Top of the Heap: Efficient Memory Error Protection of Safe Heap Objects

Kaiming Huang
The Pennsylvania State University
State College, PA, USA
kzh529@psu.edu

Mathias Payer
École Polytechnique Fédérale de Lausanne
Lausanne, Switzerland
mathias.payer@nebelwelt.net

Zhiyun Qian
University of California, Riverside
Riverside, CA, USA
zhiyunq@cs.ucr.edu

Jack Sampson
The Pennsylvania State University
State College, PA, USA
jms1257@psu.edu

Gang Tan
The Pennsylvania State University
State College, PA, USA
gxt29@psu.edu

Trent Jaeger
University of California, Riverside
Riverside, CA, USA
trentj@ucr.edu

Abstract

Heap memory errors remain a major source of software vulnerabilities. Existing memory safety defenses aim at protecting all objects, resulting in high performance cost and incomplete protection. Instead, we propose an approach that accurately identifies objects that are inexpensive to protect, and design a method to protect such objects comprehensively from all classes of memory errors. Towards this goal, we introduce the Uriah system that (1) statically identifies the heap objects whose accesses satisfy spatial and type safety, and (2) dynamically allocates such "safe" heap objects on an isolated safe heap to enforce a form of temporal safety while preserving spatial and type safety, called *temporal allocated-type safety*. Uriah finds 72.0% of heap allocation sites produce objects whose accesses always satisfy spatial and type safety in the SPEC CPU2006/2017 benchmarks, 5 server programs, and Firefox, which are then isolated on a safe heap using Uriah allocator to enforce temporal allocated-type safety. Uriah incurs only 2.9% and 2.6% runtime overhead, along with 9.3% and 5.4% memory overhead, on the SPEC CPU 2006 and 2017 benchmarks, while preventing exploits on all the heap memory errors in DARPA CGC binaries and 28 recent CVEs. Additionally, using existing defenses to enforce their memory safety guarantees on the unsafe heap objects significantly reduces overhead, enabling the protection of heap objects from all classes of memory errors at more practical costs.

CCS Concepts

• **Security and privacy → Software security engineering**.

Keywords

Heap Memory Errors, Memory Safety, Software Security, Program Analysis, Software-based Fault Isolation, Secure Allocator.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CCS '24, October 14–18, 2024, Salt Lake City, UT, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690310

ACM Reference Format:
Kaiming Huang, Mathias Payer, Zhiyun Qian, Jack Sampson, Gang Tan, and Trent Jaeger. 2024. Top of the Heap: Efficient Memory Error Protection of Safe Heap Objects. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3690310

1 Introduction

Memory errors in C/C++ programs continue to cause the most significant security problems. The White House recently stated that future software should be memory safe [126]. According to the NSA [91], Microsoft [77], and Google [124], ≈70-80% of vulnerabilities are caused by memory errors. Known since the Anderson Report [6], memory errors have led to high-impact attacks such as the Morris Worm [103], Slammer [82], Heartbleed [100], and Blastpass [8]. Memory errors are exploited by ransomware that costs organizations billions of dollars [30, 114, 131], and are even found by studies [14, 116, 137] of software produced by LLM-based code generators [33, 94, 98]. A wide variety of notable attack techniques have been discovered to exploit memory errors [15–18, 39, 49, 53, 63, 67, 68, 93, 101, 115, 135, 136, 150, 151] that enable attackers to gain control of process memory.

Among these errors, *heap* memory errors are notorious for their prevalence and severity, posing significant challenges for developers. There are several reasons why heap memory is more prone to errors. First, heap memory may be used to store objects of variable size, which can also change through reallocations. Errors in tracking sizes of these objects can violate ***spatial safety***, where memory operations access locations outside expected bounds. Second, heap objects are often complex, consisting of hierarchically structured data types that may be cast into multiple views to different formats depending on the program context. Mistakes in interpreting the layouts of data structures can violate ***type safety***, where memory operations may access objects at invalid offsets and/or using incorrect data types. Third, unique to heap memory is the challenge of managing dynamic allocation and deallocation. Unlike stack memory, heap memory management in C/C++ requires manual intervention, causing errors that violate ***temporal safety*** where pointers may not be initialized before use (i.e., UBI) and/or may be used after deallocation (i.e., UAF). Finally, heap objects may be accessed across multiple threads complicating the challenges above.

Despite years of efforts to develop heap memory safety defenses, heap data is not systematically protected from all three classes of memory errors comprehensively. Researchers have produced techniques to enforce memory safety requirements to prevent spatial memory errors [4, 27, 83, 110], type memory errors [38, 51, 59, 146], and temporal memory errors [13, 31, 58, 72, 73, 84, 130, 139–143, 148, 149] independently, as well as multiple classes of memory errors [25, 28, 29, 69, 86, 88, 105]. However, none of these defenses have been widely-adopted in production (although several are used in fuzz testing), as they often incur significant overheads. Production defenses such as AutoSlab [66], kalloc_type [102], and PartitionAlloc [36] only focus on temporal safety, Microsoft Defender [78] preserves the integrity of heap metadata for code with memory errors. However, such defenses only make exploitation harder (i.e., memory errors and exploitations are still possible). Thus, a lingering question is how to introduce heap protection that can be adopted in practice (i.e., is effective and efficient) and can serve as a foundation for enforcing memory safety comprehensively for the entire heap.

We observe that producing defenses for heap memory is very difficult because any defense must satisfy three ***mutually conflicting*** properties: An ideal defense must provide memory protection against all ***classes*** of memory errors with ***coverage*** across all memory objects for reasonable ***cost*** in performance and memory overheads. The past 20 years of research has demonstrated that it is very hard to find a silver bullet that achieves all three goals simultaneously [41]. Traditionally, researchers have focused on the goal of complete coverage across all objects, but some memory operations may occur frequently causing a high overhead for the defenses, resulting in trade-offs on protecting only one or some of the memory error classes [4, 11, 25, 80, 83, 105]. An alternative approach, exemplified by DataGuard [40], targets protection against all memory error classes, but only for objects that can be protected efficiently, finding that over 90% of stack objects can be protected for 4.3% overhead on SPEC CPU2006 benchmarks. DataGuard enables a significant fraction of "safe" stack objects to be protected comprehensively from memory errors, but does not address heap objects nor consider the impact on the remaining unsafe objects.

In this paper, we examine the challenge of memory safety validation to protect *safe heap objects*[1], whose memory references must comply with all classes of memory safety, from being corrupted by memory errors on other unsafe objects for *low cost*. Our broader vision is that such an approach will reduce the overhead of applying defenses to protect the remaining heap objects comprehensively as well. Similar ideas have been examined in selective symbolic execution [117] and removing redundant type checks [145]. We will demonstrate: (1) this idea makes heap memory protection efficient, (2) protecting safe heap objects against all memory error classes provides promising exploit mitigation, and (3) applying existing defenses only to the remaining unsafe heap objects offers the same security guarantees at lower cost.

One idea would be to apply the techniques for stack memory safety validation [40] to heap objects. Unfortunately, the original methods do not apply to the heap. First, heap objects typically have more complex representations and a greater number of aliases, are involved in more dynamic changes and shared between threads, and have much longer lifetimes than stack objects. As a result, we must design new approaches for static safety validation, while addressing the greater number of false positives that may be produced by static analysis of heap usage. Second, no general static analysis exists to validate temporal safety for heap objects. As a result, defenses have been proposed to enforce temporal safety at runtime, but these efforts have shown that fully eliminating temporal errors at runtime incurs high performance costs [28, 69]. Thus, we must adapt this idea strategically and effectively to enforce temporal safety at runtime, while keeping costs low.

To address these challenges, we develop the Uriah system to (1) validate heap objects whose accesses must always satisfy ***spatial*** and ***type safety*** and (2) enforce ***isolation*** and a form of ***temporal safety*** over validated heap objects in (1) to preserve their spatial and type safety at runtime. First, Uriah provides static memory safety validation methods that ensure that all operations that access a heap object through all its aliases must satisfy both spatial and type safety, while accounting for the challenges above, including reallocations and concurrent access. Second, given results showing that the type-safe memory reuse can be enforced efficiently to prevent temporal errors exploits [66, 81, 129], Uriah leverages this idea to enforce a more restrictive form of type-safe memory reuse to maintain spatial and type safety at runtime for all the validated heap objects on a separate *safe heap* against temporal exploits, and isolate it from memory errors in accesses to objects on the regular (unsafe) heap. This version of type-safe memory reuse, called *temporal allocated-type safety*, only allocates objects with the same fields of the same size and type for all fields in each memory region within an isolated safe heap. The result is the following ***security guarantee***: For objects whose own accesses all satisfy spatial and type safety statically, which we call *safe objects* in this paper, Uriah ensures that no access to any object can violate the spatial and type safety of any safe object at runtime. Uriah protects safe objects from any memory errors in accesses to unsafe objects by separating safe objects onto a safe heap isolated using SFI. Further, Uriah protects the spatial and type safety of safe objects at runtime from temporal exploits that leverage dangling pointers to the safe heap by enforcing temporal allocated-type safety.

By employing Uriah, we find major benefits in protecting memory safety for heap objects. First, Uriah finds that 72.0% of heap allocation sites can be validated to produce objects that satisfy spatial and type safety statically for a variety of programs, including Firefox, servers (nginx and httpd), and the SPEC CPU2006/2017 benchmarks. This represents a substantial increase relative to the 39.9% found equivalently safe by employing prior techniques (see Table 3). Uriah protects all objects produced at these allocation sites by isolation on the safe heap. This accounts for 73.6% of the memory object allocations made by the SPEC CPU2006 benchmarks at runtime. Second, Uriah exhibits only 2.9% and 2.6% runtime and 9.3% and 5.4% memory overhead for the SPEC CPU2006 benchmarks and the SPEC CPU2017 benchmarks, respectively. We compare Uriah's memory and runtime overheads to seven state-of-the-art defenses (see Table 4), finding that Uriah has lower overheads for allocating all objects for all but one system, but stronger security guarantees. As a defense, Uriah prevents exploitation of all 102 heap vulnerabilities in Cyber Grand Challenge binaries and 28 recent heap CVEs. More importantly, while Uriah does not protect objects on

[1] "Heap objects" refers to all objects allocated to an allocation site, per static analysis.

```
1  static uint64_t read_indexed_address (uint64_t idx,
2                        struct comp_unit *unit){
3    ...
4    bfd_byte *info_ptr; // used for memory access
5    size_t offset;
6    offset += unit->dwarf_addr_offset;
7    if (offset < unit->dwarf_addr_offset
8        || offset > file->dwarf_addr_size
9        || file->dwarf_addr_size - offset < unit->offset_size)
10       // wrong check always true
11     return 0;
12   info_ptr = file->dwarf_addr_buffer + offset;
13   return bfd_get_64 (unit->abfd, info_ptr);
14   //unsafe memory access
15 }
16
17 static struct dwarf_block * read_blk (bfd *abfd,
18         bfd_byte **ptr, bfd_byte *end, size_t size){
19   ...
20   struct dwarf_block *block;
21   block = (struct dwarf_block*) bfd_alloc (abfd, ...)
22 }
```

Listing 1: Spatial memory error for vulnerability CVE-2023-1579, which allows attackers to exploit a buffer over-read through `unit` to access objects aliased by `block`

the unsafe heap, it reduces the number of unsafe objects, lowering the effort to enforce memory safety overall. To examine this, we combine Uriah with existing defenses, TDI [81] and CAMP [69], saving ≈70% of their original overhead to efficiently enforce their security guarantee over the unsafe heap.

The contributions of Uriah include:

- We build the Uriah system for heap memory safety enforcement, which provides memory safety validation to determine which heap objects satisfy spatial and type safety and a heap allocator that protects these objects by enforcing temporal allocated-safety over an isolated safe heap. Heap objects in the safe heap are isolated from memory errors efficiently.
- Uriah includes new analyses for spatial and type safety validation that address the challenges of heap objects, such as complex representations, dynamic resizing, multi-threading, and up/down-casts. In addition, Uriah applies new symbolic execution analyses to validate unsafe cases, converting them to safe cases when all unsafe aliases lack feasible execution paths.
- Our evaluation shows that Uriah protects 72.0% of the allocation sites of CPU2006/2017 benchmarks, 5 server programs, and Firefox, with only 2.9% and 2.6% runtime and 9.3% and 5.4% memory overheads, on the SPEC CPU2006/2017 benchmarks, respectively, while preventing exploits of 28 known CVEs and CGC programs.

2 Motivation

In this section, we motivate our research by examining how heap memory errors may be exploited and past research on mitigating such errors, showing the limitations, and proposing our ideas.

2.1 Exploiting Heap Memory Errors

Despite decades of work on mitigation, memory errors remain an active source of vulnerabilities, particularly for the heap. Over the past 10 years, there have been more than 10,000 CVEs of heap memory errors, including hundreds in the current calendar year.

