Automated Use-After-Free Detection and Exploit Mitigation: How Far Have We Gone?

Binfa Gui, Wei Song, *Senior Member, IEEE*, Hailong Xiong, and Jeff Huang, *Member, IEEE*

Abstract—C/C++ programs frequently encounter memory errors, such as Use-After-Free (UAF), buffer overflow, and integer overflow. Among these memory errors, UAF vulnerabilities are increasingly being exploited by attackers to disrupt critical software systems, leading to serious consequences, such as remote code execution and data breaches. Researchers have proposed dozens of approaches to detect UAFs in testing environments and to mitigate UAF exploit in production environments. However, to the best of our knowledge, no comprehensive studies have evaluated and compared these approaches. In this paper, we shed light on the current UAF detection and exploit mitigation approaches and provide a systematic overview, comprehensive comparison, and evaluation. Specifically, we evaluate the effectiveness and efficiency of publicly available UAF detection and exploit mitigation tools. The experimental results show that static UAF detectors are suitable for detecting intra-procedural UAFs but are not sufficient to detect inter-procedural UAFs in real-world programs. Dynamic UAF detectors are still the first choice for detecting inter-procedural UAFs. Our evaluation also demonstrates that the runtime overhead of existing UAF exploit mitigation tools is relatively stable whereas the memory overhead may vary dramatically with respect to different programs. Finally, we envision potential valuable future research directions.

Index Terms—Use-After-Free, vulnerability detection, exploit mitigation, program analysis, survey

1 Introduction

Use-after-free (UAF), that is, dangling pointer dereferences on the freed memory object, is top security vulnerability [1], [2]. Previous research shows that in the NVD database, 80.14% of UAFs from 2007 to 2016 are rated as critical or high severity [3], with an increasing number of UAFs every year. Meanwhile, UAFs are increasingly being exploited by malicious attackers for arbitrary code execution, silent data corruption, or data leaks [4]. In languages with manual memory management, such as C/C++ [5], [6], which is indispensable for developing efficient low-level system programs, programmers do not always allocate and free memory properly. As a result, UAFs can exist in a variety of real-world programs from operating systems and web browsers to desktop and mobile applications, exposing a huge attack surface.

To date, dozens of techniques have been proposed to detect UAF in testing environments and to prevent UAF exploits in production environments. In terms of detection, most existing solutions rely exclusively on dynamic analysis by instrumenting programs at the IR-level [6], [7], [8], [9], [10], [11], [12] or binary level [13], [14], [15], [16], [17], [18], [19]. While maintaining zero or low false alarms, dynamic analysis approaches have low code coverage and high performance overhead. Few static analysis approaches are dedicated to detecting UAFs [3], [20], [21], [22], although some general-purpose tools are available [23], [24], [25], [26], [27], [28]. In terms of prevention (exploit mitigation), most existing solutions attempt to protect against UAF exploits by eliminating dangling pointers [4], [29], [30], [31], [32], [33] or implicitly checking every pointer dereference [34], [35]. An alternative approach is to change the memory allocator to mitigate UAF exploits [2], [36], [37], [38], [39], [40], [41]. Some solutions [42], [43], [44] focus on garbage collection using reference counting mechanisms to automatically release memory objects. Other studies attempt to limit the damage of UAF exploits [45], [46], [47], [48], [49]. However, no comprehensive study on this topic to evaluate and compare existing solutions exists.

To bridge this gap, we provide a comprehensive classification and comparison of currently available UAF detection and exploit mitigation techniques. Beginning with the background on UAF vulnerabilities, we organize current technologies and approaches into a taxonomy according to their objectives, i.e., detection or exploit mitigation. We then further classify, compare and evaluate these techniques. Our goal is not only to provide a comparison of current UAF mitigation approaches but also to identify potential valuable research directions.

To the best of our knowledge, this work is the first study in this field to comprehensively compare and evaluate current UAF detection and prevention approaches. Although the most recent study [50] reviews approaches for detecting memory security issues, it does not provide a systematic overview of UAF detection and exploit mitigation techniques. To better evaluate the available detection and prevention approaches, we have created and collected a set of

• *Binfa Gui, Wei Song, and Hailong Xiong are with the School of Computer Science and Engineering, Nanjing University of Science and Technology, Nanjing 210094, China.*
E-mail: guibinfa@gmail.com, wsong@njust.edu.cn, 1512960062@qq.com.
• *Jeff Huang is with the Parasol Laboratory, Texas A&M University, College Station, TX 77843-3112 USA. E-mail: jeff@cse.tamu.edu.*

Manuscript received 2 April 2021; revised 21 September 2021; accepted 18 October 2021. Date of publication 21 October 2021; date of current version 14 November 2022.
This work was supported by the National Natural Science Foundation of China under Grant 61761136003.
(Corresponding author: Wei Song.)
Recommended for acceptance by L. Mariani.
Digital Object Identifier no. 10.1109/TSE.2021.3121994

benchmarks and made them publicly available. According to our extensive experimental evaluation, we have the following overall findings:

1) Static (UAF) detectors can effectively detect intra-procedural UAFs but are not sufficient to detect inter-procedural UAFs in real-world programs. Dynamic (UAF) detectors are still the first choice for detecting inter-procedural UAFs.
2) Although the memory reuse delay mechanism used by dynamic detectors can help to detect more UAFs, it does not fundamentally solve the limitation of dynamic detectors. These dynamic detectors still miss UAFs in our evaluation.
3) Existing work on UAF exploit mitigation focuses more on reducing the runtime overhead, whereas the memory overhead reduction is not regarded as a big concern, because the memory overhead on some programs is still very large.

The remainder of this paper is organized as follows: We introduce the background of UAFs and discuss how we collected and filtered related work in Section 2. We provide an overview of UAF detection and exploit mitigation approaches in Section 3. We present the available UAF detection and exploit mitigation approaches in the form of a taxonomy in Sections 4 and 5, respectively. Section 6 presents an empirical evaluation of UAF detection techniques and publicly available tools. Section 7 discusses potential valuable directions for future work. Finally, Section 8 concludes this paper.

2 Background

In this section, we first give a brief introduction to UAFs before describing the methodology used to conduct the survey.

2.1 Temporal Errors and UAFs

The scope of temporal errors is large: UAFs are just one of the four types, and the other three types of temporal errors are double-free, use-after-scope, and use-after-return. In this case, a UAF vulnerability refers to accessing the freed heap object through a dangling pointer, a pointer that still points to the old object released. A double-free refers to releasing the freed heap object again through the dangling pointer. Accessing a stack variable beyond its scope through a dangling pointer is called a use-after-scope vulnerability. Moreover, a use-after-return vulnerability refers to accessing an invalid stack variable through a dangling pointer after the function returns. Broadly speaking, temporal errors also include uninitialized reads [51].

From a broad perspective, UAFs are essentially equivalent to temporal errors (when temporal errors such as uninitialized reads are not considered), and they are interchangeable [8], [32], [52]. In this case, these terms both refer to the same memory error, which occurs when the program accesses a freed memory object through a dangling pointer. The freed memory accessed may lie on the heap or stack.

2.2 Literature Collection

We describe the strategy we followed to collect UAF-related work and the criteria for inclusion in this paper.

We use the following two approaches to collect work related to UAF. The main approach is to search online databases using different keywords to find UAF-related research. If a keyword appears in the title, abstract, or text of the paper provided in the search results, we manually verify whether the paper is actually related to UAF. The keywords we use are "Use-After-Free", "UAF", "temporal error", and "dangling pointer". The online databases we search include *Google Scholar*,[1] *DBLP*,[2] *SEMANTIC SCHOLAR*,[3] and other online academic databases, including *ACM Digital Library*,[4] *IEEE Xplore*,[5] *Springer*,[6] *Wiley*,[7] and *Elsevier*.[8] Another approach is to read the most recent (2019-2020) top-tier international conferences (including IEEE S&P, CCS, USENIX Security, USENIX ATC, PLDI, ICSE, ESEC/FSE, ASE, ISSTA) papers related to UAF detection or prevention, and then search UAF-related papers from their references and find UAF-related work that cites these papers. We use these two approaches iteratively until we cannot find new papers or approaches related to UAFs. Note that UAF-related work is constantly emerging: we collect UAF-related work that exists prior to March 2021.

The collection process obtained almost all work related to UAFs. We then judge these studies to determine whether to include them in this paper based on the following factors: the reputation of the journal or conference in which the paper was published, the innovation (subjectively judged) of the paper, and the number of citations (should be more than 10 if the paper was published three years ago) of the paper, the publication year (mostly after the year 2000) of the paper. Note that we focus on separate C/C++ programs and thus the papers related to the Linux kernel and assembly are excluded [53], [54], [55], [56], [57], [58], [59], [60], [61], which are beyond our scope. In addition, we exclude papers that we believe are not of high importance.

3 Overview: Taxonomy and Approaches

On the basis of the considerable differences in their goals and application scenarios, existing approaches can be classified into two categories: detection [3], [9], [10], [20] and exploit mitigation strategies [34], [36], [37], [38], [39], [62], [63], [64]. Detection approaches are designed to identify UAFs and help developers fix bugs in their programs, regardless of whether there is an attacker or whether the UAFs can be exploited. Exploit mitigation approaches employ various strategies to prevent potential attackers from exploiting UAFs in a program. Moreover, compared to mitigation approaches, detection approaches have a built-in error reporting mechanism, a high performance budget, and a high false alarm tolerance [31], [50]. In this section, we

1. https://scholar.google.com/
2. https://dblp.uni-trier.de/
3. https://www.semanticscholar.org/
4. https://dl.acm.org/
5. https://ieeexplore.ieee.org/Xplore/home.jsp
6. https://www.springer.com/gp
7. https://onlinelibrary.wiley.com/
8. https://www.elsevier.com/

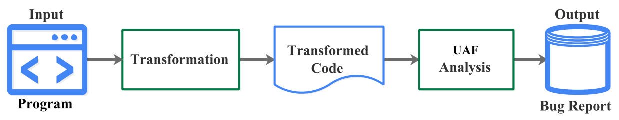

Fig. 1. Static analysis process.

first provide an overview of UAF detection and exploit mitigation approaches.

3.1 Detection

According to when detection occurs, detection approaches can be classified into two categories: static analysis and dynamic analysis. Static analysis finds potential UAFs in a program by analyzing the source or machine code without executing the program, while dynamic analysis reports potential UAFs during program execution. Since static analysis and dynamic analysis are quite different, we summarize the steps followed by the approaches in these two categories. The general process in each category is a summary of the steps that are common to existing approaches.

3.1.1 *Static Analysis*

Fig. 1 presents a generic static analysis process for detecting UAFs. A brief description of each phase is provided below.

Transformation. Transformation means to transform the program to obtain program representations derived directly from the source or binary code without executing the program. If the program's source code is available, the source code can be transformed into a program representation, such as LLVM IR [3], symbolic representation [25], abstract syntax tree [27], [28], and other intermediate representations [21], [26]. If only the binary code is available, then the binary code must be disassembled into analyzable executable intermediate representations, such as the abstract memory model [20].

UAF Analysis. After completing the transformation phase, the obtained transformed code, i.e., program representations, is then used for UAF analysis. If the transformed code is derived using symbolic execution [25], it can be represented as a series of constraints and then be resolved with various SMT solvers. If the transformed code takes the form of intermediate representations (IR) [3], [21] or other program representations [20], [26], [27], [28], then various analyses can be performed based on the transformed code, such as value-flow analysis [65] and control-flow analysis [66], to obtain the desired UAF information.

Bug Report. After the UAF analysis phase is completed, whether the constraint solving solution or another analysis approach is followed, the desired UAF information is obtained and then used for further analysis. Next, further analysis of UAFs (manual or automatic) is conducted to generate and report the final UAF vulnerabilities.

3.1.2 *Dynamic Analysis*

Fig. 2 presents a generic dynamic analysis process for detecting UAFs. The solid and dashed lines correspond to two different implementations: static instrumentation and dynamic instrumentation. The main difference between the two is whether the program needs to be executed when it is instrumented. An overview of each implementation is provided as follows.

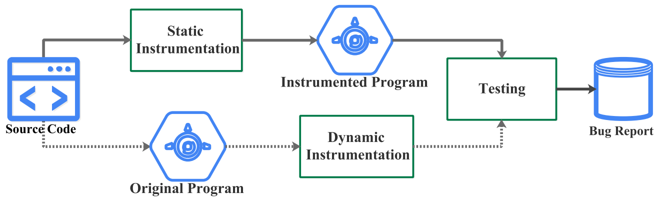

Fig. 2. Dynamic analysis process.

Static Instrumentation. Static instrumentation falls into two categories based on whether source code is needed: compile-time instrumentation and static binary rewriting (a.k.a static binary translation).

Compile-time instrumentation [7], [9], [10], [11] occurs before the program executes, that is, the compiler compile and link phase. A typical compile-time instrumentation approach consists of two parts: instrumentation module and runtime library. The instrumentation module modifies the code to add memory checks, and the runtime library provides runtime support for memory checks instrumented by the instrumentation module. In general, the source code is analyzed to obtain the target instrument locations; then, the corresponding memory checks are instrumented. Next, the malloc, free, and related functions are replaced with the runtime libraries in the compiler link phase. After that, we obtain the instrumented program. Finally, we execute the program continuously and take the test case as the input. In addition, if execution traces are recorded, the execution traces can be analyzed to determine whether there are other potential UAFs.

Static binary rewriting (a.k.a static binary translation) [13], [14] occurs at the binary level before program execution. The main idea of static binary rewriting is to recover sufficient information from the binary program and then use this information to obtain the target instrumentation locations. Memory checks are inserted at the appropriate locations and code is rewritten to generate an executable instrumented program. The next steps are similar to compile-time instrumentation, which executes the program continuously and takes test cases as inputs. In contrast to the subsequent dynamic binary translation, static binary rewriting has no runtime parsing and translation overhead and provides performance close to that of compile-time instrumented programs.

Dynamic Instrumentation. According to the different techniques implemented, dynamic instrumentation can be classified into two categories: dynamic binary translation and library interposition.

Dynamic binary translation (DBT) [15], [18] instruments the program during execution, and there is no need to prepare the program in any way. A typical DBT-based approach is that when the program is executed, it first translates the instructions executed by the program, then inserts the instrument code into the translated instructions, and finally re-executes the new instructions obtained. Therefore, when used to detect UAFs, the DBT-based approach is executed first, followed by the target program, while continuously taking test cases as inputs.

Library interposition [17] differs from dynamic binary translation in that library interposition uses library interposers to intercept calls to library functions. A library interposer is a shared library that can be preloaded into the program using the environment variables provided by the Linux operating system. The method can then intercept, monitor and operate all the library function calls of interest in the program. When used to detect UAFs, the library interposer intercepts all function calls related to memory allocation and calls the memory allocation implemented by the library interposer. Whenever a program requests or frees a memory object, the library interposer can perform memory checks on all freed objects that exist in the quarantine. If a modification to a freed memory object is detected, the library interposer reports a UAF.

