Fat Pointers for Temporal Memory Safety of C

JIE ZHOU, University of Rochester, USA
JOHN CRISWELL, University of Rochester, USA
MICHAEL HICKS*, Amazon and University of Maryland, USA

Temporal memory safety bugs, especially use-after-free and double free bugs, pose a major security threat to C programs. Real-world exploits utilizing these bugs enable attackers to read and write arbitrary memory locations, causing disastrous violations of confidentiality, integrity, and availability. Many previous solutions retrofit temporal memory safety to C, but they all either incur high performance overhead and/or miss detecting certain types of temporal memory safety bugs.

In this paper, we propose a temporal memory safety solution that is both efficient and comprehensive. Specifically, we extend Checked C, a spatially-safe extension to C, with temporally-safe pointers. These are implemented by combining two techniques: fat pointers and dynamic key-lock checks. We show that the fat-pointer solution significantly improves running time and memory overhead compared to the disjoint-metadata approach that provides the same level of protection. With empirical program data and hands-on experience porting real-world applications, we also show that our solution is practical in terms of backward compatibility—one of the major complaints about fat pointers.

CCS Concepts: • **Security and privacy** → **Software security engineering**; • **Software and its engineering** → **General programming languages**.

Additional Key Words and Phrases: Temporal Memory Safety, Fat Pointers, Checked C

ACM Reference Format:
Jie Zhou, John Criswell, and Michael Hicks. 2023. Fat Pointers for Temporal Memory Safety of C. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 86 (April 2023), 32 pages. https://doi.org/10.1145/3586038

1 INTRODUCTION

A temporal memory safety violation occurs when a program dereferences a pointer whose referent memory object has already been freed (*use after free* or *UAF*), frees a pointer more than once (*double free*), or frees a pointer that does not point to the start of a heap object (*invalid free*). Exploiting a temporal safety-violating bug may allow an attacker to read or write an arbitrary memory location and thereby to steal information, corrupt memory, or even execute arbitrary code [Afek and Sharabani 2007; Enumeration 2020; Phantasmagoria 2005; Xu et al. 2015]. Unfortunately, the past decade has seen an increase in such exploits used in the real world [Cimpanu 2020; Miller 2019; Nagaraju et al. 2013].

One can enforce temporal memory safety by associating with each memory object, at allocation time, a distinct *lock*, and with each pointer a *key*. An object's lock is invalidated when it is freed. At each pointer dereference, a dynamic check confirms the pointer's key matches the referent object's lock; if not, it signals a UAF. Prior solutions [Austin et al. 1994; Nagarakatte et al. 2010;

*Work completed prior to starting at Amazon.

Authors' addresses: Jie Zhou, University of Rochester, USA, jiezhou@rochester.edu; John Criswell, University of Rochester, USA, criswell@cs.rochester.edu; Michael Hicks, Amazon and University of Maryland, USA, mwh@cs.umd.edu.

2475-1421/2023/4-ART86
https://doi.org/10.1145/3586038

Patil and Fischer 1997; Xu et al. 2004] have explored this approach but incur high memory and/or performance overhead. For example, CETS [Nagarakatte et al. 2010] incurs 48% performance overhead on selected SPEC CPU2006 benchmarks. PTAuth [Farkhani et al. 2021] and ViK [Cho et al. 2022] lower the performance overhead on SPEC to 26% and 9%, respectively, but they trade security and scalability for speed. Both use a small key space (10–16 bits), and they require a predetermined relatively small maximum object size—for objects larger than the maximum, PTAuth may raise false alarms while ViK may miss safety violations (see Section 8.1 for details).

When experimenting with key-lock check approaches, we found that a crucial reason for their high cost is the use of *disjoint* data structures, such as look-up tables, to maintain the association between keys/locks and pointers/objects. Doing so keeps objects and pointers unchanged from their legacy representation, which makes it easy for compiled-to-be-safe code to interoperate with unchanged (e.g., library) code. However, the approach requires the compiler to add instructions to locate the keys and locks at pointer propagations and dereferences.

An alternative to making metadata disjoint from a pointer is to store it *in place*, yielding a kind of *fat pointer*.[1] Fat pointers were oft-proposed for enforcing spatial memory safety (i.e., bounds checks) [Jim et al. 2002; Necula et al. 2002] but fell out of favor because of both high memory- and run-time overhead and difficulties interoperating with legacy components. However, we observe that fat pointers do not present the same interoperability issues when temporal memory safety checks are included with a new *language feature* rather than added automatically as part of a *compilation strategy* for unmodified C code. This is because a language extension can provide different pointer types that expose to the programmer the difference in representation between safe and legacy pointers, enabling programmers to choose when and how to convert between the two representations. However, it is an open question whether fat pointers could be a practical solution for writing new temporally safe C code or to retrofit temporal memory safety to existing C code.

To that end, we extended the Checked C programming language [Elliott et al. 2018; Li et al. 2022; Tarditi 2021] with new types of temporally safe pointers, storing keys in place with pointers, and locks in place with pointed-to objects. Checked C is a new safe extension to C which efficiently enforces spatial memory safety, providing a solid foundation on which we can build to explore our in-place key-lock strategy for temporal memory safety. Checked C's type system provides strong security guarantees at compile time, eliminating common dangerous C idioms such as arbitrary casts that could break the security benefits of our safe pointers. As Checked C is a proper programming language, programmers can naturally address any compatibility issues caused by the use of fat pointers, whether initially or during program maintenance, whereas doing so would be far more challenging when working with an automatic compiler transformation [Nagarakatte et al. 2009; Necula et al. 2005].

We implemented our new pointer types in the Checked C compiler, which is based on Clang and LLVM [Lattner and Adve 2004]. By design, these pointers extend Checked C's spatially-safe pointers, so in a full implementation they would be subject to spatial safety checks. We have delayed this (conceptually straightforward, though nontrivial) integration effort in order to first evaluate the effectiveness of in-place metadata for temporal memory safety checking. To do so, we ported Olden (a small pointer-intensive benchmark suite), one pointer-intensive SPEC benchmark, three real-world applications, and the engine and HTTP protocol part of a large ubiquitous program `curl` to use temporally safe pointers and measured the resulting performance and memory overhead. We

[1] Some works (e.g., Nagarakatte et al. [2015]) refer to all techniques that associate pointers with metadata as fat pointers. In this paper, we use "fat pointers" to only mean pointers with in-place metadata.

compared our Checked C solution with CETS [Nagarakatte et al. 2010]—the state-of-the-art key-lock check approach[2] using disjoint metadata—and show that our in-place mechanism significantly reduces performance overhead (29% vs. 92%) and memory overhead (72% vs. 202%) on Olden.

While Checked C can be used for writing new code, we also recognize that programmers will want to port existing C code to gain memory safety. We therefore report on our experience porting the benchmarks and applications. On average, we can port 1–2 K lines of code per person-day. We believe that porting can be facilitated by adapting 3C [Machiry et al. 2022], a semi-automated porting tool from C to spatially-safe Checked C, to work with our temporally safe pointers.

In summary, we make the following contributions:

- We explore the benefits and costs of using fat pointers to retrofit temporal memory safety to C. We add four new types of safe pointers to Checked C to provide *full* temporal memory safety. We implemented the two most common types for 64-bit systems.
- We evaluated the new safe pointers by measuring their performance and memory overhead. We show that our fat pointer solution is *efficient* in that it incurs significantly lower performance and memory overheads compared with a disjoint metadata mechanism.
- We show that our solution is *practical* in terms of backward compatibility with legacy C code. We support this claim with empirical program data and our experience of porting real-world applications.

Roadmap. We first briefly review Checked C in Section 2. We then describe the motivations for our design choices and details of the new safe pointers in Section 3. We discuss the issue of backward compatibility with legacy C libraries in Section 4. Next, Section 5 covers the implementation. We describe the performance and memory consumption evaluation in Section 6. After that, we report our experience of porting the benchmarks in Section 7. In Section 8, we compare our work with other key-lock check works in details, and we also discuss and compare with two other major types of temporal memory safety solutions. We then briefly summarize the most important directions for future work and conclude the paper in Section 9.

2 BACKGROUND ON CHECKED C

Checked C [Elliott et al. 2018] is an extension to C that enforces spatial memory safety. It takes inspiration from prior work on safe-C dialects [Condit et al. 2007; Jim et al. 2002; Kowshik et al. 2002] but differs in that it favors *easy incremental porting of and interoperation with legacy code*. This section briefly introduces three key concepts of Checked C. Design details can be found in the language specifications [Tarditi 2021] while Li et al. [2022] provide a formal model of Checked C.

Checked Pointers. Checked C extends C with three new types of spatially-safe checked pointers [Elliott et al. 2018]. `ptr<T>` is a pointer to a single type-`T` object and thus cannot be used in pointer arithmetic; `array_ptr<T>` and `nt_array_ptr<T>` ("null-terminated" `array_ptr`) are array types and can be used in pointer arithmetic expressions. A checked array pointer is associated with a *bounds expression* that delineates the pointed-to array's size. Bounds expressions are normal program expressions and serve as an *invariant* for checking the validity of pointer-related operations: the compiler rejects the program if it detects a violation (e.g., an out-of-bounds access) statically and inserts dynamic bounds checks when it cannot prove safety statically. Bounds expressions are not stored *in place*, i.e., as "fat" pointers [Jim et al. 2002; Necula et al. 2005]. This improves backward compatibility with legacy C code as the run-time pointer representation remains unchanged [Tarditi

[2] As mentioned earlier, PTAuth [Farkhani et al. 2021] and ViK [Cho et al. 2022] are more recent but lack the same security and scalability benefits of CETS and our approach; see Section 8.1.

```
char *strncpy(char *dst : itype(array_ptr<char>) count(len),
              char *src : itype(array_ptr<char>) count(len),
              size_t len);
```

Fig. 1. Bounds-safe Interface for `strncpy`

2021]. Checked C strictly prohibits casting or assigning a raw C pointer to a checked pointer, which prevents forging checked pointers from unsafe sources.

Checked Region. Checked C allows mixing uses of checked and raw C pointers, which helps incremental conversion of legacy C programs. Programmers can use the `_Checked` keyword to explicitly put a block of code (from a single statement to a whole source file) into a checked region within which uses of legacy C pointer types are disallowed and spatial safety is provably assured [Li et al. 2022]. Similarly, programmers can use the `_Unchecked` keyword to enclose code in an *unchecked region* that disables the compiler's memory safety checks.

Bounds-Safe Interface. Checked C provides bounds-safe interfaces (BSI) for better interaction between checked and unchecked code (legacy libraries and unported source code). Programmers can declare a BSI for an unchecked function with `itype` parameters so the function can be called by both checked and unchecked code. `itype` (short for *inter-op type*) is a new keyword introduced by Checked C to annotate pointers to be used in both checked and unchecked contexts. Notably, the compiler treats an `itype` pointer as a raw C pointer when it is passed to or used in unchecked C code, and treats it as a checked pointer (enforcing necessary memory safety checks) in checked regions. Figure 1 shows an example of the BSI for `strncpy`; checked code can pass an `array_ptr` to `strncpy` via the BSI, and unchecked code can pass in a raw C pointer.

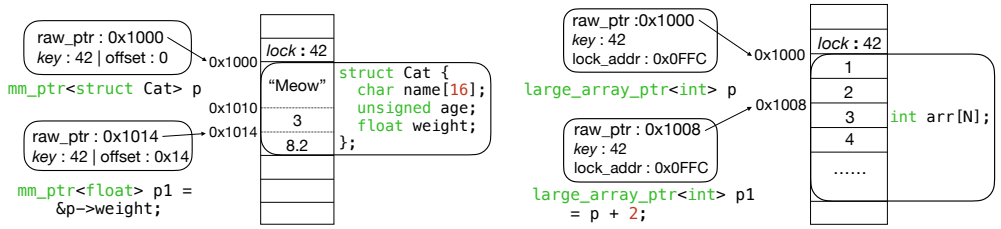
3 TEMPORALLY MEMORY SAFE POINTERS

This section presents our new approach to integrating temporally memory-safe pointers in Checked C. We begin with the motivations and overview of our design and then describe the details of our various pointer types and their metadata, including how we manage locks and keys. We discuss how we handle backward compatibility with legacy libraries in Section 4.

3.1 Design Overview

3.1.1 Goal. We aim to prevent *all* temporal safety violations, of which there are three varieties: *use after free* (UAF), *double free*, and *invalid free*. While UAFs can happen with pointers to stack objects, they are arguably challenging to exploit [Lee et al. 2015; Nagaraju et al. 2013]. As a result, they are ignored by most related work [Ainsworth and Jones 2020; Cho et al. 2022; Farkhani et al. 2021; Liu et al. 2018; Shen and Dolan-Gavitt 2020; Shin et al. 2019; van der Kouwe et al. 2017; Wickman et al. 2021]. Our design aims to detect them nevertheless to ensure comprehensive temporal memory safety enforcement. At present, we do not check for spatial safety violations via the new pointers; our focus is to prototype temporal safety enforcement and its performance. Though the engineering effort of adding spatial checks is nontrivial (estimated to be at least one person-year; more details in §5.4.2), they are essentially orthogonal to temporal safety support (e.g., they do not affect pointer/object representation).

3.1.2 In-place metadata. Prior work has enforced temporal safety by checking that a pointer's *key* matches its object's *lock* on a dereference. Key-lock metadata is typically kept in data structures disjoint from the pointer and object [Gui et al. 2021; Nagarakatte et al. 2010; Patil and Fischer 1997; Xu et al. 2004]. For example, CETS [Nagarakatte et al. 2010] uses a two-level lookup table to locate its keys and locks. Disjoint metadata approaches afford good backward compatibility with



(a) Two `mm_ptrs` to a `struct`. One points to the beginning and another points to the middle of it.

(b) Two `mm_ptrs` to a large array of integers greater than 4 GB.

Fig. 2. Structure of Temporally Safe Fat Pointers

legacy code and lowers the possibility of metadata corruption when spatial memory safety is not assured. However, to query and update the metadata, a program must first *dynamically* locate it, and the lookup procedure can be very slow. Inspired by this observation, we decided to see whether pointers with in-place keys (i.e., *fat* pointers [Jim et al. 2002; Necula et al. 2005]) and objects with in-place locks could make metadata accesses significantly faster.