A program contains a memory error when a program's memory reference may violate a memory safety property. Researchers have identified three classes of memory safety properties defined below.

- ***Spatial Safety***: Every pointer that may reference the object must only access memory within the object's allocated region.
- ***Type Safety***: Every pointer that may reference the object must only access the same data types for each offset and each field.

	Defense	*Spatial*	*Type*	*Temporal*	*Scope*
Uriah (*this work*)	A	✓	✓	✓†	H*
CCured [20, 86, 87]	R	✓	✓	×	S & H
Checked-C [29, 148] ***EffectiveSan*** [28]	R	✓	✓	UAF	S & H
ASan [105]	R	✓	×	UAF	S & H
Baggy-Bounds [4] ***SoftBound*** [83] ***Low-Fat*** [26, 27]	R	✓	×	×	S & H
HexType [51] ***TypeSan*** [38] ***CaVer*** [59]	R	×	✓	×	S & H
DangSan [130] ***DangNull*** [58] ***FreeSentry*** [142]	R	×	×	UAF	H
SAFECode [25]	A	×	✓	UAF†	S & H
FFMalloc [134] ***MarkUs*** [2]	A	×	×	UAF	H
Cling [3]	A	×	×	UAF†	H
Type-After-Type [129]	A	×	×	UAF†	S & H
DieHard [11] ***DieHarder*** [90] ***FreeGuard*** [108] ***TDI*** [81]	A	✓	×	✓†	H
CAMP [69]	A&R	✓	×	UAF	S & H
DataGuard [40]	A	✓	✓	✓	S*
Safe Stack [55]	A	✓	×	×	S*

Table 1: Comparison of Uriah with Previous Works. *Defense* includes custom allocator (A) or runtime checks (R). "✓" indicates an approach protects that class of memory errors and "×" when it does not. UAF in the *Temporal* column indicates that the protection includes use-after-free, double-free, and invalid-free, but not use-before-initialization. †indicates the approach enforces a well-defined subset of temporal safety, such a temporal type safety [129]. The *Scope* column indicates the protection covers stack (S) and/or heap (H), possibly for a subset of objects that adhere to a property (e.g., are "safe") "∗".

- ***Temporal Safety***: Every pointer that may reference the object must not be used to access the object's allocated region before being assigned to the object nor after the object's deallocation.

Listing 1 shows an example of a heap vulnerability (CVE-2023-1579) in `binutils`. A size check against the `unit->offset_size` field at line 9 always returns true, because this is the incorrect field for this size check; the check should be made against the `unit->addr_size` field. This error enables the `info_ptr` pointer assigned in line 12 to reference out-of-bounds memory, eventually resulting in heap buffer over-reads in the function called at line 13. In this case, the `block` allocated at line 21 is often exploited, and, as it may alias any data on the heap allocated by bfd, an attacker can read any heap data by illicitly accessing memory through `block`.

To exploit heap memory errors, attackers utilize a memory error in accessing one object to exploit other objects. We refer to an object whose accesses have memory errors as a ***vulnerable object***. In this example, the vulnerable object aliased by `unit` has a spatial memory error that permits attackers to read outside its allocation. We call the objects that can be accessed illicitly due to a memory error, ***target objects***. In this example, heap objects aliased by `block` are target objects. Existing, commonly-adopted defenses cannot prevent the exploitation effectively. For example, ASLR [12] can be bypassed by *disclosure attack* [21, 113] and cannot prevent illicit reads, as

the target object is often allocated at a known or configurable [23, 115, 132] offset. Spatial defenses, such as ASan [105], are known to present high costs. Other defenses [36, 66, 102] only focus on temporal safety, neglecting spatial and type errors.

2.2 Limitations of Heap Memory Defenses

Researchers have explored a variety of techniques to enforce a subset or all classes of memory safety, but none of these techniques have yet found broad acceptance in production systems. Most current defenses aim for full ***coverage*** of vulnerable objects, but have issues enforcing comprehensive memory safety for all ***classes*** of memory errors for reasonable performance and memory ***costs***, hindering their deployment. We characterize software-based defenses in Table 1. Initially, runtime defenses were designed for a single class of memory errors, such as spatial error protections [4, 27, 83, 86, 105] that restrict memory operations on objects within bounds metadata, type error protections [28, 38, 51, 59, 73, 146] that validate type casts at runtime, and temporal error protections that often seek to invalidate pointers when memory is deallocated [28, 58, 130, 142, 148]. However, despite optimizations, these defenses have performance overheads that prevented their adoption for production systems. Defenses that enforce memory safety for multiple classes of memory errors, such as ASan [105] and EffectiveSan [28], are applied commonly for detecting vulnerabilities in fuzzing. ASan performance has been optimized [147], but its use of red-zones can be circumvented, limiting its effectiveness in preventing exploits. Checked-C [29] introduces additional language semantics for spatial and type safety, but faces backward compatibility issues with legacy codebases. Checked-C was extended to enforce temporal safety with compatible use of fat pointers [148], but the original compatibility issues of Checked-C persist.

More recently, researchers have proposed defenses that leverage secure allocators [2, 11, 25, 69, 81, 90, 108, 129, 134] to prevent exploitation to target objects through temporal errors. Most of these defenses aim to prevent UAF [2, 25, 69, 129, 134], but some works [11, 81, 90, 108] include prevention of UBI. Interestingly, researchers have found that enforcing type-safe memory reuse [25, 81, 129] is an effective approach to prevent dangling pointer misuse by associating data regions with a single data type, as such exploits typically result in type confusion. Unfortunately, the pioneer SAFECode [25] ignores spatial safety and requires an exact alias analysis to apply its per-function, per-type allocation pool for all heap objects, so it can only be applied to small programs and is expensive. Recent work, Type-After-Type [129] demonstrates such a technique can be enforced efficiently. However, it places objects from the same allocation site into the same pool, risking type errors if multiple types are dynamically determined and allocated at that site. Other works focus on restricting memory reuse [2, 134], trading memory overhead for preventing dangling pointer misuse (See Table 4). In a broader sense of memory safety, the impact for all the secure allocators is limited as they do not account for all classes of memory errors, as attacks on spatial and/or type errors remain possible on objects allocated by these systems.

2.3 Protecting Target Objects without Checks

In practice, attacks often exploit memory errors in accesses to vulnerable objects to corrupt target objects [16, 17, 39, 49, 63, 67, 135, 136, 150, 151]. As a result, defenses that prevent accesses to vulnerable objects from reaching target objects can block many attack options. Recent work has proposed an approach to achieve this goal in the classes-coverage-cost space of memory defenses. DataGuard [40] proposes to protect stack objects that pass a static memory safety validation for all classes of memory safety from memory errors in accesses to other objects, finding that protection can be achieved at low cost using multiple stacks, such as Safe Stack [55], without runtime checks. Although this solution is biased toward enforcement of all classes of memory safety at a low cost at the expense of complete coverage, such a trade-off is worthwhile, as over 85% of stack objects only have memory references that satisfy all classes of memory safety in a large study of over 1,200 Linux packages [43]. The overhead for protecting memory-safe stack objects in SPEC CPU2006 benchmark programs is only 4.3% [40].

Thus, a research question is whether and how memory safety validation can be applied to identify heap objects that can be protected from all classes of memory errors cheaply. Unfortunately, heap usage introduces challenges that Dataguard's stack analyses cannot handle. For spatial safety, DataGuard's use of *value range analysis* [109] relies on the predefined, fixed-size objects. However, heap objects may be resized dynamically, and even reallocated, as well as being accessed across multiple threads. For type safety, DataGuard only validates integer type casts, considering any cast between non-identical compound types as unsafe. Heap objects are much more likely to be involved in the latter (approximately 60%, see Table 3). For temporal safety, DataGuard performs a static liveness analysis, which is not feasible for heap objects. Heap objects have much longer lifetimes than stack objects, so researchers have not yet produced a satisfactory static analysis to validate the temporal safety of heap objects. In addition, all these static analyses suffer more false positives (i.e., falsely classifying objects as unsafe) due to the greater over-approximation of aliases to heap objects. DataGuard utilizes symbolic execution to validate legal executions to remove a significant fraction of these false positives, but the depths of the def-use chains for aliases to heap objects often exceed the limits that DataGuard uses to prevent path explosion.

As a result, a method to protect heap objects from memory errors that retains low overhead requires a variety of different problems to be solved. The spatial and type memory safety validation analyses must be extended to solve challenges specifically for validating heap objects effectively. The additional complexity of these analysis problems and heap analysis in general will result in more false positives, so we need alternative methods to resolve the greater number of false positives that will occur while retaining good scalability. Finally, we need runtime methods to enforce temporal safety while preserving spatial and type safety. Inspired by the fact that illicit memory accesses caused by temporal errors require (re)use memory of different types, existing temporal defenses for heap objects enforce type-safe memory reuse to ensure only objects of the same type are allocated in each memory region [25, 129], but the aim for complete coverage introduces limitations (See Section 2.2).

3 Uriah Overview

To ensure the security guarantee (Section 1) of Uriah, we aim to: (1) statically validate the heap objects whose accesses must satisfy spatial and type safety, which form its *allocated-type* (Definition 1),

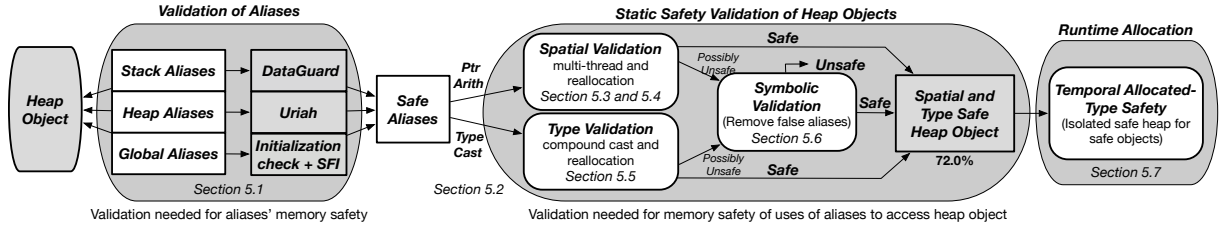

Figure 1: Overview of the Uriah approach.

and (2) prevent exploits of temporal errors at runtime from permitting accesses that violate the allocated-type of validated safe heap objects in (1), through enforcing *temporal allocated-type safety* (Definition 2) and isolation. The Uriah allocator ensures that only safe objects with identical memory layout of sizes and types are (re)allocated in the same memory locations. Memory (re)uses are restricted to the same allocated-type for validated safe heap objects, preventing temporal exploits that corrupt their allocated-type[2].

Definition 1. The **allocated-type** of a heap object (consists of n fields) is a tuple $(S, T, \{(f_i, s_i, \tau_i, o_i)\}_{i=1}^{n})$ where:

- S is the declared total size of the object in memory.
- T represents the declared type of the object.
- $\{(f_i, s_i, \tau_i, o_i)\}_{i=1}^{n}$ is the set of quadruples for each field f_i of the object, where s_i is the size of field f_i, τ_i is the type of field f_i, and o_i is the offset of f_i in the object.
- No two fields overlap, i.e., for any pair of distinct fields f_i and f_j with offsets o_i and o_j, sizes s_i and s_j, respectively, the intervals $[o_i, o_i + s_i)$ and $[o_j, o_j + s_j)$ do not intersect.

Definition 2. Temporal allocated-type safety is a property that requires a memory region be used to access objects of one allocated-type. Memory reuses are restricted to objects of one declared size and type for all fields, without partial overlapping.

Figure 1 shows an overview of the Uriah approach. This approach must: (1) validate that ***accesses to all the aliases of heap objects*** must satisfy memory safety, (2) validate that ***all accesses to a heap object*** through all of its aliases must satisfy spatial and type safety, and (3) enforce temporal allocated-type safety on heap objects found to satisfy (1) and (2) to prevent exploitation of temporal errors by preserving spatial and type safety in all allocations. First, Uriah collects all possible aliases of heap objects conservatively. Aliases may reside in the stack, heap, or global regions, so Uriah validates heap aliases and leverages the DataGuard analysis [40] for stack aliases and adapts that analysis for global aliases conservatively (Section 5.1). Second, Uriah validates that all uses of aliases to access heap objects must satisfy spatial and type safety. Uriah validates spatial safety considering the impact of reallocations (Section 5.2) and multi-threading (Section 5.3), and type safety considering the impact of compound type casts and reallocations (Section 5.4). Uriah uses symbolic execution to remove false aliases and infeasible unsafe paths to validate false positives due to the over-approximations of static analysis (Section 5.5). Third, heap objects validated to satisfy spatial and type safety are allocated on a single isolated safe heap that enforces temporal allocated-type safety (Section 5.6). Unsafe heap objects are isolated on an unsafe heap, and existing protection can be applied to the unsafe heap.

