Refactoring the FreeBSD Kernel with Checked C

Junhan Duan,* Yudi Yang,* Jie Zhou, and John Criswell
Department of Computer Science
University of Rochester

***Abstract*—Most modern operating system kernels are written in C, making them vulnerable to buffer overflow and buffer over-read attacks. Microsoft has developed an extension to the C language named Checked C which provides new source language constructs that allow the compiler to prevent NULL pointer dereferences and spatial memory safety errors through static analysis and run-time check insertion. We evaluate the use of Checked C on operating system kernel code by refactoring parts of the FreeBSD kernel to use Checked C extensions. We describe our experience refactoring the code that implements system calls and UDP and IP networking. We then evaluate the refactoring effort and the performance of the refactored kernel. It took two undergraduate students approximately three months to refactor the system calls, the network packet (mbuf) utility routines, and parts of the IP and UDP processing code. Our experiments show that using Checked C incurred no performance or code size overheads.**

***Index Terms*—memory safety, safe C, FreeBSD**

I. Introduction

Most modern operating system (OS) kernels, such as Linux [14] and FreeBSD [39], are implemented in C. However, due to C's lack of memory safety and type safety, OS kernels written in C can have exploitable memory safety errors. These memory safety errors may lead to the kernel having denial of service (DoS) [3], [4], [6], [8], privilege escalation [4], [8], and arbitrary code execution [6], [7] vulnerabilities. Memory safety errors in OS kernels are extremely dangerous as kernels form the foundation of the entire software stack; successful exploitation of kernel memory safety errors can lead to memory corruption and information disclosure [16]. Such attacks can even bypass defenses such as Data Execution Prevention (DEP) [2] and Address Space Layout Randomization (ASLR) [19], [41].

To mitigate these vulnerabilities, one could rewrite the OS kernel in a safe language; examples include Singularity [30] (Sing# [26]), Mirage [38] (OCaml), Redox [24] and Tock [36] (Rust), and Biscuit [23] (Go). However, rewriting the OS kernel requires significant programmer effort. Another approach is to use automatic compiler transformations to enforce memory safety [21], [22] or Control Flow Integrity (CFI) [9]. Memory safety transformations on kernel code have high overhead (some kernel latency increase by as much as 35× [21]) while CFI approaches [20], [27], [29] fail to mitigate non-control data attacks [17].

Checked C [25], [44] is a new safe C dialect from Microsoft that extends the C language with new pointer types for which the compiler automatically performs NULL pointer checks and array bounds checks; the compiler either proves that use of the pointer is safe at compile time or inserts run-time checks to ensure safety at run-time. Checked C has low performance overhead (only 8.6% on selected benchmarks [25]), and its spatial safety checks mitigate both control-data [43], [48] and non-control data [17] attacks.

In this paper, we investigate the use of Checked C in refactoring parts of an OS kernel to be spatially memory safe. Specifically, we quantify the performance and code size overheads of using Checked C on OS kernel code and identify challenges and solutions to using Checked C on OS kernel code. We chose FreeBSD 12 stable to harden with Checked C as FreeBSD uses Clang and LLVM [35] as its default compiler; Checked C is based on Clang and LLVM [25]. Since there are millions of lines of code inside the FreeBSD kernel, we refactored kernel components that we think are part of the kernel's attack surface. Specifically, we refactored code that copies data between application memory and kernel memory i.e., code using the `copyin()` and `copyout()` functions [5], as such code, if incorrect, would allow applications to read or corrupt the OS kernel's memory. We also refactored code in the TCP/IP stack as the TCP/IP stack processes data from untrusted networks. Specifically, we modified the `mbuf` utility routines [46] which manipulate packet headers and parts of the UDP and IP processing code.

The rest of this paper is organized as follows: Section II provides background on Checked C. Section III describes the threat model. Section IV discusses how we refactored the FreeBSD kernel to use Checked C features. Section V discusses our refactoring efforts. Section VI presents the results of our evaluation on the refactored FreeBSD kernel. Section VII discusses related work, and Section VIII concludes.

II. Background on Checked C

Checked C [25], [44] is a new safe C dialect. Checked C's design distinguishes itself from previous safe-C works [18], [32], [34] in that it enables programmers to easily convert existing C code into safe Checked C code incrementally, and it maintains high backward compatibility. Currently, Checked C provides no temporal memory safety protections, so our work does not discuss temporal memory safety. In this section, we briefly introduce Checked C's key features: new types of pointers (Section II-A), checked regions (Section II-B), and bounds-safe interfaces (Section II-C).