Similar to previous key-lock approaches [Austin et al. 1994; Cho et al. 2022; Farkhani et al. 2021; Nagarakatte et al. 2010; Patil and Fischer 1997], we associate each pointer with a key and its referent with a lock. Memory allocation sets the lock to a unique value and returns a pointer with a key set to the same value as the lock; memory deallocation invalidates the lock to a value that will never be used for any key. On a dereference, the compiler inserts a check (omitted if proven safe) to confirm that the pointer's key matches its referent's lock. A failed key check signals a temporal memory safety violation. Different from disjoint key-lock methods [Gui et al. 2021; Nagarakatte et al. 2010; Patil and Fischer 1997], our solution locates the key as part of a fat pointer, and locates the lock just before the referent object. Thus, the location of a key is *statically* known, and the location of a lock is either statically known or can be computed by a few simple arithmetic and bitwise instructions (§3.2). Consequently, metadata propagation, updates, and validity checks are much faster.

3.1.3 Checked C. There are two main benefits offered by Checked C as a host language for temporally safe pointers with in-place metadata. First, Checked C ensures that metadata will not be corrupted or fabricated. Checked C's checked pointers (§2) can only originate from a heap allocation, the address of a stack/global/thread-local object, or from another checked pointer. Checked C's type system disallows casting a raw C pointer to a checked pointer. This eliminates the possibility of forging checked pointer metadata from untrusted sources at compile time. Second, fat pointers integrate well with Checked C's approach for enforcing spatial memory safety. Fat pointer approaches for spatial safety usually have at least two fields added per pointer, (base and upper bound or size) [Jim et al. 2002; Necula et al. 2005]. Additional metadata for temporal memory safety would make fat pointers heavier, thus causing slow metadata propagation and updates. However, Checked C achieves spatial memory safety with low performance overhead [Duan et al. 2020; Elliott et al. 2018] *without* using fat pointers. Therefore, we may afford to combine its current mechanism with fat pointers to realize full memory safety efficiently. Additionally, Checked C gives programmers fine-grained control over the source code, permitting manual handling of backward compatibility issues—a major concern for fat pointer approaches.

3.2 Pointer to Singleton Memory Objects

We add four new types of checked pointers to Checked C. `mm_ptr<T>`[3] extends Checked C's `ptr<T>` (§2) and thus types a pointer to a single memory object of type `T`, while `mm_array_ptr<T>` extends `array_ptr<T>` and types a pointer to an array of objects of type `T`; the latter allows pointer arithmetic and array subscripts but the former does not. `large_ptr<T>` and `large_array_ptr<T>` play similar roles, but may point to exceptionally large objects. We cover each in the coming subsections, starting in this subsection with `mm_ptr<T>`.

An `mm_ptr<T>` consists of three logical components: a raw C pointer to its referent memory object of type `T`, a key, and an offset used to compute the location of the lock. `T` can be any singleton data type: a primitive type, a `struct`, or a `union`. Although a lock is always located right before its object, and pointer arithmetic is disallowed on `mm_ptr`, it is common for programs to use the address-of operator '&' to compute the address of an inner field of a `struct` and to assign the result to a pointer. When a checked pointer points to the beginning of a `struct`, it knows the location of the lock at compile time, but when the address-taken field is not the first field of the `struct`, a checked pointer would lose track of the lock's location. We solve this problem by adding a second piece of metadata which contains the offset of the pointer from the referent's start address. The key and the offset share one single 64-bit integer.[4] We describe the bits allotted for the key and offset in Section 3.4.

Figure 2a shows an example of two `mm_ptrs` pointing to a `struct Cat`. Eight extra bytes for the lock are allocated at the beginning of the memory object (it may also need to allocate eight more bytes of padding to properly align the first byte of the memory object), and the lock is set to a unique number (42 in Figure 2a). In Figure 2a, both `mm_ptrs` have the same key value and share the same lock. Pointer `p` points to the beginning of the `struct` and thus has offset `0` while `p1` is created by an address-of expression and has offset `0x14`.

The dynamic temporal memory safety check for a pointer dereference is straightforward: the compiler inserts instructions to extract the key from the `mm_ptr`, compute the lock's address by simply subtracting the offset from the `mm_ptr`'s raw C pointer, load the lock, and check if the key matches the lock. At memory deallocation, both the referent's memory and the lock and any added padding are released, and the lock is set to a reserved value that is never used for any key. As a result, there is no need to invalidate any `mm_ptr` to the freed memory because the key of a dangling pointer will never match the lock. Additionally, although not a pointer dereference, using the `->` and `[]` operators to compute the address of an inner object of a `struct` or an array pointed to by a dangling checked pointer (e.g., `&p->obj` and `&p[i]`) will be checked and caught as a runtime error.

3.3 Pointer to Arrays

Similar to singleton memory objects, the lock of an array is also located right before the object. Since array pointers allow pointer arithmetic, they must track the location of the lock when they do not point to the start of an array. `mm_array_ptr` has the same inner structure as `mm_ptr`, and the `offset` subfield is the distance between the current raw C pointer and beginning of the array's first element. `mm_array_ptr<T>` extends Checked C's `array_ptr<T>` and can point to an array of any data type, including an array of pointers.[5] Pointer dereference checking is the same as that done for `mm_ptrs` (§3.2).

For pointer arithmetic, we follow Checked C's design [Tarditi 2021]: the result of a pointer arithmetic operation on `mm_array_ptr` can be either an `mm_ptr` or an `mm_array_ptr`, depending

[3] "mm" stands for "memory management".
[4] Our current prototype only supports 64-bit systems; Section 3.5 describes a straightforward design for 32-bit systems.
[5] More about Checked C's type system is in Li et al. [2022] and Chapter 2 and Chapter 5 of Tarditi [2021].

Table 1. Program Statistics. For **Shared Array of Ptr**, **Call** is the number of call sites to library functions with double-pointer argument(s). **Largest** and **Total** are the size of the largest shared array and total size of shared arrays of pointers, respectively.

Program	LOC	Largest struct	Largest Heap Obj	Shared Array of Ptr Call	Largest	Total	Program	LOC	Largest struct	Largest Heap Obj	Shared Array of Ptr Call	Largest	Total
500.perlbench	291 K	8 KB	25 MB	5	0	0	curl-7.79.1	122 K	8 KB	10 MB	29	200 B	200 B
502.gcc	972 K	30 KB	5 MB	0	0	0	ffmpeg-n4.1.7	1.1 M	64 MB	48 MB	46	0	0
505.mcf	3 K	648 B	289 MB	0	0	0	httpd-2.4.46	204 K	8 KB	2 KB	383	200 B	3 KB
519.lbm	1 K	216 B	204 MB	0	0	0	nginx-1.21.1	145 K	32 KB	224 KB	2	0	0
525.x264	73 K	33 KB	4 MB	2	0	0	openssl-3.0.0	403 K	15 KB	16 MB	181	280 B	1 KB
557.xz	20 K	64 KB	320 MB	0	0	0	php-7.4.9	1.2 M	1 MB	9 MB	92	48 B	48 B
538.imagick	174 K	228 KB	5 MB	0	0	0	redis-6.2.6	148 K	10 MB	3 MB	26	0	0
544.nab	16 K	720 B	1 MB	0	0	0	sqlite-3.37.0	598 K	8 KB	74 MB	12	0	0

on the type of pointer to which the result is assigned. In both cases, the resulting pointer shares the same key as the source pointer, and the offset is updated based on the arithmetic. C programs also use the address-of operator '&' to create a pointer to an element of an array. In standard C, an address-of expression with index `i` of an array pointer `p` is semantically equivalent to a pointer addition operation of `p` and `i`, i.e., `&p[i] == p + i`. However, Checked C differentiates these two types of operations. By default, an address-of expression of an `array_ptr` generates a `ptr` or an `array_ptr` with its bounds set to zero (disallowing pointer dereferences). Programmers must manually do a dynamic bounds cast [Tarditi 2021] if they want to use the result as an array pointer. Like pointer arithmetic, we also follow Checked C's design on address-of expressions on an element of an array: by default, the result of an address-of expression on an `mm_array_ptr` is an `mm_ptr`. Programmers should use a pointer arithmetic operation if they want to use the result as an `mm_array_ptr`.

3.4 Key and Offset Allotment

The 64-bit metadata field of both `mm_ptr` and `mm_array_ptr` is split into two subfields: a key and an offset, as illustrated in Figure 2a. By default, we use the highest 32 bits for the key and the remaining 32 bits for the offset, allowing memory objects of up to 4 GB in size. This should suffice for most programs. (Currently the *entire* address space of WebAssembly is 4 GB [WebAssembly 2021].) We measured the largest `struct` and heap object of all the C programs in the SPEC CPU2017 benchmark suite and eight popular large open-source C programs, totaling 5.5 million lines of code. Table 1 shows the statistics. We measured the size of `struct` using an LLVM IR pass. To collect dynamic data, we used LLVM's test-suite to run the SPEC benchmarks with the `train` dataset. For `httpd` and `Nginx`, we used `ab` [Apache Software Foundation 2022] to fetch random files ranging from 1 MB to 32 MB from a local server. The remaining six programs all have extensive built-in test cases. The largest `struct` is 64 MB from `ffmpeg-n4.1.7`, and the largest dynamic heap object is only 320 MB from `557.xz`.

We reserve integer 0 as the invalid key value. Thirty-two bits offer over 4 billion different keys which should suffice for most programs. (Wickman et al. [2021] counted the *accumulated* number of heap allocations of the SPEC CPU2006 benchmarks, and the largest is 365 million.) For security-sensitive programs, we provide a compiler option to increase the number of bits for the key to 48: we leverage the unused highest 16 bits of a raw C pointer on a 64-bit system and combine it with the 32 bits in the key metadata subfield. Forty-eight bits provide over 281 trillion keys. Consequently, the odds of key collision are extremely low. Alternatively, we can provide compiler options to control the key-offset allotment within the 64 bits of metadata. For example, if programmers are certain that a target program never allocates objects larger than 16 MB, they can configure the

compiler to use 40-bit keys (over 1 trillion keys) and 24-bit offsets. When compiled with such an option, the compiler will insert size checks before memory allocation operations to guarantee that the compiled program will not allocate objects larger than the maximum offset.

Ideally, the runtime should generate a *unique* and *random* key for each new allocation, but the performance cost can be prohibitive—e.g., x86-64's random number generation instruction `RDRAND` [Intel Corporation 2019] takes around 100 to 1,500 CPU cycles on Intel processors and up to 2,500 cycles on AMD processors [Fog 2021]. Additionally, to guarantee the uniqueness of keys, the runtime would need to maintain a set of keys in use and check whether a newly generated key is already in use, exacerbating the performance cost. We describe our key generation for x86-64 in Section 5.2. Other architectures may take different approaches.

3.5 Pointers to Exceptionally Large Objects

As Section 3.4 explains, `mm_ptr` and `mm_array_ptr` can point to a memory object up to 4 GB in size. To support the rare case when a *single* memory object is greater than 4 GB, we add two types of checked pointers (`large_ptr` and `large_array_ptr`) for exceptionally large `structs` and arrays, respectively. Similar to a few prior works [Nagarakatte et al. 2010; Patil and Fischer 1997; Xu et al. 2004], these two checked pointers both have two separate 64-bit metadata fields: one for the key and one for the address of the lock. Figure 2b shows an example. A `large_ptr` can also be used to point to a single element of an exceptionally large array pointed to by `large_array_ptr` because the distance between the element and the lock of the array can be as large as the array.

We considered using the 3-field structure of large pointers for all checked pointers. It simplifies the language design and implementation, and it provides tremendously more keys and consequently lowers the possibility of key collisions compared with the 32-bit or even 48-bit key space. It is also our choice for 32-bit systems because a single 32-bit field is insufficient to provide both high entropy for keys and to support large object sizes.

However, we choose the current design over the universal structure option mainly due to performance and memory consumption concerns. Specifically, a 3-field pointer takes 192 bits on 64-bit systems, and for the AMD64 ABI [Lu et al. 2020], function parameters and return value of structure types that contain two eight-byte integers should be passed by registers, while structures of more than two eight-byte integers should be passed via the stack memory. A compiler may opt to break a structure argument into multiple scalar arguments, but it is not guaranteed. Considering that DRAM is usually two orders of magnitude slower than registers [Gregg 2020], the universal 3-field pointer design is potentially prohibitive in terms of performance.

3.6 Lock Management

Because C uses different memory management mechanisms for the heap and the stack, our Checked C extensions manage the locks for heap (§3.6.1) and stack (§3.6.2) objects differently. Additionally, we add locks for certain address-taken global objects (§3.6.3). Next, we describe how our enhanced Checked C compiler manages locks for the three types of memory.

3.6.1 Heap. We add custom memory allocator/deallocator wrappers, dubbed `mm_alloc`/`mm_free`, to a runtime library. In addition to the requested memory for an object, an `mm_alloc` allocates extra bytes for the lock and padding for the system's alignment requirement. It generates a new key, sets the lock to the key, and returns a new checked pointer with the key and an offset of zero. `mm_free` frees the requested memory plus the lock and padding, and it invalidates the lock before it returns.