[2]Similar to temporal type safety [129], temporal allocated-type safety does not prevent exploits that reuse memory of different objects of the same allocated-type.

The Uriah approach for protecting heap objects draws inspiration from the *Anna Karenina Principle* [52][3]. Uriah aims to maximize the number of safe target objects (i.e., "happy family") that may be co-located in the isolated *safe heap* (i.e., creating the largest "happy family") by (1) validating spatial and type safety statically, and (2) enforcing temporal allocated-type safety at runtime to prevent illicit accesses caused by temporal errors. Despite the complexity in structure (e.g., compound types) and usage (e.g., reallocation and multi-threading) of heap objects, we find that many allocation sites only produce heap objects whose accesses can be proven to satisfy spatial and type safety. No runtime checks are necessary on safe objects, as Uriah's heap allocator ensures that accesses to safe objects satisfy temporal allocated-type safety. Uriah isolates these safe objects from illicit accesses from the unsafe heap via SFI.

4 Threat Model

We assume that every program protected by Uriah may have any classes of memory errors on the heap. We also assume that every heap object that Uriah has deemed unsafe may have memory error. We assume adversaries will try to attack any unsafe heap object to corrupt safe heap objects. We aim to protect the safe heap objects from being affected by such attacks by the construction of Uriah. Protection of unsafe heap objects can be realized by applying existing defenses on Uriah's unsafe heap objects only.

To ensure that the CFG we analyze is an over-approximation of program executions to guarantee the soundness of Uriah's static analyses, we assume the presence of CFI [1], which implies that the corresponding indirect call analysis is sound , so that attackers cannot leverage the remaining unmitigated exploits by Uriah to synthesize malicious control flows beyond the expected CFG. We further assume the code memory is not writable, and the data memory is not executable [70]. To ensure that Uriah operates correctly, we assume the computed safety constraints are protected from tampering, and the underlying allocator (i.e., TcMalloc) is free from flaws and maintains the integrity of its state and metadata.

5 Design

In this section, we detail Uriah's design to perform the tasks outlined Figure 1. Soundness[4] (ensuring that no unsafe objects are misclassified as safe) is discussed individually in each section.

5.1 Collecting and Validating Aliases

All aliases that may alias each heap object must be identified to validate that all possible accesses to each heap object satisfy spatial

[3]From the opening line in Tolstoy's *Anna Karenina* [127]: "All happy families are alike; each unhappy family is unhappy in its own way."

[4]As defined by the static analysis community [112], sound analysis over-approximates the program's executions. We also summarize soundness in the extended paper [42].

and type safety and to validate the memory safety of accesses to the aliases themselves. Assume program executions are restricted to the computed CFG via CFI (Section 4), Uriah computes an overapproximation of all aliases from the stack, heap, and global regions for each heap object inter-procedurally in a context and flow-sensitive way, using static value-flow analysis [119] (SVF) on a program-dependence graph [71] (PDG) representation. SVF and PDG are both claimed to be sound[5]. SVF [119–123] uses *heap cloning* [5, 57, 133] for flow-sensitive aliases analysis [10] at intra-procedural level, the result is fed into PDG [71], which uses a parameter-tree model to compute context-sensitive aliases inter-procedurally.

A challenge is to verify the safety of all aliases to each heap object, regardless of whether they are stored in the stack, heap, and global region. The memory safety of accesses to stack aliases (67.29% of all aliases) is validated using DataGuard [40]. Heap aliases (31.88% of all aliases) will be validated by Uriah as heap objects as described in the remaining subsections. Global aliases make up only 0.83% of the aliases in all the tested benchmarks[6], so we provide a conservative approach. Global aliases are either: (a) singletons, (b) fields of global objects that only have singleton fields, or (c) fields of global objects that have fields of compound types. With Uriah, we validate global aliases of classes (a) and (b), which constitute over 87.3% of all global aliases, as follows. First, no spatial errors are possible for accesses to singleton objects or objects with only singleton fields (i.e., no pointer arithmetic except field access). Second, Uriah detects any type casts of these global aliases, as described in Section 5.4. Third, Uriah requires global aliases to be initialized immediately following their declaration. Uninitialized global aliases are deemed unsafe to avoid temporal errors. Since protecting global memory is not a contribution of this work, we assume SFI techniques may be applied to protect safe global aliases from being corrupted by memory errors on accesses to unsafe global memory similar to how we isolate the safe heap, but implementing that is future work.

5.2 Spatial Safety Validation

A heap object is validated to be *spatially safe* if all of its aliases must only access memory locations within the object's memory region, defined by its size. Uriah requires that all heap objects must be declared to have or always be bounded by a constant size, which forms the *spatial constraint* checked in spatial safety validation. If an object's size is not a constant, the object is classified as unsafe.

CCured [86] finds that only aliases used in pointer arithmetic operations may violate spatial safety, Uriah applies sound context-sensitive [107] value-range analysis [40, 109, 125] to validate the spatial safety of each heap object using Algorithm 1 for all pointer arithmetic operations on each of its aliases. Each alias is checked to comply with the collected constraints (i.e., *size*) for all accesses with an *index* (i.e., a position of the object between 0 and *size*) along the def-use chain (line 3). When referencing an object through an alias, the reference may use *offset* (i.e., in pointer arithmetic) to change the index. The initial index must be less than the size of heap object (line 4). The offset is required to be constant and the index+offset

Algorithm 1: Spatial Safety Validation for Heap Objects

```
   Input: object - the heap object to be validated
   Output: classification of the object (safe or unsafe)
1  function SpatialValidation(object)
2      for each alias of object do
3          alias.index = def(alias)
4          if (alias.index ≥ object.size then
5              object.safety(unsafe) return
6          if SingleThread(alias) then
7              for each use of alias to access object do
8                  if IsConstant(offset) then
9                      if use.offset < 0 or !(0 ≤ (alias.index +
                           use.offset) < object.size then
10                         object.safety(unsafe) return
11                     if Increment_Index(use)) then
12                         alias.index + = use.offset
13         else if MultiThread(alias) then
14             MultiThreadValidation(object, alias)
15     object.safety(safe) return
```

must always be between 0 and the size of the object for every use (line 5-9). Uriah classifies an object as unsafe if any access uses a negative offset to prevent underflows, as the PDG overapproximates possible executions. The index may be incremented by the offset in some cases (line 10-11) and needs to be updated in such cases. The validation for multithreading (line 13) is discussed in Section 5.3.

The challenge is that spatial safety validation must account for reallocations since heap objects can be *dynamically-resized* through reallocations. Previous approaches [4, 26, 28, 69, 81, 83] treat reallocations as new allocations, ignoring the impact of unsafe reallocations on accesses using the original aliases. Reallocation can compromise accesses through the original aliases by: (1) shrinking the object's size, as accesses using the original aliases to the reallocated object may exceed the new/reallocated bounds, and (2) moving the object to a different location, which can turn the original aliases into dangling pointers if not managed correctly. For case (1), Uriah classifies the objects whose sizes are reduced through reallocations (which is rare) as unsafe. Uriah limits the cases objects can be reallocated safely only to those that extend the size of the last field or append fields to the object at the end of its original data type (which are common cases for reallocation). Aliases to the reallocated object will be analyzed (i.e., for their def-use chains) using the newly reallocated size as the *size* in Algorithm 1. For case (2), accesses using the original (possibly dangling) aliases will be evaluated using the original size for their def-use chains to detect an spatial errors relative to the original object memory rather. The temporal safety concerns for case (2) are addressed in Section 5.6.

5.3 Spatial Validation with Concurrency

As heap objects may be *shared* among threads (i.e., can be accessed concurrently in multiple threads), spatial safety validation of Uriah must account for concurrent accesses to heap objects through aliases while indexes may be modified in different threads[7]. The problem is to identify the heap objects that may be accessed concurrently and then reason about the safety of such concurrent accesses

[5]Computing global aliases may suffer from the well-known scalability and precision issues for large programs, e.g., Linux Kernel [61]. We leave the adoption of the proposed technique for future work.

[6]Using global aliases to reference heap objects raises both security and performance concerns [27, 28, 61], and is thus explicitly inadvisable in much production software (e.g., a stated in reference to the Linux kernel [9], Chrome [35], and Nginx [89]).

[7]Race conditions, as not memory errors, are excluded in this paper.

Algorithm 2: Spatial Safety Validation for Shared Objects

```
   Input: object - the heap object used in concurrent context
   Input: alias - the alias to the heap object
   Output: classification of the object (safe or unsafe)
1  function MultiThreadValidation(object, alias)
2      if IsShared(object) then
3          if !IsConstant(Threads_Set) then
4              object.safety(unsafe) return
5          for each use of alias used in multiple threads do
6              for each thread in Threads_Set do
7                  if use.offset < 0 or !(0 ≤ (alias.index + use.offset) <
                     object.size then
8                      object.safety(unsafe) return
9                  if Increment_Index(use)) then
10                     alias.index += use.offset
11     classify(object, SAFE) return
```

Algorithm 3: Type Safety Validation for Heap Objects and Aliases

```
   Input: object - the memory object to be validated
   Output: classification of the object (SAFE or UNSAFE)
1  function TypeValidation(object)
2      for each type cast cast(TN, T) for each alias of object do
3          if IsIntegerCast(cast) then
4              DataGuardTypeValidation(TN, T)
5          else
6              if T! = TN then
7                  layout_TN ← GetTypeLayout(TN)
8                  layout_T ← GetTypeLayout(T)
9                  prefix_TN ← GetPrefix(layout_TN, sizeof(T))
10                 for each field f_T in layout_T do
11                     f_TN ← GetField(f_T, prefix_TN)
12                     if offset(f_T) ≠ offset(f_TN) or !f_TN then
13                         object.safety(unsafe) return
14     object.safety(safe) return
```

conservatively. While SVF provides intra-thread CFGs [118] that identify the aliases accessed in functions that may be executed concurrently in multiple threads, such a method introduces a large number of false positives, as many heap objects accessed in the intra-thread CFG may not be used concurrently (i.e., are thread-local). The challenge is to remove false positives while still maintaining an overapproximated set of objects that may be accessed concurrently.

Uriah identifies the shared heap objects from the heap objects accessed in an intra-thread CFG by determining whether they are assigned to any alias that may be used concurrently in multiple threads. Such aliases should be used in intra-thread CFG and either: (1) are on the heap, (2) are in global memory, or (3) are on the stack and passed as parameters of thread creation API. Other aliases used in the intra-thread CFG are thread-local. Since the construction of inter-procedural, intra-thread CFGs and SVF aliasing are claimed sound [56, 118], Uriah over-approximates the heap objects that may be shared and the aliases that may access them concurrently.

To reason about spatial safety conservatively, updates to the aliases of shared heap objects must consider whether the aliases may be used concurrently in multiple threads or not. Algorithm 2 shows the spatial validation of shared heap objects among threads. Accessing shared heap objects consists of two cases, depending on whether the alias is: (1) thread-local or (2) used in multiple threads. We observe that case (1) is the common method for accessing shared heap objects in production software. For example, Nginx and httpd create thread-local copies of aliases to access shared heap data. For case (1), since the alias is thread-local, aliases in different threads are used independently and can be examined using Algorithm 1 discussed in Section 5.2 (thus excluded in Algorithm 2). For case (2), where the number of threads can be concretized statically , we extend the value-range analysis to calculate the index by accumulating the access range (line 5-10) across memory operations in all threads. For example, if 2 threads access the heap object through incrementing the alias by 3 and 5, then the index calculated by Uriah will be increased by 8. Accumulation only applies to accesses that increment index of the alias using constant offsets. Other accesses (e.g., access to fields without changing the index of the alias) do not need to be accumulated. Scenarios where the index is incremented while the number of threads cannot be determined statically are unsafe (line 3-4). Thread-local objects are validated in Section 5.2. If the calculated index is greater than the size of the heap object, then the object is unsafe (line 7-8). For reallocations on shared heap objects, Uriah relies on the common convention that the reallocation operation is atomic[8] (i.e., no race between reallocation and memory access to the same heap object). Spatial constraints are updated for all threads upon reallocation. Shared heap objects are unsafe if reallocated to variable size in concurrent contexts.

5.4 Type Safety Validation

As defined in Section 2.1, A heap object is considered *type safe* only if all of its aliases only access objects of the same data types at each offset per the definition of its allocated-type, see Definition 1. The task is to identify the types used by aliases to access the object and verify if type casts among these types ensure type safety of all accesses. CCured [86] finds that only aliases used in type cast operations may violate type safety, and identified that *upcast* must be safe [87]. They define an *upcast* is a type cast from type TN to type T when the layout of T in memory is a prefix of the layout of TN. At the time of the CCured work (around 2005), they claimed that 63% of type casts are between identical types, and of the remaining casts, 93% are safe upcasts. Thus, a significant benefit may be seen by validating safe type casts statically. However, SAFECode conservatively classifies all objects in type casts as unsafe, CCured only applied its approach to casts where the type T is an explicit subtype in TN, ignoring cases where the subtype T does not physically exist, but instead of its all its fields are present as a prefix of type TN, as such cases are prefixes using allocated-types. Moreover, in recent systems, only C++ upcasts are classified [28, 38, 51, 59, 145] by leveraging class hierarchy and RTTI. Thus, redundant runtime checks are still applied to casts that can be validated as type safe.