A. New Pointer Types

Checked C extends C with three new types of pointers for spatial memory safety: `ptr<T>`, `array_ptr<T>`, and

*Junhan Duan and Yudi Yang are co-primary authors.

```
int val = 10;
char s[10] = "hello";
ptr<int> p1 = &val;
array_ptr<T> p2 : count(sizeof(s)) = s;
```

Listing 1. Checked Pointer Declaration

```
array_ptr<char> s : count(7) = "abcdef";
array_ptr<char> s1 : count(5) =
  dynamic_bounds_cast<array_ptr<char>>(s, count(5));
```

Listing 2. Checked C Bounds Casts

nt_array_ptr<T>. ptr<T> pointers are used to point to singleton objects and disallow pointer arithmetic. Listing 1 shows a simple example of defining a ptr<T> pointer. array_ptr<T> pointers are for arrays and thus allow pointer arithmetic. When initialized, it must be associated with a *bounds expression*. Listing 1 shows an example of an array_ptr<T> to an array of characters. The bounds expression count(sizeof(s)) indicates the length of the array. The array length can either be a constant or an expression. The bounds expression enables the compiler to do static compile-time bounds checking and to insert dynamic runtime checks when it cannot determine at compile-time if a pointer dereference is safe. Note that although the bounds information is semantically coupled with an array_ptr<T>, Checked C's implementation avoids real fat pointer representation (such as what Cyclone [32] does) to maintain backward compatibility. nt_array_ptr<T> is a special case of array_ptr<T> with the constraint that it can only point to NULL-terminated arrays. Besides bounds checking, Checked C also does NULL pointer checking for pointer dereference. In our work on the FreeBSD kernel, we primarily use ptr<T> and array_ptr<T>.

Checked C uses strict type checking: it disallows direct assignment between checked and unchecked pointers [25]; in addition, checked pointers with different bounds expressions cannot be assigned to each other because Checked C treats them as pointers of different types. When such assignment is necessary, Checked C provides a dynamic_bounds_cast<T>() operator to use. A dynamic bounds cast is an operation which compares the bounds information before and after the cast and determines whether the assignment is legal: the source pointer's bounds must be equal or larger than the destination pointer. Listing 2 shows a simple example of dynamic_bounds_cast<T>().

B. Checked Regions

Programmers can write code that uses both the new checked pointers and legacy unchecked pointers. This enables easy incremental conversion of legacy code. However, unchecked pointers can break the memory safety benefit ensured by checked pointers in code blocks where checked and unchecked pointers are mixed. To guarantee spatial memory safety, programmers can annotate a block of code (a source file, a function, or even a single statement) with the *_Checked* keyword [25]. In a checked region, all pointers must be checked pointers, and code in checked regions cannot call variadic functions or functions that have not been previously declared. Ruef et al. [44] prove that checked regions cannot induce spatial memory safety violations. Note that checked pointers can still be used in unchecked regions, and spatial memory safety checks are performed on those pointers.

C. Bounds Safe Interface

To achieve better interaction between checked and unchecked regions, Checked C provides *Bounds-safe interfaces* [25]. Bounds-safe interfaces allow the compiler to assign bounds expressions to a function's unchecked pointer arguments. Such interfaces serve two purposes. First, a bounds-safe interface on a function prototype allows code to pass a checked pointer to a function that takes unchecked pointers as input. In this case, the compiler performs a static bounds check (similar to assignment between checked pointers) on the checked pointer when the function is called. This allows unmodified legacy code to take checked pointers as input. Listing 3 shows an example. Second, a bounds-safe interface allows legacy code to pass an unchecked pointer to a function that takes checked pointers as input. In this case, the compiler will create a checked pointer within the body of the function that points to the same memory as the unchecked pointer that is passed into the function; the bounds of the checked pointer is specified by the bounds-safe interface. This ensures that the checked code within the function does not cause a spatial memory safety error when using the pointer passed from unmodified legacy code.

III. Threat Model

Our threat model assumes that the OS kernel is benign but that it may have exploitable spatial memory safety errors such as buffer overflows [42] and buffer overreads [47]. We assume that attackers may be legitimate users on the system that can write programs that invoke system calls or remote entities that can send arbitrary packets to the system over an IPv4 or IPv6 TCP/IP network. While the OS kernel could also have exploitable temporal memory safety errors [10], we exclude these from our threat model.

IV. Implementation

Since we started this project in November 2019, we used version 96e4565 of the Checked C compiler [1] which was, at that time, the latest version of Checked C. The compiler can compile the original FreeBSD 12 kernel.

We refactored two components of the FreeBSD kernel. First, we modified some system call code that copies data between user-space memory and kernel-space memory using copyin(), copyout() and copyinstr() [5]. If system call code passes a length argument that is too large to these functions, an application using the system call may be able to

[1] https://github.com/microsoft/checkedc-clang

```
itype_for_any(T) void* memcpy(
  void* dest : itype(array_ptr<T>) byte_count(len),
  void* src : itype(array_ptr<T>) byte_count(len),
  size_t len) : itype(array_ptr<T>) byte_count(len);

void foo(void) {
  int len_a = 5;
  int len_b = 4;
  array_ptr<char> a : count(len_a) =
      malloc<array_ptr<char>>(len_a);
  array_ptr<char> b : count(len_b) =
      malloc<array_ptr<char>>(len_b);
  memcpy<array_ptr<char>>(b, a, len_b);
}
```

Listing 3. Bounds-Safe Interface Causing Compilation Error

```
void foo(void){
  int len_a = 5;
  int len_b = 4;
  array_ptr<char> a : count(len_a) =
      malloc<array_ptr<char>>(len_a);
  array_ptr<char> b : count(len_b) =
      malloc<array_ptr<char>>(len_b);
  memcpy<array_ptr<char>>(b,
      dynamic_bounds_cast<array_ptr<char>>(a,
      count(len_b)), len_b);
}
```

Listing 4. Correct Bounds-Safe Interface with Dynamic Bounds Cast

read private kernel data or corrupt kernel memory. Second, we hardened parts of the network stack as it processes network packets from potentially untrusted sources; errors in manipulating network packets might lead to remotely exploitable code. Specifically, we modified the library routines that manipulate network packets (called `mbufs` [39]) and then modified parts of the IP and UDP protocol code to use checked pointers.

