Checked C for Safety, Gradually

Draft, 20 May 2017

Andrew Ruef
University of Maryland

Archibald Samuel Elliott
University of Washington

Michael Hicks
University of Maryland

David Tarditi
Microsoft Research

ABSTRACT

This paper presents Checked C, an extension to C designed to support spatial safety, implemented in Clang and LLVM. Checked C's design is distinguished by its focus on backward-compatibility, developer usability, and enabling highly performant code. Like past approaches to a safer C, Checked C employs a form of *checked pointer* whose accesses can be statically or dynamically verified. Performance evaluation on a set of standard benchmark programs shows overheads to be relatively low. More interestingly, Checked C introduces the notion of a *checked region*. Inspired by the *blame theorem* from *gradual typing*, checked regions can be held blameless as the source of a violation, meaning it must have arisen from unchecked code. We formalize and prove this property. To assist programmers in migrating legacy code to checked C, we have implemented a rewriting tool that introduces the use of checked pointers, where safe. Experiments on several legacy programs show it to be fast and effective.

1 INTRODUCTION

Vulnerabilities that compromise *memory safety* are at the heart of many devastating attacks. Memory safety has two aspects. *Temporal safety* is ensured when memory is never used after it is freed. *Spatial safety* is ensured when any pointer dereference is always within the memory allocated to that pointer. *Buffer overruns*—a spatial safety violation—still constitute a frequent and pernicious source of vulnerability, despite their long history. During the period 2012–2016, buffer overruns were the source of 9.7% to 18.4% of CVEs reported in the NIST vulnerability database [37], with the highest numbers occurring in 2016. During that time, buffer overruns were the leading single cause of CVEs.

Spatial safety violations commonly arise when programming low-level, performance critical code in C and C++. While a type-safe language disallows such violations [49], using one is impractical when low-level control and high performance are needed. Building on research from projects such as Cyclone [27] and Deputy [57], modern languages like Rust [43] and Go [21] provide a promising balance of safety and performance, but to use them requires programmer retraining and extensive rewrites of legacy code.

As discussed in depth in Section 6, several efforts have attempted to make C programs safe. Static analysis tools [2, 6, 30] aim to find vulnerabilities pre-deployment, but may miss bugs, have trouble scaling, or emit too many alarms. Security mitigations, such as CFI [1] and DEP, can mute the impact of vulnerabilities by making them harder to exploit, but provide no guarantee; e.g., data leaks and mimicry attacks may still be possible. Several efforts have aimed to provide spatial safety by adding run-time checks; these include CCured [35], Softbound [34], and ASAN [45]. The added checks can add substantial overhead and can complicate interoperability with legacy code if pointer representations are changed. Lower overhead can be achieved by reducing safety, e.g., by checking only writes, or ignoring overruns within a memory region (e.g., from one stack variable to another, or one struct field to another). In the end, no existing approach is completely satisfying.

This paper presents a new effort towards achieving a spatially-safe C that we call *Checked C*. Checked C borrows many ideas from prior safe-C efforts but ultimately differs in that its design focuses on interoperability, developer usability, and enabling highly performant code. Checked C and legacy C can coexist, so developers are able to port legacy code incrementally. This approach does allow for defects and vulnerabilities in non-converted regions of the program. However, taking inspiration from recent work on *gradual typing* [31, 46, 53], Checked C gives developers a way to distinguish "checked" from "unchecked" regions. The former can be held blameless as the source of any safety violation, and thus software assurance attention can be focused on the latter.

Technically speaking, Checked C's design has three key features. First, all pointers in Checked C are represented as in normal C—no changes to pointer format are imposed. This eases interoperability.