We find that any upcast where the type resulting type is an exact prefix of the allocated-type satisfies temporal type safety [129] for

[8] POSIX realloc() employs mutexes internally to avoid data corruption in races on metadata [97]. Intel suggests adoption of thread-safe memory management API [45], e.g., Intel oneTBB [46]. Allocators [32, 50, 64] are designed to leverage thread-local storage/cache and per-thread freelists. Applications such as Nginx and Apache Httpd utilize atomic reallocation operations and tend to only use realloc on thread-local data.

allocated-types, providing more opportunities for type-safe casts. To identify safe type casts, Uriah performs type safety validation following Algorithm 3 for each type cast operations for all aliases of the heap object[9]. DataGuard [40] only validates safe type casts among integers. Uriah reuses DataGuard's type safety validation for integer casts while focusing on casts among compound types (line 2-4). Uriah validates compound type casts to determine if two data types are *compatible*, which we call the *compatible-type-cast analysis* (line 6-14). Two types in a cast are compatible types if the types are identical or the cast is an upcast, other type casts are considered unsafe. Different from C++ where T usually serves as a field of TN due to inheritance and coercion, we observed that in many cases, T is not contained in TN for C programs, but instead all of its fields are. The compatible-type-cast analysis does not require T to be contained in TN, as long as all the fields of the T are at the same offsets and of the same sizes and types in TN, cast from TN to T ensures that an access at any offset within T references the same type (line 10-13). Also, it is worth mentioning that type safety validation on shared heap objects among threads is validated similarly since the safety of type cast is regardless of whether it can be executed concurrently. The compatible-type-cast analysis is built on the sound definition of upcasts in CCured [20, 87]. Our validation is conservative, requiring the types to match concretely and exactly to retain soundness.

Type safety validation is complicated by several challenges: (1) heap objects can be dynamically typed (e.g., through polymorphism in C++); (2) interchange of `void*`, `char*`, and other types (e.g., `memcpy` takes `void*` as parameter for arbitrary types); (3) heap memory is not always assigned concrete types upon allocation (e.g., the `auto` keyword may result in the pointer being assigned `void*`); (4) reallocations change original allocated-type (e.g., by extending the length of a field), and (5) memory safety concerns of using unions. These challenges may make it harder to determine the correct memory layout of an object. Uriah resolves challenges in determining an object's type (i.e., cases (1-3) by delaying type assignment. Uriah does not consider `void*` or `char*` as a concrete type. For heap objects whose allocated-type cannot be concretized or aliased by `void*` or `char*` pointers, Uriah delays assignment of a concrete type until the object is interpreted as a specific type. If the type cannot be concretized statically, such objects are classified as unsafe. For (4), Uriah only allows reallocations that extend the size of the last field or append fields to the object at the end of its original data type (see Section 5.2). For complied cases, a new allocated-type is assigned to the heap object using the new declared type. For (5), LLVM represents unions as structures. Union members are declared as distinct SSA variables [76] through type casts from such structures to the member. Uriah validates such casts using type safety validation before validating all following operations.

5.5 Symbolic Validation

Because Uriah's static analysis over-approximates the possible executions, it may classify a heap object as unsafe that could really be safe, producing false positives. Researchers have identified two approaches to remove false positives found from static analysis using symbolic execution: (1) executing all paths to verify legal executions [40], and (2) pruning infeasible paths until only compliant paths remain [143, 144]. Using (1) for the heap, we find that many false positives are generated by infeasible aliasing (i.e., pointers cannot be defined to reference the object) and operations (i.e., pointers uses cannot access the object), due to the over-approximation of static analysis, indicating the executions that result in the objects to be classified as unsafe is not feasible (i.e., infeasible path).

Instead, on top of (1), Uriah applies symbolic execution in (2) to prune infeasible paths. Specifically, Uriah removes infeasible paths until all unsafe aliases can be removed as false positives. False positives may occur because: (1) an object cannot be assigned (i.e., defined) to an alias on a path with an unsafe operation (i.e., infeasible definition); (2) an object cannot be used by an alias in an unsafe operation (i.e., infeasible use); and (3) the path cannot be executed in a manner that causes the unsafe operation (i.e., infeasible path). First, to prune infeasible definitions, Uriah symbolically executes each unsafe alias from its declaration to definitions that involve later unsafe uses to determine whether the definition is reachable in this path. Second, to prune infeasible uses, Uriah symbolically executes from the pointer definition to its unsafe uses following its def-use chain. In both of these cases, infeasibility is detected by the failure of the symbolic execution engine (S2E [19]) to generate path constraints. Finally, the infeasible paths are detected by evaluating the path constraints on the def-use chains. All possibly unsafe paths are considered before an object is reclassified as safe, preventing any unsafe object from being classified as safe. If after removing infeasible aliases and operations, all the remaining aliases of a heap object can be validated to be safe for spatial and type safety for the remaining operations, the heap object is reclassified as safe.

Symbolic execution suffers from scalability issues (e.g., path explosion). Uriah applies loop canonicalization [74], simplification, and unrolling [44, 75] features, and state merging [22]. Uriah limits functions executed symbolically via a configurable depth to avoid path explosion and classify heap objects in such cases as unsafe.

Uriah*'s Heap Allocator Operations*	
Allocation	Allocate object X of allocated-type T to region labeled for T. X must be validated to satisfy spatial and type safety.
Deallocation	Dealloc object X of allocated-type T, returning the memory region to the free-list of allocated-type T to restrict its reuse to T.
Reallocation	Reallocates the object of original allocated-type T to new allocated-type T'. The object is deallocated in the memory for T and allocated in the memory for T'.

Table 2: Operational semantics for the Uriah safe heap allocator

5.6 Uriah Runtime Allocation

Uriah enforces *temporal allocated-type safety* for heap objects validated to satisfy spatial and type safety in an isolated, safe heap. Uriah's temporal allocated-type safety prevents temporal errors prior to initialization and after deallocation by initializing all pointers on allocation and enforcing type-safe reuse on reallocation, respectively. Recall from Section 3 that temporal allocated-type safety requires that objects of only one allocated-type may be allocated in each memory region to prevent a dangling pointer from being used to access data of a different allocated-type. However, there are several issues with the current approaches that prevent

[9]We consider all C and C++ type casting operations. C++ offers 5 types of casts: *static_cast*, *dynamic_cast*, *reinterpret_cast*, *const_cast*, and *C-style_cast*. Among them, *dynamic_cast* and *const_cast* have no security concerns [38, 51, 59], *C-style_cast* is translated to other casts. We concentrate on validating *static_cast* and *reinterpret_cast*.

	Total	*VR-Spatial*	*Uriah-Spatial*	*CCured-Type*	*CTCA-Type*	*Uriah-Type*	*VR-Spatial+ CCured-Type*	*Uriah-Spatial+ Uriah-Type*
Firefox	26,162	19,857 (75.9%)	20,432 (78.1%)	14,101 (53.9%)	19,700 (75.3%)	20,040 (76.6%)	12,270 (46.9%)	18,392 (70.3%)
nginx	954	705 (73.9%)	785 (82.3%)	585 (61.3%)	766 (82.3%)	819 (85.5%)	521 (54.6%)	744 (78.0%)
httpd	1,074	662 (61.6%)	816 (76.0%)	825 (76.8%)	918 (85.5%)	942 (87.7%)	575 (53.5%)	760 (70.8%)
proftpd	1,707	1,275 (74.7%)	1,380 (80.8%)	596 (34.9%)	1,201 (70.4%)	1,366 (80.0%)	458 (26.8%)	1,174 (68.8%)
sshd	378	270 (71.4%)	310 (82.0%)	170 (45.0%)	284 (75.1%)	304 (80.4%)	144 (38.1%)	274 (72.5%)
sqlite3	761	614 (80.7%)	655 (85.7%)	382 (50.2%)	567 (74.5%)	587 (77.1%)	316 (41.5%)	513 (67.4%)
SPEC2006	–	71.1%	79.6%	37.7%	80.3%	85.0%	29.8%	72.2%
SPEC2017	–	68.9%	83.4%	41.8%	79.0%	83.9%	28.2%	75.6%
AVERAGE	–	**72.2%**	**81.0%**	**50.2%**	**77.8%**	**82.0%**	**39.9%**	**72.0%**

Table 3: Incremental Safety Improvement of Uriah, and Comparison with CCured. *Total* column shows the total number of heap objects (i.e., allocation sites). We omitted *CCured-Spatial* column since heap objects are always involved in pointer arithmetic, resulting in the *CCured-Spatial* to be around 0 for all benchmarks. *CCured-type* column shows the number of heap objects are not aliased by any pointer used in type casts. *VR-Spatial* column represents the number of heap objects passed value-range analysis, *Uriah-Spatial* column represents the number of heap objects passed the complete Uriah's static spatial safety validation. *CTCA-Type* column represents the number of heap objects passed Compatible-type-cast analysis, *Uriah-Type* column represents the number of heap objects passed the complete Uriah's static type safety validation. The *VR-Spatial+CCured-Type* column shows the number of heap objects passed value-range analysis and CCured-type analysis. *Uriah-Spatial+Uriah-Type* column shows the number of safe heap objects passed Uriah's complete static safety validation. *SPEC 2006* and *SPEC 2017* rows show the average percentage, individual results of SPEC benchmarks are shown in Table 3. *Average* row shows the average number of all tested benchmarks.

them from enforcing temporal allocated-type safety and preserving spatial and type safety. We first outline Uriah's allocator operations and then examine how it resolves issues in prior approaches to enforce spatial, type, and temporal allocated-type safety.

5.6.1 *The Safe Heap.*

Uriah implements *temporal allocated-type safety* over objects in the safe heap. Table 2 describes the API of Uriah's allocator for the safe heap. First, Uriah supports *allocation*, which allocates objects using type-specific freelists. All memory regions begin untyped and are assigned an allocated-type on the first allocation to that memory region. Second, Uriah supports *deallocation*, which places the memory region on a per-type free list to preserve the *allocated-type* of the memory region for future allocations. Uriah maintains the allocated-type with each allocation's metadata. Third, Uriah supports *reallocation*, which may change the size of the object allocated (i.e., changing the allocated-type). Reallocations that result in a new allocated-type of the heap object will result in the heap object being moved out from the original pool and allocated in the pool corresponding to the new allocated-type. Original aliases, though may be dangling pointers without proper handling, can only reference objects of the original allocated-type.

5.6.2 *Isolation from the Unsafe Heap.*

The unsafe heap region is built on top of the tcmalloc's span and page heap memory management scheme. For 64-bit system, currently only 48 bits are used for addressing, with 1 bit to distinguish kernel and user space, up to 128TB memory can be used for user space. Uriah reserves 1TB for the unsafe heap region. Uriah forces all access to unsafe objects to only access memory in the unsafe heap by performing bit-masking upon memory operations. Thus, even if a pointer is illicitly modified to an address outside of the unsafe heap, Uriah will restrict the pointed address to within unsafe heap through bit-masking. Thus, any memory errors on operations to objects located in the unsafe heap cannot be exploited for crafting pointers that reference memory outside the unsafe heap region.

5.6.3 *Security Implications of the Safe Heap.*

Spatial Safety: SAFECode [25] and Type-After-Type [129] do not enforce spatial safety within the heap. SAFECode uses per-type heaps, so spatial errors are limited to objects of same data type, but they recommend additional runtime checks. Other systems [81] allocate objects separated by guard pages, but spatial errors can evade guard pages. Uriah's static safety validation ensures that any object added to the safe heap must satisfy spatial safety.

Type Safety: SAFECode [25] enforces type safety by only pooling objects of the same data type into each per-type heap. However, SAFECode's method for reusing per-type heaps leads to significant overheads [47]. Uriah's static safety validation ensures that any object added to the safe heap must satisfy type safety. Objects of multiple types can be added to the same safe heap (i.e., in different locations) since they are guaranteed to satisfy spatial safety.

Use-Before-Initialization (UBI): UBI is possible on the stack, heap, or global aliases of heap objects. For stack pointers, Uriah leverages DataGuard [40] to ensure that safe stack aliases are never used before initialization. For heap pointers, current techniques to enforce temporal type safety [25, 69, 129] do not explicitly prevent UBI attacks on heap pointers, techniques to zero memory on initialization [80] is expensive. However, Uriah's heap allocator is built on top of TcMalloc, which already zeroes memory upon request from the OS [34, 37]. Uriah detects uninitialized global aliases and classifies the aliased heap object as unsafe (Section 5.1).

