateRawWithImmortalMap	Arg	Call	Won't Fix	-	✓
5	Factory::NewFixedArrayWithFiller	Arg	Call	Fixed	Use *rooted* pointer	✓
6	Logger::ICEvent	Arg	Call	Fixed	Use *rooted* pointer	✓
7	Logger::MapEvent	Arg	Call	Fixed	Use *rooted* pointer	✗
8	Map::DeprecateTransitionTree	Arg	Call	Submitted	-	✗
9	MapUpdater::ConstructNewMap	Call	Call	Fixed	Use *rooted* pointer	✓
10	NativeRegExpMacroAssembler::CheckStackGuardState	Arg	Call	Won't Fix	-	✗
11	ObjectDeserializer::Deserialize	Call	Store	Fixed	Remove *compact-gc*	✗
12	PartialDeserializer::Deserialize	Call	Call	Fixed	Remove *compact-gc*	✗
13	SourceTextModule::AddAsyncParentModule	Arg	Call	Fixed	Use *rooted* pointer	✗
14	ToPropertyDescriptorFastPath	Call	Call	Fixed	Relocate *compact-gc*	✗
15	V8HeapExplorer::AddEntry	Arg	Store	Fixed	Remove *compact-gc*	✓

In SpiderMonkey, rootAnalysis detected 30 *use-after-cgc* cases, which were all false-positives, while CGSan did not find any bugs. Although rootAnalysis employed target-specific heuristics, e.g. that SpiderMonkey suppresses *compact-gc* with `AutoSuppressGC`, to reduce false-positives, it had more false-positives than CGSan due to its path-insensitivity. For example, rootAnalysis concluded that `JSDependentString::new_` had a *def-cgc-use* pair where the implicit *compact-gc* function was `AllocateString<JSDependentString, js::NoGC>`, which did not trigger *compact-gc* internally. However, CGSan recognized that `AllocateString<JSDependentString, js::NoGC>` did not trigger *compact-gc* and concluded there is no *def-cgc-use* pair in `JSDependentString::new_`.

In the case of V8, gcmole detected 8 *use-after-cgc* cases. After manual verification, these were classified as 4 false-positives and 4 bugs. A false-positive was marked as "Won't Fix" and 3 false-positives were caused by path-insensitivity of gcmole. For instance, even though there were only *def-cgc-use* nodes in `FutexEmulation::Wait` function without CFG edges connecting them, gcmole concluded it was a *use-after-cgc* bug. All 4 bugs were also found by CGSan. In Table 6, `Prev.` column presents which cases were also discovered by gcmole. 10 cases were not found by gcmole due to its wrong heuristics and incompleteness. For example, gcmole missed case 1 and case 13 due to the wrong `no_gc` mark and missing to check `this`, which compilers implicitly passed.

8 Patch Strategies

Based on our study and bug fixes by developers, we summarize three general patch strategies for *use-after-cgc* bugs. Table 6 shows which strategy developers employed to patch.

```
 template <typename Derived, typename Shape> ExceptionStatus
   BaseNameDictionary<Derived, Shape>::CollectKeysTo(
     Handle<Derived> dictionary, KeyAccumulator* keys) {
   ...
-    Derived raw_dictionary = *dictionary;
     for (InternalIndex i : dictionary->IterateEntries()) {
       Object k;
+      Derived raw_dictionary = *dictionary;
       ...
```

(a) A patch of the bug in Figure 7a by relocating *Def* and *GC*.

```
 void SourceTextModule::AddAsyncParentModule(
-  Isolate* isolate, Handle<SourceTextModule> module) {
+  Isolate* isolate, Handle<SourceTextModule> module,
+  Handle<SourceTextModule> parent) {
+  Handle<ArrayList> async_parent_modules(
+      module->async_parent_modules(), isolate);
   Handle<ArrayList> new_array_list =
-      ArrayList::Add(
-        isolate, handle(async_parent_modules(), isolate), module
-      );
-  set_async_parent_modules(*new_array_list);
+      ArrayList::Add(isolate, async_parent_modules, parent);
+  module->set_async_parent_modules(*new_array_list);
 }
```

(b) A patch of the bug in Figure 7b by using the *rooted* pointer.

Figure 8: Two patches of *use-after-cgc* bugs in Figure 7.

Remove *Compact-gc*. An intuitive patch strategy is to remove *compact-gc*. If there is no *compact-gc*, objects in *unrooted* pointers will not be moved, and so pointers will not be invalidated. However, this strategy reduces memory efficiency due to the absence of *compact-gc*.

Use *Rooted* Pointer. Another intuitive patch strategy is to use *rooted* pointers instead of *unrooted* pointers. During *compact-gc*, *rooted* pointers will be updated and safe to use afterward. Figure 8b shows an example, which is the patch for Figure 7b. Developers changed `AddAsyncParentModule` to a static function and get a *rooted* pointer, `module`, instead of an *unrooted* pointer, `this`. This mitigates *use-after-cgc* bugs but increases the performance overhead as mentioned in §2.1.