Second, the legal boundaries of pointed-to memory are specified explicitly; the goal here is to enhance human readability and maintainability while supporting efficient compilation and running times. As an example, consider the following code declarations:

```
size_t dst_count;
_Array_ptr<char> dst : count(dst_count);
```

The `_Array_ptr<char>` type is a Checked C type for a bounds-checked array, and the `count` annotation indicates how the bounds should be computed. In this case `dst`'s bounds are stored in the variable `dst_count`, but other specifications, such as pointer ranges, are also possible. Checked C also has a `_Ptr<T>` type for pointers to single `T` values. Checked type information is used by the compiler to either prove that an access is safe, or else to insert a bounds check when such a proof is too difficult. Programmers can also use annotations to help the compiler safely avoid adding unnecessary checks in performance-critical code.

Finally, Checked C supports the concept of designated *checked regions* of code. Within these regions, usage of unchecked pointers is severely restricted, and casts to checked pointers are disallowed. These restrictions, along with the above-mentioned checks, ensure that execution within the checked region is spatially safe: no failure will occur within the region assuming its checked pointers are well formed (i.e., they have not been corrupted through prior execution

of unchecked code). In short, in the parlance of gradual typing, "checked code cannot be blamed" [53] for a spatial safety violation. We formalize and prove this property.

Several prior efforts have eschewed annotations, citing the programmer cost of adding them to legacy code. However, in our experience programmers have a sense of the extents and invariants of memory objects and prefer to document and enforce them, but C gives them no easy mechanism to write them down. Still, programmers probably do not have the time or fortitude to convert their entire program, all at once, to use new language features or annotations. As such, Checked C employs an automated tool to partially rewrite a legacy application to use Checked C types. We believe this approach strikes the right balance: A best-effort analysis can be applied to the whole program to assist in porting, but once ported, a program's annotations ensure efficient checking and assist readability and maintainability. The rewriter uses a global, path-insensitive unification-based algorithm to infer when variables, structure fields, function parameters, and function return values might be converted to Checked C `_Ptr<T>` and `_Array_ptr<T>` types. It automatically rewrites the program to add the former types, and points to locations for the latter, at which the programmer can convert them by hand, adding needed bounds expressions.

Contributions. This paper makes four main contributions.

First, in Section 2, we present Checked C's design and its rationale, introducing its various features by example.

Second, in Section 3, we formalize the core ideas in the design of Checked C in a core calculus called CoreChkC. We show that, in the style of gradual typing, any misbehavior can be blamed on unchecked code—either it will misbehave directly, or could induce misbehavior in checked code.

Third, as described in Section 4, we have implemented Checked C as an extension to Clang and LLVM. Since Checked C is a backwards compatible superset of C, any project that compiles today with Clang and LLVM can compile with Checked C. Using a standard benchmark suite we show that Checked C's compiler imposes little cost to either compilation time or running time. Using the standard Olden and Ptrdist benchmark suites, we find run-time slowdown is on average 8.4% and compile-time slowdown is on average 16.2%. We find that after conversion, on average only 10.5% of the benchmark code remains in unchecked regions.

Finally, as described in Section 5 we have implemented a tool to automatically convert existing C programs to Checked C programs. This tool performs a whole-program, context- and flow-insensitive analysis to identify types that can be replaced with Checked C types, and automatically rewrites them. In about 15 minutes of work the rewriter was able to replace between 23% and 42% of C pointer types with `_Ptr<T>` types in six benchmark programs, comprising more than 290KLOC.

Checked C is under active and ongoing development. It is available as open source software on the Internet at https://github.com/Microsoft/checkedc.

2 CHECKED C DESIGN

This section presents an overview of Checked C's main features, by example.

```
void read_next(int *b, int idx, _Ptr<int>out) {
  int tmp = *(b+idx);
  *out = tmp;
}
```

Figure 1: Example use of `_Ptr<T>`

2.1 Basics

The Checked C extension extends the C language with two additional *checked pointer types*: `_Ptr<T>` and `_Array_ptr<T>`.[1] The `_Ptr<T>` type indicates a pointer that is used for dereference only and has no arithmetic performed on it, while `_Array_ptr<T>` supports arithmetic with bounds declarations provided in the type. The compiler statically or dynamically confirms that checked pointers are valid when they are dereferenced. In blocks or functions specifically designated as *checked code*, it imposes stronger restrictions to uses of unchecked pointers that could corrupt checked pointers, e.g., via aliases. We would expect a Checked C program to involve a mixed of both checked and unchecked code, and a mix of checked and unchecked pointer types.