Dangling Pointers: Another memory safety problem occurs due to the reuse of dangling pointers after reallocation/deallocation. Dangling pointers to the reallocated/deallocated safe heap objects are restricted to only reference memory of the object's allocated-type. Some allocation sites may allocate objects whose type cannot be determined statically, so some systems use allocation sites instead of types to determine the memory regions that may be allocated for objects [81, 129]. This has been found to create cases where memory regions may be reused for objects of multiple types and sizes, invalidating the temporal allocated-type safety. Because Uriah preserves allocated-type for memory reuse, it does not allocate any such objects on the safe heap, avoiding this problem.

6 Implementation

Uriah has been deployed on the x86_64 architecture, running on an Intel CPU i9-9900K with 128 GB RAM, using LLVM 10.0 on Ubuntu 20.04. The CCured framework is adapted from NesCheck [79]. We expand the original value-range analysis through the call-string approach for achieving scalable context-sensitivity [107] and covering dynamically-sized objects. For type safety validation, we eliminate the type casts generated by the compiler immediately after memory allocation, since it is the common way of allocating heap objects

	Runtime Overhead (%)									Memory Overhead (%)								
	TAT	SC	AS	DS	ES	FF	MU	URIAH-R	URIAH	TAT	SC	AS	DS	ES	FF	MU	URIAH-R	URIAH
perlbench	8.3	2800.8	174.2	251.2	824.1	8.5	22.1	8.5	**5.1**	35.7	83.5	262.4	371.4	**21.8**	124.8	28.1	39.2	24.5
bzip2	4.4	11.1	74.4	5.1	122.5	2.7	**1.7**	4.7	2.4	5.5	12.3	42.4	6.2	11.5	17.4	7.2	6.4	**3.9**
mcf	2.6	42.6	12.6	48.3	67.1	**1.2**	2.2	2.4	1.5	2.3	11.1	14.5	55.3	8.2	5.5	6.8	2.4	**1.8**
gobmk	5.6	-	56.3	11.3	225.8	7.5	17.2	5.9	**3.2**	8.2	-	1130.5	124.5	17.8	78.5	91.6	8.2	**5.2**
hmmer	2.4	7.6	42.7	2.8	324.8	6.4	27.2	2.4	**2.1**	51.5	24.5	6580.4	**12.4**	44.5	85.2	82.8	57.5	34.4
sjeng	3.6	457.2	67.5	2.7	78.2	4.1	4.2	4.0	**2.2**	3.6	8.5	18.7	**2.7**	12.2	21.6	17.4	3.8	3.1
libquantum	4.6	202.4	27.1	3.1	291.2	3.8	3.0	5.1	**2.9**	4.2	7.6	289.1	4.1	6.7	17.4	12.2	4.5	**3.5**
h264ref	6.2	423.3	23.2	2.3	655.8	**1.7**	4.8	6.0	3.2	12.6	42.7	72.4	8.5	8.8	39.5	78.4	14.1	**8.5**
lbm	3.6	10.5	34.4	3.7	43.7	2.2	2.8	3.9	**2.0**	4.6	6.8	27.2	5.7	15.4	8.9	9.8	4.9	**3.2**
sphinx3	6.8	23.1	17.5	5.8	235.1	7.6	5.5	7.0	**3.7**	38.7	94.7	1150.8	182.8	**9.1**	1182.5	34.2	41.5	24.4
milc	6.2	12.8	14.6	21.1	142.0	4.4	6.4	6.5	**3.2**	3.4	10.5	337.1	34.1	14.2	12.2	15.6	3.7	**3.2**
omnetpp	6.6	-	76.5	698.4	167.4	5.1	36.5	6.8	**4.5**	13.2	-	425.3	1125.6	12.6	436.1	73.1	15.1	**11.5**
soplex	2.2	2135.4	94.7	12.2	212.9	7.2	5.6	2.6	**2.1**	5.6	17.8	355.2	1450.0	24.1	52.5	59.3	5.8	**4.3**
namd	3.4	2020.4	27.3	3.8	66.5	8.7	**2.3**	3.9	2.5	4.4	9.8	54.2	8.5	7.5	24.4	17.6	4.8	**3.6**
astar	3.5	312.6	62.6	72.3	246.1	3.3	**2.5**	3.6	3.1	4.8	35.5	468.1	515.6	11.3	77.1	64.4	5.2	**3.7**
AVERAGE	4.7	650.8	53.7	76.2	246.9	4.9	9.6	4.9	**2.9**	13.2	28.1	748.6	260.5	15.0	145.6	39.9	14.5	**9.3**

Table 4: Runtime and Memory Overhead of URIAH Compared with Prior Works for SPEC CPU 2006. We use the following abbreviations to represent the prior works, TAT-Type-After-Type, SC-SAFECode, AS-ASan, DS-DangSan, ES-EffectiveSan, FF-FFMalloc, MU-MarkUs. The lowest overhead on a benchmark is marked as bold.

(i.e., casting to the corresponding type from void*) and it is safe. We utilize S2E [19] as the guided symbolic execution engine for removing false aliases. To reduce the path explosion of the symbolic execution, URIAH employs a depth limit, where any terminated symbolic execution implies that the related heap object is unsafe.

For runtime allocation to enforce temporal allocated-type safety, URIAH creates per-allocated-type pools on the safe heap and isolates all unsafe objects found by static validation in the unsafe heap. This is achieved by adding an additional parameter that contains the hash of the allocated-type to the allocation API in TcMalloc only for safe heap objects, while all unsafe heap objects share the same unique hash. The pools are built by leveraging the spans of TcMalloc, the unsafe heap owns a separate, isolated span. The spans and metadata are originally isolated in TcMalloc through guard pages. Once acquired, the memory will never be returned to the OS for later reuse by another span (pool), but is only reused through the pool's freelist. The metadata and freelist are also isolated.

7 Evaluation

This section focuses on evaluating how URIAH improves the security for low overhead. We conduct this evaluation by analyzing browsers, servers, SPEC CPU2006, and SPEC CPU2017 benchmarks.

7.1 Identifying Safe Heap Objects

RQ1: *How many heap objects does* URIAH *identify that can be allocated on the safe heap?* Table 3 shows the counts and percentages of safe heap objects using existing techniques (i.e., *VR-Spatial+CCured-Type*) and URIAH (URIAH-*Spatial*+URIAH-*Type*). URIAH classifies 72.0% of heap objects as statically safe in Firefox, server programs and SPEC CPU2006/2017 benchmarks on average[10].

For spatial safety, CCured classified virtually no safe heap objects, which is expected as heap objects are usually compound objects and use pointer arithmetic for field accesses. Traditional value-range analysis (*VR-Spatial*) classifies 72.2% of heap objects as satisfying spatial safety, but may misclassify unsafe heap objects as safe due to its inability to handle reallocations and concurrency. URIAH's

[10]We examined 15 out of 19 C/C++ benchmarks in SPEC CPU2006. The remaining benchmarks (gcc, xalancbmk, povray, and dealII) are not supported by the SVF analysis due to the version is too old. The newer version in SPEC CPU2017 (gcc_s and xalancbmk_s) are supported. We analyzed all 12 SPEC CPU2017 Benchmarks.

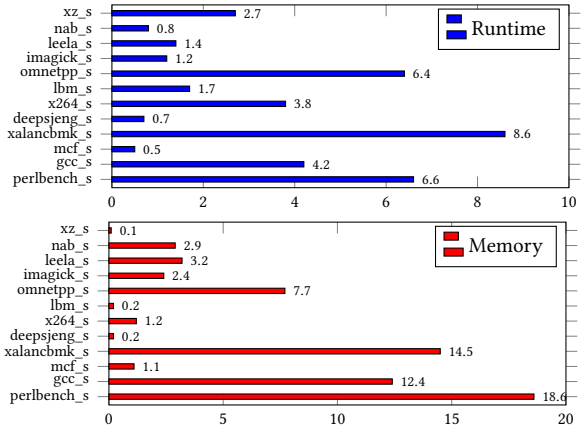

Figure 2: Runtime and Memory Overhead of URIAH on SPEC CPU2017

extensions that handle dynamic sizing and multi-threading (Section 5.2 and 5.3) and remove spurious aliasing (Section 5.5) classify around 81.0% of heap objects as safe. For type safety, CCured classifies around 50.2% of the heap objects as safe. Enhanced by the compatible-type-cast analysis (Section 5.4) and symbolic execution (Section 5.5), URIAH classifies 82.0% of the heap objects as safe, an additional 31.8% of heap objects being classified as safe. The stats for individual benchmarks can be found in the extended paper [42].

The results in Table 3 show static heap object counts, raising the question: how does URIAH perform with runtime allocations? We use heaptrack, perf, and Mtuner as the heap memory profiler to measure such data on SPEC 2006 benchmarks. If URIAH classifies a heap object as safe statically, all the corresponding runtime allocations are classified as safe as well. Based on the result, 73.6% of total runtime heap allocations are classified as safe by URIAH.

7.2 Performance Evaluation

RQ2: *What is the performance impact of isolating enforcing temporal allocated-type safety?* In this section, we evaluate the performance of URIAH on the SPEC CPU2006 and SPEC CPU2017 benchmarks. We compare URIAH with seven prior heap defenses on SPEC CPU2006 and provide result on SPEC CPU2017 (many compared works were not originally evaluated on SPEC CPU2017). All prior works are built from their open-sourced implementations without any updates. We note that defenses vary in their goals (see Table 1). All

	Foundation	Static	SymExec	Total
Firefox	17,436.4s	2,349.2s	10,386.8s	30,172.4s
nginx	2,259.8s	132.6s	323.7s	2,716.1s
httpd	249.5s	1.6s	64.2s	315.3s
proftpd	1,328.4s	212.7s	564.4s	2,105.5s
sshd	225.6s	251.0s	191.4s	668.0s
sqlite3	3,396.4s	472.8s	1,624.4s	5493.6s

Table 5: Time Elapsed in Each Phase of Uriah's Static Safety Validation. The Foundation column represents the adapted CCured analysis time. The Static column represents the time spent for spatial and type safety validation. SymExec shows the time elapsed in the symbolic execution phase. *Total* column shows the total static safety validation time - sum of the first 3 columns.

defenses except Uriah provide coverage of all objects for the memory error classes they address, but may not cover all memory error classes, whereas Uriah protects 73.6% of heap objects (72.2% of allocation sites) that satisfy spatial and type safety to enforce temporal allocated-type safety for the SPEC CPU2006 benchmarks. The Uriah-R column uses Uriah to allocate all heap objects to emulate the complete coverage of all objects for comparison.

7.2.1 Performance Overhead. Table 4 on the left side shows that Uriah has the lowest average runtime overhead on SPEC CPU2006 benchmarks of 2.9%. ASan, DangSan, and EffectiveSan enforce memory safety using runtime checks, including maintaining the metadata for use in such checks, resulting in high runtime overhead. Uriah does not require runtime checks, so it elides their overheads. HexType [51] is not listed as it only supports C++ programs.

Uriah also outperforms allocator-based defenses. Specifically, SAFECode exhibits much worse performance overhead than others (more than 20x on *perlbench* and *named*), due to SAFECode's method to reuse memory by allocating and deallocating per-type pools for each function (or call chains through escape analysis). MarkUs and FFmalloc are much more efficient since they track used memory and prohibit memory reuse, although they are still more expensive than Uriah in most cases. Type-After-Type improves performance by introducing a per-type pool allocation scheme by using an efficient memory allocator, *TcMalloc*. Unfortunately, it is impractical to compare Type-After-Type and Uriah on the same set of objects without breaking either system. For a fair comparison, we measured Uriah's overhead when it applies to all heap allocations, the results show that Uriah-R(untime) performs slightly worse, which is expected given the more precise identification of type by Uriah. Moreover, Uriah provides guarantees for spatial and type safety, and enforces the stronger temporal allocated-type safety property against temporal attacks. We also evaluated Uriah on SPEC CPU2017. See Figure 2, Uriah exhibits a 2.6% overhead on SPEC CPU2017 on average, which is reasonably low.

7.2.2 Memory Consumption. Uriah has the lowest average memory overhead on SPEC CPU2006 benchmarks of 9.3% (right side of Table 4). Uriah-R, which provides full coverage of allocation, has a slightly higher (14.5%) memory consumption than Type-After-Type (13.2%), which again is expected. Previous works consume much more memory than Uriah due to: (1) memory usage for checking mechanisms, e.g., red-zones of ASan; (2) memory usage for allocation metadata of EffectiveSan and DangSan; (3) memory usage by prohibiting memory reuse in FFmalloc and MarkUs. Systems that employ type-based reuse, like Uriah, sometime also have a higher memory utilization. SAFECode also supports reusing heap pools, which enables it to be more memory-efficient than Type-After-Type and Uriah in some cases, despite still being quite high for some programs, like perlbench and sjeng. Also, pool reuse is the main cause of SAFECode's increased runtime overhead. We have observed that the need to reuse pools is often unnecessary as allocated-types are typically reused. Again, we evaluated Uriah on SPEC CPU2017 for its memory overhead. See Figure 2, Uriah exhibits a 5.4% memory overhead on SPEC CPU2017 on average.