3.2 Exploit Mitigation

As mentioned earlier, exploit mitigation approaches aim to prevent UAFs from being exploited at runtime. In addition, most of existing exploit mitigation approaches only look at heap UAFs (including double-frees), while stack UAFs (that is, use-after-scopes and use-after-returns) are ignored. The reasons are as follows. First, stack UAFs are less likely to exist in real-world programs because they can be eliminated in advance through static checks, such as escape analysis [38]. Second, since stack memory is allocated and released very frequently, it is difficult for an attacker to reallocate the freed stack memory. As a result, heap UAFs, rather than stack UAFs, are the most commonly exploited and the most difficult to defend against [29]. Generally speaking, UAFs that are vulnerable to exploitation usually need to meet the following three conditions:

1) The program generates dangling pointers. A live pointer becomes a dangling pointer [67] once the memory object pointed to is freed.
2) An attacker can allocate a new memory object on the freed memory to which the dangling pointer points.
3) The dangling pointer is later used to access previously freed memory and perform some read and/or write operations on the previously freed memory.

When a UAF vulnerability occurs, one or more of the following situations may occur in the program:

1) If the freed memory is not reallocated and then accessed through the dangling pointer, the program may continue to function normally.
2) If the freed memory is reallocated (without the attacker involved), which results in changes to the memory contents, and the program accesses the previously freed memory through the dangling pointer, the program may function strangely and even crash.
3) Worst of all, if attackers manage to reallocate the freed memory and the program accesses the previously freed memory through the dangling pointer, serious consequences may occur, such as control-flow hijacking and data-based attacks [50].

Therefore, if we break one or more of the above conditions, attackers cannot easily exploit a UAF, thus minimizing the harm caused by UAFs. Moreover, by preventing attackers from hijacking control-flows or performing data-based attacks, we can reduce the damage caused by successful exploitation of a UAF. Dozens of] approaches are presented in the literature to protect programs from UAFs during execution [4], [29], [34], [36], [37], [38], [39], [62], [63], [64], [68], [69]. They fall into the following categories:

- *Dangling pointer elimination*. This category of approaches is devoted to eliminating dangling pointers and is characterized by detecting the presence of dangling pointers and invalidating them when appropriate. Such approaches fall into three groups: approaches based on taint tracking [29], approaches based on meta-data tracking [4], [30], [31], [70], and approaches based on intermediate pointers [69], [71].
- *Pointer dereferences checking*. This category performs memory checks on each pointer dereference and crashes the program when a UAF occurs. Such approaches fall into three groups: per-pointer metadata via fat pointer [62], [72], [73], [74], lock-and-key identifier checking [34], [52], [75], [76], [77], and hardware-based checking [32], [33], [35], [78], [79].
- *Changing memory allocators*. These approaches change the memory allocation strategy to make it difficult for attackers to exploit UAF vulnerabilities. Such approaches can be further clustered into three groups: random heap allocation [37], [39], [40], [80], [81], [82], [83], [84], [85], type-safe heap allocation [36], [86], and isolated heap allocation [2], [38], [87], [88].
- *Improve garbage collection*. These approaches focus on using various techniques to release memory objects at the appropriate time. Such approaches can be further categorized into three groups: safe C dialects [63], [89], [90], use of garbage collection libraries [42], [43], delay of memory reuse [44], [87], [91], [92] and improvement of object life-cycle management [44], [72], [91].
- *Limiting exploitation damage*. These approaches limit the damage caused by the successful exploitation of UAFs. To achieve this, these approaches take different measures to prevent virtual table hijacking [45], [46], [47], [48], [49], [64], [68], [93], [94], [95].

4 Detection

In this section, we review the state-of-the-art approaches for detecting UAFs, which are summarized and classified in Table 1 according to our proposed taxonomy. The first three columns of Table 1 show the categories, names, and availability of the tools; columns 4 and 5 summarize the languages and platforms of the programs that can be analyzed by the tools; and the last two columns describe the user interface of the tools and the key technical means.

4.1 Static Analysis Approaches

Static analysis approaches can be further classified into two groups: static source code analysis and static binary code analysis. The former includes analyzing programs based on source code and analyzing program representations derived directly from source code, such as value flow graph and

TABLE 1
Summary and Classification of Existing UAF Detectors

Category	Name	Availability	Language Support	Platform	User Interface	Means
Static Source Code Analysis	CBMC [25]	open-source	C/C++	Windows/ Linux/Unix	command line tool	bounded model checking; use constraint solving
	Coccinelle [26]	open-source	C	Linux/Unix	command line tool	conduct pattern matching based on data flow analysis
	Clang static analyzer [27]	open-source	C/C++	Windows/ Linux/Unix	command line tool	conduct control flow-sensitive analysis; inter-procedural analysis using function inling; use multiple checkers for path sensitive analysis
	Frama-C [23]	open-source	C	Linux/Unix	command line tool	conduct program safety verification based on value-flow analysis
	Klee [24]	open-source	C/C++	Linux/Unix	command line tool	based on symbolic execution uses search heuristics to get high code coverage
	TscanCode [28]	open-source	C/C++	Windows/ Linux/Unix	command line tool/ GUI available	conduct pattern matching based on control flow analysis
	UAFChecker [96]	unavailable	C/C++	Linux/Unix	not mentioned	inter-procedural analysis using function inlining and function summary
	Tac [21]	unavailable	C/C++	Linux/Unix	not mentioned	pointer alias analysis using machine learning
	CRed [3]	unavailable	C	Linux/Unix	not mentioned	use spatial-temporal context reduction; perform path-sensitive analysis
Static Binary Code Analysis	GUEB [20]	open-source	C/C++	Linux/Unix	command line tool	conduct value analysis; inter-procedural analysis using function inling
	UAFDetector [97]	unavailable	C/C++	Linux/Unix	command line tool	consider indirect jumps to construct CFG; inter-procedural analysis using function summary
Evidence-based Dynamic Analysis	Purify [16]	commercially available	C/C++	Windows/ Linux/Unix	command line tool/ GUI available	instrumentation based on object code insertion
	Valgrind [18]	open-source	C/C++	Linux/Unix	command line tool	instrumentation based on Valgrind; multi-level shadow mapping
	Dr.Memory [15]	open-source	C/C++	Windows/ Linux/Unix	command line tool	instrumentation based on DynamoRIO; multi-level shadow mapping
	DoubleTake [17]	open-source	C/C++	Linux/Unix	command line tool	execute epoch twice to detect UAF
	Mudflap [7]	unavailable	C/C++	Linux/Unix	command line tool	validity predicate assertions
	ASan [10]	open-source	C/C++	Windows/ Linux/Unix	command line tool	direct shadow mapping
	BASan [13]	unavailable	C/C++	Linux/Unix	command line tool	instrumentation based on static binary rewriting
	Frama-C/E-ACSL [8], [98], [99]	open-source	C/C++	Linux/Unix	command line tool	runtime verification based on Frama-C
	TSan [11], [12]	open-source	C/C++	Linux/Unix	command line tool	conduct data race detection
	UAFL [6]	unavailable	C/C++	Linux/Unix	command line tool	fuzzing with ASan
Prediction-based Dynamic Analysis	UFO [9]	open-source	C/C++	Linux/Unix	command line tool	use constraint solving; extended maximal thread causality model
	ConVul [100]	unavailable	C/C++	Linux/Unix	not mentioned	detect UAF based on exchangeable events

control flow graph. The latter involves analyzing programs based on machine code, including program intermediate representations derived from machine code.

4.1.1 Static Source Code Analysis

Due to the separation of memory use and free pattern promoted by object-oriented and/or procedural programming, static source code analysis faces several challenges:

1) How to address the exponential growth of the program paths with the size of the program, namely, how to efficiently infer every pointer's malloc site, use site, and free site in the program.
2) How to efficiently perform pointer alias analysis, i.e., two or more different pointers may point to the same memory object.
3) How to effectively determine all the path feasibility of UAF vulnerabilities, where pointers pointing to the same memory object has a path from the malloc site to the free site and finally the use site.

Because of these challenges, only a few general-purpose tools [23], [25], [26], [27], [28] and a few specific tools [3], [21] can be used to detect UAFs in programs.

Bounded Model Checking. CBMC [25] is a bit-precise bounded model checking tool that combines model checking [101] with satisfiability solving. CBMC first unrolls each loop into a fixed boundary before translating the program into a program annotated with UAF assertions. Then, it treats all possible program paths as constraints and solves them using an SMT solver to determine whether the program contains UAF constraint violations. Despite its high precision, CBMC only scales to small programs [102] due to the limitation of constraint solving.

Symbolic Execution. Klee [24] can detect UAFs using symbolic execution. Klee models memory with bit-level accuracy, supports different constraint solvers for constraint solving, and uses search heuristics to obtain higher code coverage. However, its capability is also limited by the path explosion problem and constraint solutions. Therefore, although Klee uses search heuristics to reduce path exploration, it only scales to small or slightly larger programs [103].

Pattern Matching. Coccinelle [26] employs a given pattern to analyze and certify vulnerabilities in C programs. Although it achieves high scalability, it also reduces the precision of vulnerability detection. Therefore, the UAFs detected by Coccinelle contain a large number of, or even all, false alarms due to the poor pointer analysis and the nature of pattern matching. TscanCode [28] extends CppCheck [104] and has the ability to detect UAFs in programs using specific syntactic patterns. TscanCode meets the difficulties in accurately identifying UAFs in real-world programs due to the lack of pointer analysis and insufficient control-flow analysis.

Abstract Interpretation. Clang static analyzer [27] performs control-flow analysis, path-sensitive analysis, and inter-procedural analysis to detect general errors, including UAFs. To achieve path-sensitiveness, it uses a variety of checkers to infer possible paths and establishes a program execution state graph over time [105], [106]. To scale to large programs, it balances effectiveness and scalability and performs limited inter-procedural analysis through function inlining technology. As a result, only a small part of the program paths is analyzed. In addition, it only performs simple pointer analysis and conservative loop analysis. Hence, it refrains from reporting excessive false alarms but inevitably misses real inter- and intra-procedural UAFs. Frama – C [23] is a value flow analysis framework that implements program safety verification. When a program does not respect UAF constraints, UAF errors are reported. However, the pointer analysis capability of Frama – C is very limited: it suffers from high false negative rates.

Hybrid Analysis. UAFChecker [96] is designed to detect UAFs, which uses function inlining and function summary techniques for inter-procedural analysis to ensure accuracy. It first builds a finite state machine to track the state of each pointer, that is, to check whether the pointer or its alias is used again after deallocating its reference object. This step can identify multiple candidate UAFs. UAFChecker then uses symbolic execution to check the satisfiability of path constraints of candidate UAFs and eliminate false positives. The results based on the Juliet Test Suite [107] show that it inevitably misses many real UAFs. Additionally, it is only applicable to small programs due to the limitations of constraint solving and the difficulty of inter-procedural analysis caused by exponential path explosion.

Pointer Analysis. Tac [21] combines pointer analysis and machine learning to detect UAFs. It first uses the pointer analysis tool SVF [65] to obtain pointer information in the program and then establishes a finite state machine to determine whether UAFs exist in the program. Tac uses a support vector machine (SVM) to improve the effectiveness of pointer alias analysis to reduce false alarms in the detection results. Although Tac scales to large programs, it cannot guarantee an effective reduction in false alarms in real-world programs due to machine learning flaws. CRed [3] can effectively detect UAFs in C programs based on demand-driven pointer analysis and spatial-temporal context reduction. It first uses SVF to obtain candidate UAF pairs in the program and then reduces false alarms via multi-stage analysis, including calling context-sensitive reduction and path-sensitive reduction. To scale to large programs, the approach trades off between calling context depth and scalability to reduce the number of program paths that need to be analyzed. Therefore, CRed inevitably misses some UAFs due to the trade-offs between scalability and call context depth. Moreover, due to its unsound modeling of array access aliases and lack of handling of linked lists, CRed may still miss real UAFs even though there is no calling context depth reduction.

4.1.2 *Static Binary Code Analysis*

Static binary code analysis faces not only similar challenges as static source analysis but also obstacles such as lack of type information and an incomplete control flow graph recovered from the binary level. There are several approaches [20], [97] in the literature to statically detect UAFs in binary code.

GUEB [20] first establishes an abstract memory model as its intermediate representation. It then conducts value analysis and pointer alias analysis to track heap operations and pointer propagation. Finally, it extracts a subgraph for each UAF detected to describe where the pointer is created, freed, and used. GUEB can detect UAFs without source code and provides better coverage than dynamic detectors, which heavily depend on test cases. However, GUEB is neither precise nor sound due to the imprecision of disassembly techniques and the unsound program points-to analysis.

In addition, GUEB encounters difficulties in analyzing real large programs because it uses function inlining technology for inter-procedural analysis, resulting in repeated analysis of functions that are called multiple times. UAFDetector [97], a recent approach, follows a behavior similar to GUEB but uses a function summary technique instead of inlining for inter-procedural analysis, thereby achieving better scalability and reducing unnecessary analysis overhead.

In addition, UAFDetector considers not only pointer aliases but also indirect jumps, so it has higher analysis integrity. However, UAFDetector also has defects. Its pointer alias analysis is incomplete, the loop handling is imprecise, and path-sensitive analysis is not conducted, so it may report a large number of false alarms in large programs.

4.2 Dynamic Analysis Approaches

According to whether the detected UAFs lie in the execution paths while the program executes, dynamic analysis falls into two categories: evidence-based dynamic analysis and prediction-based dynamic analysis.

4.2.1 *Evidence-Based Dynamic Analysis*

Most existing approaches are sanitizers that can detect UAFs under given test cases. These sanitizers usually utilize *shadow memory*, i.e., one-level or multi-level lookup tables, to store the state of memory objects. Specifically, shadow mapping maps the allocation or deallocation of each memory object into the corresponding shadow memory. Each time a memory object is accessed through a pointer, the shadow memory is queried to determine whether the accessed object is still allocated. If the accessed object is allocated, these sanitizers consider the memory access is legal. Otherwise, a UAF vulnerability is detected because the accessed object has been deallocated. As long as the freed memory is not reused, the existing sanitizers can detect the corresponding UAFs. The reasons why shadow memory is used is as follows. First, in contrast to the fat pointers that puts meta-data and data together, shadow memory stores meta-data separately, so it does not change the memory layout or introduce compatibility issues. Second, sanitizers can reduce memory overhead by increasing the level of lookup tables. Finally, sanitizers can use shadow memory to facilitate meta-data lookup through shadow mapping.