2.2 Simple pointers

Using `_Ptr<T>` is straightforward: any pointer to an object that is only referenced indirectly, without any arithmetic or array subscript operations, can be replaced with a `_Ptr<T>`. For example, one frequent idiom in C programs is an `out` parameter, used to indicate an object found or initialized during parsing. Figure 1 shows using a `_Ptr<int>` for the `out` parameter. When this function is called, the compiler will confirm that it is given a valid pointer, or null. Within the function, the compiler will insert a null check before writing to `out`. Such null checks are elided when the compiler can prove they are unnecessary.

2.3 Arrays

The `_Array_ptr<T>` type identifies a pointer to an array of values. Prior safe-C efforts sometimes involve the use of *fat pointers*, which consist both of the actual pointer and information about the bounds of pointed-to memory. Rather than changing the run-time representation of a pointer in order to support bounds checking, in Checked C the programmer associates a *bounds expression* with each `_Array_ptr<T>` type to indicate where the bounds are stored. The compiler proves that indexing an `_Array_ptr<T>` is safe or else inserts a run-time check that does so. Bounds expressions consist of non-modifying C expressions and can involve variables, parameters, and structure field members.

Figure 2 shows using `_Array_ptr<T>` with declared bounds as parameters to a function. In particular, the types of the `dst` and `src` arrays have bound expressions that refer to the function's other two respective parameters. (On struct members, bounds declarations may refer to the same struct's fields.) In the body of the function, both `src` and `dst` are accessed as expected, but potentially could result in additional compiler-inserted dynamic checks. Checks on `src` are elided because the compiler can prove that `i` $\leq$ `src_count`, the size of `src`. Checks on `dst` are elided thanks to the `_Dynamic_check`

[1] We use the C++ style syntax for programmer familiarity, and precede the names with an underscore to avoid parsing conflicts in legacy libraries.

```
void append(
  _Array_ptr<char> dst : count(dst_count),
  _Array_ptr<char> src : count(src_count),
  size_t dst_count, size_t src_count)
{
  _Dynamic_check(src_count <= dst_count);
  for (size_t i = 0; i < src_count; i++) {
    if (src[i] == '\0') {
      break;
    }
    dst[i] = src[i];
  }
}
```

Figure 2: Example use of _Array_ptr<T>

placed outside the loop. Like an `assert`, this predicate evaluates the given condition and signals a run-time error if the condition is false; unlike `assert`, this predicate is not removed unless proven redundant. Here, its existence assures the compiler that $i \leq$ `dst_count` (transitively), so no per-iteration checks are needed.

There are two other ways to specify array bounds. The first is a *range*, specified by base and bounds pointers. For example, the bounds expression on `dst` from Figure 2 could have been written `bounds(dst,dst+dst_count)` instead. The second is an alternative to `count` called `bytecount`, which can be applied to either `void*` or `_Array_ptr<void>` types. A `bytecount(n)` expression applied to a pointer `p` would be equivalent to the range `p` through `(char *)p+n`. An example of this is given at the end of this section.

Note that we can also annotate an array declaration as `_Checked`, any auto-promoted address to that array is treated as a checked `_Array_ptr<T>`. We also add a restriction that all inner dimensions of checked arrays also be checked. If inner dimensions could be unchecked, then mistakes indexing this array would not be caught. We see both of these situations in Figure 3, shortly.