Relocate *Compact-gc*. The most performant patch strategy is relocating *compact-gc*. This generally implies moving *compact-gc* function calls to either before the *unrooted* pointer definition or after the *unrooted* pointer use. For instance, the patch for Figure 7a in Figure 8a transfers an *unrooted* pointer definition for `raw_dictionary` from the outside of the loop to inside so that there is no longer *compact-gc* function call between *unrooted* pointer definition and use.

9 Discussion

We discuss our limitations due to our assumptions and issues related to supporting other targets.

Limitations. As mentioned in §7.4, CGSan has some limitations due to its assumptions. First, we assume that *compact-gc* functions must move all *unrooted* pointers, but there can be *unrooted* pointers that are not moved even after *compact-gc* like `AllocateRawWithImmortalMap`, which is marked as "Won't Fix". Also, when the memory context that *unrooted* pointers belong to is different from the memory context where *compact-gc* performed, the *unrooted* pointers will not be moved. However, we assume that they share the same context because most programs will have one memory context. Second, other threads can trigger *compact-gc* functions, but we lack a scalable lockset analysis and multi-threading-capable static symbolic execution to handle multiple threads. And we mitigated the path explosion problem during the analysis but there were timeout cases, which means that the path explosion problem still existed. We leave overcoming our limitations as future work.

Supporting Other Targets. Theoretically, *use-after-cgc* bugs can exist in any software system that employs *compact-gc*. To apply CGSan to them, it requires clear identifications of the memory cell type and explicit *compact-gc* functions. Fortunately, as we described in §2.1, these systems typically have a memory cell type and explicit *compact-gc* functions. In addition, we proved the possibility of supporting multiple targets by evaluating CGSan on two independent code bases, V8 and SpiderMonkey.

10 Related Work

In this section, we survey related works in static analysis for bug finding. We refer the reader to §2 for symbolic taint analysis and directed symbolic execution.

Many static analyzers including CGSan focus on specific bugs. For example, Wang *et al.* [28] and Deadline [31] are designed to detect double-fetch bugs in the OS kernel based on pattern matchings and the specialized symbolic checking, respectively. Also, some analyzers extract constraints from source code and report violating cases. APISan [35] finds API misuse bugs by inferring API usage from source code, LRSan [29] detects security checks in the OS kernel and finds lacking-recheck bugs, and CRIX [17] infers which variables require checks and detects missing check bugs. In addition, EECatch [25] identifies errors, infers their severity level, and detects that error handling code is over the severity of the corresponding error in the Linux kernel. Also, K-MELD [13] infers locations where allocated memory objects are expected to be released by a new ownership reasoning mechanism and finds memory leaks in the Linux kernel.

There have been several static analyzers to detect *use-after-free* bugs. They perform a pointer analysis to check whether accessed pointers are in a freed pointer set. The main challenge is to make pointer analysis scalable and precise. CRED [33] improves its scalability by spatio-temporal context reduction, TAC [32] mitigates imprecision by predicting *use-after-free* related pointer aliases with machine-learning techniques, and DCUAF [5] detects concurrent *use-after-free* bugs in the Linux kernel by local-global analysis and summary-based lockset analysis. However, they are not applicable for discovering *use-after-cgc* bugs in practice as we described in §1.

In addition, there have been many extensible frameworks for detecting bugs. µchex [9] is a scalable framework based on parsing only what analysts want to analyze instead of an entire language, which is scalable but not precise. Using µcheck, Brown *et al.* [8] discovered various bugs by pattern matching, including *use-after-free* bugs caused by garbage collection in JS bindings. Sys [10] provides a framework that detects bugs by static analysis and checks them by symbolic execution. This is similar to ours but we improve scalability by novel techniques, directed ICFG construction, and directed scheduling.

11 Conclusion

Compacting garbage collection may introduce a new kind of *use-after-free* bug, named *use-after-cgc*, if an *unrooted* pointer defined before *compact-gc* is used after *compact-gc*. In this paper, we have presented CGSan, a precise and scalable static analyzer for detecting *use-after-cgc* bugs. CGSan finds *use-after-cgc* bug candidates, which are *def-cgc-use* pairs, by intra-procedural static symbolic taint analysis. It then checks their feasibility of *def-cgc-use* pairs by under-constrained directed symbolic execution. CGSan also constructs the directed ICFGs and employs the directed scheduling to be more scalable without losing the precision. We evaluated CGSan against Google V8 and Mozilla SpiderMonkey, and we found 13 *use-after-cgc* bugs in a few hours and reported them to the vendors. We also showed that our optimization techniques improve scalability while preserving precision. Lastly, we summarized three general patch strategies for *use-after-cgc* bugs based on our study and patches by the vendors.

Acknowledgement

We thank our shepherd, Chengyu Song, and the anonymous reviewers for their helpful comments. We are also grateful to Insu Yun, Tim Becker, and Tyler Nighswander for fruitful feedback.