Dynamic Binary Translation. Purify [16], Valgrind [18] and Dr.Memory [15] instrument binary programs at execution time. To achieve this, Purify [16] utilizes object code insertion (OCI) technology, which instruments object files with additional instructions, and these instructions may vary from system to system. Valgrind [18] uses disassemble-and-resynthesize technology to achieve similar tasks. The disassemble-and-resynthesize technology first translates the machine code into one or more IR operations, inserts additional IRs to enable memory checks, and finally translates these IRs back into machine code. Dr.Memory [15] differs from Valgrind in that Dr.Memory is based on DynamoRIO [108] and performs multiple memory check optimizations. DynamoRIO uses copy-and-annotate to copy the incoming instructions verbatim: each instruction is annotated with a description of its effect. Both Valgrind and Dr.Memory split shadow memory into similar multi-level structures, i.e., multi-level shadow mapping, to reduce memory overhead and accelerate queries. The overhead of Purify, Valgrind and Dr.Memory is substantial, with performance penalties of 2x-25x, 20x and 10x, respectively.

Compile-Time Instrumentation. Mudflap [7], AddressSanitizer (ASan) [10], ThreadSanitizer (TSan) [11], [12] and Frama – C/E – ACSL [8], [98], [99] add memory checks while the program compiles. Mudflap [7] statically inserts the predicate validity assertion at the pointer dereference site (i.e., pointer's use site). The assertion depends on whether the pointer accesses valid memory, which is determined by the list of valid memory objects maintained by Mudflap. In practice, Mudflap suffers from heavy false positives in complex C/C++ programs. ASan [10] follows a similar detection approach as Valgrind and Dr.Memory. The difference is that ASan uses compile-time instrumentation to avoid the runtime instrumentation overhead. Moreover, ASan uses a direct shadow mapping scheme to reduce the overhead of querying shadow memory. The direct shadow mapping scheme maps the memory object address directly to the shadow address through the combination of direct scale and offset. A derivative of ASan, KASan [109] is designed to detect UAFs in the Linux kernel. HWASan [110] is another derivative of ASan, designed to reduce overhead with the aid of AArch64 hardware. TSan [11], [12] is a data race detector that can be used to detect UAFs. Frama – C/E – ACSL [8], [98], [99] differs from Frama – C in that it performs runtime verification and uses tokens to track allocated objects and pointers to these objects. These tokens allow inferring the state of the object pointed to by its pointer. None of the above approaches has startup runtime overhead, but they require source code. Moreover, ASan incurs 2x slowdown, while TSan, Mudflap, and Frama – C/E – ACSL have higher runtime overhead of 5x-15x, 2x-40x and 35x, respectively.

Library Interposition. DoubleTake [17] uses library interposition to instrument binary programs. It first divides the program execution into different epochs, each ending with an irrevocable system call. Then, at the beginning and end of each epoch, it checks the state of the program and determines whether any memory errors occurred (i.e., UAF, memory leaks or buffer overflow) during the epoch. If so, it re-executes the current epoch and instruments additional memory checks to determine the exact location of the error. Although DoubleTake has low overhead (5%), it can only detect writes to freed memory and not reads because its detection capability is based on the canary value (a random integer) stored in the freed memory, which cannot be corrupted by read operations.

Static Binary Rewriting. BASan [13], implemented using RetroWrite [13], can detect UAFs that exist in position-independent binaries, including third-party libraries. BASan differs from ASan in that its instrumentation occurs at the binary level, not at compile time, i.e., the IR level. RetroWrite realizes a static instrumentation module at the binary level using the reassembleable assembly technique based on the relocation information contained in position-independent binaries. This static binary instrument module is functionally equivalent to ASan's instrument module and is compatible with ASan's runtime library. RetroWrite uses this instrumentation module to insert memory checks at appropriate locations and to generate executable instrumented binaries. In other words, RetroWrite extends the applicability of ASan to the binary level. Similar to RetroWrite, Egalito [14] can also disassemble, transform, and regenerate position-independent binaries. Therefore, theoretically, Egalito can also implement BASan. The above two approaches avoid the overhead of dynamic instrumentation and adapt UAF detection to binary code. However, they also have limitations; that is, they apply only to position-independent binaries and are limited by the architecture.

The above sanitizers cannot detect UAFs if the freed memory is reallocated. To alleviate this issue, these approaches reduce the chance of UAF detection failure by quarantining freed memory objects until a dynamically calculated threshold is reached. DoubleTake may fail to detect many UAFs because it can only detect write operations to the first 128 bytes of freed memory objects. Since Mudflap can incur many false positives, it is deprecated and replaced by ASan [111]. BASan has lower overhead and is more effective [13], so it may play an important role in UAF detection of binary code.

UAFL [6] differs from the above approaches in that it is a fuzzer, not just a sanitizer. In other words, UAFL can automatically generate test cases to detect UAFs based on existing sanitizers (e.g., ASan). Compared with other standard fuzzers combined with ASan (such as AFL [112], AFLFast [113], FairFuzz [114], Angora [115], MOpt [116], and QSYM [117]), UAFL can detect more UAFs in less time. The reason is that other fuzzers use only the individual CFG (control-flow graph) edge coverage to guide test case generation, whereas UAFL models UAFs as type-state properties to guide fuzzing to generate test cases that violate UAF type-state properties. Moreover, UAFL uses information-flow analysis [6] to improve the fuzzing process. UAFuzz [118] is a similar tool, which is the first binary-level directed greybox fuzzer dedicated to UAFs.

4.2.2 Prediction-Based Dynamic Analysis

In contrast to evidence-based analysis, prediction-based analysis has the ability to predict possible UAFs based on the correct execution paths of the program. There are several approaches reported [9], [100] in the literature.

UFO [9] uses its extended maximum causality model, MaxModel [119], to predict UAFs. In general, UFO first uses TSan to obtain execution paths. It then uses MaxModel to infer additional feasible execution paths based on the obtained execution paths. Finally, it encodes both UAF violations and feasible execution paths as first-order logical constraints and uses a constraint solver (such as Z3 [120]) to solve them. Thus, UFO can predict UAFs with fewer execution paths and guarantee that each predicted UAF is true and repeatable. Since constraint solving is usually time-consuming, UFO makes a trade-off between time overhead and effectiveness. It actually uses a relaxed model that cannot predict all possible execution paths; hence, it may miss real UAFs. In addition, since UFO relies on TSan to obtain execution paths, if TSan ignores the paths where UAFs exist, UFO also misses real UAFs.

ConVul [100] can also predict concurrent UAFs. ConVul differs from UFO in that ConVul defines exchangeable events and detects UAFs based on that definition. Given events 1 and 2 are exchangeable if event 1 can occur before or after event 2. When used to predict UAF, if ConVul finds a pointer whose free site and use site are exchangeable events, then ConVul predicts a UAF. Although ConVul can detect concurrent UAFs, its ability is affected by the number of exchangeable events. If there are many other exchangeable events between the pointer's free site and use site, the real UAF may be missed.

Among the above approaches, UFO and ConVul are designed to detect concurrent UAFs due to multi-threaded scheduling. They may fail to detect UAFs in single-threaded programs or those existing within threads.

5 Exploit Mitigation

In this section, we elaborate on different kinds of exploit mitigation techniques, which are summarized and classified in Table 2 according to our proposed taxonomy. The first four columns of Table 2 show the categories, subcategories, names, and publication year of the tools; columns 5 and 6 summarize the availability of the tools and their performance overhead. The performance overhead is obtained from their original papers.

5.1 Dangling Pointer Elimination

Approaches Based on Taint Tracking. Undangle [29] uses taint tracking to detect dangling pointers. Whenever the program allocates a heap object, Undangle assigns a taint label to the return value of the allocation function and tracks the propagation of the taint label through dynamic taint tracking. Whenever the program deallocates this heap object, Undangle determines whether there are dangling pointers and which still points to the freed memory object based on the taint label and the life cycle of the pointer. While taint tracking allows for high code coverage and great precision, it incurs significant runtime and memory overhead. Thus, Undangle is not practical in many scenarios.

Approaches Based on Meta-Data Tracking. FreeSentry [4], DangNULL [70], and DangSan [30] track pointer propagation and record the relationship between pointers and objects. Each time a memory object is freed, they invalidate all dangling pointers to the object by querying the meta-data storage. DangNULL [70] employs red-black trees for meta-data storage, where each node corresponds to a memory object and maintains all pointers to the memory object. FreeSentry [4] uses two lookup tables as its meta-data storage: one records all mappings from a pointer to its reference objects, and the other records all mappings from a memory object to the pointers that reference it. DangSan [30] utilizes maps and logs to protect against dangling pointers. Each map contains a mapping of pointers to their reference objects. Logs are used to track the memory locations where pointers are stored. In contrast to DangNULL and FreeSentry, DangSan stores relationship changes between pointers and objects in logs rather than immediately updating the relationship. Therefore, DangSan has a lower runtime overhead (33%) than DangNULL and FreeSentry, 50% and 45%, respectively. In addition, FreeSentry does not support multi-threaded programs. However, DangSan may cause significant overhead in multi-threaded programs because its logs are per thread.

To further reduce overhead, pSweeper [31] employs instrumentation to prevent dangling pointer propagation and uses a concurrent thread to iteratively sweep all live pointers in the program. Hence, pSweeper has lower overhead (15%) because it avoids the overhead of maintaining the exact relationship between pointers and objects. In contrast to the above four approaches, HeapExpo[121] can track local variables and function parameters which are promoted to CPU registers and thus provide better heap protection.

Since stack UAFs are rare [3] and hard to exploit [70], the above approaches focus on mitigating heap UAFs. However, the heap protection they provide is not complete due to their trade-off between performance and effectiveness. Therefore, they cannot track pointers copied in type-unsafe ways. When these untracked pointers become dangling and dereferenced, malicious attackers may find these untracked pointers and exploit them to invade the system.

Approaches Based on Intermediate Pointers. DangDone [71] prevents dangling pointers by inserting intermediate pointers between pointers and objects. All operations performed on heap memory objects must be performed through intermediate pointers. When a pointer frees its reference object, DangDone not only deallocates the memory object but also invalidates the intermediate pointer. Thus, it prevents the dangling pointer from accessing the freed memory. Although its runtime overhead is low (1%), it prevents only dangling pointers to the start address of the heap object. To alleviate this issue, MPChecker [69] inserts intermediate pointers based on multi-level pointers and thus can additionally prevent dangling pointers to the internal addresses of objects. Moreover, MPChecker can track pointers copied in type-unsafe ways. Therefore, it provides better heap protection with larger runtime overhead (62%).

5.2 Pointer Dereferences Checking

Per-Pointer Meta-Data via Fat Pointer. CCured [62], [73], [74] uses fat pointers to perform memory checks. A fat pointer is a pointer that has other meta-data besides the address in memory. When dereferencing a pointer, CCured retrieves the pointer meta-data to determine whether the pointer dereference is permitted. Although CCured prevents illegal memory access, it changes the memory layout and inevitably introduces compatibility issues. To alleviate this issue, EffectiveSan [72] uses a low-fat pointer instead, which encodes meta-data within the pointer itself and thus eliminates compatibility issues. EffectiveSan dynamically binds each freed memory object to a special type and prevents UAFs by checking whether the accessed object is of the special type. However, attackers can bypass its protection by reallocating objects of the same type as the released object.

Lock-and-Key Identifier Checking. Many solutions attach pointers with unique labels to prevent UAF exploits [34],

TABLE 2
Summary and Classification of Existing UAF Mitigation Approaches

Category	SubCategory	Name	Year	Availability	Performance Overhead
Dangling Pointer Elimination	Approaches based on taint tracking	Undangle [29]	2012	unavailable	not directly given
	Approaches based on meta-data tracking	DangNULL [68]	2015	unavailable	54.6%
		FreeSentry [4]	2015	open-source	45%
		DangSan [30]	2017	open-source	33%
		HeapExpo [121]	2020	open-source	35%
		pSweeper [31]	2018	unavailable	15%
	Approaches based on intermediate pointers	DangDone [71]	2018	unavailable	1%
		MPChecker [69]	2019	unavailable	62%
Pointer Dereferences Checking	Per-pointer meta-data via fat pointer	CCured [62], [73], [74]	2012	unavailable	30%∼150%
		EffectiveSan [72]	2018	open-source	49%
	Lock-and-key identifier checking	CETS [34]	2010	open-source	48%
		MemSafe [75]	2013	unavailable	88%
		Wei Xu et al. [76]	2004	unavailable	162%
		Suan Hsi Yong et al. [77]	2003	unavailable	190%
		Safe-C [52]	1994	unavailable	130%-540%
	Hardware-based checking	WatchDog [35]	2012	unavailable	25%
		WatchDogLite [79]	2014	unavailable	29%
		BOGO [33]	2019	open-source	60%
		Chex86 [122]	2020	unavailable	14%
		Cornucopia [78]	2020	unavailable	2%
		Cherivoke [32]	2019	unavailable	5%
Changing Memory Allocators	Random heap allocation	DieHard [37]	2006	open-source	12%
		Archipelago [82]	2008	unavailable	6%
		DieHarder [83]	2010	unavailable	20%
		LFH [85]	2010	unavailable	not directly given
		Address Obfuscation [80]	2003	unavailable	0.9%-5.2%
		Vivek et al. [81]	2010	unavailable	not directly given
		FreeGuard [84]	2017	open-source	2%
		SlimGuard [41]	2019	open-source	not directly given
		Guarder [41]	2018	open-source	3%
		ZEUS [39]	2018	unavailable	1.20%
	Type-safe heap allocation	Cling [36]	2010	unavailable	11.40%
		Type-after-Type [86]	2018	open-source	4.30%
		Internet Explorer [88]	2014	unavailable	not directly given
	Isolated heap allocation	Dhurjati et al. [87]	2006	unavailable	4%-15%
		Oscar [38]	2017	unavailable	40%
		FFmalloc [2]	2021	partially open-source	2.3%
Improving Garbage Collection	Safe C language dialects	Fail-Safe C [63]	2009	unavailable	170%-220%
		Cyclone [89]	2002	unavailable	100%-285%
		Ironclad C++ [123]	2013	unavailable	12%
	Use garbage collection libraries	Boehm garbage collector [42], [43]	1988	open-source	not directly given
	Delay memory reuse	MarkUs [1]	2020	open-source	10%
		MemGC [124]	2020	unavailable	not directly given
	Improve object lifecycle management	Smart pointer [92]	1992	unavailable	not directly given
		CRCount [44]	2019	unavailable	22%
		SafeCode [91]	2006	unavailable	10%
Limiting Exploitation Damage	Virtual table hijacking protection	SafeDispatch [47]	2014	unavailable	2.10%
		VTPin [49]	2016	unavailable	4%
		Vip [45]	2017	unavailable	0.60%
		T-VIP [46]	2014	unavailable	2.20%
		vfGuard [48]	2015	unavailable	18.30%
		VTrust [64]	2016	unavailable	0.72%-2.2%
		VTint [68]	2015	unavailable	2%
		VTable Interleaving [93]	2016	unavailable	1%
		NoVT [95]	2021	unavailable	not directly given
		CFIXX [94]	2018	open-source	4.98%

[52], [75], [76], [77]. For example, CETS [34] assigns a specific label to each memory object and all pointers to that object. When dereferencing a pointer, it checks whether the pointer label matches the object label the pointer (should) points to. If they do not match, it crashes the program to prevent UAF exploits. Although CETS has low overhead (48%), it may mistakenly crash the program and allow UAF to occur, making it less practical in many situations [30]. Similarly, MemSafe [75] validates each pointer dereference to prevent UAF exploits, which incurs 88% slowdown.