Additionally, an `mm_free` also detects invalid free and double free bugs. Before an `mm_free` calls the underlying memory deallocator `free`, they first check the offset value and signals an invalid free bug in case of a non-zero offset. They then do a key-lock check. If the key does not match the

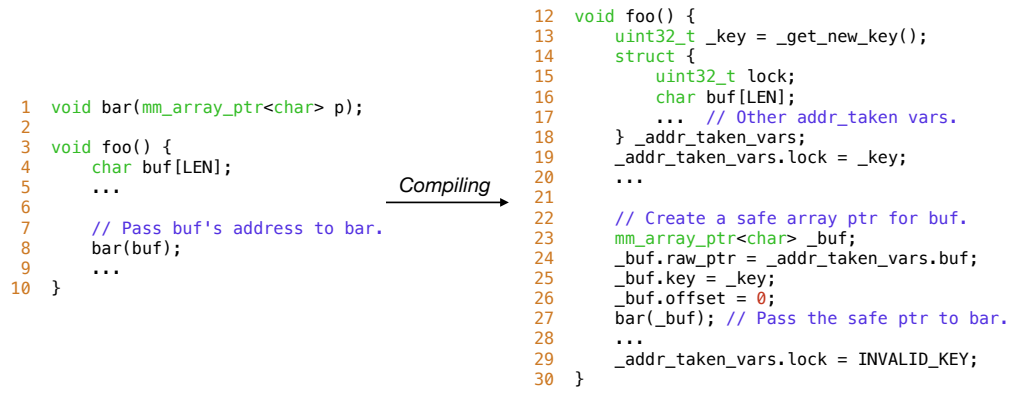


Fig. 3. Pair Address-taken Stack Variables with a Lock. The code is only for illustration purposes. The real transformation happens during LLVM IR code generation.

lock (the lock being either the reserved invalid key value or a new valid key value), a double free bug is detected. Finally, in the case when a pointer arithmetic operation overflows the offset to 0 (incidentally or maliciously), an invalid free error will be caught because chances are that the key would not match the "lock" (whatever resides before the updated pointer).

3.6.2 Stack. Most stack objects have the same lifetime as their enclosing function; however, compilers may optimize memory by reusing some stack slots for different objects. Our compiler guarantees that all address-taken stack objects allocated in the same function have the same lifetime and thus can share the same lock. Specifically, our compiler assembles all fixed-size address-taken local variables into one structure and adds a lock (plus necessary padding for alignment) to the beginning of the structure. Because each variable's offset within the structure is known statically, when the variable's address (including its sub-object if it is an aggregate) is taken and assigned to a checked pointer, the compiler can compute the pointer's offset metadata. For each variable-sized array, the compiler creates a lock for it individually. In the function prologue, the compiler inserts a call to the function in the runtime library that returns a new key; the compiler then sets all the locks in the frame to the key and sets the key metadata of every checked pointer that points to address-taken stack objects. In the function epilogue, the compiler revokes the lock(s) by setting them to the reserved invalid key value. Since a lock is a field of the `struct` that contains address-taken stack objects, the lock invalidation guarantees that the lifetime of all such `structs` lasts to the end of the function, thus preventing the compiler from reusing stack slots for different objects.

Figure 3 shows an example. The start address of a local array `buf` is passed to a function `bar` (line 8) that takes a checked array pointer argument. After compilation, `buf` is associated with a lock (line 15), and a temporary `mm_array_ptr<char>` with its key set to the lock and offset set to 0 (line 23–26) is generated for the call to `bar` (line 27).

3.6.3 Global. Global objects are never freed, and therefore, pointers to them do not suffer from use-after-free bugs (invalid frees are still possible). However, a single pointer may point to a global, stack, or heap object, depending on the execution context. For example, in Figure 3, `bar` takes a pointer to a stack object from `foo` (line 8), but another function may pass a pointer to a global object to `bar`. The compiler needs to know whether to insert a key check for such pointers. However, it is an undecidable problem to statically infer to which variables a pointer points [Hind 2001]. Our solution is to have the compiler put a lock right before each global variable whose address is taken *and* assigned to a checked pointer. We reserve one value as the lock for all global variables

because global variables are never freed. We also support directly assigning a string constant to a `mm_array_ptr<char>` because string constants are essentially static constant global variables.

We acknowledge that paring a global object with a lock and checking the validity of pointers to global objects incurs unnecessary overhead. However, this design provides a universal interface for programmers to use: a checked pointer will always be checked, and programmers do not need to consider to which type of memory it points. Our empirical experience on porting the C programs indicates that address-taken global variables are infrequent. We therefore believe our design's benefits greatly outweigh its performance and memory overheads.

External Global Objects. Address-taken `extern` global variables pose a subtle challenge. If one translation unit *declares* an `extern` global variable and assigns its address to a checked pointer while the translation unit that *defines* the variable does not assign its address to a checked pointer, the compiler will not know whether to allocate a lock for the variable. We solve this problem with a new qualifier `_checkable` and the name mangling technique that is commonly used in object-oriented programming languages such as C++. Specifically, programmers should use the `_checkable` type qualifier to label the definition (and optionally, the `extern` declaration) of an address-taken global variable. The compiler will automatically mangle the names of address-taken `extern` global declarations and the names of `_checkable` variables, and the compiler will add a lock for `_checkable` variables. An "undefined symbol" error will be raised during linking if the definition of an address-taken global variable is missing the `_checkable` keyword.

4 COMPATIBILITY WITH LEGACY C LIBRARIES

A Checked C program is rarely self-contained; it relies on legacy libraries. Such libraries' APIs (in header files) are expressed using raw pointers or bounds-safe interfaces (§2). As such, they make plain their expectations about representation: legacy and `itype`-annotated pointers have no in-place metadata. The Checked C compiler will ensure that only compatible pointers are passed to/from libraries. We provide library routines to *(un)marshal* pointers that programmers can use to satisfy compatibility requirements. We focus on the interoperability between checked code and legacy libraries in this section. We describe the challenges and our solutions specific to the interaction between checked and unchecked parts in a partially ported program in Section 7.

4.1 Type Compatibility

Recall Figure 1 from §2 which gives the bounds-safe interface for `strncpy`. Its arguments are declared with an *inter-op* type `char *p : itype(array_ptr<char>)`. With such a defintion, the Checked C compiler will force checked code to pass an `array_ptr<char>`, but unchecked code may pass a `char *`. In checked code, we can also safely call `strncpy` with an `mm_array_ptr<char>`. This is because `mm_array_ptr<char>` is *compatible* with `array_ptr<char>`.[6] That is, doing something like `x = (ptr<int>)y` where `y` is an `mm_ptr<int>` assigns only the raw pointer of `y` into `x`, skipping the key. The pointer `x` will not be temporally safe—the lock will be present in the pointed-to data, but cannot be checked; of course, legacy library code will not be doing temporal checks anyway.

Now suppose we had

```
void foo(int **p : itype(array_ptr<ptr<int>>));
```

We could *not* safely pass `foo` a pointer `q` of checked pointer type `mm_array_ptr<mm_ptr<int>>`. While we can strip off the key for `q`, that is not enough: the function `foo` will assume the array points to single-word pointers, but the caller would be passing in an array of fat pointers instead.

[6] Recall that, in the full design, all `mm*` pointers are bounds-checked.

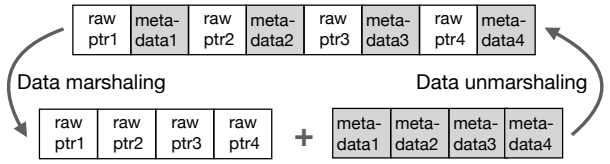


Fig. 4. Marshaling an Array of Four Fat Pointers

The compiler will prevent this mistake because it will deem types `array_ptr<ptr<int>>` and `mm_array_ptr<mm_ptr<int>>` incompatible.

A similar problem can arise when passing a `struct` with an `mm_ptr`/`mm_array_ptr` field to a library function. However, as a practical matter, this will not happen because `structs` defined in legacy libraries will not declare the use of fat pointers, even in bounds-safe interfaces; conversely, `structs` defined in applications will not be seen by libraries. We assume that programmers cannot or will not change the source code of legacy libraries, and as a result, we just need to focus on what to do in situations like the one involving `foo` above (i.e., shared arrays of fat pointers).

4.2 Marshalling

The incompatibility shown above means that, unfortunately and inevitably, metadata must be separated from some fat pointers passed to libraries. We therefore employ data marshaling [Liu et al. 2017]: to pass an array of fat pointers to a library function, the programmer must insert code that copies raw pointers within the original array into a new array of raw pointers. Our system provides data marshaling utility functions such as the one below to assist the programmer:

```
void **_marshal_ptr_array<T>(mm_array_ptr<mm_ptr<T>> p, unsigned size);
```

The function allocates a new array of raw C pointers, copies `size` raw C pointers from `p` into the new array, and returns a pointer to the new array. The returned `void **` pointer can be cast to a legacy pointer as needed. Figure 4 illustrates marshaling an array of four checked pointers.

Programmers can continue to use the original array of checked pointers after such a library call if the callee only reads the array. If the library function can write to the array, programmers have two options. First, they can continue to use the array of raw pointers returned from the marshaling procedure but at the cost of reduced temporal safety guarantees; this is possible because Checked C allows raw and checked pointers to coexist outside checked regions (§2).

Second, if programmers want to revive the checked pointers after the call, they need to write an unmarshalling procedure to reassociate the array of raw pointers and their corresponding metadata. Fortunately, this situation is uncommon; `qsort` in `libc` is the only such library function of which we are aware. We wrote a small wrapper function (16 lines) for it to call our marshalling procedure and the original `qsort` and then recover the array of checked pointers. Note that this inconvenience is shared by disjoint metadata approaches because essentially such library functions break the association between a raw pointer and its metadata and there is no easy way to recover the association both automatically and soundly. For example, CETS, a compiler-based disjoint key-lock approach [Nagarakatte et al. 2010], uses the address of a pointer as an index to locate its metadata, and therefore a `qsort` call mentioned above will invalidate a pointer's connection with its metadata. CETS writes its own version of `qsort` to solve this problem.

Performance Cost. Data marshalling could be expensive if it occurs often or must marshal large arrays. To see whether this might be the case, we analyzed the eight C programs in SPEC CPU2017 and eight large open-source C programs (totalling 5.5 million lines of code) to estimate the sizes of

shared arrays of pointers. Specifically, we instrumented programs to record the bounds of each heap object, and for each call to a library function with double-pointer argument(s), the runtime library computes the size of a possible array of pointers. For example, if a double-pointer `0x1010` is passed to a library function, and our runtime records an object ranging `0x1000`–`0x1020`, then we assume that there is a 16-byte shared array of pointers.

As Table 1 shows (see §3.4 for how we ran the programs), none of the SPEC benchmarks and only four of the eight applications use shared arrays of pointers on the heap. The largest single shared array is only 280 bytes (`openssl`); the largest total size of shared arrays is only 3 KB (`httpd`).

Programmer Effort. We estimated the effort required from programmers to add calls to marshalling/unmarshalling code manually by counting the static call sites with double-pointer argument(s). Table 1 shows that, for most programs, there is only a small number of call sites with double-pointer argument(s). The largest number is 383 from `httpd`, which has 200 K SLOC. However, this is an overestimate for the call sites that require programmers' manual intervention because many double-pointer arguments do not point to an array of pointers but only to a single pointer, e.g., `strtod` in `libc`. Our analysis pass has a whitelist of several such functions in `libc`, but there could be more. In all the 51 K lines of code that we ported for evaluation (§6 and §7), we only need to add a call to the marshalling procedure once and the `qsort` wrapper twice. In short, we believe that manual intervention for data marshaling is manageable, considering the low ratio of the number of call sites to the lines of code.

5 IMPLEMENTATION

We extended the Checked C compiler [7] (which is based on Clang and LLVM [Lattner and Adve 2004]) to support our new checked pointers. Our implementation is based on commit `2eebdf` of its `Clang` frontend[7] and commit `e5e9ba7` of its `LLVM` backend[8]. In this section, we explain the implementation of the new checked pointers (§5.1), the dynamic key-lock checking (§5.2), and the runtime library that we added for safe heap (de-)allocation and compatibility support (§5.3). We also describe the current implementation limitations for a fully thread-safe and memory-safe Checked C compiler.

5.1 Checked Pointers

We implemented `mm_ptr<T>` and `mm_array_ptr<T>` using a structure that consists of two fields: an LLVM pointer [LLVM Document 2022] for the raw C pointer and an `i64` integer for the key-offset metadata. Pointer propagation operations, e.g., assignments or pointer arithmetic, will create a new checked pointer with the source pointer's metadata (offset is updated as needed). For other regular pointer operations (such as dereferences and comparison), the raw C pointer is extracted to do the computation as what is normally done for the original C. We have not implemented the two large pointer types (§3.5) because none of our test programs use them. These two types of pointers can be implemented using a 3-field structure: an LLVM pointer for the raw pointer, an `i64` for the key, and an LLVM pointer for the address of the lock. Additionally, as Section 3.1.1 mentions, we did not implement spatial memory safety checks for the new checked pointers (more details in §5.4.2).

[7] Originally from Microsoft: https://github.com/microsoft/checkedc-clang. Now maintained by the Secure Software Development Project: https://github.com/secure-sw-dev/checkedc-llvm-project.

[8] https://github.com/microsoft/checkedc-llvm. The frontend and backend of the Checked C compiler were initially in separate repositories. They were later merged.

5.2 Dynamic Key Checks

We modified Clang's IR generator to create a function that performs a dynamic key check and to insert a call to this function before dereferencing a checked pointer (i.e., via operators `*`, `->`, and `[]`). LLVM's inlining pass will later inline the calls to the key check function to improve performance. While it would be simpler to implement the key check function in a runtime library, doing so would require link-time optimization (LTO) to inline the calls to it. LTO is undesirable in certain scenarios as it can be both time- and memory-expensive.

5.2.1 Key Generation. We reserve integer 0 as the invalid key and 1 as the key for global objects. As Section 3.4 describes, the performance overhead of generating *unique* and *random* keys for each memory allocation can be prohibitive. Another option is to omit randomness by selecting an initial key value and then updating the key predictably on each memory allocation [Nagarakatte et al. 2010]. Our current prototype takes the middle ground: it uses Intel's `RDRAND` instruction [Intel Corporation 2019] to generate a 32-bit random integer as the first key and increases the key by 1 for all subsequent requests for new keys. This creates unique keys with a degree of randomness. To improve security, our prototype could be easily enhanced to generate a new random key periodically (e.g., every 1,000 memory allocations).

5.2.2 Redundant Key Check Optimization. The initial code generation from Clang AST to LLVM IR creates many redundant key checks. A key check on pointer `p` can be safely removed if the compiler is certain that since the last check on `p`, (1) `p`'s referent is not freed, and (2) `p` is not updated to point to another object, directly by assignment or indirectly via a double-pointer to `p`. We implemented a standard local data-flow analysis to remove redundant key checks.