A. Kernel Support Routines

We first refactored support functions that are needed by both refactored and unmodified code. These functions must accept checked pointers and unchecked pointers as inputs and return values. We refactored most of these functions to have a bound safe interface using the features described in Section II-C.

We could not, however, refactor the memory copying and initialization functions (`memset()`, `memcpy()`, and `memmove()`) to use a bound safe interface as it caused safe code to fail to compile. We observed that the bounds safe interface requires that the bounds of a checked pointer argument be *equal* to the separate bounds argument. However, in some cases, these functions are used to copy or initialize a *subset* of the memory pointed to by a pointer. For example, the code in Listing 3 is memory-safe, but our version of the Checked C compiler fails to compile it: the length for array `a` is 5 while the amount of data to be copied is 4. The variable `a` and the formal parameter `src` have different types due to the size mismatch [44].

We must use a dynamic bounds cast (Section II-A) to create a new pointer that has its bound equal to the memory area needing to be initialized or copied. The Checked C compiler will insert a dynamic check on the cast to ensure the new bounds is no greater than the target pointer's bounds. Listing 4 adds a dynamic bounds cast to reduce the length of the destination buffer to equal the amount of data to be copied; this code compiles and executes correctly.

To ease the refactoring burden, we created macros that wrap `memset()`, `memcpy()`, and `memmove()`. These macros use a dynamic bound cast on its checked pointer input to reduce the pointer's bounds to the correct size before calling the original library functions which we modified to use a bounds safe interface. Code using checked pointers can use our wrapper macros to avoid compiler errors conveniently.

We also modified the function prototype of the kernel's `malloc()` and `free()` functions to provide a bounds safe interface so that they work with both checked and unchecked C pointers. FreeBSD has an optimization in which it inlines the code that zeros the memory object if the size of the memory object is a constant. We disabled this optimization as it is implemented using a macro wrapper around `malloc()`; Checked C does not support macros with arguments that take both checked and unchecked C pointers like bounds safe interfaces do.

```
itype_for_any(T) int copyin(const void *udaddr,
   void *kaddr:itype(array_ptr<T>) byte_count(len),
       size_t len);

itype_for_any(T) int copyout(const void
    *kaddr:itype(array_ptr<T>) byte_count(len),
       void *udaddr, size_t len);

itype_for_any(T) int copyinstr(const void *udaddr,
   void *kaddr:itype(array_ptr<T>) byte_count(len),
      size_t len, size_t *lencopied);
```

Listing 5. Refactored `copyin()`, `copyout()` and `copyinstr()`

B. Copyin and Copyout Functions

The FreeBSD kernel's `copyin()`, `copyout()` and `copyinstr()` [5], [39] functions copy data between user-space application memory and kernel memory. Each ensures that the kernel can recover if the pointer to user-space memory is invalid; however, they do not ensure that the kernel buffer is large enough for the copy operation. These functions are implemented in assembly language to maximize performance; we cannot implement them efficiently in C. Furthermore, we want both checked code and unchecked code to use a single copy of these functions to avoid unnecessary code duplication.

Listing 5 shows the basic idea of how we refactored the `copyin()`, `copyout()` and `copyinstr()` [5] function prototypes to provide a bounds safe interface. For the convenience of reading, trivial keywords (e.g. __restrict) are removed from the source code. The kernel address pointer can be either a checked or unchecked pointer. When checked pointers are passed to `copyin()`, `copyout()` and `copyinstr()` in our refactored kernel, the Checked C compiler will ensure that the kernel buffer is large enough to contain the data that

TABLE I
CURRENTLY MODIFIED SYSTEM CALLS

Functions	System Calls
`copyin()` & `copyout()`	`sys_read()`, `sys_readv()`, `sys_pread()`, `sys_preadv()`, `sys_write()`, `sys_writev()`, `sys_pwrite()`, `sys_pwritev()`, `sys_sendto()`, `sys_sendmsg()`, `sys_sendfile()`, `sys_recvfrom()`, `sys_recvmsg()`
`copyinstr()`	`sys_open()`, `sys_chmod()`, `sys_chown()`

is to be copied. Section IV-C describes how we refactored the system call code to pass checked pointers to `copyin()`, `copyout()` and `copyinstr()`.

C. System Call Modifications

Since FreeBSD has more than 100 system calls, we chose to modify a subset of them. Table I shows the system calls that we refactored.