Hardware-Based Checking. Many solutions utilize dedicated hardware support to perform memory checks [32], [33], [35], [79], [122], [125]. Watchdog [35] follows the behavior similar to CETS. The difference is that Watchdog uses micro-operations to retrieve labels with pointers and thus it achieves a lower runtime overhead (25%). WatchdogLite [79] extends Watchdog with a fixed set of additional instructions and incurs 29% slowdown. Since the Intel MPX [126] may incur severe performance degradation [126], BOGO [33], which is built on top of the Intel MPX, incurs 60% slowdown when checking memory access. Other hardware-based solutions include CHERIvoke [32] and Cornucopia [78].

5.3 Changing Memory Allocators

Random Heap Allocation. DieHard [37] uses randomization to provide probabilistic temporal safety. To achieve this goal, it assumes that the heap space is unlimited and randomly allocates memory objects on the heap that may never be released. Therefore, an attacker cannot easily reallocate objects on the freed memory, thereby reducing the possibility of exploiting UAF. Archipelago [82] extends DieHard by compacting infrequently used objects and thus reduces the memory overhead. DieHarder [83] extends DieHard using a spare layout to provide more stable randomization entropy.

In contrast to DieHard and its derivatives, low fragment heap (LFH) of Windows 8 [85] prevents UAF exploits through random chunk selection. Address obfuscation [80] achieves probabilistic memory safety by randomizing the heap address and adding random intervals to memory regions. Vivek *et al.* [81] propose that for each heap allocation request, the probability of a successful attack can be reduced by returning a memory chunk containing a random extra memory block. Similar to FreeGuard [84] and Guarder [40], SlimGuard [41] allows users to specify the security level they need. For each memory allocation, SlimGuard randomly selects a memory chunk from the memory allocation pool to provide stable randomized entropy.

ZEUS [39] differs from the above approaches in that its randomness lies not only in the size and heap address of the memory object but also in the random offsets introduced before each member field, which violates the alignment of the internal member fields of the memory object. By implementing fine-grained randomization, Zeus can effectively protect programs from UAF exploits.

Type-Safe Heap Allocation. Cling [36] performs object-type inference at runtime and prevents UAF exploits by allowing freed memory to be reallocated only to store heap objects of the same type. For each memory allocation, the returned memory chunk is either never allocated or has previously stored a freed object of the same type. Thus, Cling can prevent UAF exploits between objects of different types. Type – after – Type [86] follows similar behavior to Cling. The difference is that Type – after – Type uses static analysis to infer object type and achieves type-safe memory reuse on the stack for the first time. Furthermore, its runtime overhead (4.3%) is lower than that of Cling (11.4%). However, it shares the same design limitation as Cling; that is, it cannot prevent UAF exploits between objects of the same type.

Microsoft Internet Explorer [88] divides memory objects into different types and allocates different memory areas for each type of memory object. Although this approach greatly increases the difficulty for attackers to exploit UAF vulnerabilities, it can still be bypassed by reallocating objects of the same type as the released object [127].

The above approaches, both random heap allocation and type-safe heap allocation, cannot completely defend against attacks on the heap. These approaches only increase the difficulty for an attacker to reallocate objects on the freed memory. The protection they provide to the program can be circumvented in various ways, such as heap spraying [128] and heap feng shui [129].

Isolated Heap Allocation. Dhurjati *et al.* [87] allocate a new virtual page for each allocated memory object and thus avoid dangling pointer dereferences to newly allocated objects. Moreover, to reduce memory overhead, they map different virtual pages to the same physical page. Inspired by Dhurjati *et al.*, Oscar [38] proposes an optimized approach based on page permissions that can completely prevent UAF exploits. In addition, Oscar does not require source code and supports more memory operations. Therefore, it is more practical. Nevertheless, in memory-intensive programs, it incurs considerable runtime overhead (60%). To further reduce overhead without sacrificing security, FFmalloc [2] allocates a batch of pages each time, and uses different policies according to the requested allocation size. As a result, it achieves lower runtime overhead (2.3%).

5.4 Improving Garbage Collection

Safe C Language Dialects. Fail – Safe C [63] implements a fully ANSI C compliant compiler to provide complete memory safety features. It uses a garbage collection mechanism to eliminate the threat posed by dangling pointers, that is, to release memory objects only at the right time. Other safe C dialects exist, namely Cyclone [89], [90] and Ironclad C + + [123]. While these C dialects achieve complete memory security, it is expensive to maintain compatibility with the C/C++ specification and transfer legacy programs into corresponding C programs.

Use of Garbage Collection Libraries. Many researchers focus on adding garbage collection libraries, such as the Boehm garbage collector [42], [43], to automatically release memory objects. With garbage collection libraries, the program can not only prevent attacks but also prevent program crashes due to UAFs. However, most garbage collection algorithms consume more memory because they delay the deallocation of memory objects until there is not sufficient memory or the program explicitly requests it. Worst of all, some dangling pointers may survive for a long time, which further increases the memory overhead of garbage collection. Another key limitation is that they may free objects that are still referenced by hidden pointers [1], thus causing UAFs themselves.

Delay Memory Reuse. Many mitigation techniques delay memory reuse until a criterion is met [1], [16], [18], [44], [63], [87], [91], [124], [130]. For example, MemGC [124] delays memory reuse until the total size of unreferenced chunks reaches a dynamically calculated threshold (that is, criterion). After that, MemGC performs a mark-and-sweep operation to recycle unreferenced chunks. However, it only applicable to small programs because its reference-counting mechanism is not robustness. Scudo Malloc [130] places freed objects in quarantine and does not actually free them until a criterion is met, which helps partially alleviate UAFs exploits by reducing the determinism of allocation and deallocation patterns. The design limitation is that its protection is probabilistic. MarkUs [1] differs from other memory reuse-delay approaches in that it delays memory reuse until it cannot find dangling pointers to freed objects.

Improve Object Life Cycle Management. Smart pointers [92] constitute a class that wraps a bare C++ pointer. During program execution, the smart pointer tracks the reference count of the memory object and frees the memory object when the reference count is reduced to zero. Although smart pointers can automatically release memory objects, for legacy code, the cost of converting bare pointers to smart pointers is prohibitively high. Therefore, smart pointers work only for newly developed code, not legacy code. To this end, CRCount [44] uses its self-implemented reference-

TABLE 3
Eight Popular Programs Gathered From GitHub

Program	Version	KLOC	Language
sed	4.7	26	C++
bison	3.0.4	31	C++
libevent	2.1.9	121	C++
tesseract-ocr	4.0	158	C++
redis	5.0.4	182	C++
zfs	0.8.0	768	C++
gzip	1.6	644	C
ghostscript	9.23	1,745	C++

TABLE 4
Nine Widely Used Programs With Known UAFs

Program	Version	KLOC	Language
gohttp	-	0.5	C
libheif	1.4.0	12	C
lrzip	0.631	22	C
lua	5.3.5	29.7	C++
nasm	2.14.02	38	C++
nasm	2.14rc16	57	C++
binaryen	1.38.22	98	C++
jasper	1.900.1	114	C++
binutils	2.31.1	333	C++

counting mechanism to calculate the reference count of each memory object, where each reference corresponds to a pointer. Although CRCount works with legacy code, it only performs intra-procedural backward data-flow analysis and thus may miss some pointers that are copied in type-unsafe ways. As a result, CRCount may release memory objects prematurely, which instead leads to UAFs. SafeCode [91] implements a custom allocation pool in which memory objects are freed only when the reference count of all memory objects is reduced to zero. To track pointer references, it performs field-insensitive static analysis and is therefore only suitable for small programs.

5.5 Limiting Exploitation Damage

Virtual Table Hijacking Protection. Some solutions focus on protecting virtual function calls indexed by virtual tables to defend against possible UAF exploits in C++ dynamic dispatch. These solutions are implemented at either the binary [46], [48], [68] or source code level [45], [47], [49], [64], [95]. For example, SafeDispatch [47] inserts runtime checks to ensure that when virtual functions in a class are invoked, the control flow cannot be arbitrarily modified. When an object is freed, VTpin [49] lets the virtual table pointer of the object point to a secure virtual table address. Vip [45] first uses partial pointer analysis to discover the legitimate target set of virtual calls from separately compiled modules and then performs integrity checks at runtime. Although these approaches can protect against arbitrary code execution caused by overwriting virtual table pointers, they cannot prevent information leakage caused by exploiting UAFs. Furthermore, when attackers target other pointers instead of virtual table pointers, these approaches are ineffective. Attackers can still exploit UAFs via various techniques, such as heap spraying [128] and heap feng shui [129].

6 Evaluation

The purpose of empirical evaluation is to assess the effectiveness and efficiency of existing publicly-available UAF detection and exploit mitigation approaches in C/C++ programs. We use a set of benchmarks and real-world programs to answer the following research questions (RQs).

- *RQ1*: Can the static detectors detect UAFs in the benchmarks and real-world programs?
- *RQ2*: Can the dynamic detectors detect UAFs in the benchmarks and real-world programs?
- *RQ3*: What is the difference between the dynamic and static detectors in detecting UAFs?
- *RQ4*: What is the performance (runtime) overhead and memory overhead of the dynamic detectors? Can they scale to large programs?
- *RQ5*: How effective are the exploit mitigation tools in preventing UAFs in real-world programs?
- *RQ6*: What is the runtime overhead and memory overhead of the exploit mitigation tools?

6.1 Experimental Setup

Methodology. To answer RQ1, we evaluate six publicly-available static detectors: Clang static analyzer [27], CBMC [25], Klee [24], Coccinelle [26], Frama − C [23], and TscanCode [28]. In terms of GUEB [20], although it is open-source, we cannot successfully compile it. To answer RQ2 and RQ4, we evaluate seven publicly-available dynamic detectors: Valgrind [18], Dr.Memory [15], ASan [10], TSan [11], [12], DoubleTake [17], Frama − C/E − ACSL [23], and UFO [9]. To answer RQ3, we compare the above six static detectors and the seven dynamic detectors. Other static detectors and dynamic detectors are not considered because their prototypes are either unavailable or unable to run. To answer RQ5 and RQ6, we evaluate five publicly-available exploit mitigation tools: MarkUs [1], SlimGuard [41], Guarder [40], FreeGuard [84], and DieHarder [83]. For the remaining exploit mitigation approaches in Section 5, their prototypes are either unavailable or cannot be successfully run [2].

Benchmarks. To answer RQ1, we use two benchmarks, namely JTS [107] and *dataset_1* (cf. Table 3), which are collected from previous UAF-related work [3], [21], [107]. Moreover, JTS is a set of carefully planned small programs (case studies) representing real-world errors covering various data types and control flows. Each small program in JTS has one and only one real UAF. To answer RQ2, we use three benchmarks, namely JTS benchmark [107], our crafted benchmark *dataset_2* and benchmark *dataset_3* (cf. Table 4). Since JTS contains only 400 small programs that read data from freed memory objects, we adapt the original small programs by changing the read operations into write operations on freed memory objects. Hence, benchmark *dataset_2* also contains 400 small programs. Benchmark *dataset_3* contains nine widely-used programs with known UAFs identified by CVE-ID. We use existing proofs of concept (i.e., test cases) confirmed by the developers to trigger the UAFs in benchmark *dataset_3*. To answer RQ3, we also apply static detectors to benchmark *dataset_3* and compare the

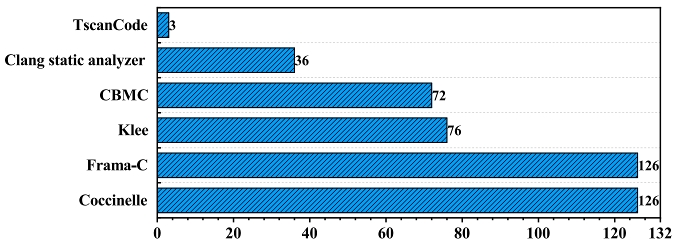

Fig. 3. Completeness for C-only benchmark in JTS: Tscancode (2.3%), Clang static analyzer (27.0%), CBMC (54.0%), Klee (57.0%), Framc-C (95.4%) and Coccinelle (95.4%).

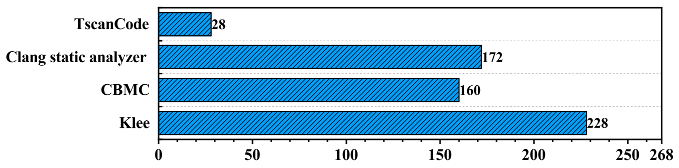

Fig. 4. Completeness for C++-only benchmark in JTS: Tscancode (10.4%), Clang static analyzer (64.2%), CBMC (59.7%), Klee (85.1%).

experimental results of static detectors and dynamic detectors. We use C/C++ benchmark from SPEC CPU2006 (employing the reference input) to answer both RQ4 and RQ6. We also use benchmark *dataset_3* to answer RQ5.

Our experiments are conducted in 64-bit Ubuntu 18.04 installed on a VMware Workstation 15.04 with 2 quad-core Intel Core i7-7700 CPUs and 16 GB memory. For the experiments to answer RQ4 and RQ6, we repeat each experiment 10 or 20 times (depending on the deviations of different runs) to ensure accuracy, and we use the average value as the final result. This is sufficient because the standard deviations of different runs mostly range from 0.5 to 2.0, which are very slim. In addition, similar to the existing work [2], [5], [80], all benchmarks and real-world programs are compiled with "O2" optimization level. During these experiments, we only run the virtual machine, in which we only run programs related to the experiment. Therefore, we believe that this setting does not affect the fairness of the final experimental results. Except SPEC CPU2006, all the other benchmarks we use are publicly-available.[9]

6.2 Results and Analysis

In the following, we report on the experimental results to answer RQ1-RQ6, respectively.

6.2.1 RQ1: Effectiveness of Static Detectors

Effectiveness on JTS. We first use JTS to establish the ground-truth of static detectors. Each small program in JTS consists of 100-500 lines of code and has one and only one UAF. Since Coccinelle and Frama – C are only applicable to C programs, the 400 small programs we use in JTS are divided into a C-only benchmark and C++-only benchmark, containing 132 and 268 UAFs, respectively. The results for the C-only benchmark and C++-only benchmark in JTS are shown in Figs. 3 and 4, respectively.