References

[1] gcmole. https://github.com/v8/v8/tree/master/tools/gcmole.

[2] rootanalysis. https://github.com/mozilla/gecko-dev/tree/master/js/src/devtools/rootAnalysis.

[3] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2006.

[4] Domagoj Babic, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. Statically-directed dynamic automated test generation. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 12–22, 2011.

[5] Jia-Ju Bai, Julia Lawall, Qiu-Liang Chen, and Shi-Min Hu. Effective static analysis of concurrency use-after-free bugs in linux device drivers. In *Proceedings of the USENIX Annual Technical Conference*, pages 255–268, 2019.

[6] Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, and David Brumley. Your exploit is mine: Automatic shellcode transplant for remote exploits. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 824–839, 2017.

[7] Joel F Bartlett. Compacting garbage collection with ambiguous roots. In *ACM SIGPLAN Lisp Pointers*, pages 3–12, 1988.

[8] Fraser Brown, Shravan Narayan, Riad S. Wahby, Dawson Engler, Ranjit Jhala, and Deian Stefan. Finding and preventing bugs in javascript bindings. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 559–578, 2017.

[9] Fraser Brown, Andres Nötzli, and Dawson Engler. How to build static checking systems using orders of magnitude less code. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 143–157, 2016.

[10] Fraser Brown, Deian Stefan, and Dawson Engler. Sys: A static/symbolic tool for finding good bugs in good (browser) code. pages 199–216, 2020.

[11] Jacques Cohen and Alexandru Nicolau. Comparison of compacting algorithms for garbage collection. *ACM Transactions on Programming Languages and Systems*, 5(4):532–553, 1983.

[12] Heming Cui, Gang Hu, Jingyue Wu, and Junfeng Yang. Verifying systems rules using rule-directed symbolic execution. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 329–342, 2013.

[13] Navid Emamdoost, Qiushi Wu, Kangjie Lu, and Stephen McCamant. Detecting kernel memory leaks in specialized modules with ownership reasoning. In *Proceedings of the Network and Distributed System Security Symposium*, 2021.

[14] Patrice Godefroid, Michael Y. Levin, and David A Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium*, pages 151–166, 2008.

[15] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983.

[16] Kangjie Lu and Hong Hu. Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 1867–1881, 2019.

[17] Kangjie Lu, Aditya Pakki, and Qiushi Wu. Detecting missing-check bugs via semantic- and context-aware criticalness and constraints inferences. In *Proceedings of the USENIX Security Symposium*, pages 1769–1786, 2019.

[18] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S Foster, and Michael Hicks. Directed symbolic execution. In *International Static Analysis Symposium*, pages 95–111, 2011.

[19] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 2019.

[20] Jiang Ming, Dinghao Wu, Jun Wang, Gaoyao Xiao, and Peng Liu. StraightTaint: Decoupled offline symbolic taint analysis. In *Proceedings of the International Conference on Automated Software Engineering*, pages 308–319, 2016.

[21] Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. TaintPipe: Pipelined symbolic taint analysis. In *Proceedings of the USENIX Security Symposium*, pages 65–80, 2015.

[22] MITRE. CVE-2019-13696. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-13696, 2019.

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

[24] Ben Niu and Gang Tan. Modular control-flow integrity. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 577–587, 2014.

[25] Aditya Pakki and Kangjie Lu. Exaggerated error handling hurts! an in-depth study and context-aware detection. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 1203–1218, 2020.

[26] David A. Ramos and Dawson Engler. Under-constrained symbolic execution: Correctness checking for real code. In *Proceedings of the USENIX Security Symposium*, pages 49–64, 2015.

[27] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 317–331, 2010.

[28] Pengfei Wang, Jens Krinke, Kai Lu, and Gen Li. How double- fetch situations turn into double-fetch vulnerabilities: A study of double fetches in the linux kernel. In *Proceedings of the USENIX Security Symposium*, pages 1–16, 2017.

[29] Wenwen Wang, Kangjie Lu, and Pen-Chung Yew. Check it again: Detecting lacking-recheck bugs in os kernels. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 1899–1913, 2018.

[30] Paul R Wilson. Uniprocessor garbage collection techniques. In *International Workshop on Memory Management*, pages 1–42, 1992.

[31] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. Precise and scalable detection of double-fetch bugs in os kernels. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 661–678, 2018.

[32] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. Machine-learning-guided typestate analysis for static use-after-free detection. In *Proceedings of the Annual Computer Security Applications Conference*, pages 42–54, 2017.

[33] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *Proceedings of the International Conference on Software Engineering*, pages 327–337, 2018.

[34] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: a practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the USENIX Security Symposium*, pages 745–761, 2018.

[35] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. APISan: Sanitizing API usages through semantic cross-checking. In *Proceedings of the USENIX Security Symposium*, pages 363–378, 2016.