	Uriah	*Uriah-R*	*TAT*
SunSpider	1.6%	2.3%	2.1%
Octane 2.0	0.1%	0.3%	-0.2%
Dromaeo JS	2.4%	3.3%	3.8%
Dromaeo DOM	0.8%	1.4%	1.1%

Table 6: Runtime Overhead of Uriah vs. Type-After-Type on Firefox.

7.2.3 Static Analysis Time. Table 5 shows the static safety validation time of Uriah on all the benchmarks we analyzed. In general, the spatial and type safety validation approaches by Uriah are efficient given the **Static** column indicates the least amount of time among the three phases. The CCured analysis and constraint extraction i.e., **Foundation**, takes more time since Uriah leverages the analysis in a context-sensitive manner. We note that the time for static safety validation is a one-time cost for each program, as long as no updates are made to the program. The static analysis time for SPEC benchmarks can be found in the extended paper [42].

7.2.4 Firefox. We applied Uriah on the Firefox browser (changeset ad179a6f) to further evaluate its runtime overhead on real-world applications. We evaluated four commonly-used Firefox benchmarks, namely SunSpider, Octane 2.0, Dromaeo JS, and Dromaeo DOM. Note that we only compared with Type-After-Type in this section since: (1) Uriah shares a similar approach for runtime allocation (i.e., enforcing type-based temporal safety) with Type-After-Type and (2) for all the existing frameworks evaluated in Table 4, only EffectiveSan was evaluated on Firefox originally. Since EffectiveSan is mostly a runtime checking mechanism (i.e., Sanitizer), it exhibits much higher overhead than the allocator-based designs.

As shown in Table 6, Uriah incurs less overhead than Type-After-Type for all four benchmarks tested. However, a key reason is that Uriah only applies its type-based pool allocation for the safe heap objects (i.e., 70.3% of allocation sites), while Type-After-Type applies a similar technique to all the heap allocations. For a fair comparison, we also include the results when Uriah is applied to all heap allocations (i.e., Uriah-R). Uriah generally incurs a slightly higher overhead than Type-After-Type, which is expected given that it distinguishes types more strictly and precisely (i.e., allocated-type) than Type-After-Type, creating more pools. However, the additional overhead of Uriah is modest for Firefox.

7.3 Impact on Mitigating Exploitation

RQ3: *Does* Uriah *improve the security against exploitation of heap memory errors?* While Uriah may expand the range of safe objects, a legitimate concern is whether this (alone) effectively prevents exploits or only reduces the overheads of applying additional defenses. Uriah places heap objects that cannot be validated as satisfying spatial or type safety on an unsafe heap, so perhaps these unsafe heap objects are both the vulnerable objects and target objects exploited in attacks. For evaluating the ability of Uriah to mitigate

CVEs Evaluated	
Spatial	2023-27781; 2023-27117; 2023-27249; 2023-27103; 2023-23456; 2023-1655; 2023-1579; 2023-0433; 2023-0288; 2023-0051
Type	2023-23454; 2023-1078; 2023-1076; 2022-27882; 2021-21861; 2021-21860; 2021-3578; 2021-28275
Temporal	2023-27320; 2023-25136; 2023-22551; 2023-0358; 2023-1449; 2022-47093; 2022-45343; 2022-43680; 2022-43286; 2022-4292

Table 7: Recent CVEs evaluated under the URIAH defense.

attacks on memory errors for heap objects, we choose to examine vulnerabilities in the DARPA CGC binaries and 28 recent CVEs.

7.3.1 Impact on Mitigating Exploits on CGC Binaries. To assess the security implications of URIAH, we apply it on the DARPA CGC Binaries [24]. We picked all 73 Binaries with heap memory errors, including 65 spatial errors, 17 type errors, and 20 temporal errors.

We say that URIAH successfully mitigates the attack when the vulnerable object is classified as unsafe and isolated in the unsafe heap and the target object is classified as safe and allocated in the safe heap. Additionally, temporal errors may also be mitigated by temporal allocated-type safety enforcement. URIAH successfully mitigates all 65 spatial error and 17 type error exploits. For type error cases. 4 of them are incompatible casts, 13 are bad casts on integers (serving as bounds or access ranges) that change signedness, objects are unsafe if index is used in bad cast. 12 of the temporal errors are UAF and 8 are UBI. Of the 20 cases, 6 of them are mitigated by URIAH's static safety validation since the exploits are combined with spatial or type errors. 14 of them are mitigated by URIAH's runtime allocation to ensure temporal allocated-type safety. The UBI cases trigger crashes using original PoC exploit scripts, since URIAH's heap allocator zeros heap memory upon requesting memory from OS, preventing exploitation. Thus, URIAH successfully mitigated all 102 heap memory errors of the 73 CGC binaries.

7.3.2 Impact on Mitigating Exploits on Recent CVEs. We evaluated 28 recent CVEs (see Table 7). We picked all C/C++ heap CVEs in userspace programs in 2023 from CVE database reported before June 2023 when we began the evaluation. Selected CVEs must meet the following requirements: (1) open-sourced; (2) at least one PoC and/or PoV is available; and (3) a patch has been released. These requirements are necessary as we must pinpoint the vulnerable/target object and run the memory safety validation for the program.

We evaluated the exploitation paths used in the available exploits (PoCs). The goal is to find how often advanced exploits corrupt safe target objects to determine whether isolating safe objects can reduce real attack targets. Of course, some vulnerabilities could be exploited by attacking unsafe target objects instead in other exploitation paths, but isolating safe objects for low cost remains worthwhile as it can increase exploit development cost. A challenge is that recent CVEs often do not have a corresponding PoV (i.e., proof-of-vulnerability, exploit write-up/script) released. Thus, we do not know the target objects of any example exploit. However, we can evaluate URIAH's ability to identify vulnerable objects in the PoC/PoV as unsafe. URIAH successfully identified vulnerable objects for all 18 spatial/type error CVEs, by classifying them as unsafe. Runtime allocation of URIAH ensures that those vulnerable objects are isolated on the unsafe heap to block illicit access to the safe heap objects. For the 10 CVEs that relate to temporal memory errors, since vulnerable objects are not involved in spatial/type errors, URIAH's static validation classified them as safe and prevents exploits by enforcing temporal allocated-type safety on the safe heap, preventing temporal exploits (e.g., UAF/DF) that aim at reusing the memory for different allocated-type.

	SPEC CPU2006		**SPEC CPU2017**	
	Native	***w/* URIAH**	***Native***	***w/* URIAH**
TDI	8.4% / 15.5%	2.5% / 3.7%	12.5% / 18.6%	4.4% / 7.1%
CAMP	54.9% / 237.7%	16.8% / 72.3%	21.3% / 127.5%	8.2% / 40.6%

Table 8: Overhead Reduction of Applying TDI and CAMP to URIAH Unsafe Heap. Overhead is represented using the form "(runtime) / (memory)".

7.4 Combining URIAH with Existing Protections

RQ4: *What is the security impact and performance improvement of applying existing protections to the* URIAH *unsafe heap?* We applied two recent frameworks of protecting heap memory safety on URIAH's unsafe heap, namely TDI [81] and CAMP [69].

The results in Table 8, showcase significant reductions in both runtime and memory overhead when protections are only applied to URIAH's unsafe heap (≈70% of their original overhead). These findings highlight the effectiveness of integrating protections with URIAH. More importantly, all the CVEs presented in their paper, namely 28 from CAMP and 4 from TDI, remain prevented by only applying these defenses to URIAH's unsafe heap, highlighting the effectiveness of integrating existing protections with URIAH in reducing costs while keeping the their original security guarantee.

8 Discussion

While we have seen that URIAH does mitigate existing exploits, as previously discussed, there remain classes of attacks that URIAH does not directly protect against: (1) attacks on unsafe objects; (2) attacks where the target and vulnerable objects are the same; and (3) temporal attacks on (re)using memory of objects of the same allocated-type. By design, URIAH does not prevent attacks on heap objects that fail spatial or type safety validation (i.e., unsafe objects), as these unsafe objects are allocated in the unsafe heap. For example, URIAH does not prevent the exploitation of CVE-2022-23088 [7], in which the attacker overflows a field of a structure and gains access to another field. However, URIAH does greatly reduce the fraction of heap objects that are placed in the unsafe heap and are thus exposed to attacks. Fortunately, other defenses can be applied on unsafe heap with reduced cost, as illustrated in Section 7.4.

In some cases, attackers can perform exploits by reusing the memory of the same allocated-type of vulnerable object, but such attacks can be difficult to implement [66]. While Uriah does not completely prevent such temporal exploits through UAF/DF, it does prevent such reuse attacks that (1) require spatial and/or type errors (e.g., DirtyCred [68]) or (2) reuse memory of unsafe objects for safe objects of the same allocated-type (i.e., by isolation). To assess the efficacy of attacks by reusing memory of the same type, we examine the recent DirtyCred attack. DirtyCred allows an attacker to modify authentication credentials by reusing the memory of the same type, but we found that DirtyCred also requires exploitation of other memory errors to enable such reuse, which URIAH prevents successfully. Note that advanced analyses to augment the memory safety validation of URIAH can be adopted, such as T-prunify [145].

9 Related Work

To prevent spatial errors, researchers have proposed techniques to validate that memory accesses are within an object's memory bounds on each reference [27, 83, 110]. Static analysis techniques are

employed to remove checks for objects that can be proven to only be accessed safely [4, 86]. To prevent type errors, runtime checks such as UBSan [73] and VTrust [146] worked only for polymorphic classes of objects, CaVer [59] and TypeSan [38] aimed to include the non-polymorphic objects and C-style type casts but introduced high overhead due to inefficient metadata tracking and inserted checks. Hextype [51] minimizes overhead and increases coverage for various allocation patterns but only supports C++. EffectiveSan [28] includes C programs, but it only deals with upcasts in C++. To prevent temporal errors, detectors of potential temporal safety violations on the heap using static analysis are proposed [31, 72, 139–141, 143, 149]. Unfortunately, all works present soundness limitations. Alternatively, runtime defenses have been proposed by leveraging dangling pointer elimination [58, 130, 142], pointer dereference checking [20, 28, 84, 86, 87, 148], limiting exploitability [13], modifying memory allocators to introduce new memory allocation [108, 129, 134], and garbage collection schemes [2, 25].

Hardware-assisted defenses [54, 62, 65, 92, 104, 106, 111, 138] are emerging to reduce costs introduced by software-based defenses by applying recent hardware features (e.g., ARM PA) and fat-pointer design. However, the applicability of such approaches is usually limited due to the dependency on the specific hardware platform and feature. Moreover, research has already shown that such hardware features can be compromised [99]. Also, the idea of employing cryptography for memory protection and pointer integrity has been proposed [60, 85, 95, 96, 128]. However, these techniques do not ensure memory safety for all objects (e.g., pointers only) and only selectively protect sensitive memory. Moreover, many approaches utilize the unused bits of pointers for encoding the metadata of checking. Given the trend of evolving the OS into 64-bit mode by Intel [48], such designs may no longer be feasible in the near future.

10 Conclusion

We present the Uriah system that enforces temporal allocated-type safety while preserving spatial and type safety of the validated safe heap objects. Uriah leverages static safety validation approaches to validate heap objects that are free from spatial and type memory errors, proposes an efficient memory allocator to isolate objects that passes static safety validations on a separate safe heap, and enforces temporal allocated-type safety on the safe heap. Distinct from previous approaches that tried to detect and eliminate memory errors, Uriah is designed to isolate the targets of exploitation from illicit accesses from vulnerable objects to target objects to thwart attacks. Uriah isolates objects produced at 72.0% of heap allocation sites from exploitations of spatial and type errors, as well as temporal attacks that violate temporal allocated-type safety for low runtime and memory overhead. Combining Uriah with existing protections drastically reduces their overhead while preserving their original security guarantee, offering insights of making existing and future heap memory error defenses more targeted and efficient.

Acknowledgment

We thank our Shepherd and the anonymous reviewers for their insightful feedback. This work was supported by CNS-1801534, ERC Horizon 2020 grant 850868, and SNSF PCEGP2_186974. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agency.

References

[1] Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2005), CCS '05, ACM, p. 340–353.

[2] Ainsworth, S., and Jones, T. M. Markus: Drop-in use-after-free prevention for low-level languages. In *2020 IEEE Symposium on Security and Privacy, SP 2020* (San Francisco, CA, USA, 2020), IEEE, pp. 578–591.

[3] Akritidis, P. Cling: A memory allocator to mitigate dangling pointers. In *Proceedings of the 19th USENIX Conference on Security* (2010).