As shown in Fig. 3, Coccinelle, Frama – C, Klee, CBMC, Clang static analyzer and TscanCode find 126, 126, 76, 72, 36 and 3 UAFs, respectively. In Fig. 3, the completeness of Coccinelle, Frama – C, Klee, CBMC, Clang static analyzer and TscanCode is 95.4%, 95.4%, 57.0%, 54.0%, 27.0% and 2.3%, respectively. As presented in Fig. 4, Klee, CBMC, Clang static analyzer and TscanCode find 228, 160, 172 and 28 UAFs, respectively. In Fig. 4, the completeness of Klee, CBMC, Clang static analyzer and TscanCode is 85.1%, 59.7%, 64.2%, and 10.4%, respectively. In short, none of these static detectors can detect all UAFs. In addition, none of them report any false alarms. Therefore, the correctness of their results on JTS benchmark is 100%.

Effectiveness on Real-World Programs. We then evaluate whether these static detectors can discover new UAFs in the benchmark *dataset_1*. We ignore Klee and Frama – C because we cannot apply them to the benchmark *data_set1*. Table 5 summarizes the experimental results. Clang static analyzer issues 18 warnings, including seven real UAFs, in 4,204 seconds (1.17 hours). The other static detectors are either terminated within one day for two programs (CBMC) or are impractical, reporting no UAFs (TscanCode) or only false alarms (Coccinelle).

CBMC spends more than one day to analyze the programs (except *libevent* and *tesseract-ocr*) in *data_set1*. Moreover, CBMC does not detect any real UAFs or produce any false positives. TscanCode spends a total of 8,450 seconds without finding any UAFs. Coccinelle reports 21 warnings in 139 seconds, all of which are false positives. Therefore, their correctness and completeness are zero because no real UAFs are identified. Clang static analyzer spends only 4,204 seconds (1.17 hours) to analyze the eight programs (3,675+ KLOCs) in *data_set1*. Moreover, it reports seven real UAFs and 11 false positives, with a correctness of 38%.

In summary, for large programs, Clang static analyzer outperforms the other static detectors. However, when analyzing small C programs, Coccinelle and Frama – C are the best static detectors. Klee is better than other static detectors when analyzing small C++ programs.

Reasons for Detection Failure. To help understand the reasons for failure, an example of a simplified UAF in JTS is shown in Fig. 5. The pointers p and q are in different functions, but they both point to the same object allocated at Line 2. At Line 4, the memory object is freed and should no longer be accessed. However, the program accesses this freed memory object through pointer q at Line 10. All the above static detectors miss this UAF.

Coccinelle misses UAFs for the following reasons. First, given a free site, it does not search the use site upstream; that is, the free site may be wrapped, so the use site and the free site are not in the same function. In Fig. 5, the free site is located at the function *helperBad*, and the use site is located at the function *Bad*. Coccinelle cannot find this use site. The second reason is related to its limited pointer alias analysis. The third reason is related to the poor control-flow analysis. TscanCode fails to detect this UAF, mainly because its rough pattern matching rules cannot work normally due to poor control-flow analysis. Other reasons include the simple pointer analysis approach and the fact that it does not perform pointer alias analysis. Frama – C cannot detect this UAF due to its poor inter-procedural analysis.

CBMC and Klee miss UAFs for similar reasons. CBMC detects UAFs through bounded model checking using

9. https://figshare.com/s/aa5fd6dace565fcc1672

TABLE 5
Experimental Results (#T: #True Positives and #F: #False Positives)

Program	CBMC			TscanCode			Clang static analyzer			Coccinelle		
	Report		Time (secs)	Report		Time (secs)	Report		Time (secs)	Report		Time (secs)
	#T	#F		#T	#F		#T	#F		#T	#F	
sed	0	0	>86,400	0	0	74	0	2	75	0	0	7
bison	0	0	>86,400	0	0	12	0	0	112	0	2	6
libevent	0	0	3,000	0	0	52	0	2	203	0	0	4
tesseract-ocr	0	0	34,553	0	0	686	0	2	236	0	0	15
redis	0	0	>86,400	0	0	101	0	0	352	0	5	9
zfs	0	0	>86,400	0	0	204	6	4	1,357	0	2	38
gzip	0	0	>86,400	0	0	17	1	0	36	0	1	2
ghostscript	0	0	>86,400	0	0	7,304	0	1	1,834	0	11	58
Total	**0**	**0**	**>555,953**	**0**	**0**	**8,450**	**7**	**11**	**4,204**	**0**	**21**	**139**

symbolic execution. It transforms each possible program path into a set of constraints, which are then solved by an SMT solver. If the constraints are unresolvable or the result is not sufficiently accurate, the UAF can be missed. Klee is based on symbolic execution and utilizes an SMT solver to resolve constraints, so SMT solvers also affect its detection ability.

Clang static analyzer fails to detect UAFs for several reasons. First, its pointer alias analysis is simple. Second, it performs limited inter-procedural analysis through function inlining. Finally, it handles loops and arrays in a conservative manner. The reason for the false positives on *dataset_1* is that Clang static analyzer performs only limited inter-procedural analysis through function inlining and performs limited path-sensitive analysis, and thus some infeasible paths are not excluded.

6.2.2 RQ2: Effectiveness of Dynamic Detectors

Effectiveness on JTS and dataset_2. We first evaluate the dynamic detectors on JTS and *dataset_2*, and the results are shown in Figs. 6 and 7, respectively. We create *dataset_2* because DoubleTake can only detect writes to freed memory objects, but small programs in JTS do not include this situation. We do not evaluate UFO because it is used for multi-threaded programs, while JTS and *dataset_2* are single-threaded. Fig. 6 reveals that all the dynamic detectors find 362 UAFs and miss 38 UAFs. The completeness of the results on JTS is 90.5%. In addition, these approaches do not report any false positives and thus the correctness of their results on JTS is 100%. As shown in Fig. 7, the dynamic detectors find all 400 UAFs, except for DoubleTable, which finds only 191 UAFs. The completeness of ASan, Valgrind, Valgrind and Frama – C/C – ACSL on *dataset_2* is 100%, but the completeness of DoubleTake *dataset_2* is only 47.8%. In summary, Valgrind, ASan, TSan, Frama – C/C – ACSL and Dr.Memory are effective for JTS and *dataset_2*, while DoubleTake is the least effective.

Effectiveness on Real-World Programs. We then evaluate these dynamic detectors on *dataset_3* to further demonstrate their detection capabilities. We exclude Frama – C/E – ACSL from this analysis because it fails to compile all the programs in *dataset_3*. Table 6 summarizes the experimental results, where the results in parentheses represent the corresponding detection results when the quarantine is minimized. ASan finds all nine real UAFs, while Valgrind and Dr.Memory both detect six real UAFs. TSan detects five real UAFs but misses four. DoubleTake fails to detect any UAF in *dataset_3*. UFO finds one UAF in *libheif* and misses two UAFs in *lrzip* and *lua*. Moreover, UFO fails to compile for the other six programs. None of the above dynamic detectors report false positives. The completeness of ASan, Valgrind, Dr.Memory, TSan, UFO and DoubleTake is 100%, 66.7%, 66.7%, 55.6%, 11.1% and zero, respectively.

Next, we report the experimental results when changing the quarantine of Valgrind and Dr.Memory. We ignore the other dynamic detectors because they cannot minimize their quarantine. Table 6 presents the results of Valgrind and Dr.Memory when their minimum quarantines are used: as

```
char * helperBad(char * aString) {
    char * p= (char *) malloc(10);
    ...
    free(p);
    return p;
}

void Bad(){
    char * q= helperBad("BadSink");
    printLine(q);
}
```

Fig. 5. An example simplified from UAF in JTS.

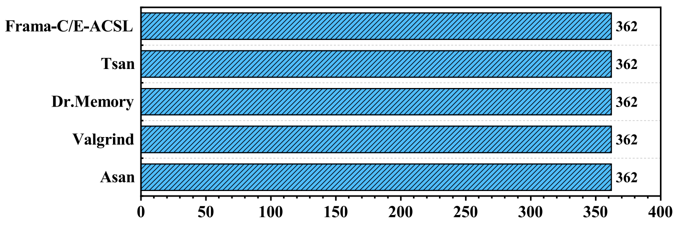

Fig. 6. Completeness for UAFs in JTS: All is 90.5%.

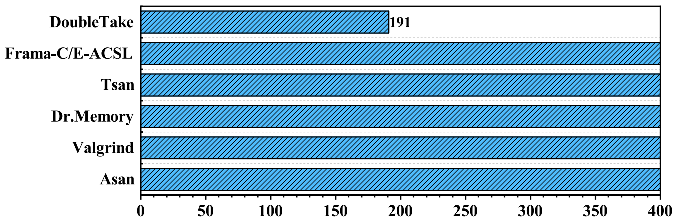

Fig. 7. Completeness for UAFs in *dataset_2*: DoubleTake is 47.8%, the rest is 90.5%.

TABLE 6
Dynamic Analysis Results on Nine Known UAFs in Programs (*dataset_3*)

Known UAFs		ASan	Valgrind	Dr.Memory	DoubleTake	TSan	UFO
Identifier	Program						
CVE-2019-12160	*gohttp*	√	√ (√)	√ (√)	×	×	×
CVE-2019-11471	*libheif*	√	× (×)	× (×)	×	×	√
CVE-2018-11496	*lrzip*	√	× (×)	× (×)	×	×	×
CVE-2019-6706	*lua*	√	√ (√)	√ (√)	×	√	×
CVE-2019-8343	*nasm*	√	√ (×)	√ (×)	×	√	×
CVE-2018-20535	*nasm*	√	√ (×)	√ (×)	×	√	×
CVE-2019-7703	*binaryen*	√	× (×)	× (×)	×	×	×
CVE-2015-5221	*jasper*	√	√ (×)	√ (×)	×	√	×
CVE-2018-20623	*binutils*	√	× (×)	× (×)	×	√	×

shown in the parentheses, they identify only two UAFs. Thus, the completeness of Valgrind and Dr.Memory is 22.2%. In addition, even when the quarantine size is increased to the maximum value, Valgrind and Dr.Memory fail to find all nine UAFs. At this pointer, we can draw the following two conclusions. First, the memory reuse delay incurred by quarantining the freed memory objects can indeed increase the possibility of detecting UAFs. Second, the memory reuse delay used by these dynamic detectors cannot fundamentally avoid missing UAFs.

In summary, ASan outperforms the other evidence-based dynamic detectors. Valgrind and Dr.Memory rank second. DoubleTake is the least effective because it fails to detect any UAF in *dataset_2*. UFO detects a real UAF, but the robustness of its prototype must be improved.

Reasons for Detection Failure. Since these evidence-based dynamic detectors have no fundamental difference in detecting UAF, they miss UAFs for the same two reasons. The main reason is that they cannot distinguish different objects allocated at the same address at execution time, which is why they cannot detect some UAFs in *dataset_3*. To alleviate this issue, these approaches use quarantines to delay memory reuse and thus increase the possibility of detecting UAFs. However, when the quarantine is exhausted, the freed memory is reallocated to store another object, causing these approaches to fail to detect UAFs, as shown in Table 6. Another reason is that they do not handle certain C library functions well, which explains the failed detection of 38 UAFs in the JTS.

UFO misses two UAFs for the following two reasons. First, it relies on TSan to obtain execution traces and can infer other possible execution paths based on the obtained execution traces. If TSan fails to record some execution traces, then UFO misses the real UAF. Second, UFO cannot predict all possible execution paths due to its trade-off between effectiveness and efficiency. Therefore, it may fail to predict the path leading to the real UAF.

6.2.3 RQ3: Comparison Between Dynamic and Static Detectors

We apply static detectors to the *dataset_3* to compare the effectiveness of dynamic and static detectors. Table 7 lists the experimental results. Clang static analyzer issues eight warnings, including one real UAF and seven false alarms. The real UAF detected by Clang static analyzer is the same as the known UAF in *gohttp*; thus, no new real UAFs are detected. Coccinelle issues 61 warnings, which are all false positives. CBMC and TscanCode fail to detect any UAFs in *dataset_3*. In this case, the completeness of Clang static analyzer is 11.1%, whereas the completeness of the other static detectors is zero. Tables 6 and 7 indicate that although the code coverage of dynamic detectors is low, the detection ability of dynamic detectors is better than that of static detectors in real-world programs. In addition, the code coverage of dynamic detectors can be improved by combining these approaches with fuzzing tools. Therefore, unless static detectors can solve the path explosion problem and further improve the accuracy of inter-procedural analysis and pointer analysis, dynamic detectors will continue to dominate inter-procedural UAF detection in practice.

TABLE 7
Static Analysis Results on Nine Known UAFs in Programs (*dataset_3*)

Known UAFs		CBMC		TscanCode		Clang static analyzer		Coccinelle	
Identifier	Program	#T	#F	#T	#F	#T	#F	#T	#F
CVE-2019-12160	*gohttp*	0	0	0	0	1	0	0	0
CVE-2019-11471	*libheif*	0	0	0	0	0	0	0	0
CVE-2018-11496	*lrzip*	0	0	0	0	0	0	0	22
CVE-2019-6706	*lua*	0	0	0	0	0	0	0	0
CVE-2019-8343	*nasm*	0	0	0	0	0	1	0	0
CVE-2018-20535	*nasm*	0	0	0	0	0	1	0	0
CVE-2019-7703	*binaryen*	0	0	0	0	0	0	0	5
CVE-2015-5221	*jasper*	0	0	0	0	0	0	0	9
CVE-2018-20623	*binutils*	0	0	0	0	0	5	0	25
Total	**0**	**0**	**0**	**0**	**0**	**1**	**7**	**0**	**61**

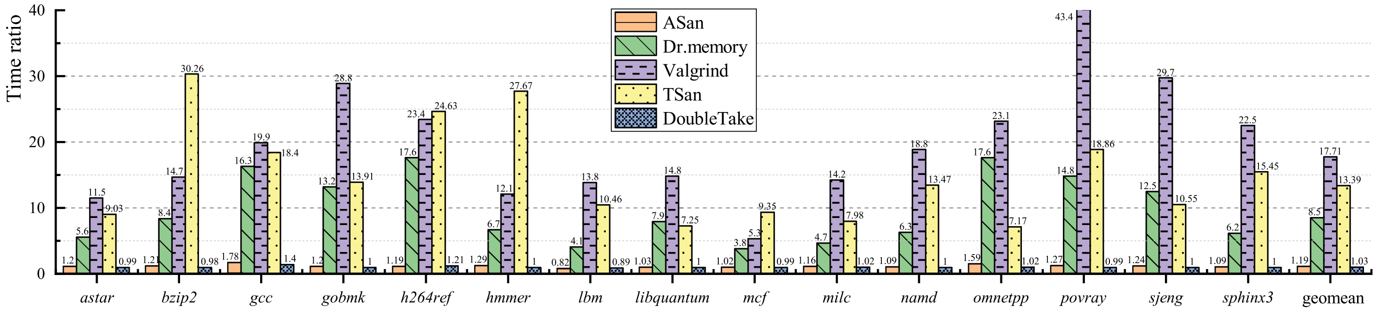

Fig. 8. Runtime overhead of different dynamic detectors on the SPEC CPU2006 benchmark.