In most cases, the kernel passes the address of local variables to the `copyin()`, `copyout()` and `copyinstr()` functions. To refactor the code, we created one new local checked pointer for every local variable passed into these functions, assigned the address of the local variable to the checked pointer, and modified calls to `copyin()`, `copyout()` and `copyinstr()` to use the new checked pointer. In some cases, we had to use `dynamic_bounds_cast()` to shrink the bounds of an existing checked pointer. For example, `uiomove_faultflag()` uses `copyin()` and `copyout()` to transfer data for scatter/gather I/O. The code loops through a list of user-space buffers and copies the data of each user-space buffer into one contiguous kernel buffer. Consequently, the size of the kernel buffer must be progressively reduced each time the loop appends new data to the kernel buffer.

D. TCP/IP Stack Modifications

An *mbuf* is a data structure that represents a single network packet [46]; it is used to represent packets for all of the network protocols supported by the FreeBSD kernel. An mbuf contains a header and a buffer area in which the packet's data is stored. A pointer in the mbuf header points to the first memory location in the mbuf occupied by data. By moving the pointer in the mbuf header, the FreeBSD kernel can quickly add and remove network protocol headers e.g., UDP and IP headers.

The FreeBSD kernel provides a set of utility routines that the network stack can use to manipulate the `mbuf` [46]. For example, `m_prepend()` adds extra space to the beginning of the data within an `mbuf` by decrementing the pointer in the `mbuf` header, allowing for efficient prepending of headers without data copying. Hence, as Figure 1 illustrates, an `mbuf` uses an individual pointer to point to the first address of the packet held within the mbuf instead of pointing to the initial address of the buffer region within the mbuf.

We refactored all the `mbuf` utility routines to be in `checked` regions so that Checked C enforces spatial memory safety. We changed all unchecked pointers to use checked

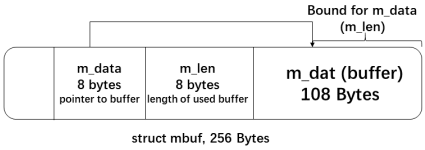

Fig. 1. Memory Layout of an mbuf Data Structure.

pointer types, modified all functions called by the routines to be in `checked` regions, and added bounds-safe interfaces for the routines. We did leave a few functions outside of `checked` regions. For example, `m_alloc()` (which allocates an `mbuf` [46]) calls a function in the kernel memory allocation library that is outside of our refactoring scope and is therefore outside a `checked` region. The memory safety of these functions requires manual inspection for now but could be proved by Checked C when we refactor more of the kernel.

Aside from the exceptions stated below, pointers and memory buffers allocated within and used by the `mbuf` utility routines have complete spatial memory safety. Additionally, the bounds-safe interfaces ensure that the `mbuf` utility routines will be memory safe when they use buffers passed in from unmodified kernel code so long as that code passes in buffers with sufficient size. Operations that manipulate these pointers, including prepending, appending, inserting, and removing, enforce spatial memory safety.

However, we could not put all `mbuf` code into checked regions. Two scenarios account for the majority of difficulty. First, some `mbuf` routines call code in other kernel subsystems that we have not yet refactored; such code must be outside checked regions. Second, some `mbuf` code uses pointer arithmetic to generate out-of-bound pointers that are used in comparisons but not dereferenced. Checked C considers such pointers to be out of bounds even though they are not used to read or write out of the bounds of their referent memory object.

After refactoring the `mbuf` code, we refactored the `ip_input()`, `ip_output()`, `udp_input()`, and `udp_output()` functions to place their code within `checked` regions. `ip_input()` and `udp_input()` process the IP and UDP headers of incoming packets, respectively, while `ip_output()` and `udp_output()` add IP and UDP headers to outgoing packets, respectively [46]. We refactored all functions called by these four functions to be in `checked` regions as well. We did not refactor the IP routing table code due to time.

V. REFACTORING EFFORT AND CHALLENGES

To evaluate the cost of our refactoring work, we measured the number of lines we added and deleted using the `git`

TABLE II
LINES OF CODE REFACTORED MEASURED IN LINES

Added	*Deleted*	*Files Modified*	*Lines in Checked Regions*
1,779	587	61	1,068

`diff --stat` command between the latest Checked C modified kernel and the original kernel source code (revision 2ade061 from the FreeBSD 12 git repository). This count includes comments and white space we added to the kernel. We also counted the number of files we modified. Our results in Table II show that we modified a small percentage of the original FreeBSD 12 kernel's 302,406 lines of code. We also counted the number of lines of code that are within checked regions; such code cannot induce a spatial memory safety violation [44]. As Table II shows, most of the code we added is within checked regions.

We also estimated our programmer effort on the project. Two undergraduate students spent about 7.3 hours a week on the project for 3 months. This includes time spent writing and debugging code, compiling the kernel, writing test suites, and reading material needed for the project. The pure coding and debugging time spent on this project is around 3 hours a week.