[4] Akritidis, P., Costa, M., Castro, M., and Hand, S. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th Conference on USENIX Security Symposium* (2009).

[5] Anand, S., and Harrold, M. J. Heap cloning: Enabling dynamic symbolic execution of java programs. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering* (2011), ASE '11, p. 33–42.

[6] Anderson, J. P. Computer security technology planning study. Tech. rep., The Mitre Corporation, Air Force Electronic Systems Division, 1972.

[7] Anonymous. CVE-2022-23088 Exploiting A Heap Overflow in the Freebsd WiFi Stack. https://www.zerodayinitiative.com/blog/2022/6/15/cve-2022-23088-exploiting-a-heap-overflow-in-the-freebsd-wi-fi-stack, 2022.

[8] Apple. About the security content of ios 16.6.1 and ipados 16.6.1. https://support.apple.com/en-us/106361. Accessed: 2024-04-10.

[9] Archives, L. K. Linux kernel coding style. https://www.kernel.org/doc/html/next/process/coding-style.html. Accessed: 2024-04-10.

[10] Barbar, M., Sui, Y., and Chen, S. Flow-sensitive type-based heap cloning. In *34th European Conference on Object-Oriented Programming, ECOOP 2020* (2020).

[11] Berger, E. D., and Zorn, B. G. Diehard: Probabilistic memory safety for unsafe languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2006), PLDI '06.

[12] Bhatkar, S., DuVarney, D. C., and Sekar, R. Address obfuscation: An efficient approach to combat a board range of memory error exploits. In *Proceedings of the 12th Conference on USENIX Security Symposium* (2003).

[13] Burow, N., McKee, D. P., Carr, S. A., and Payer, M. Cfixx: Object type integrity for c++. In *Network and Distributed System Security Symposium (NDSS)* (2018).

[14] Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[15] Chen, S., Xu, J., Sezer, E. C., Gauriar, P., and Iyer, R. K. Non-Control-Data Attacks Are Realistic Threats. In *Proceedings of the 14th Conference on USENIX Security Symposium* (2005).

[16] Chen, Y., Lin, Z., and Xing, X. A systematic study of elastic objects in kernel exploitation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (CCS 2020).

[17] Chen, Y., and Xing, X. Slake: Facilitating slab manipulation for exploiting vulnerabilities in the linux kernel. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (CCS 2019).

[18] Cheng, L., Ahmed, S., Liljestrand, H., Nyman, T., Cai, H., Jaeger, T., Asokan, N., and Yao, D. Exploitation Techniques for Data-Oriented Attacks with Existing and Potential Defense Approaches. *ACM Transactions on Privacy and Security 24*, 4 (2021).

[19] Chipounov, V., Kuznetsov, V., and Candea, G. S2e: A platform for in-vivo multi-path analysis of software systems. *SIGPLAN Not. 46*, 3 (mar 2011), 265–278.

[20] Condit, J., Harren, M., McPeak, S., Necula, G. C., and Weimer, W. Ccured in the real world. *SIGPLAN Not. 38*, 5 (may 2003), 232–244.

[21] Crane, S., Liebchen, C., Homescu, A., Davi, L., Larsen, P., Sadeghi, A.-R., Brunthaler, S., and Franz, M. Readactor: Practical code randomization resilient to memory disclosure. In *2015 IEEE Symposium on Security and Privacy*.

[22] Cyberhaven. Exponential Analysis Speedup with State Merging. http://s2e.systems/docs/StateMerging.html, 2018.

[23] Daniel, M., Honoroff, J., and Miller, C. Engineering heap overflow exploits with javascript. In *Proceedings of the 2nd Conference on USENIX Workshop on Offensive Technologies* (USA, 2008), WOOT'08, USENIX Association.

[24] Darpa. Cyber Grand Challenge. https://github.com/CyberGrandChallenge/.

[25] Dhurjati, D., Kowshik, S., and Adve, V. Safecode: Enforcing alias analysis for weakly typed languages. *SIGPLAN Not. 41*, 6 (June 2006), 144–157.

[26] Duck, Yap, and Cavallaro. Stack Bounds Protection with Low Fat Pointers. In *Proceedings of the 2017 Network and Distributed System Security Symposium*.

[27] Duck, G. J., and Yap, R. H. C. Heap bounds protection with low fat pointers. In *Proceedings of the 25th International Conference on Compiler Construction* (2016).

[28] Duck, G. J., and Yap, R. H. C. Effectivesan: Type and memory error detection using dynamically typed c/c++. *SIGPLAN Not. 53*, 4 (jun 2018), 181–195.

[29] Elliott, A. S., Ruef, A., Hicks, M., and Tarditi, D. Checked c: Making c safe by extension. In *2018 IEEE Cybersecurity Development (SecDev)*, pp. 53–60.

[30] Emsisoft. The cost of ransomware in 2020: A country-by-country analysis. https://blog.emsisoft.com/en/36665/the-cost-of-ransomware-in-2020-a-country-by-country-analysis/, 2020. Accessed on May 13, 2023.

[31] Feist, J., Mounier, L., and Potet, M.-L. Statically detecting use after free on binary code. *Journal of Computer Virology and Hacking Techniques 10* (2014).

[32] Ghemawat, S., and Menage, P. Tcmalloc: Thread-caching malloc. https://goog-perftools.sourceforge.net/doc/tcmalloc.html, 2021.

[33] GitHub. GitHub Copilot. https://copilot.github.com/.

[34] Golick, J. How tcmalloc works. https://jamesgolick.com/2013/5/19/how-tcmalloc-works.html, 2013.

[35] Google. Google c++ style guide - static and global variables. https://google.github.io/styleguide/cppguide.html#Static_and_Global_Variables.

[36] Google. Partitionalloc design. https://chromium.googlesource.com/chromium/src/+/master/base/allocator/partition_allocator/PartitionAlloc.md, 2021.

[37] Google. Restartable sequence mechanism for tcmalloc. https://github.com/google/tcmalloc/blob/master/docs/rseq.md, 2023.

[38] Haller, I., Jeon, Y., Peng, H., Payer, M., Giuffrida, C., Bos, H., and van der Kouwe, E. Typesan: Practical type confusion detection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016).

[39] Hu, H., Shinde, S., Adrian, S., Chua, Z. L., Saxena, P., and Liang, Z. Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy* (SP 2016).

[40] Huang, K., Huang, Y., Payer, M., Qian, Z., Sampson, J., Tan, G., and Jaeger, T. The taming of the stack: Isolating stack data from memory errors. In *Network and Distributed System Security Symposium* (NDSS 2022).

[41] Huang, K., Payer, M., Qian, Z., Sampson, J., Tan, G., and Jaeger, T. Comprehensive memory safety validation: An alternative approach to memory safety. *IEEE Security & Privacy* (April 2024).

[42] Huang, K., Payer, M., Qian, Z., Sampson, J., Tan, G., and Jaeger, T. Top of the heap: Efficient memory error protection of safe heap objects. https://arxiv.org/abs/2310.06397, 2024.

[43] Huang, K., Sampson, J., and Jaeger, T. Assessing the impact of efficiently protecting ten million stack objects from memory errors comprehensively. In *Proceedings of the 2023 IEEE Secure Development Conference* (IEEE SecDev 2023).

[44] Huang, Z., Lie, D., Tan, G., and Jaeger, T. Using safety properties to generate vulnerability patches. In *2019 IEEE Symposium on Security and Privacy* (SP).

[45] Intel. Intel guide for developing multithreaded application. https://www.intel.com/content/dam/develop/external/us/en/documents/gdma-2-165938.pdf. 2011.

[46] Intel. Intel oneapi threading building blocks. https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html#gs.63k1wf.

[47] Intel. Intel mpx explained - performance evaluation. https://intel-mpx.github.io/performance/, 2018. Accessed on May 23, 2023.

[48] Intel. Envisioning a simplified intel architecture. https://www.intel.com/content/www/us/en/developer/articles/technical/envisioning-future-simplified-architecture.html, 2023. Accessed on May 21, 2023.

[49] Ispoglou, K. K., AlBassam, B., Jaeger, T., and Payer, M. Block oriented programming: Automating data-only attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (CCS 2018).

[50] jemalloc. jemalloc — general purpose memory allocation functions. https://jemalloc.net/jemalloc.3.html. Accessed on Mar 7, 2024.

[51] Jeon, Y., Biswas, P., Carr, S., Lee, B., and Payer, M. Hextype: Efficient detection of type confusion errors for c++. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), CCS '17.

[52] Karenina, A. Anna karenina principle, 2023.

[53] Kellermann, M. The Dirty Pipe Vulnerability. https://dirtypipe.cm4all.com/.

[54] Kim, Y., Lee, J., and Kim, H. Hardware-based always-on heap memory safety. In *2020 Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO).

[55] Kuznetsov, V., Szekeres, L., Payer, M., Candea, G., Sekar, R., and Song, D. Code-pointer integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (2014), OSDI'14.

[56] Landi, W., and Ryder, B. G. A safe approximate algorithm for interprocedural pointer aliasing. *SIGPLAN Not. 39*, 4 (apr 2004), 473–489.

[57] Lattner, C., Lenharth, A., and Adve, V. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[58] Lee, B., Song, C., Jang, Y., Wang, T., Kim, T., Lu, L., and Lee, W. Preventing Use-after-free with Dangling Pointers Nullification. In *Proceedings of the 2015 Network and Distributed System Security Symposium* (NDSS 2015).

[59] Lee, B., Song, C., Kim, T., and Lee, W. Type casting verification: Stopping an emerging attack vector. In *Proceedings of the 2015 USENIX Security Symposium*.

[60] LeMay, M., Rakshit, J., Deutsch, S., Durham, D. M., Ghosh, S., Nori, A., Gaur, J., Weiler, A., Sultana, S., Grewal, K., and Subramoney, S. Cryptographic capability computing. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO 2021).

[61] Li, G., Zhang, H., Zhou, J., Shen, W., Sui, Y., and Qian, Z. A hybrid alias analysis and its application to global variable protection in the linux kernel. In *32nd USENIX Security Symposium* (2023).

[62] Li, Y., Tan, W., Lv, Z., Yang, S., Payer, M., Liu, Y., and Zhang, C. Pacmem: Enforcing spatial and temporal memory safety via arm pointer authentication. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (CCS 2022).

[63] Liang, Z., Zou, X., Song, C., and Qian, Z. K-leak: Towards automating the generation of multi-step infoleak exploits against the linux kernel. In *31st Annual Network and Distributed System Security Symposium, NDSS* (2024).

[64] Library, G. C. Glibc wiki - mallocinternals. https://sourceware.org/glibc/wiki/MallocInternals. Accessed on Mar 7, 2024.

[65] Liljestrand, H., Nyman, T., Wang, K., Perez, C. C., Ekberg, J.-E., and Asokan, N. Pac it up: Towards pointer integrity using arm pointer authentication. In *Proceedings of the 28th USENIX Conference on Security Symposium* (2019).

[66] Lin, Z. How autoslab changes the memory unsafety game. https://grsecurity.net/how_autoslab_changes_the_memory_unsafety_game, 2022.

[67] Lin, Z., Chen, Y., Wu, Y., Mu, D., Yu, C., Xing, X., and Li, K. Grebe: Unveiling exploitation potential for linux kernel bugs. In *2022 IEEE Symposium on Security and Privacy* (SP 2022).

[68] Lin, Z., Wu, Y., and Xing, X. Dirtycred: Escalating privilege in linux kernel. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (CCS 2022).

[69] Lin, Z., Yu, Z., Guo, Z., Campanoni, S., Dinda, P., and Xing, X. CAMP: Compiler and allocator-based heap memory protection. In *33rd USENIX Security Symposium (USENIX Security 24)* (Philadelphia, PA, Aug. 2024).

[70] Linux. Linux 2.6.7. nx (no execute) support for x86. https://lkml.org/lkml/2004/6/2/228, 2004.

[71] Liu, S., Tan, G., and Jaeger, T. Ptrsplit: Supporting general pointers in automatic program partitioning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (CCS 2017).

[72] LLVM. Clang static analyzer. https://clang-analyzer.llvm.org/, 2023.

[73] LLVM. Clang undefined behavior sanitizer. http://clang.llvm.org/docs/UsersManual.html, 2023. Accessed: 2023-05-02.

[74] Canonicaliza natural loops. LLVM documentation at https://llvm.org/docs/Passes.html#loop-simplify-canonicalize-natural-loops, 2020.

[75] Loop Simplify Form. LLVM documentation at https://llvm.org/docs/LoopTerminology.html#loop-simplify-form, 2020.

[76] Mapping High Level Constructs to LLVM IR - Union. https://mapping-high-level-constructs-to-llvm-ir.readthedocs.io/en/latest/basic-constructs/unions.html.

[77] Microsoft. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape, 2019.