Figs. 3, 4, and 6 show that static detectors can effectively detect intra-procedural UAFs. Compared with the dynamic detectors, static detectors can complete the analysis of the entire program in a short time. Hence, static detectors are more suitable for detecting intra-procedural UAFs, whereas dynamic detectors are more suitable for further detecting intra-procedural UAFs. In summary, dynamic detectors should focus on improving the effectiveness and efficiency of UAF detection, for example, by means of combinations with fuzzing tools. Static detectors should further solve the path explosion problem and improve the pointer analysis and pointer alias analysis capability.

6.2.4 RQ4: Overhead of Dynamic Detection Tools

We use the SPEC CPU2006 benchmark to evaluate the runtime overhead and memory overhead of different dynamic detectors. We ensure that all CPU cores are fully loaded in the experiment, because we observe that the utilization rate of the CPU cores is always over 98%, which also applies to the experiment to answer RQ6.

Runtime Overhead. We measure the runtime overhead by comparing the running time of each program between the version instrumented by the dynamic detection tool and the unmodified baseline version. Fig. 8 presents the experimental results. The horizontal axis lists the 15 real-world programs we use. The vertical axis is the ratio of the running time of the instrumented version to that of the unmodified baseline version. The value "1" represents the baseline, which is a perfect score, indicating that there is no additional runtime overhead. The higher the bar, the higher the runtime overhead of running the instrumented version. Specially, the value "0" indicates that the dynamic detection tool fails to instrument the program or the instrumented program is not runnable.

As illustrated in Fig. 8, the geomean runtime overhead of DoubleTake, ASan, Dr.Memory, TSan, and Valgrind is 1.03x, 1.19x, 8.50x, 13.39x, and 17.71x, respectively. DoubleTake has the lowest runtime overhead, which only incurs 3% slowdown. Valgrind has the highest runtime overhead, which incurs up to 1671% slowdown. For *povray*, Valgrind has the maximum runtime overhead (43.4x). These results demonstrate that there is great difference among the runtime overhead of different dynamic detectors.

Memory Overhead. We measure the memory overhead with the memory ratio, that is, the memory consumption of the instrumented program over that of the original program. Fig. 9 summarizes the experimental results. Like Fig. 8, the horizontal axis shows the 15 real-world programs we use; the vertical axis is the ratio of the memory consumption of the instrumented version to that of the original version. The value "1" indicates that there is no extra memory overhead. The higher the bar, the higher the memory overhead of running the instrumented version. The value "0" indicates that the the program instrumented by the dynamic detector is not runnable.

As illustrated in Fig. 9, the geomean memory overhead of DoubleTake, Dr.Memory, Valgrind, ASan, and TSan is 2.07x, 2.11x, 2.78x, 2.88x, and 4.30x, respectively. DoubleTake not only has the lowest runtime overhead, but also has the lowest memory overhead. However, as summarized in Table 6, the effectiveness of DoubleTake is poor, because the nine UAFs are all missed. TSan has the highest memory overhead. Compared with the runtime overhead, overall, there is less difference among the memory overhead of different dynamic detectors.

UFO *Performance*. In contrast to other dynamic detectors, UFO [9] is a predictive analysis tool, which can precisely predict concurrency UAFs from a single observed execution trace. Here, we use some benchmarks on *Chromium* to

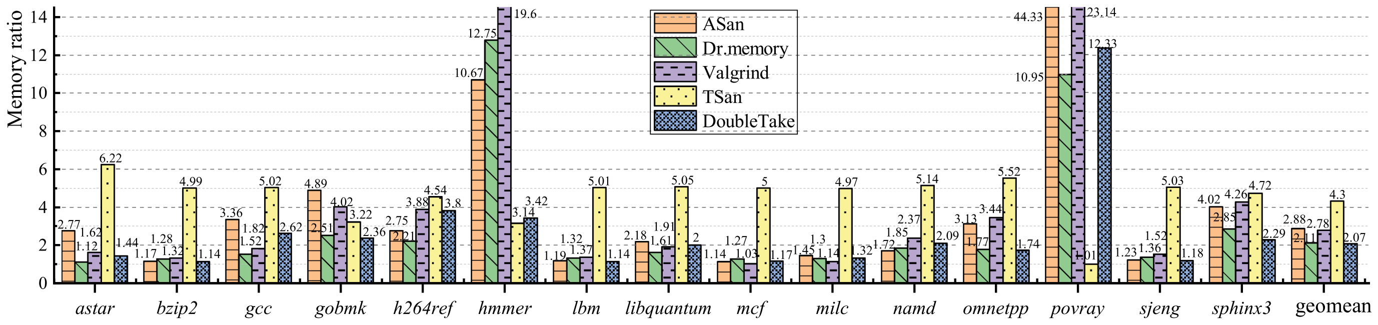

Fig. 9. Memory overhead of different dynamic detectors on the SPEC CPU2006 benchmark.

TABLE 8
Performance Evaluation on *Chromium*

Benchmark	Original	ASan	TSan	UFO
Octane	39,974	25,817	23,293	9,399
SunSpider	242	450	633	1,262
Dromaeo JS	3,087	2,402	1,904	1,552
Dromaeo DOM	3,787	2,336	1,013	589

measure the performance overhead of UFO. For a better comparison, we also measure the performance overhead of ASan and TSan on these benchmarks. The experiment results are listed in Table 8. For *Octane*, the result is the score obtained after running the benchmark. The scores are the time in ms to perform the benchmark computation [9]. The higher the score is, the better the performance. For *SunSpider*, the result is reported in milliseconds after the benchmark runs. The shorter the runtime is, the better the performance. For *Dromaeo JS* and *Dromaeo DOM*, the results are the average number of runs per second after running the benchmarks. The more runs per second, the better the performance is.

Table 8 indicates that UFO has a greater impact on *Chromium* than do ASan and TSan. For *SunSpider*, the runtime overhead of UFO is 5.21x, whereas that of ASan and TSan is 1.86x and 2.61x, respectively. For *Dromaeo JavaScript engine*, UFO introduces 2x runtime overhead, whereas that of ASan is 128% and that of TSan is 162%. The performance of the three tools on this benchmark is comparable because JavaScript is compiled by the JIT compiler and executed without instrumentation. For *Dromaeo HTML rendering*, the average slowdown for UFO is 6.42x, compared to 1.62x for ASan and 3.67x for TSan.

6.2.5 RQ5: Effectiveness of Exploit Mitigation Tools

We use the benchmark *dataset_3* to validate the effectiveness of existing publicly-available exploit mitigation tools. Some tools are ignored because their prototypes are either partially open-source or cannot be successfully run. Table 9 summarizes the experimental results, where "crash" and "no crash" listed in the "original" column are the obtained results when the original program is executed. A cell marked as "mitigation" indicates that the runtime protection provided by the corresponding exploit mitigation tool is observed to be effective; that is, the execution results with and without instrumentation are explicitly inconsistent, including 1) from "crash" to "no crash"; 2) from "no crash" to "no crash but with debugging information generated".

It follows from Table 9 that SlimGuard, FreeGuard, and Guarder clearly protect the programs from five UAFs, while DieHarder and MarkUs explicitly protect against four UAFs. For four to five UAFs, there is no evidence whether these exploit mitigation tools are effective though we believe that they (e.g., MarkUs) may defend against them. From these results, we argue that there is still room for improving the protection ability and debugging (prompt) information of the exploit mitigation tools.

6.2.6 RQ6: Overhead of Exploit Mitigation Tools

We also use the SPEC CPU2006 benchmark to evaluate the runtime overhead and memory overhead of the existing publicly-available exploit mitigation tools.

Runtime Overhead. We measure the runtime overhead by comparing the running time of each program between the version protected by the exploit mitigation tool and the baseline version without any modification. Fig. 10 summarizes the experimental results on 15 programs from the SPEC CPU2006 benchmark. The vertical axis in Fig. 10 represents the ratio of the running time between each protection system and baseline. Values greater than "1.0" indicate that the running time is slower than that of the Linux default allocator, while values less than "1.0" indicate a faster running time. The value "0" indicates that the exploit mitigation tool cannot instrument the program.

As shown in Fig. 10, the runtime overhead of these exploit mitigation tools in most programs is negligible, except for *gcc* and *omnetpp*.MarkUs incurs 157% slowdown on *gcc* and 31% slowdown on *omnetpp*. SlimGuard incurs 155% slowdown on *gcc*. Guarder incurs 131% slowdown on *gcc* and 34% slowdown on *omnetpp*. FreeGuard incurs 28% slowdown on *omnetpp*. DieHarder incurs 133% slowdown on *gcc* and 52% slowdown on *omnetpp*. Overall, the geomean runtime overhead of these exploit mitigation tools is as follows: MarkUs (1.12x), SlimGuard (1.11x), Guarder (1.11x), FreeGuard (1.07x) and DieHarder (1.12x). This indicates that these exploit mitigation tools have a slim impact on the original program speed.

Memory Overhead. We measure the memory overhead by comparing the maximum memory consumption of each

TABLE 9
Results of Different Exploit Mitigation Tools on Nine Known UAFs in Programs (*dataset_3*)

Known UAFs		Original	SlimGuard	DieHarder	FreeGuard	Guarder	MarkUs
Identifier	Program						
CVE-2019-12160	*gohttp*	no crash	mitigation	no crash	mitigation	mitigation	no crash
CVE-2019-11471	*libheif*	no crash	no crash	no crash	no crash	no crash	no crash
CVE-2019-11496	*lrzip*	no crash	no crash	no crash	no crash	no crash	no crash
CVE-2019-6706	*lua*	crash	mitigation	mitigation	mitigation	mitigation	mitigation
CVE-2019-8343	*nasm*	crash	mitigation	mitigation	mitigation	mitigation	mitigation
CVE-2018-20535	*nasm*	no crash	no crash	no crash	no crash	no crash	no crash
CVE-2019-7703	*binaryen*	crash	mitigation	mitigation	mitigation	mitigation	mitigation
CVE-2015-5221	*jasper*	crash	mitigation	mitigation	mitigation	mitigation	mitigation
CVE-2018-20623	*binutils*	no crash	no crash	no crash	no crash	no crash	no crash

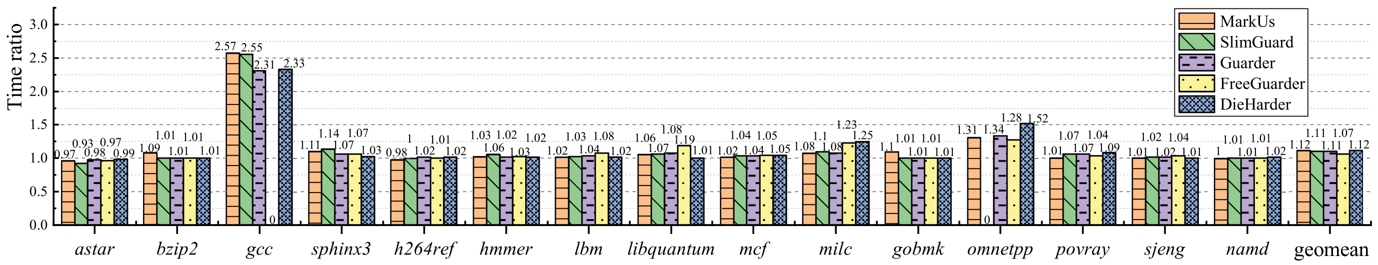
Fig. 10. Runtime overhead of different exploit mitigation tools on the SPEC CPU2006 benchmark.

program between each protection system and the baseline. The memory consumption is obtained using the *maxresident* output of the *time* utility. Fig. 11 summarizes the memory overhead of different exploit mitigation tools. Values greater than "1.0" indicate that the exploit mitigation tools introduce extra memory overhead; the value "0" indicates that the exploit mitigation tool fails to instrument and run the program. As summarized in Fig. 11, the geomean memory overhead of the exploit mitigation tools is as follows: MarkUs (1.14x), SlimGuard (1.25x), Guarder (1.39x), FreeGuard (1.77x) and DieHarder (1.08x). FreeGuard has the largest geomean memory overhead (1.77x) and largest memory overhead (19.9x). The memory overhead of Guarder is less than that of FreeGuard. DieHarder has the lowest geomean memory overhead (1.08x). It is worth mentioning that in contrast to the runtime overhead, the memory overhead varies significantly on different programs.

6.3 Evaluation Summary

Our evaluation demonstrates that static detectors can find most intra-procedural UAFs but only a few inter-procedural UAFs. Considering both effectiveness and efficiency, dynamic detectors are more suitable for detecting inter-procedural UAFs. Among existing dynamic detectors, ASan is most precise. Since the static detectors are mostly more efficient, in practice, users can first employ static detectors for a quick scan and fix, and then resort to dynamic detectors for a complementary check.

Our evaluation also illustrates the effectiveness and limitations of the memory reuse delay mechanism employed by existing dynamic detectors. Both Valgrind and Dr.Memory discover four additional UAFs by delaying memory reuse. However, they fail to find all nine UAFs even when we use as large a quarantine as possible.

Our evaluation demonstrates that although MarkUs is the most advanced tool for improving garbage collection and does well in the performance, it does not show the evidence that it defends against some UAFs. Therefore, its debugging (prompt) information can be improved to show that some UAFs which cannot be exploited or do not lead to program crashes are also protected. In addition, the runtime overhead of the mitigation approaches we evaluate is between 1x and 3x, which is relatively stable. However, the memory overhead may vary significantly with respect to different programs.

6.4 Threats to Validity

The external threat comes mainly from the benchmarks used for the runtime and memory overhead comparison of different dynamic detectors and protectors, because we only use the SPEC CPU2006 benchmark while the latest SPEC CPU 2017 benchmark is not used due to our budgetary constraints. Nonetheless, we believe that this threat is minimal because SPEC CPU2006 is still commonly used [1], [33], and the the real-world programs in SPEC CPU2006 are as diverse and complex as those in SPEC CPU2017.

The experimental results may impact the conclusions. For one thing, we manually analyze the results given by dynamic and static detectors to confirm their validity. However, since we have to manipulate the results produced by dynamic and static detectors (over 7,000 in total), we may have overlooked some issues that could affect the final results. Furthermore, despite repeated experiments, some inaccuracies may inevitably be introduced when evaluating the runtime and memory overhead of the detection and protection approaches.

7 Future Directions

In this section, we discuss potential directions for future research, as listed below.