The incremental conversion features of Checked C and its compatibility with C eased our refactoring efforts: we were able to modify functions and add small `checked` regions instead of completely rewriting the kernel. Bounds safe interfaces allowed us to reuse existing kernel functions with both checked pointers and unchecked C pointers. However, we observed a few challenges during our refactoring efforts due to Checked C's design or implementation:

1) Checked C's bounds checking cannot always deduce pointer bounds equality after performing pointer arithmetic. For example, if `p` is a checked array pointer, the compiler cannot infer that `p` and `p + x - x` have the same bounds; instead, it generates a compile-time warning (an error when compiling the FreeBSD kernel). Programmers must insert a dynamic bounds cast operation to force the compiler to check that the two pointer expressions have the same bounds.
2) As mentioned in the last paragraph of Section IV-A, Checked C does not permit programmers to write C preprocessor macros that have bounds safe interfaces that accept both checked and unchecked pointers as inputs to the same parameter. As numerous macros are used in the kernel, we were forced to write new versions of macros that take checked pointers as inputs and to refactor existing code that uses checked pointers to use the new macros. Because of this issue, we created a new set of macros for the kernel linked list implementation with 11 additional lines.
3) We observed that the FreeBSD kernel frequently allocates local variables and copies data between application memory and these local variables using `copyin()` and `copyout()`. In the original kernel, the address of these local variables can be passed to `copyin()` and `copyout()` by using the address-of operator (&). In Checked C, a separate checked pointer variable must be created to point to the local variable, and this new checked pointer variable is passed to `copyin()` and `copyout()`. As this creates many unnecessary pointer variables, it would be convenient if Checked C enhanced the address-of operator to automatically create a checked pointer in calls to functions that can take checked pointers.

VI. Evaluation

We evaluated the performance and code size overheads of our refactored FreeBSD kernel. For performance, we evaluated the following components:

- **File System**: the bandwidth for file read.
- **System Calls**: the latency of several common system calls such as `read()`, `write()`, and `open()`.
- **Pipe**: the bandwidth and transaction latency of pipe I/O using the `read()` and `write()` system calls.
- **Unix Socket**: the bandwidth and transaction latency for Unix domain sockets using the `read()` and `write` system calls.
- **UDP**: the bandwidth and transaction latency for an IPv4 UDP socket.

We used the LMBench suite [40] to measure the performance of system calls, the file system, and pipes as well as the latency of Unix domain and IPv4 UDP sockets. The `open()` system call uses our refactored `copyinstr()` function, and the file system and networking experiments utilize our refactored `read()`, `write()`, `send()`, and `recv()` system calls. We used iPerf3 [1] to measure IPv4 and IPv6 UDP bandwidth. iPerf3 sets up a client process and server process which communicate over a network; the datagram size is configurable. iPerf3 transmits data between the client and server for 10 seconds while calculating the bandwidth once per second.

We performed our experiments on a machine with one i7-7700 Intel CPU (4 cores, 8 threads) running at 3.60 GHz. Our machine had 16 GB of DRAM and a 500 GB solid state drive (SSD). We ran each benchmark 10 times. For the network tests, we ran the server and the client on the same machine. We configured a UDP server and client using iPerf3 with a buffer size of 65,507 bytes (the maximum buffer size that iPerf3 accepts) and to report bandwidth in MB/s. We also disabled the limitation on bandwidth. We switched the client mode between IPv4 and IPv6 to measure both IPv4 and IPv6 performance.

We used version 2ade061 of the stable/12 branch of FreeBSD's online repository.[2] Our baseline kernel is the original FreeBSD kernel compiled by the Checked C compiler as we wanted to measure the overhead of using the Checked C features. We compiled both the baseline kernel and the Checked C kernel using default compiler flags.

A. Performance Overhead

1) System Calls: To understand how our changes affect system call performance, we used LMBench [40] v3.0-a9 to measure the latency of several system calls and the bandwidth of file I/O, pipe I/O, and Unix domain socket I/O on the baseline and Checked C kernels. We ran each LMBench

[2] https://github.com/freebsd/freebsd/tree/stable/12

TABLE III
FREEBSD KERNEL PERFORMANCE

Version	FreeBSD12		Checked-FreeBSD12		
Latency (ns)	*Average*	*Std Dev*	*Average*	*Std Dev*	*Overhead*
`getppid`	187.96	0.61	188.23	1.25	+0.14%
`read`	260.22	0.68	259.96	1.25	−0.10%
`write`	233.14	0.96	232.75	2.23	−0.17%
`open`	1,874.72	6.47	1,881.26	5.18	+0.34%
Pipe I/O	4,573.20	38.15	4,526.57	37.79	−1.01%
Unix socket I/O	4,921.33	64.33	4,858.62	50.76	−1.27%
Bandwidth (MB/s)	*Average*	*Std Dev*	*Average*	*Std Dev*	*Overhead*
File I/O	15,076.88	48.54	14,880.79	67.30	+1.01%
Unix socket I/O	1,677.64	6.11	1,680.41	9.81	−0.16%
Pipe I/O	10,351.54	176.35	10,281.92	66.55	+0.67%

TABLE IV
UDP BENCHMARK RESULT

Version	FreeBSD12		Checked-FreeBSD12		
Latency (ns)	*Average*	*Std Dev*	*Average*	*Std Dev*	*Overhead*
UDP Socket I/O	8,133.78	44.70	8,093.38	33.67	−0.49%
Bandwidth (MB/s)	*Average*	*Std Dev*	*Average*	*Std Dev*	*Overhead*
UDP socket I/O	5,702	5.67	5,672	16.26	+0.52%
UDP6 socket I/O	5,117	12.81	5,145	19.16	−0.54%

benchmark 10 times and calculated the average and standard deviation of the 10 runs. Time is measured in nanoseconds.