Our compiler invokes the redundant key check optimization pass immediately after LLVM's `mem2reg` pass [LLVM Developer Group 2022b] which promotes stack-allocated memory objects into LLVM IR SSA virtual registers early within LLVM's pass pipeline. Our current data-flow analysis implementation is very conservative. It assumes that any function call may free any heap object, and it assumes that writes through double pointers may change to which memory object any checked pointer points.

We also add two operators, `mm_checked` and `mm_array_checked`, for programmers to label a safe pointer as already-checked so as to assist the compiler for further key check optimization. The compiler will directly mark the pointer as valid in the data-flow analysis, which may result in fewer inserted key checks. Programmers can use these operators when they are certain that a checked pointer is valid but the compiler cannot prove the validity. It is particularly helpful in the case of a loop or recursive function. However, to ensure temporal memory safety, these operators may only be used outside safe regions (§2). This optimization is inspired by Checked C's `dynamic_check` operator [Tarditi 2021] which evaluates a programmer-written boolean expression and informs the compiler that a condition about a pointer's bounds is true when the compiler is unable to determine this fact by itself. Use of our two operators is also similar to when Rust programmers opt to use `unsafe` blocks to suppress security checks when dereferencing raw pointers for performance. Prior surveys [Astrauskas et al. 2020; Evans et al. 2020] show that eliding security checks to improve performance is one of the most important reasons for writing unsafe Rust code, especially for certain types of crates.

5.3 Runtime Library

We implemented the memory allocator/deallocator wrappers, i.e., `mm_alloc`/`mm_free` (§3.6.1), and the data (un)marshalling procedures (§4.2) in a small runtime library. An `mm_alloc` internally calls one of `libc`'s `malloc` family of functions and initializes the lock and a checked pointer (§3.6.1).

`mm_free` invalidates the lock and does necessary security checks (§3.6.1) in addition to calling the original `free`.

We also implemented safe versions of `libc`'s string duplication functions `strdup`/`strndup`. These two functions are very commonly used, and the returned pointer is expected to be passed to `free`.

Our runtime library also includes several wrappers for common `libc` functions such as `strchr` and `strtok`. These library functions take an argument of array pointer, locate a substring/byte, and then return a pointer to the located target. Our function wrapper takes a checked array pointer (call it `p`), calls the corresponding library function with the raw pointer of `p`, creates a new checked array pointer by adding to `p` the difference between `p`'s raw pointer and the returned raw pointer of the library function, and then returns the new checked pointer. These function wrappers improve the coverage of checked pointers because, without them, a program using a pointer returned from these `libc` functions must use the returned raw pointer instead of a checked pointer.

Our current prototype does not provide safe wrappers for `mmap` and `munmap`. Supporting these two system calls is much more complicated. When `mmap` is only used as a normal memory allocator (i.e., mapping anonymous, non-shared, read/write memory), programmers can simply use our `mm_alloc` to replace it; alternatively, we can add a wrapper to `mmap` to handle this simple case. If `mmap` is used to allocate read-only memory, the runtime would be unable to conveniently initialize and invalidate the lock as an `mm_alloc` does. We can create a wrapper that maps the memory read/write, initializes the lock, and then uses `mprotect` to make the memory read-only. Supporting other use cases (e.g., shared memory and memory-mapped files) may be possible, but their support is not straightforward, so we leave it to future work. There is only one use of `mmap`/`munmap` in `thttpd` in our evaluation's benchmarks (Table 3). We leave its return value as a raw pointer.

5.4 Limitations

5.4.1 Multithreading Support. Our work is thread-safe for multithreaded programs that are data-race free. However, our approach suffers two limitations for programs that have certain data races. First, if there is a data race between a pointer dereference and a memory deallocation on the same pointer, our approach may miss a UAF bug if the free happens in the middle of the key checking. This is a common limitation shared by all key-lock checking approaches. Second, if there is a data race between reading and updating a checked pointer, the read may see a half-updated pointer because the update is not atomic by default. These two problems can be solved by making the key-checking and pointer update atomic.

We can make key-lock checking atomic using a mutex. For checked pointer updates, we see three possible solutions for our x86-64 implementation. First, we can use a 128-bit atomic compare-and-swap instruction (`CMPXCHG16` [Intel Corporation 2021]) to update a regular checked pointer. Second, we can use XMM registers and SIMD instructions [Intel Corporation 2021] to update a regular checked pointer, i.e., moving the two 64-bit fields of a checked pointer to/from an XMM register before/after updating the checked pointer. The third option is to use a lock, which could potentially incur severe performance overhead. Atomically updating checked pointers to large memory objects (which are 192 bits in size) will require a lock. In addition, we can apply static analysis techniques (such as LOCKSMITH [Pratikakis et al. 2011]) to detect data races as an optimization for performance.

5.4.2 Unimplemented Functionality. Table 2 summarizes our current prototype's unimplemented features for a fully memory-safe Checked C compiler. Notably, we did not integrate Checked C's spatial memory safety checks with our new checked pointers. The integration does not introduce many new *design* complications because the spatial and temporal memory safety checks are semantically independent, but the *implementation* is nontrivial. We therefore opted to first build

Table 2. Unimplemented Functionality for a Fully Memory-safe Checked C Compiler

Functionality	Explanation
Fully-safe pointers	The new checked pointers (§3) do not enforce spatial memory safety checks as Checked C does.
Large pointers	The two large pointers (§3.5) are not implemented. They are not used in the evaluation.
itype	No extention to Checked C's `itype` to support the new pointers. Using pointer casts to enable calls to unchecked C functions sufficed for porting the applications we used in the evaluation.

a prototype to evaluate our temporal-safe fat pointers. For a full implementation when building upon our current prototype, we need to modify the AST-to-LLVM-IR code generation to integrate the original Checked C's spatially-safe pointers with our new checked pointers for both spatial and temporal memory safety checks. By default, the original Checked C's `Clang` assumes that pointers are lowered to an LLVM IR singleton type (an `LLVM::PointerType`, which is a 64-bit integer on 64-bit systems). However, due to the additional in-place key-offset metadata, we violate that assumption by lowering pointers to an LLVM IR struct (§5.1). We need to handle the code lowering for *all* types of expressions/statements that involve pointers, and this is a labor-intensive task due to C's extraordinarily freestyle grammar and unrestricted use of pointers.

We believe that the performance and memory overhead evaluation (§6) is valid even without a full implementation. The overhead of spatially-safe Checked C mainly comes from the dynamic bounds checks and null pointer checks. The overhead of our temporally-safe pointers comes from two sources: key-lock checking and key-offset metadata propagation. Because the causes of overhead are independent, we estimate that the overall overhead would be roughly the sum of the spatial and temporal memory safety overheads.

6 EVALUATION

We evaluated the performance and memory consumption overhead incurred by our new checked pointers for temporal memory safety. We also compared our approach with CETS [Nagarakatte et al. 2010], a disjoint key-lock checking mechanism which we believe is the most relevant related work. We describe the benchmarks used for evaluation and the reasons for choosing them in Section 6.1, experimental setup in Section 6.2, and performance and memory evaluation in Section 6.3 and Section 6.4, respectively. Finally, we report our porting experience of the benchmarks in Section 7.

6.1 Benchmarks

Our evaluation requires programs that use our new checked pointers. Consequently, we needed to port existing programs to use our new pointer types. Since such modifications must be done manually and our team had limited engineering person-effort available, porting large benchmark suites such as SPEC or large applications like `Nginx` in their entirety was impractical. Therefore, for our evaluation, we ported the Olden benchmark suite, one SPEC benchmark (`429.mcf`), three mature real-world applications/libraries, and the engine and the HTTP components of `curl` to use our new checked pointers. Table 3 provides brief descriptions of these programs and their source lines of code (LoC) computed using `cloc` [AIDanial 2022].

Olden Benchmarks and mcf. We chose the Olden benchmark suite [Luk and Mowry 1996; Rogers et al. 1995] for three reasons. First and foremost, Olden is a pointer-intensive benchmark suite that emphasizes dynamic recursive data structures such as binary trees and linked lists. *All* related work that retrofits metadata to pointers shows much higher performance and memory overhead on their pointer-intensive benchmarks than other programs [Burow et al. 2018; Farkhani et al. 2021; Lee et al. 2015; Liu et al. 2018; Nagarakatte et al. 2010; Shen and Dolan-Gavitt 2020; van der Kouwe

Table 3. Description and Statistics of Programs for Evaluation

Program	Description	LoC	Ported	Porting Effort	# of Safe Ptr	# of Safe Alloc	# of Safe Free	Input for Evaluation
Olden	Data structure benchmark	5,027	4,206	—	298	28	0	LLVM Test-suite
429.mcf	Vechicle scheduling	1,574	Full	1 day	129	4	3	SPEC ref dataset
thttpd-2.29	Lightweight HTTP server	8,360	Full	—	426	47	36	4 KB–32 MB random files
parson-2d7b3dd	JSON parser	2,783	Full	—	826	17	31	328 KB–232 MB JSON files
lzfse-1.0	Data (de-)compressor	3,383	Full	2 days	148	6	6	Silesia corpus and enwik9
curl-7.79.1	Network data transferer	122 K	31 K	16 days	1,770	267	539	Built-in tests

et al. 2017]. We believe that such benchmarks will expose the worst performance and memory overheads that our work can incur. Second, it is a relatively small benchmark suite amenable to manual porting. Third, the original Checked C paper used it for evaluation [Elliott et al. 2018]. We similarly chose `429.mcf` from SPEC CPU2006 as it is also small yet pointer intensive.

Real-world Applications. We did a survey on recent temporal memory safety violation CVEs and found that two types of programs are often vulnerable. The first are long-running programs such as web servers. This is because long-running programs need to constantly free allocated objects to prevent memory exhaustion, enabling the possibility of temporal memory safety bugs. Examples include CVE-2019-5096 and CVE-2020-1647. The second type of vulnerable program is one that parses input from untrusted sources, especially complex files such as PDFs. This is because the parsing code usually needs to frequently allocate and deallocate buffers. For example, CVE-2021-36088 is a double free bug in the JSON parsing component of a log processing application; other examples include CVE-2018-1000039 and CVE-2019-17534. We therefore looked for lightweight long-running programs and programs that process input files from untrusted sources. Table 3 lists the three applications we chose: the `thttpd` [Poskanzer 2018] HTTP server, the `parson` [Gabis 2021] JSON parser, and the `lzfse` [Apple Inc. 2017] data compressor.

We also ported portions of `curl` [Stenberg 2022]. `curl` is a command line tool for transferring data with URLs via network. It consists of an engine and a library that supports over 20 network protocols. We chose `curl` for three reasons. First, it is a ubiquitous tool that is "used daily by virtually every Internet-using human on the globe" [Stenberg 2022]. Second, there are nine temporal memory safety vulnerabilities in its list of 135 published security vulnerabilities as of February 2023, and all nine vulnerabilities were reported since 2016. [curl 2022]. Third, we would like to test our work's interoperability between the checked code and the original C code in a partially ported large program. To that end, we ported `curl`'s engine and its HTTP protocol component (its most used protocol).

6.2 Experimental Setup

Checked Benchmarks. We manually ported the benchmarks to use our new checked pointers. Specifically, we replaced all the calls to the `malloc` family of functions and `free` with their corresponding allocator and deallocator wrappers, respectively (§5.3); we also replaced all related pointers with `mm_ptr<T>` and `mm_array_ptr<T>` except those that cannot be ported mainly due to backward compatibility issues with original C code, e.g., those returned from a `libc` function call.

Compilers and System. We used the original LLVM 8.0.0 compiler, upon which our Checked C compiler is built, to compile all the unmodified benchmarks as the baseline. We compiled the ported programs with our Checked C compiler. To compare with a disjoint key-lock checking mechanism, we used the SoftBoundCETS compiler [Nagarakatte 2014; Nagarakatte et al. 2015] (CETS for short)

to compile unmodified benchmarks. The currently open-sourced stable CETS compiler is based on LLVM 3.4; we ported it to LLVM 8.0.0 and disabled its spatial memory safety component for a more accurate comparison. Unfortunately, CETS cannot compile three of the Olden benchmarks (`bh`, `em3d`, and `mst`), `mcf`, and our real-world applications correctly due to bugs in its runtime library and assertion failures in its compiler.

We compiled all programs with the standard `-O3` optimizations for all three settings (baseline, Checked C, and CETS). In addition, we used LLVM's linker `LLD` to do link-time optimization (LTO) on Olden. This is because CETS' runtime library is compiled separately from applications, and it contains many functions that can only be inlined with LTO. Thus, using LTO considerably improves the performance of programs compiled by CETS. We did not use LTO for other programs.

We conducted all the experiments on an Ubuntu 20.04.1 OS running on a machine with an Intel i7-7700 CPU (8 logical cores), 16 GB of DRAM, and 256 GB of SSD.

6.3 Execution Time Overhead

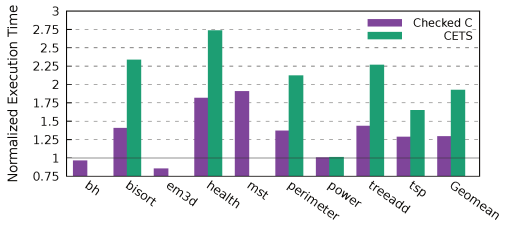
We ran the baseline and our checked version of each test program 20 times and computed their average execution time/transfer throughput/(de)compression rate.

6.3.1 Olden and mcf. We used the LLVM test suite [LLVM Developer Group 2022a] to evaluate Olden's execution time. We excluded the `voronoi` benchmark because it frequently uses an unsafe code pattern (casting an integer to a pointer) that was not supported by the original (and hence our) Checked C. We used the outputs from running the baseline programs as the expected results for our Checked C and the CETS-compiled benchmarks. The LLVM test-suite verified that both versions have the same output as the baseline. We modified the inputs to increase the input size so that all benchmarks (except for `perimeter`, `power`, `treeadd`) run for at least five seconds. The `perimeter` program runs for less than 2 seconds regardless of its input size, and `power` takes no user inputs. For `treeadd`, the input is the depth of a binary tree to build; the CETS-compiled `treeadd` crashes with a segmentation fault with an input greater than 26. We therefore chose the largest input (26) that permits the CETS-compiled `treeadd` to execute.