- The cutting-edge static detection approaches are still not satisfactory. While the fundamental limitations of static analysis cannot be overcome, some issues and disadvantages of existing static UAF detectors could be mitigated by combining static analysis with

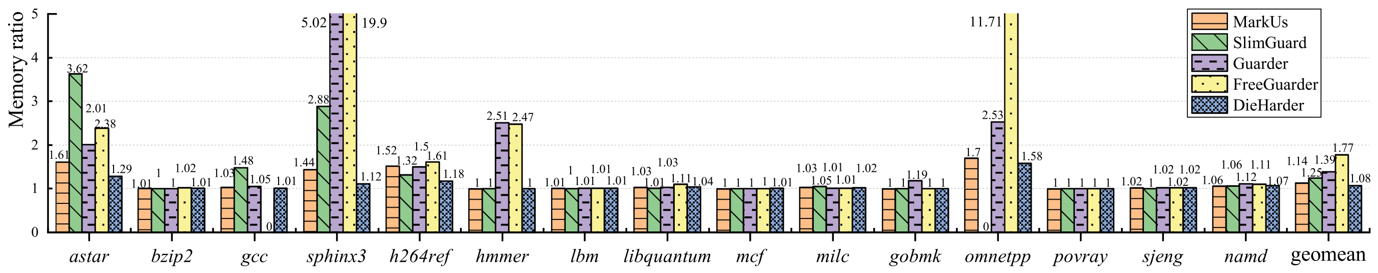
Fig. 11. Memory overhead of different exploit mitigation tools on the SPEC CPU2006 benchmark.

ever-changing advanced techniques such as more efficient symbolic execution approaches and state-of-the-art machine learning techniques.

- When the freed memory is reused, all existing dynamic detection approaches (sanitizers) may miss some UAFs in sequential programs. Therefore, future work can address this limitation.
- At present, researchers are focused on detecting UAFs, but rarely pay attention to how to design test cases to effectively trigger UAFs. Promising future work is to propose new testing approaches to effectively generate test cases that can trigger UAFs.
- Some UAFs, no matter whether or not they can be exploited, are not protected by some existing UAF exploit mitigation approaches. Therefore, the exploit mitigation approaches can be further enhanced to provide with better protection ability and debugging information.
- Although many mitigation approaches can provide temporal safety by changing memory allocation strategies, they aim to reduce runtime overhead while memory overhead has not been fully valued by the community. Future work can focus on memory-constrained scenarios, such as IoT, to provide temporal safety with lower memory overhead.
- In terms of improving garbage collection, the existing approaches are either simple to implement but not completely safe, or completely safe but difficult to implement. In practice, they may release memory objects prematurely, which cause UAFs instead. We can explore more effective ways to release memory objects actively at the appropriate time.
- Researchers for UAFs are mainly focused on C/C++ programs, and there is little research on programs written in emerging languages such as Rust [131]. Therefore, filling this gap is another promising research direction.

8 Conclusion

We have presented a taxonomy of existing UAF detection and exploit mitigation techniques. Additionally, we have conducted an extensive and intensive empirical comparison to evaluate the effectiveness and efficiency of currently available UAF detection and exploit mitigation tools. Our experimental results demonstrate that static detectors can effectively detect intra-procedural UAFs, while dynamic detectors can effectively detect inter-procedural UAFs. Our evaluation also reveals that the exploit mitigation techniques can be further improved to enhance the protection ability and to reduce the memory overhead. Since the potential risks of UAFs are high, we believe this study can help readers understand the progress of existing research as well as the advantages and disadvantages of various UAF defense techniques, shedding light on future research directions for addressing UAFs.

Acknowledgments

The authors would like to thank all reviewers for their insightful and constructive feedback to this work.

References

[1] S. Ainsworth and T. M. Jones, "Markus: Drop-in use-after-free prevention for low-level languages," in *Proce. IEEE Symp. Secur. Privacy*, 2020, pp. 860–860.

[2] B. Wickman *et al.*, "Preventing use-after-free attacks with fast forward allocation," in *Proc. 30th USENIX Secur. Symp.*, 2021, pp. 2453–2470.

[3] H. Yan, Y. Sui, S. Chen, and J. Xue, "Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities," in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2018, pp. 327–337.

[4] Y. Younan, "FreeSentry: Protecting against use-after-free vulnerabilities due to dangling pointers," in *Proc. Annu. Netw. Distrib. Syst. Secur. Symp.*, 2015, pp. 1–15.

[5] Z. Shen and B. Dolan-Gavitt , "Heapexpo: Pinpointing promoted pointers to prevent use-after-free vulnerabilities," in *Proc. Annu. Comput. Secur. Appl. Conf.*, 2020, pp. 454–465.

[6] H. Wang *et al.*, "Typestate-guided fuzzer for discovering use-after-free vulnerabilities," in *Proc. Int. Conf. Softw. Eng.*, 2020, pp. 999–1010.

[7] F. C. Eigler, "Mudflap: Pointer use checking for C/C++," in *Proc. 1st Annu. GCC Developers' Summit*, 2003, pp. 57–70.

[8] K. Vorobyov, N. Kosmatov, J. Signoles, and A. Jakobsson, "Runtime detection of temporal memory errors," in *Proc. Int. Conf. Runtime Verification*, 2017, pp. 294–311.

[9] J. Huang, "UFO: predictive concurrency use-after-free detection," in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2018, pp. 609–619.

[10] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *Proc. USENIX Annu. Techn. Conf.*, 2012, pp. 309–318.

[11] K. Serebryany and T. Iskhodzhanov, "ThreadSanitizer: Data race detection in practice," in *Proc. Workshop Binary Instrum. Appl.*, 2009, pp. 62–71.

[12] K. Serebryany, A. Potapenko, T. Iskhodzhanov, and D. Vyukov, "Dynamic race detection with LLVM compiler," in *Proc. Int. Conf. Runtime Verification*, 2011, pp. 110–114.

[13] S. Dinesh, N. Burow, D. Xu, and M. Payer, "Retrowrite: Statically instrumenting COTS binaries for fuzzing and sanitization," in *Proc. IEEE Symp. Secur. Privacy*, San Francisco, CA, USA, 2020, pp. 1497–1511.

[14] D. W.-King *et al.*, "Egalito: Layout-agnostic binary recompilation," in *Proc. 25th Int. Conf. Architectural Support Program. Lang. Oper. Syst.*, 2020, pp. 133–147.

[15] D. Bruening and Q. Zhao, "Practical memory checking with dr. memory," in *Proc. Annu. IEEE/ACM Int. Symp. Code Gener. Opti.*, 2011, pp. 213–223.

[16] R. Hastings and B. Joyce, "Purify: Fast detection of memory leaks and access errors," in *Proc. Winter USENIX Conf.*, 1991, pp. 125–136.

[17] T. Liu, C. Curtsinger, and E. D. Berger, "Doubletake: Fast and precise error detection via evidence-based dynamic analysis," in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2016, pp. 911–922.

[18] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2007, pp. 89–100.

[19] A. Fioraldi, D. C. D'Elia, and L. Querzoni, "Fuzzing binaries for memory safety errors with QASan," in *Proc. IEEE Secure Develop.*, 2020, pp. 23–30.

[20] J. Feist, L. Mounier, and M.-L. Potet, "Statically detecting use after free on binary code," *J. Comput. Virol. Hacking Tech.*, vol. 10, no. 3, pp. 211–217, 2014.

[21] H. Yan, Y. Sui, S. Chen, and J. Xue, "Machine-learning-guided typestate analysis for static use-after-free detection," in *Proc. Annu. Comput. Secur. Appl. Conf.*, 2017, pp. 42–54.

[22] D. Dewey, B. Reaves, and P. Traynor, "Uncovering use-after-free conditions in compiled code," in *Proc. Int. Conf. Availability, Rel. Secur.*, 2015, pp. 90–99.

[23] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c - A software analysis perspective," in *Proc. Int. Conf. Softw. Eng. Formal Methods*, 2012, pp. 233–247.

[24] C. Cadar *et al.*, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. USENIX Symp. Oper. Syst. Des. Implementation*, 2008, pp. 209–224.

[25] D. Kroening and M. Tautschnig, "CBMC–C bounded model checker," in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2014, pp. 389–391.

[26] M. C. Olesen, R. R. Hansen, J. L. Lawall, and N. Palix, "Coccinelle: Tool support for automated CERT C secure coding standard certification," *Sci. Comput. Program.*, vol. 91, pp. 141–160, 2014.

[27] Clang, Accessed: Mar. 2021. [Online]. Available: http://clang-analyzer.llvm.org/

[28] TscanCode, Accessed: Mar. 2021. [Online]. Available: https://github.com/Tencent/TscanCode/

[29] J. Caballero, G. Grieco, M. Marron, and A. Nappa, "Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities," in *Proc. ACM SIGPLAN Int. Symp. Softw. Testing Anal.*, 2012, pp. 133–143.

[30] E. Van Der Kouwe, V. Nigade, and C. Giuffrida, "DangSan: Scalable use-after-free detection," in *Proc. Eur. Conf. Comput. Syst.*, 2017, pp. 405–419.

[31] D. Liu, M. Zhang, and H. Wang, "A robust and efficient defense against use-after-free exploits via concurrent pointer sweeping," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 1635–1648.

[32] H. Xia *et al.*, "Cherivoke: Characterising pointer revocation using cheri capabilities for temporal memory safety," in *Proc. Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2019, pp. 545–557.

[33] T. Zhang, D. Lee, and C. Jung, "BOGO: Buy spatial memory safety, get temporal memory safety (almost) free," in *Proc. Int. Conf. Architectural Support Program. Lang. Oper. Syst.*, 2019, pp. 631–644.

[34] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "CETS: compiler enforced temporal safety for c," in *Proc. Int. Symp. Memory Manage.*, 2010, pp. 31–40.

[35] S. Nagarakatte, M. M. Martin, and S. Zdancewic, "Watchdog: Hardware for safe and secure manual memory management and full memory safety," in *Proc. Annu. Int. Symp. Comput. Archit.*, 2012, pp. 189–200.

[36] P. Akritidis, "Cling: A memory allocator to mitigate dangling pointers," in *Proc. USENIX Secur. Symp.*, 2010, pp. 177–192.

[37] E. D. Berger and B. G. Zorn, "Diehard: Probabilistic memory safety for unsafe languages," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2006, pp. 158–168.

[38] T. H. Dang, P. Maniatis, and D. Wagner, "Oscar: A practical page-permissions-based scheme for thwarting dangling pointers," in *Proc. USENIX Secur. Symp.* 2017, pp. 815–832.

[39] M. Zhang and S. Zonouz, "Use-after-free mitigation via protected heap allocation," in *Proc. IEEE Conf. Dependable Secure Comput.*, 2018, pp. 1–8.

[40] S. Silvestro, H. Liu, T. Liu, Z. Lin, and T. Liu, "Guarder: A tunable secure allocator," in *Proc. Secur. Symp.*, 2018, pp. 117–133.

[41] B. Liu, P. Olivier, and B. Ravindran, "SlimGuard: A secure and memory-efficient heap allocator," in *Proc. Int. Middleware Conf.*, 2019, pp. 1–13.

[42] H.-J. Boehm and M. Weiser, "Garbage collection in an uncooperative environment," *Softw. Prac. Exp.*, vol. 21, no. 12, pp. 807–820, 1988.

[43] H. Boehm, A. Demers, and M. Weiser, "A garbage collector for c and C++," 2002. [Online]. Available: https://www.hboehm.info/gc/

[44] J. Shin, D. Kwon, J. Seo, Y. Cho, and Y. Paek, "CRCount: Pointer invalidation with reference counting to mitigate use-after-free in legacy C/C++," in *Proc. Annu. Netw. Distrib. Syst. Secur. Symp.*, 2019, 1–15.

[45] X. Fan, Y. Sui, X. Liao, and J. Xue, "Boosting the precision of virtual call integrity protection with partial pointer analysis for C++," in *Proc. ACM SIGPLAN Int. Symp. Softw. Testing Anal.*, 2017, pp. 329–340.

[46] R. Gawlik and T. Holz, "Towards automated integrity protection of C++ virtual function tables in binary programs," in *Proc. Annu. Comput. Secur. Appl. Conf.*, 2014, pp. 396–405.

[47] D. Jang, Z. Tatlock, and S. Lerner, "Safedispatch: Securing C++ virtual calls from memory corruption attacks," in *Proc. Annu. Netw. Distrib. Syst. Secur. Symp.*, 2014, pp. 1–15.

[48] A. Prakash, X. Hu, and H. Yin, "VfGuard: Strict protection for virtual function calls in cots C++ binaries," in *Proc. Annu. Netw. Distrib. Syst. Secur. Symp.*, 2015, pp. 1–15.

[49] P. Sarbinowski, V. P. Kemerlis, C. Giuffrida, and E. Athanasopoulos, "VTPin: Practical Vtable hijacking protection for binaries," in *Proc. Annu. Comput. Secur. Appl. Conf.*, 2016, pp. 448–459.

[50] D. Song *et al.*, "SoK: Sanitizing for security," in *Proc. IEEE Symp. Secur. Privacy*, 2019, pp. 1275–1295.

[51] A. Milburn, H. Bos, and C. Giuffrida, "SafeInit: Comprehensive and practical mitigation of uninitialized read vulnerabilities," in *Proc. 24th Annu. Netw. Distrib. Syst. Secur. Symp.*, San Diego, California, USA, 2017, pp. 1–15.

[52] T. M. Austin, S. E. Breach, and G. S. Sohi, "Efficient detection of all pointer and array access errors," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 1994, pp. 290–301.

[53] W. Xu *et al.*, "From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2015, pp. 414–425.

[54] J.-J. Bai, J. Lawall, Q.-L. Chen, and S.-M. Hu, "Effective static analysis of concurrency use-after-free bugs in linux device drivers," in *Proc. USENIX Annu. Techn. Conf.*, 2019, pp. 255–268.

[55] P. Deligiannis, A. F. Donaldson, and Z. Rakamaric, "Fast and precise symbolic analysis of concurrency bugs in device drivers (t)," in *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2015, pp. 166–177.

[56] D. Engler and K. Ashcraft, "RacerX: Effective, static detection of race conditions and deadlocks," *ACM SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 237–252, 2003.

[57] S. Hong and M. Kim, "Effective pattern-driven concurrency bug detection for operating systems," *J. Syst. Softw.*, vol. 86, no. 2, pp. 377–388, 2013.

[58] V. Vojdani, K. Apinis, V. Rõtov, H. Seidl, V. Vene, and R. Vogler, "Static race detection for device drivers: The goblint approach," in *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2016, pp. 391–402.

[59] J. W. Voung, R. Jhala, and S. Lerner, "Relay: Static race detection on millions of lines of code," in *Proc. Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2007, pp. 205–214.

[60] L. Chirica, D. Martin, T. Dreisbach, J. Peetz, and A. Sorkin, "Two parallel euler run time models: The dangling reference, impostor environment, and label problems," in *Proc. ACM SIGPLAN Notices*, 1973, pp. 141–151.

[61] D. Berry, Z. Erlich, and C. Lucena, "Pointers and data abstractions in high level languages–i: Language proposals," *Comput. Lang.*, vol. 2, no. 4, pp. 135–148, 1977.

[62] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer, "Ccured in the real world," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2003, pp. 232–244.

[63] Y. Oiwa, "Implementation of the memory-safe full ANSI-C compiler," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2009, pp. 259–269.