Our results in Table III show that our changes induce almost no performance overhead when accounting for standard deviation. As the Checked C compiler can optimize away some checks [25], we believe the low overhead is due to the Checked C compiler optimizing away bounds checks. As Section V states, in many calls to `copyin()` and `copyout()`, the function calling `copyin()` or `copyout()` is passing a pointer to a local variable. It is trivial for the compiler to prove that the length passed to `copyin()` or `copyout()` is identical to the length of the local variable, especially since the same expression is used as the length of buffer to which the checked pointer points and as the length of data that `copyin()` or `copyout()` should transfer.

2) UDP & IP: Since LMBench [40] v3.0-a9 does not provide a complete UDP bandwidth test, we only used LMBench to measure UDP latency. We used iPerf3 [1] to measure the bandwidth; we report the bandwidth iPerf3 measured for its server process. As the results in Table IV show, our changes incur nearly no overhead. This is expected; we only refactored the code that processes UDP and IP headers, and this code is only used once per packet when sending or receiving a UDP packet.

B. Code Size Overhead

We measured sizes of the compiled kernel binaries with the `ls -l` command. Since the Checked C compiler inserts dynamic checks into the refactored (checked) kernel, we expect the checked version to be larger than the original kernel. However, to our surprise, the original baseline kernel is 31,295,952 bytes, while the checked kernel is 31,295,448 bytes, which is 504 bytes smaller. We disassembled both kernels and checked a sample function that shrank and observed that some functions that we have not modified shrank in size in the checked kernel. When compiling the Checked C kernel, the Checked C compiler makes different instruction selection choices that cause certain instruction sequences to be shorter in the checked kernel. For example, in `sigexit_handler()`, a function we did not modify, the compiler generated a shorter `mov` instruction for the checked kernel.

We also counted the number of dynamic checks inserted by the Checked C compiler. For each dynamic check, the compiler inserts code to execute an undefined instruction `ud2` [31] and directs the control flow to it if the run-time check fails. We disassembled both kernels and found 173 additional `ud2` instructions in the checked kernel, which indicates the compiler inserted at least 173 dynamic checks.

C. Checked C Compiler vs. Original Clang Compiler

When compiling code that does not use Checked C features, the Checked C compiler should produce code that has the same performance as when it is compiled with the original Clang compiler. To verify this, we compiled the original FreeBSD kernel with both the original unmodified Clang 9.0 compiler and the Checked C compiler which is based on Clang 9.0. We then ran all our benchmarks again on both kernels; we used a different machine for this experiment, but its specifications are identical to our first machine save for a smaller 256 GB SSD. We found that both kernels have the same performance. Hence, the overhead from the modified FreeBSD kernel must be due to our kernel modifications and any dynamic checks that Checked C introduces.

VII. RELATED WORK

A. Safe C Dialects

There are a few safe C dialects that provide spatial and/or temporal memory safety. Similar to Checked C, Cyclone [32] combines static analysis and dynamic checking to catch out-of-bounds accesses; unlike Checked C, Cyclone uses “fat” pointers - integrating bounds information with raw C pointers - to allow pointer arithmetic. Consequently, it breaks backward compatibility when interacting with legacy library code. Deputy [18] extends C’s type system with dependent types to incorporate pointers’ bounds information into types; like Checked C, it maintains backwards compatibility by avoiding fat pointers but also requires programmers to annotate their code. SafeDrive [49] applied Deputy to secure Linux device drivers and to automatically restart failed device drivers. Our work is similar to the refactoring work in SafeDrive but evaluates the effects of refactoring core kernel code. Control-C [34] is a subset of C specifically targeting real-time embedded systems. It restricts certain C features (e.g., it disallows casts between any pointer type) so that all the memory safety

checking can be done at compile time. Although incurring no runtime overhead, it is not suitable for refactoring general-purpose OS kernels due to its restrictions.

B. OS Written in a Safe Programming Language

Many research and prototype OS kernels have been written in memory-safe and type-safe programming languages, including SPIN [13] (Modula-3), JX [28] and KaffeOS [11], [12] (Java), Singularity [30] (Sing# [26]), Mirage [38] (OCaml), Redox [24] and Tock [36] (Rust), and Biscuit [23] (Go). A few common obstacles hinder wide adoption of such operating systems. First, these OS kernels suffer from compatibility issues with existing low-level programs such as device drivers. Second, it takes significant programmer effort to rewrite an OS kernel. Checked C's features allowed us to add spatial memory safety to the FreeBSD kernel incrementally and cheaply while using FreeBSD's existing device drivers.