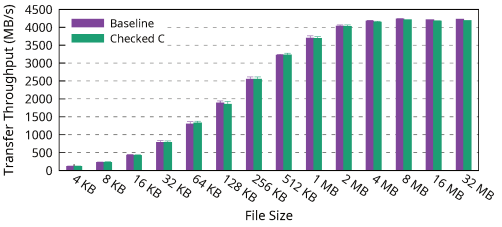
Figure 5a shows the normalized execution time of our Checked C and CETS. Overall, Checked C incurred a geometric mean of 29.1% overhead on the nine benchmarks, and CETS slowed down the six programs by 92.2%. For those six programs that can be compiled correctly by CETS, Checked C's overhead is 36.3%. The vastly improved performance of Checked C over CETS is mainly attributed to two factors. First, Checked C adds 8 bytes of metadata for each raw C pointer for temporal memory safety, while CETS adds 16 bytes [Nagarakatte et al. 2010]. As a result, CETS's pointer propagation is slower. Second, and more importantly, although using the same key-lock check mechanism, Checked C can locate the key and lock significantly faster than CETS. CETS puts a pointer's key and the address of the lock together and *dynamically* locates the metadata based on the address of the pointer. Figure 6 shows the comparison of the assembly code of the key check procedures of dereferencing a pointer, compiled by Checked C and CETS, respectively. Although having the same number of instructions, Checked C only uses one memory instruction (loading the lock), while CETS uses four. Since memory instructions are usually two orders of magnitude slower than arithmetic and bitwise instructions [Gregg 2020], it is understandable why Checked C's in-place metadata is considerably faster to use than CETS's disjoint metadata.

`mcf` is a benchmark for single-depot vehicle scheduling. It is highly pointer intensive: Nagarakatte et al. [2010] report that CETS incurred the highest overhead (175%) in all CETS' tested SPEC CPU2006 benchmarks. In contrast, our Checked C slowed `mcf` down by 64.2%.

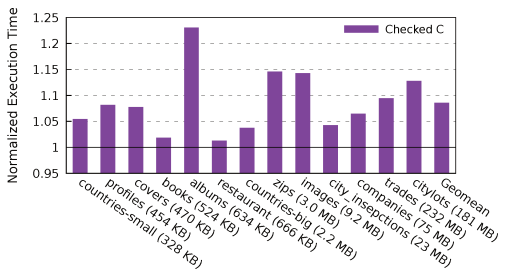
6.3.2 thttpd. We used the Apache benchmarking tool `ab` [Apache Software Foundation 2022] to fetch files from a `thttpd` server running on the same machine, and we measured the average



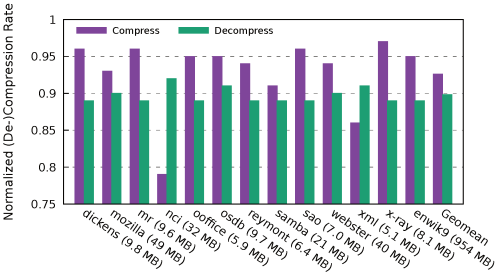
(a) Olden Execution Time Normalized to Baseline



(b) Transfer Throughput of a Local `thttpd` Server



(c) Checked C Performance on `parson`



(d) Checked C Performance on `lzfse`

Fig. 5. Checked C Performance Overhead.

```
mov %esi,%eax           # copy offset from key-offset to eax
shr $0x20,%rsi          # right-shift key-offset to get key
lea -0x8(%rdi),%rcx     # subtract space for lock from raw ptr
sub %rax,%rcx           # subtract offset to get lock's address
cmp %esi,(%rcx)         # key-lock checking
jne 202150 <foo+0x20>   # jump to routine for failed check
```

Checked C's key check for dereferencing a checked pointer argument

```
mov 0x2849(%rip),%rax  # load ptr to metadata area to rax
mov 0x20(%rax),%rsi    # load key to rsi
mov 0x28(%rax),%rax    # load address of lock to rax
mov (%rax),%rdx        # load lock to rdx
cmp %rsi,%rdx          # key-lock checking
jne 201806 <foo+0x26>  # jump to routine for failed check
```

CETS' key check for dereferencing a pointer argument

Fig. 6. Comparison of Checked C's and CETS's Key Check Procedures. The code is from a function (compiled with `-O3`) dereferencing a pointer argument. For Checked C, the raw C pointer is in `$rdi`, and the key-offset is in `$rsi`.

transfer throughput. We used files of random contents (generated by `Python`'s `random` library) ranging from 4 KB to 32 MB because the baseline server reaches full throughput for files larger than 16 MB. We configured `ab` to run with 10,000 requests at a concurrency level of 8. Note that `ab`'s concurrency level option configures how many requests to send at a time *sequentially* but not in different threads. We confirmed that `ab` did not hog all our machine's CPU resources so that the performance measurement for `thttpd` was not affected. Figure 5b shows the performance of the baseline and Checked C `thttpd`. When only considering the throughput (i.e., when ignoring noise in the experiments), Checked C incurs 0.4% overhead on average; when also considering the standard deviations, Checked C introduced no measurable overhead for each file.

6.3.3 parson. `parson` is a JSON parsing library [Gabis 2021]. It has a test suite to verify its correctness, and our Checked C version of `parson` passed all 339 test cases. However, the built-in test files are too small for performance evaluation, so we collected larger JSON files: one is a dataset for testing MongoDB [Özler 2019]; the other is a large JSON file (189.9 MB) about a city's districts [Zeiss 2012]. We used `libc`'s `clock_gettime` timer with the `CLOCK_MONOTONIC` flag to measure the execution time. Specifically, the main computation parses a JSON file into a JSON value (a structure representing the whole file), serializes the value to a string, and deserializes the

Table 4. Memory Overhead. **RSS** and **WSS** are the maximum RSS and average WSS of one run of a program, respectively. **Min**, **Max**, **Mean** is min, max, and geomean of all benchmarks (for Olden), or all input files (for thttpd, parson, and lzfse), or all test cases (for curl). mcf only has one input and thus having no min and max.

		Olden		mcf	thttpd	parson	lzfse		curl
		Checked C	CETS				Compress	Decompress	
RSS	**Min**	0	70%	—	7.1%	-0.1%	-1.4%	-12.6%	22.4%
	Max	150%	250%	—	9.9%	69.6%	0.7%	1.2%	26.9%
	Mean	72%	202%	74.9%	9.5%	11.2%	-0.2%	-1.2%	25.1%
WSS	**Min**	5%	250%	—	—	-17.2%	-8.2%	-9.2%	—
	Max	138%	1,977%	—	—	88.3%	2.8%	53.1%	—
	Mean	44%	889%	44.7%	—	10.5%	-3.5%	9.6%	—

string back to the JSON value. We excluded the smallest three JSON files from the MongoDB dataset because they take too little time (around 1 ms) to process, and thus their coefficient of variation are too high (over 20%). We also excluded binary JSON files because `parson` does not support binary inputs. Figure 5c shows Checked C `parson`'s normalized execution time. The geometric mean overhead is 8.5%, and the highest overhead is 23% on `albums`. Only four input files slowed `parson` down more than 10%, and only one slowed `parson` down more than 15%.

6.3.4 lzfse. `LZFSE` is a lossless data compression algorithm developed by Apple aiming for a high compression ratio [Apple Inc. 2017]. We chose the Silesia compression corpus [Deorowicz [n. d.]] for evaluation. Although small, the Silesia corpus covers a wide range of data types (plain text, PDF, executables, etc.) and is used by other popular compressors such as `lz4` and `zstd` for benchmarking. We also used a large (994 MB) data file `enwik9` [Mahoney 2021], which is also commonly used for benchmarking compressors. Figure 5d shows the average compression/decompression rate of Checked C over the baseline (a higher rate is better). Checked C incurred an average of 7.4% overhead for compression, with the minimum and maximum being 3.4% and 20.9%. Checked C `lzfse` slowed down by less than 5% for 5 of the 13 data files and less than 10% for 11 of them. For decompression, Checked C's average overhead is 10.2%, and the performance degradation is reasonably consistent, ranging from 7.7% to 11.2%.

6.3.5 curl. `curl` has comprehensive built-in tests. We excluded tests 1014, 1119, 1135, and 1167. Test 1014 fails with the original `curl` compiled by the vanilla `clang` compiler. The other three tests do not execute `curl` but perform sanity checks on the `curl` source code e.g., checking if file names or function names follow the naming convention. The names of and the symbols in the header files of our runtime library (§5.3) break these tests. Our partially-ported `curl` passed the remaining 1,134 tests. The test suite launches local servers to/from which `curl` sends and retrieves data. The test suite reports the execution time of each test and of all test cases combined. On average, our checked version took 299.7 s while the baseline `curl` took 298.7 s in total, incurring 0.4% overhead.

6.4 Memory Overhead

The key, lock, and necessary memory padding for alignment (§3.6.1) incurs memory overhead. We measured the memory overhead of Olden and our applications using Checked C; we also compared our overheads with CETS' memory overheads on Olden. We used the `wss` tool [Gregg 2018] to measure the maximum resident set size (RSS) (total memory consumption at a moment) and average working set size (WSS) (memory consumption in a period of time). Table 4 summarizes the results.

6.4.1 Olden and mcf. On average, Checked C consumed 72% more memory than the baseline for RSS and 44% for WSS, and CETS' RSS and WSS overhead are 202% and 889%, respectively. There are two reasons why Checked C's memory consumption is significantly less than CETS'. First, Checked C adds 8 bytes of metadata for each pointer, while CETS adds 16 bytes, and CETS uses a trie-based table for metadata lookup [Nagarakatte et al. 2010]. Second, in order to reduce the calls to allocate memory for the key and lock address metadata of each allocation, CETS preallocates two memory pools to store the metadata for the heap and the stack, which incurs unnecessary memory overhead (empty space in the pools). In contrast, our Checked C does not consume unneeded memory for metadata except for the padding (8 bytes for each lock on 64-bit systems) needed to properly align the start address of memory objects.

Nagarakatte et al. [2010] do not report CETS' memory overheads for `mcf`. Our Checked C incurs a RSS overhead of 74.9% and WSS overhead of 44.7%. As Section 6.3.1 mentions, `mcf` is a highly pointer-intensive program and thus represents an extreme case for both performance and memory overhead.

6.4.2 Applications. Checked C incurred a rather consistent RSS overhead of 7.1% to 9.9% on `thttpd`. We did not report `thttpd`'s average WSS because its WSS stays very low most of the time (when we believe the server is idle) and peaks periodically (most likely to process new connections and new data). For `parson`, the RSS and WSS overheads are 11.2% and 10.5%, respectively. Note that we observed negative memory overhead because `wss` measured the memory periodically and we used the same time interval for baseline and the checked version, it is possible that the tool missed the real max RSS of a checked run and recorded the real max RSS of the baseline.

For `lzfse`, Checked C introduces negligible RSS and WSS overhead for compression and at most 1.2% RSS overhead for decompression. The geomean of WSS overhead for decoding is 9.6%. Although `lzfse` is a CPU-intensive program and uses pointers frequently, it does not use data structures that contain many pointers: the `struct` for maintaining the encoder's state only has four checked pointers, and the one for decoding has six. Additionally, most of the buffers allocated by `lzfse` are of single-byte type (`char` or `uint8_t`), and they are usually manipulated by one checked pointer. These reasons largely explain why `lzfse` has around 9% performance overhead but very low memory overhead for most tasks.

Since most of `curl`'s test cases execute for less than one second, we measured the memory consumption of the nine tests that run longer than 5 seconds. Like `thttp`, we do not report the WSS as it remains low most of the time and peaks occasionally. We observed that `curl`'s RSS stays low and stable during the execution (5.74 MB–6.04 MB for the baseline), and our checked version incurs an overhead of 22.4% to 26.9%, with a geomean of 25.1%. We believe that a major contributor to the overhead is the `unordered_set` used to track the safe pointers (§7.3). A pointer itself plus the metadata needed by the `unordered_set` to store it can exceed 40 bytes [Lemire 2016].

7 PORTING EXPERIENCE

Checked C is a language extension, and the intention is that programmers write and maintain safe code directly. This gives them explicit control over the use of safe pointers, offering a durable approach to safety and one that affords easier long-term maintenance compared to relying solely on automated transformations, which have struggled to support all of C reliably. For example, Burow et al. [2018] reported that CETS [Nagarakatte et al. 2010] raised both false positives and false negatives on the Juliet suite.

Nevertheless, the security and maintenance benefits come at a price for *legacy* programs: programmers must port them to use our new safe pointers, as we did for our test programs. However, we believe that the porting work could be partially automated by extending 3C [Machiry et al.

2022], a tool that assists in porting legacy C to spatially-safe Checked C. 3C can automatically convert 67.9% of raw C pointers to be checked, and follow-on usage modes help programmers iteratively port the rest [Machiry et al. 2022]. As our new safe pointers are a strict extension to Checked C's type system, the core approach of 3C should be easily adaptable to our pointers.

In this section, we describe how we ported the test programs to use our new safe pointers (§7.1), the manual porting effort (§7.2), and the challenges (§7.3) we experienced in the process. We also compare using Rust versus Checked C for porting existing programs and developing new programs from scratch (§7.4).

7.1 How to Port

Recall from Section 3 that our new checked pointers follow the type system of the original Checked C [Tarditi 2021]. One of the most important type rules is that a checked pointer and a raw pointer may not be directly assigned to each other. We mainly rely on compiler errors from enforcing this rule to guide the porting process. Specifically, we started porting a program by replacing its heap memory allocation calls with calls to our `mm_alloc` (§ 5.3). The pointer returned by `mm_alloc` is a checked pointer, and there will be a type mismatch error when it is assigned to a raw C pointer. The refactored checked pointer will then be propagated to other pointers. However, we must allow casting of a checked pointer to a raw pointer (the other way around is strictly prohibited) in two scenarios: passing a checked pointer to a call to a legacy library function and to an application function that will not be ported in the immediate future. In a full implementation, we should declare a bounds-safe interface (§ 4.1) for those original C functions and let the compiler automatically do the casting (mainly stripping off the metadata). Our current compiler prototype does not implement this feature yet, and we used a simple macro to manually do the casting.