2.4 Checked and unchecked regions

The safety provided by checked pointers can be thwarted by unsafe operations, such as writes to traditional pointers. For example, consider a variation of the code in Figure 1 shown below:

```
void more(int *b, int idx, _Ptr<int *>out) {
  int oldidx = idx, c;
  do {
    c = readvalue();
    b[idx++] = c;
  } while (c != 0);
  *out = b+idx-oldidx;
}
```

This function repeatedly reads an input value into `b` until a 0 is read, at which point it returns an updated `b` pointer via the checked `out` parameter. While we might expect that writing to `out` is safe, since it is a checked pointer, it will not be safe if the loop overflows `b` and in the process modifies `out` to point to invalid memory.

In a program with a mix of checked and unchecked pointers we cannot and should not expect complete safety. However, we would like to provide some assurance about which code is possibly

```
void foo(int *out) {
  _Ptr<int> ptrout;
  if (out != (int *)0) {
    ptrout = (_Ptr<int>)out; // cast OK
  } else { return; }
  _Checked {
    int b _Checked[5][5];
    for (int i = 0; i < 5; i++) {
      for (int j = 0; j < 5; j++) {
        b[i][j] = -1; // access safe
    } }
    *ptrout = b[0][0];
  }
}
```

Figure 3: Example `_Checked` block, and `_Checked` array

dangerous, i.e., whether it could be the source of a safety violation. Code review and other efforts can then focus on that code. For this purpose Checked C provides *checked regions* of code. Such code is designated specifically at the level of a file (using a pragma), a function (by annotating its prototype), or a single block (by labeling that block, similar to an `asm` block).

An example checked block is shown in Figure 3. Outside of the `_Checked`-annotated region, unchecked code casts an unchecked pointer to a checked one; this cast is a potential source of problems (if `out` was bogus) and so would not be permitted in checked code. Within the checked block, checked pointers declared inside and outside the block can be freely manipulated and the compiler performs the expected checks. The compiler also treats uses of the address-of operator `&` in a checked block as producing a checked pointer, not an unchecked one. When doing this to a `struct` field, the bounds are defined as the extent of that field.

In general, within a checked region both null and bounds checks on checked pointers are employed as usual, but additional restrictions are also imposed. In particular, explicit casts to checked pointer types are disallowed, as are reads from and writes to unchecked pointers. Checked regions may neither use varargs nor K&R-style prototypes. All of these restrictions are meant to ensure that the entire execution of a checked region is *spatially safe*. This means that assuming checked pointers have been constructed properly (in particular, they have not been corrupted due to the execution of unchecked code prior to entering the checked region), no safety violations will occur due to dereferencing a pointer into illegal memory. Section 3 makes this guarantee precise, and proves that it holds.[2]

Checked C also permits ascribing checked types to unchecked functions. This is particularly useful for checking interactions with legacy libraries. As an example, the type we give to the `fwrite` standard library function is shown in Figure 4. The first argument to the function is the target buffer whose size (in bytes) is given by the second and third arguments. The final argument is a `FILE` pointer whose type depends on whether it is being called from

[2]To be precise, Checked C's implementation goes further than what is strictly needed for safety and forbids the use unchecked pointers *anywhere* in checked blocks, even when they are not dereferenced. This is intended to encourage programmers to make all of a region checked.

```
size_t fwrite(
  const void * pointer : byte_count(size*nmemb),
  size_t size, size_t nmemb,
  FILE * stream : itype(_Ptr<FILE>));
```

Figure 4: Standard library checked interface

checked or unchecked code. For the latter, the type is given by the `itype` annotation, indicating it is expected to be a checked pointer. For the former, it is the "normal" type of the argument.

2.5 Restrictions and Limitations

Checked C's design is a work in progress and currently imposes several restrictions that we hope to relax in the near future.

First, to ensure that checked pointers are valid by construction, we require that checked pointer variables be initialized when they are declared. In addition, heap-allocated memory that contains checked pointers (like a struct or array of checked pointers) must use `calloc` so that heap-resident pointers are initialized. We plan to employ something akin to Java's *definite initialization* analysis to relax this requirement, at least somewhat.