C. Automatic Compiler Instrumentation

Automatic compiler instrumentation can also mitigate memory safety vulnerabilities. Some solutions, such as KCoFI [20], the work of Ge et al. [27], kGuard [33], and FINECFI [37] restrict control flow to prevent various control-flow hijacking attacks. Unlike our work, these systems do not mitigate non-control data attacks. KENALI [45] uses CFI [9] and Data Flow Integrity [15] to prevent memory safety attacks from performing privilege escalation attacks by protecting the OS kernel's access control mechanisms, but it does not protect other kernel data structures from theft or corruption. Secure Virtual Architecture (SVA) [21], [22] provides both spatial and temporal memory safety to an OS kernel, preventing both control flow hijacking and non-control data attacks. However, SVA incurs 28.9% to 280% performance overhead on common system calls. Checked C provides spatial memory safety with low performance overhead, but only refactored code provides protection; unmodified code can still be vulnerable.

VIII. Conclusion

To the best of our knowledge, we are the first to refactor OS kernel code to use Checked C [25]. Our current experience leads us to believe that using Checked C on OS kernel code will yield efficient spatial memory safety protections against buffer overflow [42] and buffer overread [47] attacks with a modest refactoring cost. Given our positive experience, we will continue refactoring more kernel code to use checked pointers and will investigate whether the FreeBSD community is interested in accepting our changes to the kernel upstream.

Acknowledgements

We thank David Tarditi and the anonymous reviewers for their helpful comments. This work was funded by NSF Award CNS-1618213.

References

[1] ESnet. iperf3. [Online]. Available: http://software.es.net/iperf/
[2] "A detailed description of the data execution prevention (dep," in *feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003*, retrieved Dec.2018. [Online]. Available: https://support.microsoft.com/en-us/help/875352/a-detailed-description-of-the-data-execution-prevention-
[3] "CVE-2014-6416." Available from MITRE, CVE-ID CVE-2014-6416., Sep. 15 2014. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6416
[4] "CVE-2016-1887." Available from MITRE, CVE-ID CVE-2016-1887., Jan. 13 2016. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-1887
[5] *FreeBSD Handbook*, December 2018, revision 52666. [Online]. Available: https://www.freebsd.org/doc/handbook/index.html
[6] "CVE-2019-14816." Available from MITRE, CVE-ID CVE-2019-14816., Aug. 10 2019. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-14816
[7] "CVE-2019-8555." Available from MITRE, CVE-ID CVE-2019-8555., Feb. 18 2019. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-8555
[8] "CVE-2020-12653." Available from MITRE, CVE-ID CVE-2020-12653., May 5 2020. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-12653
[9] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information Systems Security*, vol. 13, pp. 4:1–4:40, November 2009. [Online]. Available: http://doi.acm.org/10.1145/1609956.1609960
[10] J. Afek and A. Sharabani, "Dangling Pointer: Smashing the Pointer for Fun and Profit," in *Black Hat USA*, 2007.
[11] G. Back and W. C. Hsieh, "The kaffeos java runtime system," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 4, p. 583–630, Jul. 2005. [Online]. Available: https://doi.org/10.1145/1075382.1075383
[12] G. Back, P. Tullmann, L. Stoller, W. C. Hsieh, and J. Lepreau, "Techniques for the design of java operating systems," in *Proceedings of 2000 USENIX Annual Technical Conference*. San Diego, CA, USA: USENIX Association, 2000.
[13] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers, "Extensibility safety and performance in the spin operating system," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 267–283. [Online]. Available: https://doi.org/10.1145/224056.224077
[14] D. P. Bovet and M. Cesati, *Understanding the LINUX Kernel*, 2nd ed. Sebastopol, CA: O'Reilly, 2002.
[15] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, ser. OSDI '06. USA: USENIX Association, 2006, p. 147–160.
[16] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek, "Linux kernel vulnerabilities: State-of-the-art defenses and open problems," in *Proceedings of the Second Asia-Pacific Workshop on Systems*, 2011, pp. 1–5.
[17] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data Attacks Are Realistic Threats," in *Proceedings of the 14th USENIX Security Symposium (SEC)*, Baltimore, MD, 2005, pp. 12–12. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251398.1251410
[18] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. C. Necula, "Dependent types for low-level programming," in *Proceedings of the 16th European Symposium on Programming*, ser. ESOP'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 520–535. [Online]. Available: http://dl.acm.org/citation.cfm?id=1762174.1762221
[19] K. Cook, "Kernel address space layout randomization," *Linux Security Summit*, 2013.
[20] J. Criswell, N. Dautenhahn, and V. Adve, "KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, ser. SP '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 292–307. [Online]. Available: https://doi.org/10.1109/SP.2014.26
[21] J. Criswell, N. Geoffray, and V. Adve, "Memory Safety for Low-level Software/Hardware Interactions," in *Proceedings of the 18th Conference on USENIX Security Symposium*, ser. SSYM'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 83–100. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855768.1855774

[22] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve, "Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '07. New York, NY, USA: ACM, 2007, pp. 351–366. [Online]. Available: http://doi.acm.org/10.1145/1294261.1294295

[23] C. Cutler, M. F. Kaashoek, and R. T. Morris, "The benefits and costs of writing a posix kernel in a high-level language," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18. USA: USENIX Association, 2018, p. 89–105.

[24] R. Developers, "The redox operating system." [Online]. Available: https://doc.redox-os.org/book/

[25] A. S. Elliott, A. Ruef, M. Hicks, and D. Tarditi, "Checked C: Making C Safe by Extension," in *2018 IEEE Cybersecurity Development (SecDev)*, Sep. 2018, pp. 53–60.