7.2 Porting Effort

We quantitatively measured the porting effort by person-days. We interspersed our time between porting programs and implementing our compiler in the early development periods because we often needed to fix compiler bugs or implement missing language features while porting. We therefore did not measure the porting effort for `Olden`, `thttpd`, and `parson`. For `curl`, we ported its engine which starts the program and does initialization work such as parsing user inputs. We omit a large file that prints out `curl`'s manual. The rest of the engine is around 12 KLoC. We also ported the main body of HTTP and the major components upon which it depends. Additionally, we ported some utility functions shared by different protocols. In total, we ported 31 KLoC of `curl`.

Table 3 shows the ported LoC and person-days we spent. Note that the LoC is not the lines of code that was changed but is the size of the source file or function we modified. For example, if we modified 5 lines of a 30 line function, we report 30 ported lines of code. A "fully-ported" function means that all pointers are checked, with two exceptions. First, pointers to global variables will not be ported as long as they are not assigned to a checked pointer. Second, pointers returned from a legacy library or unported application code will remain untouched.

On average, we ported ~1–2 KLoC per person-day, depending on the program's complexity. We expect that this rate is a slight underestimate compared to a real-world production environment, for two reasons. First, we were unfamiliar with these codebases. Developers familiar with the code should be able to port faster, e.g., starting from the least complex components. Second, the reported effort also includes time diagnosing problems that were caused by bugs in our compiler or runtime library. A more robust implementation of our toolchain would eliminate such debugging time.

Table 3 also gives the number of checked pointers, calls to `mm_alloc`, and calls to `mm_free`. We counted the number of checked pointers in `struct` definitions, local pointer variable declarations,

function parameters, and function return types. We report these numbers because they reflect an estimate on how many inline changes we need to make manually.

7.3 Challenges

As our work is built upon the original Checked C, we face similar challenges [Elliott et al. 2018] when porting a legacy C program. Additionally, we experienced new challenges due to the use of in-place metadata for temporal memory safety. All the challenges discussed in this section are specific to *partially*-ported application code. A newly-written or fully-ported Checked C program are not subject to these challenges, even directly linked with legacy libraries.

Invalid free. The first major challenge is that ported code may pass a checked pointer to legacy code (essentially removing the metadata and passing the raw pointer) which then attempts to free the pointer. While we are not aware of common C library functions that free application pointers, it is possible that an unported function will free an argument passed from a ported one. This would cause a runtime error: due to the lock and memory padding (§3.2), the raw pointer of a checked pointer does not actually point to the start address returned from a heap allocator, and freeing such a pointer is illegal. We propose two solutions. First, the runtime can record the generated checked pointers by `mm_alloc` in a set, and the compiler instruments all calls to original `free` to dynamically check if the target pointer is in the checked pointer set. Our current prototype uses this solution for `curl` implemented using C++'s `unordered_set`. We regard this solution as a "debug mode" as the performance penalty may get high for allocation-intensive programs.

The second solution is to instrument all calls to the original `malloc` so that it always allocates the space for a lock and memory padding as `mm_alloc` does, and to instrument calls to original `free` to adjust the pointer before freeing. This solution is more predictable as it takes a constant number of operations to free a pointer at the cost of allocating extra memory for each heap object. Note that a fully-ported program does not need to pay the cost of any of the two solutions.

Functions used by ported and unported code. The second major challenge is how to port functions with pointer arguments and/or return values used by both already-ported and unported code of a program. The simplest solution is to follow the original Checked C's convention by using bounds-safe interface and `itype` for the function prototype (§2). However, it will lose temporal memory safety inside the function body and/or for the returned pointer. The essential reason is that the metadata for spatial memory safety, i.e., bounds information, is *explicit* and controllable by programmers, while the metadata for temporal memory safety is *implicit* and transparent to programmers. For example, when porting a function like `strncpy` (Figure 1), the spatial bound is provided as an argument. Programmers can therefore port this function using the bounds information. Both checked and unchecked code can call it, and the compiler will enforce spatial memory safety. However, this is infeasible for our safe pointers because the unchecked part has no corresponding metadata to offer.

Programmers can choose to leave the body of such a function unported, which loses temporal memory safety, or they can port it and all its callers. However, the second option can be challenging for large programs. Our current strategy is hybrid: for commonly used small functions, we wrote a safe version for them, and we keep other shared functions unchanged. We acknowledge that this approach adds extra maintenance burden for programmers. However, it provides more memory safety, and the original function can be removed when all callers are ported.

Variadic functions. Variadic functions (e.g., `printf`) are inherently unsafe [Biswas et al. 2017]. The original Checked C [Elliott et al. 2018] disallows them in safe regions (§2); we likewise only allow

them outside safe regions. Pointer arguments from a variable argument list should be conservatively assumed to be raw pointers, even though they may actually be from checked pointers.

7.4 Discussion: Comparing to Rewriting or Developing from Scratch in Rust

An alternative to Checked C for safe systems programming is Rust [Mozilla 2023]. We do not have quantitative experimental data about the required manual effort of porting a C program to Checked C versus Rust, but we believe porting to Checked C would be much easier mainly because Checked C is an extension to C rather than a completely new language with a very different programming model. As Section 7.1 describes, porting a C program to Checked C mainly requires changing raw pointers to checked pointers, changing memory (de)-allocation calls to the corresponding safe wrappers, and (in a complete implementation) declaring bounds-safe interfaces for legacy code. In addition, the second paragraph of Section 7 notes that it should be possible to adapt 3C [Machiry et al. 2022] to support our temporally-safe pointers. In contrast, porting C to Rust is a more drastic change; thus, developing (semi-)automatic tools is significantly more challenging.

That said, we believe that, in general, for a completely new project, Rust may be a better choice due to the advancements it has over Checked C (e.g., built-in synchronization primitives [Mozilla 2023]). However, if there are really strict constraints for performance and/or memory overhead, Checked C may still be favorable. It is now a widely held impression that Rust is on par with C in performance (perhaps partially due to [Pereira et al. 2017]). However, a recent study [Zhang et al. 2023] shows that when running the exact same functionalities and algorithms, Rust takes 1.75x of execution time compared to C. One major reason why some projects rewritten in Rust indeed have competitive performance with their C counterparts is that these new projects are not a mere line-to-line porting of the original C programs but completely refactored and optimized ones. We doubt that if those old C projects go through the same level of refactoring and optimization that is done on those Rust programs, they would see a large performance gain as well.

8 RELATED WORK

In general, our work is related to memory safety and techniques of retrofitting metadata for raw C/C++ pointers, which covers an extremely rich literature that we cannot practically discuss in its entirety. We therefore focus on related work that tackles the temporal memory safety of C. There are three main directions of work: checking the validity of pointer dereferences (§8.1), invalidating dangling pointers (§8.2), and safe memory allocation (§8.3). We also briefly discuss related work that demands special architectural support for temporal memory safety (§8.4).

8.1 Dynamic Key-lock Checks

Our Checked C work falls into the category of dynamic key-lock checks.[9] We compare such approaches in this section; Table 5 summarizes the comparisons. Particularly, Section 8.1.1 discusses in detail one type of approach named *low-fat pointers* and compares our approach with ViK [Cho et al. 2022] (the latest published work before ours in this category).

UW-Pascal [Fischer and LeBlanc 1980] is the first work that proposed dynamic key-lock checks and applied it to a dialect of Pascal. Safe-C [Austin et al. 1994] adds a newly created lock to a hash table and removes the lock at memory deallocation. A pointer dereference invokes a search for the lock. The search time could be linear when the hash table grows large. In contrast, our Checked C

[9] A key or lock is also referred to as a *capability* [Austin et al. 1994; Burow et al. 2018; Patil and Fischer 1997; Xu et al. 2004], *lock-key* [Patil and Fischer 1997], or *ID* [Cho et al. 2022]. For clarity, we uniformly call the metadata for a pointer a key and the metadata for a memory object a lock.

Table 5. Comparison of Key-Lock Check Approaches. **FP**: False Positive (reporting false bugs); **FN**: False Negative (missing bugs). We assume both a program and its libraries are protected with the security mechanism in the first column; otherwise, it will suffer false negatives as long as it is directly linked with legacy library code. "**?**" means that we cannot infer the information from the paper or publicly available source code.
[*] CETS provides an API for programmers to manually handle the case of casting a pointer to an integer and then casting the integer back to a pointer; otherwise, it may incur false positives.

Key-Lock Check Work	No FP	No FN	No Stack UAF	Key Size	No Manual Intervention	No Arch Support	Supports 32-bit and 64-bit Sys.
Safe-C [Austin et al. 1994]	✓	✗	✓	32 bits	?	✓	✓
Guarding [Patil and Fischer 1997]	✓	✗	✓	32 bits	✗	✓	✓
Xu & Sekar [Xu et al. 2004]	✓	✗	✓	32 bits	?	✓	✓
CETS [Nagarakatte et al. 2010]	✓[*]	✓	✓	64 bits	✓[*]	✓	✓
CUP [Burow et al. 2018]	✓	✓	✓	31 bits	✗	✓	✗
Arm MTE [Arm Ltd. 2019b]	✓	✗	✗	4 bits	✗	✗	✗
PTAuth [Farkhani et al. 2021]	✗	✗	✗	16 bits	✗	✗	✗
ViK [Cho et al. 2022]	✓	✗	✗	10 bits	✗	✓	✗
Checked C	✓	✓	✓	32 bits	✗	✓	✓

key-lock checks always take constant time. CUP [Burow et al. 2018] repurposes a 64-bit pointer to embed a 31-bit key that works as an index into a metadata table to check the validity of the pointer. CUP requires transforming all code and will otherwise suffer severe compatibility problems due to the radical change of pointer representation. Our Checked C also changes the representation of pointers but maintains good backward compatibility (§4).

Guarding [Patil and Fischer 1997] and Xu et al. [Xu et al. 2004] pair a pointer with additional `struct`(s) of metadata and put the locks in disjoint arrays. A pointer's metadata contains a key and the address of the lock in the array. Key checks take constant time, but the dynamically managed lock array mechanism is slower than storing a lock together with its memory object, as our Checked C does. Similarly, CETS [Nagarakatte et al. 2010] stores locks in disjoint arrays and takes more memory instructions than Checked C for metadata propagation and key checks (§6.3.1).

8.1.1 Low-fat Pointers. On 64-bit systems, the higher order bits of a pointer are usually unused. For example, systems running on x86-64 and ARM64 processors only use the lower 48 bits of the 64-bit virtual address space [Arm Ltd. 2019a; Intel Corporation 2021]. A set of techniques, dubbed *low-fat pointers* by Kwon et al. [2013], utilize these unused pointer bits to store metadata.

Several systems use low-fat pointers to detect UAF errors. ARM MTE [Arm Ltd. 2019b] is a hardware extension that embeds a 4-bit tag (key) in a pointer's first byte and adds a 4-bit tag (lock) to every 16 bytes of memory; it provides new native instructions to manipulate the keys and locks. Two recent compiler-based key-lock approaches, PTAuth [Farkhani et al. 2021] and ViK [Cho et al. 2022], also leverage low-fat pointers. PTAuth computes the key using the unused bits and ARM's Pointer Authentication Code [Arm Ltd. 2019a] feature. ViK [Cho et al. 2022] uses 10 bits for keys. Like our work, PTAuth and ViK place the lock right before the referent memory object. PTAuth and ViK focus on heap UAFs while MTE and our solution handles both heap and stack.

In general, low-fat pointers have two major advantages over our fat-pointer approach. First, propagating pointer metadata induces no overhead although there is a small overhead of clearing the metadata bits when the pointer is expected to be a raw C/C++ pointer, e.g., before being passed to a legacy library function. Second, it does not suffer the compatibility problem of passing an array of pointers to legacy code (§4) unless the array itself will be modified, e.g., by `qsort` (§4.2).

```
1  mov %rax,%rbx
2  shr $63,%rbx
3  test %ebx,%ebx
4  jne .key_check
5  .key_check:
6  mov %rax,%rbx
7  shr $48,%rbx
8  mov %bx,%cx
9  and $0x1f,%rcx
10 mov %rax,%rdx
11 and $0xfffffffff000,%rdx
12 shl $7,%rcx
13 or %rcx,%rdx
14 mov (%rdx),%rsi
15 xor %rsi,%rbx
16 shl $48,%rbx
17 mov %rax,%rcx
18 and $0xffffffffffff,%rcx
19 or %rbx,%rcx
```

Fig. 7. Key-check procedure of ViK for User-space Programs. The target pointer is in `$rax`.

However, low-fat pointers suffer other compatibility issues. When a low-fat pointer is passed to a legacy library function and returned, the metadata will be lost because legacy code is unaware of the metadata. Ideally, the returned raw pointer should be refilled with correct metadata. Unfortunately, this is extremely challenging to do *automatically* as it requires a precise pointer analysis on the library code. One can opt to omit key-lock checks on pointers returned from library functions, as ViK does [Cho et al. 2022]. Our work likewise must address this challenge: we solve it with corresponding library function wrappers that recover the metadata (§5.3), and we believe low-fat pointers can adopt our solution. Additionally, a program itself may be using the higher order bits of a pointer for special purposes; low-fat pointers break such programs, but our approach does not.

Low-fat pointers have two other common drawbacks. Because the number of usable bits in a pointer is limited, not only is the key space significantly smaller than our fat-pointer solution (Table 5), but it may be difficult to scale to programs that use large memory objects. To locate the lock, PTAuth [Farkhani et al. 2021] performs a dynamic backward search starting from the raw pointer until a key-lock check passes or the search hits a preset maximum distance. PTAuth therefore suffers false positives when the distance between a pointer and its lock is greater than the *preset* threshold. ViK [Cho et al. 2022] uses a portion of the unused bits combined with a prefixed heap allocation alignment to locate the lock. By default, it does not protect objects larger than 4 KB. While 4 KB covers 98% of heap objects in Linux kernels [Cho et al. 2022], our results in Table 1 show that it is insufficient for general programs. In contrast, our Checked C enhancement places no restrictions on object size.