Second, `_Array_ptr<T>` values can be dereferenced following essentially arbitrary arithmetic; e.g., if `x` is an `_Array_ptr<int>` we could dereference it via `*(x+y-n+1)` and the compiler will insert any needed checks to ensure the access is legal. However, *updates* to `_Array_ptr<T>` values are currently more limited. For example, we might like to replace the loop in Figure 2 with the following:

```
for (size_t i = 0; i < src_count; i++) {
  if (*src == '\0') {
    break;
  }
  *dst = *src;
  src++; dst++;
}
```

The problem is that the bounds declared for `src` are tantamount to the range (`src`,`src+src_count`), which would mean that updating `src` to `src+1` would invalidate them, as the upper bound would be off by one. In the meantime, this sort of arithmetic would be allowed by assigning `src` and `dst` to temporary variables; updating these variables would be OK because the bounds would be in terms of `src` and `dst`, which would remain invariant. We are working to support flow-sensitive bounds so that, in this case, the update `src++` would update the bounds to (`src`,`src+src_count-i`).

Third, Checked C lacks support for checked, zero-terminated arrays. Ideally we would like the bounds to be specified as the (dynamic) location of a zero terminator. For now, the programmer must alias the pointer and use `strlen` to compute a bound.

Finally, some elements of our static analysis for confirming safe usage are designed but not fully implemented or tested. We elaborate on these in Section 4.

3 FORMALISM: CORECHKC

This section presents a formal language CoreChkC that models the essence of Checked C. The language is designed to be simple but nevertheless highlight Checked C's key features: checked pointers; checked code blocks, which are prevented from using unchecked

Mode	m	::=	$c \mid u$
Word types	τ	::=	$\texttt{int} \mid \texttt{ptr}^m\omega$
Types	ω	::=	$\tau \mid \texttt{struct}\ T \mid \texttt{array}\ n\ \tau$
Expressions	e	::=	$n^\tau \mid x \mid \texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2$
		|	$\texttt{malloc@}\omega \mid (\tau)e$
		|	$e_1 + e_2 \mid \&e{\rightarrow}f$
		|	$*e \mid *e_1 = e_2 \mid \texttt{unchecked}\ e$
Structdefs	D	$\in$	$T \rightharpoonup fs$

Figure 5: CoreChkC Syntax

pointers and certain casts; and unchecked code blocks, which may manipulate pointers as they wish. After presenting the syntax, semantics, and type system of CoreChkC, we state and prove its key guarantee: Checked code preserves well-typing, which ensures well-defined behavior, while any observed undefined behavior can be blamed on unchecked code.

3.1 Syntax

The syntax of CoreChkC is presented in Figure 5. Types τ classify word-sized objects while types ω also include multi-word objects. The type $\texttt{ptr}^m\omega$ types a pointer, where m identifies its *mode*: mode c identifies a Checked C safe pointer, while mode u represents an unchecked pointer. In other words $\texttt{ptr}^c\tau$ is a checked pointer type `_Ptr<`τ`>` while $\texttt{ptr}^u\tau$ is an unchecked pointer type $\tau*$. Multi-word types ω include `struct` records, and arrays of type τ having size n, i.e., $\texttt{ptr}^c\texttt{array}\ n\ \tau$ represents a checked array pointer type `_Array_ptr<`τ`>` with bounds n. We assume `structs` are defined separately in a map D from struct names to their constituent field definitions.

Programs are represented as expressions e; we have no separate class of program statements, for simplicity. Expressions include integers n^τ, local variables x, which are introduced by let-bindings $\texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2$; there is no type annotation on variable x because it can be inferred from context. Constant integers n are annotated with type τ to indicate their intended type is. As in an actual implementation, pointers in our formalism are represented as integers. Annotations help formalize type checking and the safety property it provides; they have no effect on the semantics except when τ is a checked pointer, in which case they facilitate null and bounds checks. Local variables can only hold word-sized objects, so all `structs` can only be accessed by pointers.