[26] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi, "Language support for fast and reliable message-based communication in singularity os," in *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, ser. EuroSys '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 177–190. [Online]. Available: https://doi.org/10.1145/1217935.1217953

[27] X. Ge, N. Talele, M. Payer, and T. Jaeger, "Fine-grained control-flow integrity for kernel software," in *Proceedings of the 1st IEEE European Symposium on Security and Privacy (EuroS&P)*, Saarbrücken, Germany, March 2016, pp. 179–194.

[28] M. Golm, M. Felser, C. Wawersich, and J. Kleinöder, "The JX operating system," in *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '02. USA: USENIX Association, 2002, p. 45–58.

[29] S. Gravani, M. Hedayati, J. Criswell, and M. L. Scott, "Iskios: Lightweight defense against kernel-level code-reuse attacks," *arXiv preprint arXiv:1903.04654*, 2019.

[30] G. C. Hunt and J. R. Larus, "Singularity: Rethinking the software stack," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 2, p. 37–49, Apr. 2007. [Online]. Available: https://doi.org/10.1145/1243418.1243424

[31] *Intel 64 and IA-32 Architectures Software Developer's Manual*, Intel Corporation, January 2019, order Number: 325462-069US.

[32] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, "Cyclone: A Safe Dialect of C," in *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '02. Berkeley, CA, USA: USENIX Association, 2002, pp. 275–288. [Online]. Available: http://dl.acm.org/citation.cfm?id=647057.713871

[33] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis, "Kguard: Lightweight kernel protection against return-to-user attacks," in *Proceedings of the 21st USENIX Conference on Security Symposium*, ser. Security'12. USA: USENIX Association, 2012, p. 39.

[34] S. Kowshik, D. Dhurjati, and V. Adve, "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*, ser. CASES '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 288–297. [Online]. Available: https://doi.org/10.1145/581630.581678

[35] C. Lattner and V. Adve, "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)*, Palo Alto, CA, 2004, pp. 75–86. [Online]. Available: http://dl.acm.org/citation.cfm?id=977395.977673

[36] A. Levy, B. Campbell, B. Ghena, D. B. Giffin, P. Pannuto, P. Dutta, and P. Levis, "Multiprogramming a 64kb computer safely and efficiently," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 234–251. [Online]. Available: https://doi.org/10.1145/3132747.3132786

[37] J. Li, X. Tong, F. Zhang, and J. Ma, "Fine-CFI: Fine-grained control-flow integrity for operating system kernels," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 6, pp. 1535–1550, 2018.

[38] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library operating systems for the cloud," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 461–472. [Online]. Available: https://doi.org/10.1145/2451116.2451167

[39] M. K. McKusick, G. V. Neville-Neil, and R. N. M. Watson, *The Design and Implementation of the FreeBSD Operating System*, 2nd ed. Addison-Wesley Professional, 2014.

[40] L. McVoy and C. Staelin, "lmbench: Portable Tools for Performance Analysis," in *Proceedings of the 7th USENIX Annual Technical Conference (ATC)*, San Diego, CA, January 1996, pp. 23–23. [Online]. Available: http://dl.acm.org/citation.cfm?id=1268299.1268322

[41] S. Nicula and R. D. Zota, "Exploiting stack-based buffer overflow using modern day techniques," *Procedia Computer Science*, vol. 160, pp. 9–14, 2019.

[42] A. One, "Smashing the Stack for Fun and Profit," *Phrack*, vol. 7, November 1996, http://www.phrack.org/issues/49/14.html [Online; accessed 11-March-2019].

[43] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-Oriented Programming: Systems, Languages, and Applications," *ACM Transactions on Information Systems Security (TISSEC)*, vol. 15, no. 1, pp. 2:1–2:34, Mar. 2012.

[44] A. Ruef, L. Lampropoulos, I. Sweet, D. Tarditi, and M. Hicks, "Achieving Safety Incrementally with Checked C," in *Principles of Security and Trust*, F. Nielson and D. Sands, Eds. Cham: Springer International Publishing, 2019, pp. 76–98.

[45] C. Song, B. Lee, K. Lu, W. R. Harris, T. Kim, and W. Lee, "Enforcing kernel security invariants with data flow integrity," in *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.

[46] W. R. Stevens and G. R. Wright, *TCP/IP Illustrated*, 1st ed. Addison-Wesley Professional, 1995, vol. 2.

[47] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter, "Breaking the Memory Secrecy Assumption," in *Proceedings of the 2nd European Workshop on System Security (EUROSEC)*, Nuremburg, Germany, 2009, pp. 1–8.

[48] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning, "On the Expressiveness of Return-into-libc Attacks," in *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection (RAID)*, Menlo Park, CA, 2011, pp. 121–141.

[49] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, E. Brewer, E. Brewer, and E. Brewer, "Safedrive: Safe and recoverable extensions using language-based techniques," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 45–60. [Online]. Available: http://dl.acm.org/citation.cfm?id=1298455.1298461