[78] Microsoft. Customize exploit protection. https://learn.microsoft.com/en-us/microsoft-365/security/defender-endpoint/customize-exploit-protection?view=o365-worldwide, 2022.

[79] Midi, D., Payer, M., and Bertino, E. Memory safety for embedded devices with nescheck. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* (ASIA CCS 2017).

[80] Milburn, A., Bos, H., and Giuffrida, C. Safelnit: Comprehensive and practical mitigation of uninitialized read vulnerabilities. In *Network and Distributed System Security Symposium* (NDSS 2017).

[81] Milburn, A., Van Der Kouwe, E., and Giuffrida, C. Mitigating information leakage vulnerabilities with type-based data isolation. In *2022 IEEE Symposium on Security and Privacy* (SP 2022).

[82] Moore, D., Paxson, V., Savage, S., Shannon, C., Staniford, S., and Weaver, N. Inside the Slammer worm. *IEEE Security & Privacy 1*, 4 (2003), 33–39.

[83] Nagarakatte, S., Zhao, J., Martin, M. M., and Zdancewic, S. Softbound: Highly compatible and complete spatial memory safety for c. In *2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[84] Nagarakatte, S., Zhao, J., Martin, M. M., and Zdancewic, S. Cets: Compiler enforced temporal safety for c. In *Proceedings of the 2010 International Symposium on Memory Management* (ISMM 2010).

[85] Nasahl, P., Schilling, R., Werner, M., Hoogerbrugge, J., Medwed, M., and Mangard, S. Cryptag: Thwarting physical and logical memory vulnerabilities using cryptographically colored memory. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security* (ASIA CCS 2021).

[86] Necula, G. C., Condit, J., Harren, M., McPeak, S., and Weimer, W. Ccured: Type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst. 27*, 3.

[87] Necula, G. C., McPeak, S., and Weimer, W. Ccured: Type-safe retrofitting of legacy code. *SIGPLAN Not. 37*, 1 (jan 2002), 128–139.

[88] Neugschwandtner, M., Sorniotti, A., and Kurmus, A. Memory categorization: Separating attacker-controlled data. In *Detection of Intrusions and Malware, and Vulnerability Assessment* (Cham, 2019), Springer International Publishing.

[89] Nginx. Nginx development guide - common pitfalls. https://nginx.org/en/docs/dev/development_guide.html#common_pitfalls. Accessed: 2024-04-10.

[90] Novark, G., and Berger, E. D. Dieharder: Securing the heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (2010).

[91] NSA-CSS. Nsa releases guidance on how to protect against software memory safety issues, 2022.

[92] Oleksenko, O., Kuvaiskii, D., Bhatotia, P., Felber, P., and Fetzer, C. Intel

mpx explained: A cross-layer analysis of the intel mpx system stack. *ACM SIGMETRICS Performance Evaluation Review* (2019).
[93] One, A. Smashing the stack for fun and profit. *Phrack magazine* (1996).
[94] OpenAI. ChatGPT. https://chat.openai.com/. Accessed on May 13, 2023.
[95] Palit, T., Firose Moon, J., Monrose, F., and Polychronakis, M. Dynpta: Combining static and dynamic analysis for practical selective data protection. In *2021 IEEE Symposium on Security and Privacy* (SP 2021).
[96] Proskurin, S., Momeu, M., Ghavamnia, S., Kemerlis, V. P., and Polychronakis, M. xmp: Selective memory protection for kernel and user space. In *2020 IEEE Symposium on Security and Privacy* (SP 2020).
[97] ptmalloc. realloc(3) - linux man page. https://linux.die.net/man/3/realloc.
[98] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. Language models are unsupervised multitask learners. *OpenAI Blog* (June 2019).
[99] Ravichandran, J., Na, W. T., Lang, J., and Yan, M. Pacman: Attacking arm pointer authentication with speculative execution. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (ISCA 2022).
[100] Riku, Antti, Matti, and Mehta, N. Heartbleed. http://heartbleed.com/, 2014.
[101] Roemer, R., Buchanan, E., Shacham, H., and Savage, S. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur. 15*, 1 (2012).
[102] Security, A. Towards the next generation of xnu memory safety: kalloc_type. https://security.apple.com/blog/towards-the-next-generation-of-xnu-memory-safety/, 2022.
[103] Seeley, D. A Tour of the Worm. https://www.cs.unc.edu/~jeffay/courses/nidsS05/attacks/seely-RTMworm-89.html.
[104] Serebryany, K. Arm memory tagging extension and how it improves c/c++ memory safety. *login Usenix Mag. 44* (2019).
[105] Serebryany, K., Bruening, D., Potapenko, A., and Vyukov, D. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (ATC 2012).
[106] Sharifi, R., and Venkat, A. Chex86: Context-sensitive enforcement of memory safety via microcode-enabled capabilities. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture* (ISCA 2020).
[107] Sharir, M., and Pnueli, A. *Two approaches to interprocedural data flow analysis.*
[108] Silvestro, S., Liu, H., Crosser, C., Lin, Z., and Liu, T. Freeguard: A faster secure heap allocator. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (CCS 2017).
[109] Simon, A. Value-range analysis of c programs: Towards proving the absence of buffer overflow vulnerabilities, 2008.
[110] Simpson, M. S., and Barua, R. K. MemSafe: Ensuring the Spatial and Temporal Memory Safety of C at Runtime. In *Proceedings of the 2010 10th IEEE Working Conference on Source Code Analysis and Manipulation* (SCAM 2010).
[111] Sinha, K., and Sethumadhavan, S. Practical memory safety with rest. In *Proceedings of the 2018 Annual International Symposium on Computer Architecture*.
[112] Smaragdakis, Y., and Kastrinis, G. Defensive Points-To Analysis: Effective Soundness via Laziness. In *32nd European Conference on Object-Oriented Programming* (ECOOP 2018).
[113] Snow, K. Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., and Sadeghi, A.-R. Just-in-time Code Reuse: On the Effectiveness of Fine-grained Address Space Layout Randomization. In *Proceedings of the 34th IEEE Symposium on Security and Privacy* (S&P 2013).
[114] Sophos. The state of ransomware. https://news.sophos.com/en-us/2024/04/30/the-state-of-ransomware-2024, 2024.
[115] Sotirov, A. Heap feng shui in javascript. *Black Hat Europe* (2007).
[116] Stamatogiannakis, M., Bos, H., Giuffrida, C., Mavroudis, V., and Papadopoulos, S. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy* (S&P 2022).
[117] Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., and Vigna, G. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium* (NDSS 2016).
[118] Sui, Y., Di, P., and Xue, J. Sparse flow-sensitive pointer analysis for multithreaded programs. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization* (CGO 2016).
[119] Sui, Y., and Xue, J. Svf: Interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th International Conference on Compiler Construction* (2016).
[120] Sui, Y., and Xue, J. Value-flow-based demand-driven pointer analysis for c and c++. *IEEE Transactions on Software Engineering 46*, 8 (2018), 812–835.
[121] Sui, Y., and Xue, J. On-demand strong update analysis via value-flow refinement. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (FSE 2016).
[122] Sui, Y., Ye, D., and Xue, J. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Transactions on Software Engineering (TSE) 40*, 2 (2014).
[123] Sui, Y., Ye, D., and Xue, J. Static memory leak detection using full-sparse value-flow analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis* (ISSTA 2012).
[124] Taylor, A., Whalley, A., Jansens, D., and Oskov, N. An update on memory safety in chrome. https://security.googleblog.com/2021/09/an-update-on-memory-safety-in-chrome.html, 2021.
[125] Teixeira, D., and Pereira, F. M. Q. The Design and Implementation of a Non-Iterative Range Analysis Algorithm on a Production Compiler. In *The 15th Brazilian Symposium on Programming Languages* (SBLP 2011).
[126] The White House. Press release: Future software should be memory safe. https://www.whitehouse.gov/oncd/briefing-room/2024/02/26/press-release-technical-report/. FEBRUARY 26, 2024.
[127] Tolstoy, L. *Anna Karenina.* Wordsworth Editions, Tsarist Russia, 1995.
[128] Unterguggenberger, M., Schrammel, D., Lamster, L., Nasahl, P., and Mangard, S. Cryptographically enforced memory safety. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (CCS 2023).
[129] van der Kouwe, E., Kroes, T., Ouwehand, C., Bos, H., and Giuffrida, C. Type-after-type: Practical and complete type-safe memory reuse. In *Proceedings of the 34th Annual Computer Security Applications Conference* (ACSAC 2018).
[130] van der Kouwe, E., Nigade, V., and Giuffrida, C. Dangsan: Scalable use-after-free detection. In *2017 European Conference on Computer Systems*.
[131] Ventures, C. Cybersecurity ventures' ransomware damage report. https://cybersecurityventures.com/cybersecurity-500/.
[132] Wang, Y., Zhang, C., Zhao, Z., Zhang, B., Gong, X., and Zou, W. MAZE: Towards automated heap feng shui. In *30th USENIX Security Symposium* (2021).
[133] Whaley, J., and Lam, M. S. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation* (2004).
[134] Wickman, B., Hu, H., Yun, I., Jang, D., Lim, J., Kashyap, S., and Kim, T. Preventing Use-After-Free attacks with fast forward allocation. In *30th USENIX Security Symposium* (2021).
[135] Wu, W., Chen, Y., Xing, X., and Zou, W. Kepler: facilitating control-flow hijacking primitive evaluation for linux kernel vulnerabilities. In *Proceedings of the 28th USENIX Security Symposium* (SEC 2019).
[136] Wu, W., Chen, Y., Xu, J., Xing, X., Gong, X., and Zou, W. FUZE: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. In *27th USENIX Security Symposium* (SEC 2018).
[137] Xu, F. F., Alon, U., Neubig, G., and Hellendoorn, V. J. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming* (MAPS 2022).
[138] Xu, S., Huang, W., and Lie, D. In-fat pointer: Hardware-assisted tagged-pointer spatial memory safety defense with subobject granularity protection. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS 2021).
[139] Yan, H., Sui, Y., Chen, S., and Xue, J. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *Proceedings of the 2018 International Conference on Software Engineering*.
[140] Yan, H., Sui, Y., Chen, S., and Xue, J. Machine-learning-guided typestate analysis for static use-after-free detection. In *Proceedings of the 33rd Annual Computer Security Applications Conference* (ACSAC 2017).
[141] Ye, J., Zhang, C., and Han, X. Poster: Uafchecker: Scalable static detection of use-after-free vulnerabilities. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (CCS 2014).
[142] Younan, Y. FreeSentry: Protecting Against Use-after-free Vulnerabilities Due to Dangling Pointers. In *22nd Annual Network and Distributed System Security Symposium* (NDSS 2015).
[143] Zhai, Y., Hao, Y., Zhang, H., Wang, D., Song, C., Qian, Z., Lesani, M., Krishnamurthy, S. V., and Yu, P. Ubitect: A precise and scalable method to detect use-before-initialization bugs in linux kernel. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (ESEC/FSE 2020).
[144] Zhai, Y., Hao, Y., Zhang, Z., Chen, W., Li, G., Qian, Z., Song, C., Sridharan, M., Krishnamurthy, S. V., Jaeger, T., and Yu, P. Progressive scrutiny: Incremental detection of ubi bugs in the linux kernel. In *Proceedings 2022 Network and Distributed System Security Symposium* (NDSS 2022).
[145] Zhai, Y., Qian, Z., Song, C., Sridharan, M., Jaeger, T., Yu, P., and Krishnamurthy, S. V. Don't waste my efforts: Pruning redundant sanitizer checks of developer-implemented type checks. In *USENIX Security Symposium* (2024).
[146] Zhang, C., Carr, S. A., Li, T., Ding, Y., Song, C., Payer, M., and Song, D. Vtrust: Regaining trust on virtual calls. In *Symposium on Network and Distributed System Security* (NDSS 2016).
[147] Zhang, Y., Pang, C., Portokalidis, G., Triandopoulos, N., and Xu, J. Debloating address sanitizer. In *31st USENIX Security Symposium* (SEC 2022).
[148] Zhou, J., Criswell, J., and Hicks, M. Fat pointers for temporal memory safety of c. *Proc. ACM Program. Lang. 7*, OOPSLA1 (2023).
[149] Zhu, K., Lu, Y., and Huang, H. Scalable static detection of use-after-free vulnerabilities in binary code. *IEEE Access 8* (2020).
[150] Zou, X., Hao, Y., Zhang, Z., Pu, J., Chen, W., and Qian, Z. Syzbridge: Bridging the gap in exploitability assessment of linux kernel bugs in the linux ecosystem. In *31st Annual Network and Distributed System Security Symposium* (NDSS 2024).
[151] Zou, X., Li, G., Chen, W., Zhang, H., and Qian, Z. SyzScope: Revealing High-Risk security impacts of Fuzzer-Exposed bugs in linux kernel. In *31st USENIX Security Symposium* (SEC 2022).