Checked pointers are constructed using $\texttt{malloc@}\omega$; for simplicity, we do not consider numeric arguments to `malloc`, but just include the type. Thus, `malloc@int` produces a pointer of type $\texttt{ptr}^c\texttt{int}$ while `malloc@(array 10 int)` produces a pointer of type $\texttt{ptr}^c\texttt{(array 10 int)}$. Unchecked pointers can only be produced by the cast operator, $(\tau)e$, e.g., by doing $(\texttt{ptr}^u\texttt{int})\texttt{malloc@int}$. Casts can also be used to coerce between integer and pointer types and between different multi-word types.

Pointers are read via the $*$ operator, and assigned to use the $=$ operator. To read or write `struct` fields, a program can take the address of that field and read or write that address, e.g., $x{\rightarrow}f$ is equivalent to $*(\&x{\rightarrow}f)$. To read or write an array, the programmer can use pointer arithmetic to access the desired element, e.g., $x[i]$ is equivalent to $*(x + i)$.

Heap	H	$\in$	$\mathbb{Z} \rightharpoonup \mathbb{Z} \times \tau$
Result	r	::=	e | `Null` | `Bounds`
Contexts	E	::=	_ | `let` x `=` E `in` e
		|	$E + e \mid n + E$
		|	$\&E \rightarrow f \mid (\tau)E$
		|	$*E \mid *E = e \mid *n = E$
		|	`unchecked` E
Fields	fs	::=	τ `f` | τ `f;` fs

Figure 6: Semantics Definitions

By default, CoreCHKC expressions are assumed to be checked. Expression e in `unchecked` e is unchecked, giving it additional freedom. From T-Cast we can see that casting to checked pointers is only permitted in unchecked code. Doing so permits accessing the contents of memory in essentially arbitrary ways. Unchecked pointers may only be read or written in unchecked mode.

Design Notes. CoreCHKC leaves out many interesting C language features. We do not include an operation for freeing memory, since this paper is concerned about spatial safety, not temporal safety. CoreCHKC models statically sized arrays but supports dynamic indexes; supporting dynamic sizes is interesting but less important compared to the complexity it adds the formalism. We do not model control operators or function calls, whose addition would be straightforward. Function calls $f(e')$ can be modeled by `let` x `=` e' `in` e, where we can view x as function f's parameter, e as its body, and e' as its actual argument. Calls to unchecked functions from checked code can thus be simulated by having an `unchecked` e expression for e_2. We chose to make checked mode the default in the formalism, but making it unchecked would have been equally easy. We do not have a corresponding `checked` e expression to embed within an unchecked one; this could also be handled straightforwardly, with the effect of passing data affected by unchecked code to checked code modeled by the value returned from `unchecked` e.

3.2 Semantics

Figure 7 gives a small-step operational semantics for Checked C expressions, defining judgment $H; e \longrightarrow^m H; r$. Here, as shown in Figure 6, H is a *heap*, which is a partial map from integers (representing pointer addresses) to type-annotated integers n^τ. m is the *mode* of evaluation, which is either c for checked mode, or u for unchecked mode. Finally, r is a *result*, which is either an expression e, `Null` (indicating a null pointer dereference), or `Bounds` (indicating an out-of-bounds array access). An unsafe program execution occurs when the expression reaches a "stuck" state — the program is not an integer n^τ, and yet no rule applies. Notably, this could happen if trying to dereference a pointer n that is actually invalid, i.e., $H(n)$ is undefined. The semantics is implicitly parameterized by struct map D.