Key-check Comparison with ViK. An ideal performance comparison with ViK [Cho et al. 2022] would evaluate both systems on the same machine running the same benchmarks. However, this is impossible because ViK is not available. We therefore built assembly code for ViK's core key-check procedure and qualitatively compared it to our own. Figure 7 shows the assembly code of ViK's key-check procedure.[10] ViK targets only a subset of heap objects and it uses the highest-order bit to indicate whether a pointer points to a protected object, as Lines 1–4 of Figure 7 show. Lines 6–13 compute the lock's location. Lines 15–19 does key-lock checking by computing the `xor` of the metadata in the pointer and the metadata for the object, putting the result into the pointer's metadata region. If the `xor` result is 0, the "check" passes because it generates a valid C pointer; a failed "check" results in an non-canonical pointer which generates a trap if dereferenced. ViK chose this implementation to avoid a conditional branch [Cho et al. 2022], and we believe this is to improve performance.

ViK's key-check procedure uses many more instructions than ours (Figure 6) because it encodes three pieces of metadata inside a regular pointer; it thus takes more instructions to extract it. Conversely, our approach uses a separate register to store two 32-bit pieces of metadata. ViK's initial check (lines 1–4) also puts extra pressure on the branch target buffer. ViK's major performance advantage over ours is that it does not incur overhead for propagating pointer metadata. Additionally, two other factors improve ViK's performance at the cost of weaker security. First, it only protects objects smaller than a predetermined size. Second, it omits key checks on pointer dereferences

[10] The ViK paper does not provide concrete assembly code. The code in Figure 7 is based on our understanding of the paper and direct correspondence with one of ViK's authors.

that are arguably challenging to exploit—e.g., it only checks the first dereference of a pointer in a function. In contrast, our solution checks the validity of *every* pointer dereference.

8.2 Dangling Pointer Invalidation

Another way to prevent use-after-free bugs is to invalidate dangling pointers so that later dereferences will raise an exception or error. DANGNULL [Lee et al. 2015], FreeSentry [Younan 2015], DangSan [van der Kouwe et al. 2017], and pSweeper [Liu et al. 2018] are all compiler-based approaches that instrument programs to record the points-to relations of a program and invalidate dangling pointers after their referents are freed. They mainly differ in the data structures used to maintain the point-to relations. However, they all maintain metadata in disjoint data structures, which is an important reason for their high performance overhead that partially motivated our design choice for in-place metadata. MemSafe [Simpson and Barua 2013] invalidates a pointer indirectly by invalidating its bounds information at memory deallocation. A failed bounds check indicates a UAF bug. This mechanism can be enhanced with similar techniques described in Section 3.6.1 to catch double free and invalid free bugs.

One common limitation of dangling pointer invalidation systems is that a dangling pointer is usually invalidated to a reserved value of which a later dereference will crash the program or invoke an exception handling procedure, but any other non-dereference uses, such as pointer arithmetics or pointer comparison, are permitted and thus may cause incorrect program execution without raising attentions. This is not a problem for our Checked C solution as it does not modify a pointer's value after its referent is deallocated. In addition, a more severe limitation of these systems is that they transform a program's compiler IR and only track pointers *explicitly* written to memory and ignore pointers in virtual registers. Consequently, dangling pointers in registers will escape pointer invalidation. HeapExpo [Shen and Dolan-Gavitt 2020] shows that dangling pointers stored in virtual registers are common, and previous works [Lee et al. 2015; Liu et al. 2018; van der Kouwe et al. 2017; Younan 2015] failed to detect 10 of 19 real-world UAF bugs due to this omission. HeapExpo solved this problem by also tracking virtual registers albeit with additional performance and memory overhead. In contrast, Checked C does not suffer this limitation because every checked pointer will be checked regardless of where it is placed when translated to the compiler's intermediate representation.

8.3 Safe Memory Allocation and Deallocation

Use-after-free bugs become dangerous when freed memory regions get reallocated and filled with new data. One way to mitigate this problem is to lower the possibility of reusing memory. DieHard(er) [Berger and Zorn 2006; Novark and Berger 2010], FreeGuard [Silvestro et al. 2017], and Guarder [Silvestro et al. 2018] allocate heap objects to random locations to provide probabilistic protection again UAF bugs. SAFECode [Dhurjati et al. 2006], SVA [Criswell et al. 2007] and Cling [Akritidis 2010] only allow reusing memory for the same type of objects. Another class of work—Electric Fence [Perens 1993], PageHeap [Microsoft Incorporation [n. d.]], Dhurjati et al. [Dhurjati and Adve 2006], Oscar [Dang et al. 2017], and FFmalloc [Wickman et al. 2021]—goes even further to never reuse virtual memory. Garbage collection techniques have also been explored, including conservative GC for C [Boehm 1993, 2002] and reference counting [Shin et al. 2019]. One state-of-the-art work in this category, MarkUs [Ainsworth and Jones 2020], achieves low performance overhead by utilizing extra CPU cores to do live-object traversal.

In general, secure memory allocators often trade memory for security and performance. Consequently, the memory overhead can be prohibitive for certain programs. Our Checked C extension's memory overhead is mainly proportional to the number of live checked pointers. There are two other common limitations of secure allocators. First, they are for the heap and thus UAF bugs

of the stack are possible. Second, some secure allocators delay freeing memory after a `free` is called [Ainsworth and Jones 2020; Shin et al. 2019; Wickman et al. 2021], which would allow one type of UAF: dereferencing a dangling pointer to its own stale memory. This usually does not have security implications but is still an undefined behavior. Our Checked C does not suffer from these two limitations because it handles the stack and does not delay memory deallocation.

8.4 Architectural Support

Several works proposed architectural changes or utilized existing hardware extensions for temporal memory safety, such as Arm MTE [Arm Ltd. 2019b] described in Section 8.1. Watchdog [Nagarakatte et al. 2012] and WatchdogLite [Nagarakatte et al. 2014] implemented CETS [Nagarakatte et al. 2010] in hardware by adding new instructions and cache dedicated for accessing and managing pointer and object metadata. Similar to MemSafe [Simpson and Barua 2013], BOGO [Zhang et al. 2019] also catches UAF bugs by checking a pointer's bounds information. It leverages Intel MPX [Intel Corporation 2019] (now deprecated [Intel Corporation 2021]) to invalidate the bounds of dangling pointers, and dereferencing dangling pointers will be caught by MPX as bounds violations. CHERIvoke [Xia et al. 2019] and Cornucopia [Wesley Filardo et al. 2020] periodically scan the address space to revoke capabilities to freed memory, accelerated by new changes to the CHERI architecture [Woodruff et al. 2014]. Our solution, in contrast, requires no specialized hardware components or changes to current architectures.

9 CONCLUSION AND FUTURE WORK

This paper presents a new fat-pointer scheme to retrofit temporal memory safety to C. We demonstrated that when built on a solid foundation—in our case, Checked C plus key-lock checking—fat pointers can provide *full* temporal memory safety *efficiently*. We showed that, on a pointer-intensive benchmark suite, use of fat pointers significantly improves both execution time and memory overhead compared to using disjoint metadata (29% vs. 92% of performance overhead and 72% vs. 202% of memory overhead on Olden). Overhead on three full applications was also low. With findings from analyzing large open-source C programs and with our hands-on experience of porting real-world applications, we also showed that our solution does not suffer serious backward compatibility issues with legacy C code—a formerly major concern about fat pointers.

There are four main directions for future work. The first is to integrate Checked C's spatial memory safety checks into our new checked pointers to realize fully memory-safe checked pointers. Second, we will explore extending the 3C converter [Machiry et al. 2022] to support our new checked pointers. The third is to add multithreading support for the new checked pointers. Finally, there is room for improving the efficacy of the redundant key check optimization and for removing unnecessary metadata propagation.

Open Source. We open-sourced all the artifacts of this paper, including our Checked C compiler, the checked versions of the benchmarks (except `429.mcf` of SPEC as it is a commercial product), and the scripts we used for evaluation. They are available at https://doi.org/10.5281/zenodo.7511299.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their valuable feedback. We also thank David Tarditi for helping us initiate this project. In addition, we thank Sreepathi Pai, Deian Stefan, and Aravind Machiry for their insightful comments and suggestions on an early draft of the paper. This work was funded by a research gift from Microsoft Research and NSF award CNS-1955498.

REFERENCES

Jonathan Afek and Adi Sharabani. 2007. Dangling pointer: Smashing the Pointer for Fun and Profit. (2007). https://www.blackhat.com/presentations/bh-usa-07/Afek/Whitepaper/bh-usa-07-afek-WP.pdf

AIDanial. 2022. cloc: Count Lines of Code. https://github.com/AlDanial/cloc

Sam Ainsworth and Timothy M. Jones. 2020. MarkUs: Drop-in use-after-free prevention for low-level languages. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 578–591. https://doi.org/10.1109/SP40000.2020.00058

Periklis Akritidis. 2010. Cling: A Memory Allocator to Mitigate Dangling Pointers. In *Proceedings of the 19th USENIX Conference on Security* (Washington, DC) *(USENIX Security'10)*. USENIX Association, Berkeley, CA, USA, 12–12. http://dl.acm.org/citation.cfm?id=1929820.1929836

Apache Software Foundation. 2022. ab - Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html

Apple Inc. 2017. LZFSE compression library and command line tool. https://github.com/lzfse/lzfse

Arm Ltd. 2019a. *Arm Architecture Reference Manual: Armv8, for Armv8-A architecture profile.* DDI 0487E.a.

Arm Ltd. 2019b. Armv8.5-A Memory Tagging Extension. https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf

Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. 2020. How Do Programmers Use Unsafe Rust? *Proc. ACM Program. Lang.* 4, OOPSLA, Article 136 (nov 2020), 27 pages. https://doi.org/10.1145/3428204

Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. 1994. Efficient Detection of All Pointer and Array Access Errors. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation* (Orlando, Florida, USA) *(PLDI '94)*. ACM, New York, NY, USA, 290–301. https://doi.org/10.1145/178243.178446

Emery D. Berger and Benjamin G. Zorn. 2006. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. ACM, New York, NY, USA, 158–168. https://doi.org/10.1145/1133981.1134000

Priyam Biswas, Alessandro Di Federico, Scott A. Carr, Prabhu Rajasekaran, Stijn Volckaert, Yeoul Na, Michael Franz, and Mathias Payer. 2017. Venerable Variadic Vulnerabilities Vanquished. In *Proceedings of the 26th USENIX Conference on Security Symposium* (Vancouver, BC, Canada) *(SEC'17)*. USENIX Association, USA, 183–198.

Hans-Juergen Boehm. 1993. Space Efficient Conservative Garbage Collection. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* (Albuquerque, New Mexico, USA) *(PLDI '93)*. Association for Computing Machinery, New York, NY, USA, 197–206. https://doi.org/10.1145/155090.155109

Hans-J. Boehm. 2002. Bounding Space Usage of Conservative Garbage Collectors. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon) *(POPL '02)*. Association for Computing Machinery, New York, NY, USA, 93–100. https://doi.org/10.1145/503272.503282

Nathan Burow, Derrick McKee, Scott A. Carr, and Mathias Payer. 2018. CUP: Comprehensive User-Space Protection for C/C++. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security* (Incheon, Republic of Korea) *(ASIACCS '18)*. Association for Computing Machinery, New York, NY, USA, 381–392. https://doi.org/10.1145/3196494.3196540

Haehyun Cho, Jinbum Park, Adam Oest, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doupé, and Gail-Joon Ahn. 2022. ViK: Practical Mitigation of Temporal Memory Safety Violations through Object ID Inspection. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS 2022)*. Association for Computing Machinery, New York, NY, USA, 271–284. https://doi.org/10.1145/3503222.3507780

Catalin Cimpanu. 2020. Chrome: 70% of all security bugs are memory safety issues. https://www.zdnet.com/article/chrome-70-of-all-security-bugs-are-memory-safety-issues/

Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. 2007. Dependent Types for Low-level Programming. In *Proceedings of the 16th European Symposium on Programming* (Braga, Portugal) *(ESOP'07)*. Springer-Verlag, Berlin, Heidelberg, 520–535. http://dl.acm.org/citation.cfm?id=1762174.1762221

John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. 2007. Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, Washington, USA) *(SOSP '07)*. ACM, New York, NY, USA, 351–366. https://doi.org/10.1145/1294261.1294295

curl. 2022. curl security problems. https://curl.se/docs/security.html

Thurston H.Y. Dang, Petros Maniatis, and David Wagner. 2017. Oscar: A Practical Page-Permissions-Based Scheme for Thwarting Dangling Pointers. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 815–832. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/dang

Sebastian Deorowicz. [n. d.]. Silesia compression corpus. http://sun.aei.polsl.pl/~sdeor/index.php?page=silesia Accessed: 09-03-2021.

Dinakar Dhurjati and Vikram Adve. 2006. Efficiently Detecting All Dangling Pointer Uses in Production Servers. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '06)*. Washington, DC, USA, 269–280. https://doi.org/10.1109/DSN.2006.31

Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. 2006. SAFECode: Enforcing Alias Analysis for Weakly Typed Languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. Association for Computing Machinery, New York, NY, USA, 144–157. https://doi.org/10.1145/1133981.1133999

Junhan Duan, Yudi Yang, Jie Zhou, and John Criswell. 2020. Refactoring the FreeBSD Kernel with Checked C. In *2020 IEEE Secure Development (SecDev)*. 15–22. https://doi.org/10.1109/SecDev45635.2020.00018

A. S. Elliott, A. Ruef, M. Hicks, and D. Tarditi. 2018. Checked C: Making C Safe by Extension. In *2018 IEEE Cybersecurity Development (SecDev)*. 53–60. https://doi.org/10.1109/SecDev.2018.00015

Common Weaknesses Enumeration. 2020. Use After Free. https://cwe.mitre.org/data/definitions/416.html

Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. 2020. Is Rust Used Safely by Software Developers?. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 246–257. https://doi.org/10.1145/3377811.3380413

Reza Mirzazade Farkhani, Mansour Ahmadi, and Long Lu. 2021. PTAuth: Temporal Memory Safety via Robust Points-to Authentication. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association. https://www.usenix.org/conference/usenixsecurity21/presentation/mirzazade

Charles N. Fischer and Richard J. LeBlanc. 1980. The Implementation of Run-Time Diagnostics in Pascal. *IEEE Transactions on Software Engineering* SE-6, 4 (1980), 313–319.