[64] C. Zhang *et al.*, "Vtrust: Regaining trust on virtual calls," in *Proc. Annu. Netw. Distrib. Syst. Secur. Symp.*, 2016, pp. 1–26.

[65] Y. Sui and J. Xue, "SVF: Interprocedural static value-flow analysis in LLVM," in *Proc. Int. Conf. Compiler Construction*, 2016, pp. 265–266.

[66] F. E. Allen, "Control flow analysis," *ACM SIGPLAN Notices*, vol. 5, no. 7, pp. 1–19, 1970.

[67] J. Afek and A. Sharabani, "Dangling pointer: Smashing the pointer for fun and profit," 2007. [Online]. Available: https://www.blackhat.com/presentations/bh-usa-07/Afek/Whitepaper/bh-usa-07-afek-WP.pdf

[68] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song, "VTINT: Protecting virtual function tables' integrity," in *Proc. Annu. Netw. Distrib. Syst. Secur. Symp.*, 2015, pp. 1–15.

[69] W. Qiang, W. Li, H. Jin, and J. Surbiryala, "MPChecker: Use-after-free vulnerabilities protection based on multi-level pointers," *IEEE Access*, vol. 7, pp. 45961–45977, 2019.

[70] B. Lee *et al.*, "Preventing use-after-free with dangling pointers nullification," in *Proc. Annu. Netw. Distrib. Syst. Secur. Symp.*, 2015, pp. 1–15.

[71] Y. Wang *et al.*, "Dangdone: Eliminating dangling pointers via intermediate pointers," in *Proc. Asia-Pacific Symp. Internetware*, 2018, pp. 6:1–6:10.

[72] G. J. Duck and R. H. Yap, "Effectivesan: Type and memory error detection using dynamically typed C/C++," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2018, pp. 181–195.

[73] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "Ccured: Type-safe retrofitting of legacy software," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 3, pp. 477–526, 2005.

[74] G. C. Necula, S. McPeak, and W. Weimer, "CCured: Type-safe retrofitting of legacy code," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2002, pp. 128–139.

[75] M. S. Simpson and R. K. Barua, "MemSafe: Ensuring the spatial and temporal memory safety of C at runtime," *Softw. Pract. Exp.*, vol. 43, no. 1, pp. 93–128, 2013.

[76] W. Xu, D. C. DuVarney, and R. Sekar, "An efficient and backwards-compatible transformation to ensure memory safety of C programs," *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, vol. 29, no. 6, pp. 117–126, 2004.

[77] S. H. Yong and S. Horwitz, "Protecting C programs from attacks via invalid pointer dereferences," in *Proc. 11th ACM SIGSOFT Symp. Found. Softw. Eng. 9th Eur. Softw. Eng. Conf., Helsinki, Finland*, 2003, pp. 307–316.

[78] N. W. Filardo, *et al.*, "Cornucopia: Temporal safety for CHERI heaps," in *Proc. IEEE Symp. Secur. Privacy*, 2020, pp. 1507–1524.

[79] S. Nagarakatte, M. M. Martin, and S. Zdancewic, "Watchdoglite: Hardware-accelerated compiler-based pointer checking," in *Proc. Annu. IEEE/ACM Int. Symp. Code Gener. Opt.*, 2014, pp. 175–184.

[80] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address obfuscation: An efficient approach to combat a broad range of memory error exploits," in *Proc. USENIX Secur. Symp.*, 2003, pp. 291–301.

[81] V. Iyer, A. Kanitkar, P. Dasgupta, and R. Srinivasan, "Preventing overflow attacks by memory randomization," in *Proc. IEEE Int. Symp. Softw. Rel. Eng.*, 2010, pp. 339–347.

[82] V. B. Lvin, G. Novark, E. D. Berger, and B. G. Zorn, "Archipelago: Trading address space for reliability and security," in *Proc. Int. Conf. Architectural Support Program. Lang. Oper. Syst.*, 2008, pp. 115–124.

[83] G. Novark and E. D. Berger, "Dieharder: securing the heap," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2010, pp. 573–584.

[84] S. Silvestro, H. Liu, C. Crosser, Z. Lin, and T. Liu, "Freeguard: A faster secure heap allocator," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 2389–2403.

[85] C. Valasek, "Understanding the low fragmentation heap," *Black Hat USA*, 2010. [Online]. Available: http://illmatics.com/Understanding_the_LFH.pdf

[86] E. Van Der Kouwe , T. Kroes, C. Ouwehand, H. Bos, and C. Giuffrida, "Type-after-type: Practical and complete type-safe memory reuse," in *Proc. Annu. Comput. Secur. Appl. Conf.*, 2018, pp. 17–27.

[87] D. Dhurjati and V. Adve, "Efficiently detecting all dangling pointer uses in production servers," in *Proc. Int. Conf. Dependable Syst. Netw.*, 2006, pp. 269–280.

[88] M. V. Yason, "Understanding IE's new exploit mitigations: The memory protector and the isolated heap," 2014. [Online]. Available: http://securityintelligence.com/understanding-ies-new-exploit-mitigations-the-memory-protector-and-the-isolated-heap

[89] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of c," in *Proc. USENIX Annu. Techn. Conf., Gen. Track*, 2002, pp. 275–288.

[90] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney, "Region-based memory management in cyclone," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2002, pp. 282–293.

[91] D. Dhurjati, S. Kowshik, and V. Adve, "Safecode: Enforcing alias analysis for weakly typed languages," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2006, pp. 144–157.

[92] D. R. Edelson, "Smart pointers: They're smart, but they're not pointers," in *Proc. C++ Conf.*, 1992, pp. 1–19.

[93] D. Bounov, R. G. Kici, and S. Lerner, "Protecting C++ dynamic dispatch through vtable interleaving," in *Proc. Symp. Netw. Distrib. Syst. Secur.*, 2016, pp. 1–15.

[94] N. Burow, D. McKee, S. A. Carr, and M. Payer, "Cfixx: Object type integrity for C++ virtual dispatch," in *Proc. Symp. Netw. Distrib. Syst. Secur.*, 2018, pp. 1–14.

[95] M. Bauer and C. Rossow, "Novt: Eliminating C++ virtual calls to mitigate vtable hijacking," in *Proc. IEEE Eur. Symp. Secur. Privacy*, 2021, pp. 863–879.

[96] J. Ye, C. Zhang, and X. Han, "Poster: Uafchecker: Scalable static detection of use-after-free vulnerabilities," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2014, pp. 1529–1531.

[97] K. Zhu, Y. Lu, and H. Huang, "Scalable static detection of use-after-free vulnerabilities in binary code," *IEEE Access*, vol. 8, pp. 78713–78725, 2020.

[98] K. Vorobyov, N. Kosmatov, and J. Signoles, "Detection of security vulnerabilities in c code using runtime verification: An experience report," in *Proc. Int. Conf. Tests Proofs*, 2018, pp. 139–156.

[99] K. Vorobyov, J. Signoles, and N. Kosmatov, "Shadow state encoding for efficient monitoring of block-level properties," *ACM SIGPLAN Notices*, vol. 52, no. 9, pp. 47–58, 2017.

[100] Y. Cai, B. Zhu, R. Meng, H. Yun, L. He, P. Su, and B. Liang, "Detecting concurrency memory corruption vulnerabilities," in *Proc. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2019, pp. 706–717.

[101] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Handbook of Satisfiability*, Adv. Comput., vol. 58, pp. 117–148, 2003.

[102] K. Vorobyov and P. Krishnan, "Comparing model checking and static program analysis: A case study in error detection approaches," in *Proc. SSV*, 2010, pp. 1–7.

[103] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surveys*, vol. 51, no. 3, pp. 1–39, 2018.

[104] D. Marjamäki, "Cppcheck: A tool for static C/C++ code analysis," 2013. [Online]. Available: http://cppcheck.sourceforge.net

[105] T. Kremenek, "Finding software bugs with the clang static analyzer," *Apple Inc*, 2008. [Online]. Available: https://llvm.org/devmtg/2008-08/Kremenek_StaticAnalyzer.pdf

[106] Detecting Critical Control Flow with Clang Static Analyzer, March. 2021 (last accessed). [Online]. Available: http://llvm.org/devmtg/2018–04/slides

[107] Juliet Test Suite 1.3, Accessed: Mar. 2021. [Online]. Available: https://samate.nist.gov/SRD/testsuite.php/

[108] D. Bruening and S. Amarasinghe, "Efficient, transparent, and comprehensive runtime code manipulation," Ph.D. dissertation, Dept. Elect. Eng. Massachusetts Inst. Technol., MA, USA, 2004.

[109] The Kernel Address Sanitizer, Accessed: Mar. 2021. [Online]. Available: https://www.kernel.org/doc/html/latest/dev-tools/kasan.html

[110] HWAddressSanitizer, Accessed: Mar. 2021. [Online]. Available: https://www.kernel.org/doc/html/latest/dev-tools/kasan.html

[111] Mudflap Pointer Debugging, Accessed: Mar. 2021. [Online]. Available: https://gcc.gnu.org/wiki/Mudflap_Pointer_Debugging/

[112] M. Zalewski, "American fuzzy lop," 2014. [Online]. Available: http://lcamtuf.coredump.cx/afl/

[113] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as Markov chain," *IEEE Trans. Softw. Eng.*, vol. 45, no. 5, pp. 489–506, May 2019.

[114] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proc. ACM/IEEE Int. Conf. Autom. Softw. Eng.*, 2018, pp. 475–485.

[115] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *Proc. IEEE Symp. Secur. Privacy*, 2018, pp. 711–725.

[116] C. Lyu *et al.*, "MOPT: Optimized mutation scheduling for fuzzers," in *Proc. USENIX Secur. Symp.*, 2019, pp. 1949–1966.

[117] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A practical concolic execution engine tailored for hybrid fuzzing," in *Proc. USENIX Secur. Symp.*, 2018, pp. 745–761.

[118] M.-D. Nguyen, S. Bardin, R. Bonichon, R. Groz, and M. Lemerre, "Binary-level directed fuzzing for use-after-free vulnerabilities," in *Proc. 23rd Int. Symp. Res. Attacks, Intrusions Defenses*, San Sebastian, Spain, 2020, pp. 47–62.

[119] J. Huang, P. O. Meredith, and G. Rosu, "Maximal sound predictive race detection with control flow abstraction," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2014, pp. 337–348.

[120] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2008, pp. 337–340.

[121] Z. Shen and B. Dolan-Gavitt , "Heapexpo: Pinpointing promoted pointers to prevent use-after-free vulnerabilities," in *Proc. Annu. Comput. Secur. Appl. Conf., Virtual Event* / Austin, TX, USA, 2020, pp. 454–465.

[122] R. Sharifi and A. Venkat, "Chex86: Context-sensitive enforcement of memory safety via microcode-enabled capabilities," in *Proc. ACM/IEEE 47th Annu. Int. Symp. Comput. Archit.*, 2020, pp. 762–775.

[123] C. DeLozier, R. Eisenberg, S. Nagarakatte, P.-M. Osera, M. M. Martin, and S. Zdancewic, "Ironclad C++: A library-augmented type-safe subset of C++," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2013, pp. 287–304.

[124] *MemGC: Use-After-Free Exploit Mitigation in Edge and IE on Windows 10*, Accessed: Mar. 2021. [Online]. Available: https://securityintelligence.com/memgc-use-after-free-exploit-mitigation-in-edge-and-ie-on-windows-10/

[125] S. Das, R. H. Unnithan, A. Menon, C. Rebeiro, and K. Veezhinathan, "SHAKTI-MS: A RISC-V processor for memory safety in C," in *Proc. ACM SIGPLAN/SIGBED Int. Conf. Lang. Compilers Tools Embedded Syst*, 2019, pp. 19–32.
[126] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, "Intel MPX explained: A cross-layer analysis of the intel MPX system stack," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 2, no. 2, pp. 1–30, 2018.
[127] A.-A. Hariri, S. Zuckerbraun, and B. Gorenc, "Abusing silent mitigations," *BlackHat USA*, 2015. [Online]. Available: https://www.ecirtam.net/autoblogs/autoblogs/lamaredugoffrblog_6aa4265372739b936776738439d4ddb430f5fa2e/media/2b8f35e4.WP-Hariri-Zuckerbraun-Gorenc-Abusing_Silent_Mitigations.pdf
[128] M. Daniel, J. Honoroff, and C. Miller, "Engineering heap overflow exploits with javascript," in *Proc. 2nd USENIX Workshop Offensive Technol.*, 2008. [Online]. Available: http://www.usenix.org/events/woot08/tech/full_papers/daniel/daniel.pdf
[129] A. Sotirov, "Heap feng shui in javascript," *Black Hat Europe*, 2007. [Online]. Available: http://www.phreedom.org/research/heap-feng-shui/heap-fengshui.html
[130] Scudo Malloc, Accessed: Mar. 2021. [Online]. Available: https://llvm.org/docs/ScudoHardenedAllocator.html
[131] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, "Safe systems programming in rust: The promise and the challenge," *Commun. ACM*, vol. 64, no. 4, pp. 144–152, 2021.

Binfa Gui received the ME degree from the Nanjing University of Science and Technology, China, in 2021. He is currently working toward the master's degree under supervision by Professor Wei Song. He is currently a software engineer with ByteDance. His research interests include program analysis, software security, and software engineering.

Wei Song (Senior Member, IEEE) received the PhD degree from Nanjing University, China. He is currently a full professor with the School of Computer Science and Engineering, Nanjing University of Science and Technology, China. He was a visiting scholar with Technische Universität München, Germany. He was invited to the Schloss Dagstuhl Seminar "Integrating Process-Oriented and Event-Based Systems" held in August 2016. He has authored or coauthored in top computer science journals, including *IEEE Transactions on Cloud Computing*, *IEEE Transactions on Dependable and Secure Computing*, *IEEE Transactions on Knowledge and Data Engineering*, *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Services Computing*, and *IEEE Transactions on Software Engineering*, and premier software engineering conferences, including ASE, ESEC/FSE, ICSE, and ISSTA. His research interests include software engineering, services and cloud computing, program analysis, and process mining.

Hailong Xiong received the BS degree in 2020 from the Nanjing University of Science and Technology, China, where he is currently working toward the master's degree with the School of Computer Science and Engineering. His research interests include program analysis, software security, and software engineering.

Jeff Huang (Member, IEEE) received the PhD degree from the Hong Kong University of Science and Technology in 2012. From 2013 to 2014, he was a postdoctoral researcher with the University of Illinois at Urbana-Champaign. He joined Texas A&M University, College Station, TX, USA, in 2014, where he is currently an associate professor with the Department of Computer Science & Engineering. He has authored or coauthored extensively in premier software engineering conferences and journals, including PLDI, OOPSLA, ICSE, FSE, ISSTA, *ACM Transactions on Software Engineering and Methodology*, and *IEEE Transactions on Software Engineering*. His research interests include developing intelligent techniques and tools for improving software performance and reliability based on fundamental program analyses and programming language theory.