The semantics is defined in the standard manner using *evaluation contexts* E, given in Figure 6. Contexts are designed to ensure a unique decomposition of any expression e into context E and (smaller) expression e_0 that ensures a left-to-right evaluation order. We write $E[e_0]$ to mean the expression that results from substituting e_0 into the "hole" _ of context E. Rule C-Exp defines normal evaluation. It decomposes an expression e into a context E and expression e_0 and then evaluates the latter via $H; e_0 \rightsquigarrow H'; e_0'$ as defined in the bulk of the figure (and discussed below). The annotation m is the evaluation mode, which is restricted by the $mode(E)$ function, also given in Figure 7. The rule and this function ensure that when evaluation occurs within e in some expression `unchecked` e, then it does so in unchecked mode u; otherwise it may be in checked mode c. Rule C-Halt halts evaluation due to a failed null or bounds check; this works by simply discarding the outer context E, thus terminating the program.

Rules with prefix E- define the core computation semantics, and are fairly standard. Rule E-Binop produces an integer n_3 that is the sum of arguments n_1 and n_2. When n_1 is a checked pointer to an array and n_2 is an `int`, result n_3's type annotation is annotated as a pointer to an array with its bounds suitably updated.[3] Otherwise, n_3's type is just the same as n_1's type. E-Deref and E-Assign check the bounds of checked array pointers: the length l must be positivefor the dereference to be legal. The rule permits the program to proceed for non-checked or non-array pointers (but the type system will forbid them). Rule E-Amper takes the address of a `struct` field, according to the type annotation on the pointer. E-Malloc allocates a checked pointer by finding a string of free heap locations and initializing each to 0, annotated to the appropriate type. Here, types(D, ω) returns n types, where these are the types of the corresponding memory words; e.g., if ω is a `struct` then these are the types of its fields (looked up in D), or if ω is an array τ of length k, then we will get back k τ's. E-Let uses a substitution semantics for local variables; notation $e[x \mapsto n^\tau]$ means that all occurrences of x in e should be replaced with n^τ. E-Unchecked returns the result of an unchecked block.

Rules with prefix X- describe failures due to bounds checks and null checks; these rules complement E-Assign and E-Deref.

3.3 Typing

Typing judgment $\Gamma \vdash_m e : \tau$ says that expression e has type τ under environment Γ when in mode m. Heap H and struct map D are implicit parameters of the judgment; they are not shown because they are invariant in proof derivations. Unchecked expressions are always checked in mode u, otherwise we may use either mode. To avoid clutter, the rules elide annotation m when it could be either u or c.

Γ maps local variables x to types τ, and is used in rules T-Var and T-Let as usual. Rule T-Int ascribes type τ to constant n^τ when τ is an integer, when it is an unchecked pointer type (so dereferencing is only possible in unchecked code, and failure there is an option), when n is 0 (and thus dereferencing it in checked mode would produce `Null`) or it has type $\texttt{ptr}^c(\texttt{array}\ 0\ \tau')$ (since dereferencing it would produce `Bounds`).

Rule T-PtrC ensures checked pointers of type $\texttt{ptr}^c\omega$ are consistent with the heap. This works by checking that the pointed-to memory has types consistent with ω. When doing this, we add n^τ to Γ to properly handle cyclic heap structures, per rule T-VConst. A key feature of T-PtrC is that it effectively confirms that all pointers reachable from the given one are consistent; it says nothing about

[3] Here, $l - n_2$ is natural number arithmetic: if $n_2 > l$ then $l - n_2 = 0$.