Agner Fog. 2021. *4. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs.* Technical Report. https://www.agner.org/optimize/instruction_tables.pdf Accessed: 07-19-2021.

Krzysztof Gabis. 2021. parson: Lightweight JSON library written in C. https://github.com/kgabis/parson

Brendan Gregg. 2018. How To Measure the Working Set Size on Linux. https://www.brendangregg.com/blog/2018-01-17/measure-working-set-size.html Accessed: 10-05-2021.

Brendan Gregg. 2020. *Systems Performance: Enterprise and the Cloud, 2nd Edition.* Addison-Wesley.

Binfa Gui, Wei Song, and Jeff Huang. 2021. UAFSan: An Object-Identifier-Based Dynamic Approach for Detecting Use-after-Free Vulnerabilities. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 309–321. https://doi.org/10.1145/3460319.3464835

Michael Hind. 2001. Pointer Analysis: Haven't We Solved This Problem Yet?. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (Snowbird, Utah, USA) *(PASTE '01)*. Association for Computing Machinery, New York, NY, USA, 54–61. https://doi.org/10.1145/379605.379665

Intel Corporation 2019. *Intel 64 and IA-32 Architectures Software Developer's Manual.* Intel Corporation. Order Number: 325462-069US.

Intel Corporation 2021. *ntel® 64 and IA-32 Architectures Software Developer's Manual.* Intel Corporation. Order Number: 253665-075US.

Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference (ATEC '02)*. USENIX Association, Berkeley, CA, USA, 275–288. http://dl.acm.org/citation.cfm?id=647057.713871

Sumant Kowshik, Dinakar Dhurjati, and Vikram Adve. 2002. Ensuring Code Safety without Runtime Checks for Real-Time Control Systems. In *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems* (Grenoble, France) *(CASES '02)*. Association for Computing Machinery, New York, NY, USA, 288–297. https://doi.org/10.1145/581630.581678

Albert Kwon, Udit Dhawan, Jonathan M. Smith, Thomas F. Knight, and Andre DeHon. 2013. Low-Fat Pointers: Compact Encoding and Efficient Gate-Level Implementation of Fat Pointers for Spatial Safety and Capability-Based Security. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (Berlin, Germany) *(CCS '13)*. Association for Computing Machinery, New York, NY, USA, 721–732. https://doi.org/10.1145/2508859.2516713

Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO'04)*. IEEE Computer Society, Palo Alto, CA, 75–86. http://dl.acm.org/citation.cfm?id=977395.977673

Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. 2015. Preventing Use-after-free with Dangling Pointers Nullification. In *NDSS*.

Daniel Lemire. 2016. The memory usage of STL containers can be surprising. https://lemire.me/blog/2016/09/15/the-memory-usage-of-stl-containers-can-be-surprising/

Liyi Li, Yiyun Liu, Deena L. Postol, Leonidas Lampropoulos, David Van Horn, and Michael Hicks. 2022. A Formal Model of Checked C. In *Proceedings of the Computer Security Foundations Symposium (CSF)*.

Daiping Liu, Mingwei Zhang, and Haining Wang. 2018. A Robust and Efficient Defense against Use-after-Free Exploits via Concurrent Pointer Sweeping. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 1635–1648. https://doi.org/10.1145/3243734.3243826

Shen Liu, Gang Tan, and Trent Jaeger. 2017. PtrSplit: Supporting General Pointers in Automatic Program Partitioning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2359–2371. https://doi.org/10.1145/3133956.3134066

LLVM Developer Group. 2022a. LLVM Test Suite. https://llvm.org/docs/TestSuiteGuide.html

LLVM Developer Group. 2022b. Promote Memory to Register. https://llvm.org/docs/Passes.html#mem2reg-promote-memory-to-register

LLVM Document. 2022. llvm::PointerType Class Reference. https://llvm.org/doxygen/classllvm_1_1PointerType.html

H.J. Lu, Michael Matz, Milind Girkar, Jan Hubička, Andreas Jaeger, and Mark Mitchell. 2020. *System V Application Binary Interface AMD64 Architecture Processor Supplement.* https://gitlab.com/x86-psABIs/x86-64-ABI Version 1.0.

Chi-Keung Luk and Todd C. Mowry. 1996. Compiler-Based Prefetching for Recursive Data Structures. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages an d Operating Systems* (Cambridge, Massachusetts, USA) *(ASPLOS VII)*. Association for Computing Machinery, New York, NY, USA, 222–233. https://doi.org/10.1145/237090.237190

Aravind Machiry, John Kastner, Matt McCutchen, Aaron Eline, Kyle Headley, and Michael Hicks. 2022. C to Checked C by 3C. In *Proceedings of the ACM Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA)*. https://arxiv.org/abs/2203.13445

Matt Mahoney. 2021. Large Text Compression Benchmark. http://mattmahoney.net/dc/text.html Accessed: 09-03-2021.

Microsoft Incorporation. [n. d.]. How to use Pageheap.exe in Windows XP and Windows 2000. https://support.microsoft.com/en-gb/help/286470/how-to-use-pageheap-exe-in-windows-xp-windows-2000-and-windows-server.

Matt Miller. 2019. Trends, challenge, and shifts in software vulnerability mitigation. https://github.com/microsoft/MSRC-Security-Research/tree/master/presentations/2019_02_BlueHatIL BlueHat IL.

Mozilla. 2023. Rust Programming Language. https://www.rust-lang.org/.

Swamy Shivaganga Nagaraju, Cristian Craioveanu, Elia Florio, and Matt Miller. 2013. Software Vulnerability Exploitation Trends. Microsoft Technical Report.

Santosh Nagarakatte. 2014. SoftBoundCETS for LLVM+Clang version 34. https://github.com/santoshn/softboundcets-34 Accessed: 07-25-2021.

Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Watchdog: Hardware for Safe and Secure Manual Memory Management and Full Memory Safety. In *Proceedings of the 39th Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '12)*. IEEE Computer Society, USA, 189–200.

Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2014. WatchdogLite: Hardware-Accelerated Compiler-Based Pointer Checking. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Orlando, FL, USA) *(CGO '14)*. Association for Computing Machinery, New York, NY, USA, 175–184. https://doi.org/10.1145/2544137.2544147

Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2015. Everything You Want to Know About Pointer-Based Checking. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 32)*, Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 190–208. https://doi.org/10.4230/LIPIcs.SNAPL.2015.190

Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. ACM, New York, NY, USA, 245–258. https://doi.org/10.1145/1542476.1542504

Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2010. CETS: Compiler-Enforced Temporal Safety for C. In *Proceedings of the 2010 International Symposium on Memory Management* (Toronto, Ontario, Canada) *(ISMM '10)*. ACM, 31–40. https://doi.org/10.1145/1806651.1806657

George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: Type-Safe Retrofitting of Legacy Software. *ACM Trans. Program. Lang. Syst.* 27, 3 (May 2005), 477–526. https://doi.org/10.1145/1065887.1065892

George C. Necula, Scott McPeak, and Westley Weimer. 2002. CCured: Type-Safe Retrofitting of Legacy Code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon) *(POPL '02)*. Association for Computing Machinery, New York, NY, USA, 128–139. https://doi.org/10.1145/503272.503286

Gene Novark and Emery D. Berger. 2010. DieHarder: Securing the Heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (Chicago, Illinois, USA) *(CCS '10)*. ACM, New York, NY, USA, 573–584. https://doi.org/10.1145/1866307.1866371

Harish Patil and Charles Fischer. 1997. Low-Cost, Concurrent Checking of Pointer and Array Accesses in C Programs. *Softw. Pract. Exper.* 27, 1 (Jan. 1997), 87–110.

Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2017. Energy Efficiency across Programming Languages: How Do Energy, Time, and Memory Relate?. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering* (Vancouver, BC, Canada) *(SLE 2017)*. Association for Computing Machinery, New York, NY, USA, 256–267. https://doi.org/10.1145/3136014.3136031

Bruce Perens. 1993. Electric Fence. https://linux.die.net/man/3/efence.

Phantasmal Phantasmagoria. 2005. The Malloc Maleficarum. https://dl.packetstormsecurity.net/papers/attack/MallocMaleficarum.txt

Jef Poskanzer. 2018. thttpd - tiny/turbo/throttling HTTP server. https://acme.com/software/thttpd/

Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. 2011. LOCKSMITH: Practical Static Race Detection for C. *ACM Trans. Program. Lang. Syst.* 33, 1, Article 3 (jan 2011), 55 pages. https://doi.org/10.1145/1889997.1890000

Anne Rogers, Martin C. Carlisle, John H. Reppy, and Laurie J. Hendren. 1995. Supporting Dynamic Data Structures on Distributed-memory Machines. *ACM Trans. Program. Lang. Syst.* 17, 2 (March 1995), 233–263. https://doi.org/10.1145/201059.201065

Zekun Shen and Brendan Dolan-Gavitt. 2020. HeapExpo: Pinpointing Promoted Pointers to Prevent Use-After-Free Vulnerabilities. In *Proceedings of the 36th Annual Computer Security Applications Conference (ACSAC '20)*. Association for Computing Machinery.

Jangseop Shin, Donghyun Kwon, Yeongpil Cho Jiwon Seo, and Yunheung Paek. 2019. CRCount: Pointer Invalidation with Reference Counting to Mitigate Use-after-free in Legacy C/C++. In *NDSS*.

Sam Silvestro, Hongyu Liu, Corey Crosser, Zhiqiang Lin, and Tongping Liu. 2017. FreeGuard: A Faster Secure Heap Allocator. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2389–2403. https://doi.org/10.1145/3133956.3133957

Sam Silvestro, Hongyu Liu, Tianyi Liu, Zhiqiang Lin, and Tongping Liu. 2018. Guarder: A Tunable Secure Allocator. In *Proceedings of the 27th USENIX Conference on Security Symposium* (Baltimore, MD, USA) *(SEC'18)*. USENIX Association, USA, 117–133.

Matthew S. Simpson and Rajeev K. Barua. 2013. MemSafe: Ensuring the Spatial and Temporal Memory Safety of C at Runtime. *Softw. Pract. Exper.* 43, 1 (Jan. 2013), 93–128. https://doi.org/10.1002/spe.2105

Daniel Stenberg. 2022. cURL: A command line tool and library for transferring data with URLs. https://curl.se/

David Tarditi. 2021. *Extending C with Bounds Safety and Improved Type Safety*. Technical Report. https://github.com/microsoft/checkedc/tree/master/spec/bounds_safety Accessed: 07-14-2021.

Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. 2017. DangSan: Scalable Use-after-free Detection. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. ACM, 405–419. https://doi.org/10.1145/3064176.3064211

WebAssembly. 2021. Memory64. https://github.com/WebAssembly/memory64/blob/main/proposals/memory64/Overview.md

Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, David Chisnall, Jessica Clarke, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Alfredo Mazzinghi, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, Timothy M. Jones, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. 2020. Cornucopia: Temporal Safety for CHERI Heaps. In *2020 IEEE Symposium on Security and Privacy (SP)*. 608–625. https://doi.org/10.1109/SP40000.2020.00098

Brian Wickman, Hong Hu, Insu Yun, Daehee JangJungWon Lim, Sanidhya Kashyap, and Taesoo Kim. 2021. Preventing Use-After-Free Attacks with Fast Forward Allocation. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Vancouver, B.C. https://www.usenix.org/conference/usenixsecurity21/presentation/wickman

Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Bro oks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. 2014. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecture* (Minneapolis, Minnesota, USA) *(ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 457–468. http://dl.acm.org/citation.cfm?id=2665671.2665740

Hongyan Xia, Jonathan Woodruf, Sam Ainsworth, Nathaniel W. Filardo, Michael Roe, Alexander Richardson, Peter Rugg, Peter G. Neumann, Simon W. Moore, Robert N. M. Watson, and Timothy M. Jones. 2019. CHERIvoke: Characterising Pointer Revocation Using CHERI Capabilities for Temporal Memory Safety. In *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. ACM, New York, NY, USA, 545–557. https://doi.org/10.1145/3352460.3358288

Wei Xu, Daniel C. DuVarney, and R. Sekar. 2004. An Efficient and Backwards-compatible Transformation to Ensure Memory Safety of C Programs. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering* (Newport Beach, CA, USA) *(SIGSOFT '04/FSE-12)*. ACM, New York, NY, USA, 117–126. https://doi.org/10.1145/1029894.1029913

Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. 2015. From Collision To Exploitation: Unleashing Use-After-Free Vulnerabilities in Linux Kernel. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS '15)*. ACM, New York, NY, USA, 414–425. https://doi.org/10.1145/2810103.2813637

Yves Younan. 2015. FreeSentry: Protecting Against Use-After-Free Vulnerabilities Due to Dangling Pointers. In *NDSS*.

Mirco Zeiss. 2012. Really big json file representing san francisco's subdivision parcels. https://github.com/zemirco/sf-city-lots-json

Tong Zhang, Dongyoon Lee, and Changhee Jung. 2019. BOGO: Buy Spatial Memory Safety, Get Temporal Memory Safety (Almost) Free. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. ACM, New York, NY, USA, 631–644. https://doi.org/10.1145/3297858.3304017

Yuchen Zhang, Yunhang Zhang, Georgios Portokalidis, and Jun Xu. 2023. Towards Understanding the Runtime Performance of Rust. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 140, 6 pages. https://doi.org/10.1145/3551349.3559494

Hakan Özler. 2019. A curated list of JSON / BSON datasets from the web in order to practice / use in MongoDB. https://github.com/ozlerhakan/mongodb-json-files

Received 2022-10-28; accepted 2023-02-25