E-Binop	$H; n_1^{\tau_1} + n_2^{\tau_2}$	$\leadsto$	$H; n_3^{\tau_3}$	where $n_3 = n_1 + n_2$ $\tau_1 = \texttt{ptr}^c(\texttt{array}\ l\ \tau)\ \wedge\ \tau_2 = \texttt{int}\ \Rightarrow$ $\tau_3 = \texttt{ptr}^c(\texttt{array}\ l'\ \tau)$ where $l' = l - n_2$ $\tau_3 = \tau_1$ otherwise
E-Cast	$H; (\tau) n^{\tau'}$	$\leadsto$	$H; n^{\tau}$	
E-Deref	$H; *n^{\tau}$	$\leadsto$	$H; n_1^{\tau_1}$	where $n_1^{\tau_1} = H(n)$ $\tau = \texttt{ptr}^c(\texttt{array}\ l\ \tau')\ \Rightarrow\ l > 0$
E-Assign	$H; *n^{\tau} = n_1^{\tau_1}$	$\leadsto$	$H'; n_1^{\tau_1}$	where $H(n)$ defined $\tau = \texttt{ptr}^c(\texttt{array}\ l\ \tau')\ \Rightarrow\ l > 0$ $H' = H[n \mapsto n_1^{\tau_1}]$
E-Amper	$H; \&n^{\tau} \rightarrow f_i$	$\leadsto$	$H; n_0^{\tau_0}$	where $\tau = \texttt{ptr}^{m'}\texttt{struct}\ T$ $D(T) = \tau_1 f_1; \ldots; \tau_k f_k$ for $1 \le i \le k$ $n_0 = n + i\ \wedge\ \tau_0 = \texttt{ptr}^{m'}\tau_i$
E-Malloc	$H; \texttt{malloc@}\omega$	$\leadsto$	$H', n_1^{\texttt{ptr}^c\omega}$	where sizeof$(\omega) = k$ and $n_1 \ldots n_k$ are consecutive $n_1 \neq 0$ and $H(n_1) \ldots H(n_k)$ are undefined $\tau_1, \ldots, \tau_k = \text{types}(D, \omega)$ $H' = H[n_1 \mapsto 0^{\tau_1}] \ldots [n_k \mapsto 0^{\tau_k}]$
E-Let	$H; \texttt{let}\ x\ =\ n^{\tau}\ \texttt{in}\ e$	$\leadsto$	$H; e[x \mapsto n^{\tau}]$	
E-Unchecked	$H; \texttt{unchecked}\ n^{\tau}$	$\leadsto$	$H; n^{\tau}$	
X-DerefOOB	$H; *n^{\tau}$	$\leadsto$	$H; \texttt{Bounds}$	where $\tau = \texttt{ptr}^c(\texttt{array}\ 0\ \tau_1)$
X-AssignOOB	$H; *n^{\tau} = n_1^{\tau_1}$	$\leadsto$	$H; \texttt{Bounds}$	where $\tau = \texttt{ptr}^c(\texttt{array}\ 0\ \tau_1)$
X-DerefNull	$H; *0^{\tau}$	$\leadsto$	$H; \texttt{Null}$	where $\tau = \texttt{ptr}^c\omega$
X-AssignNull	$H; *0^{\tau} = n^{\tau'}$	$\leadsto$	$H; \texttt{Null}$	where $\tau = \texttt{ptr}^c\omega$

C-Exp

$$\frac{e = E[e_0] \qquad m = mode(E) \vee m = u \qquad H; e_0 \leadsto H'; e_0' \qquad e' = E[e_0']}{H; e \longrightarrow^m H'; e'}$$

C-Halt

$$\frac{e = E[e_0] \qquad m = mode(E) \vee m = u \qquad H; e_0 \leadsto H'; r \text{ where } r = \texttt{Null} \text{ or } r = \texttt{Bounds}}{H; e \longrightarrow^m H'; r}$$

$mode(\_)$	=	c
$mode(\texttt{unchecked}\ E)$	=	u
$mode(\texttt{let}\ x\ =\ E\ \texttt{in}\ e)$	=	
$mode(E + e)$	=	
$mode(n + E)$	=	
$mode(\&E \rightarrow f)$	=	
$mode((\tau)E)$	=	
$mode(*E)$	=	
$mode(*E = e)$	=	
$mode(*n = E)$	=	$mode(E)$

Figure 7: Semantics

other parts of the heap. For example, if some set of checked pointers is only reachable via unchecked pointers then we are not concerned whether the former are consistent, since they cannot be accessed (directly) from checked pointers.

Rules T-Amper and T-Let are unsurprising. Rule T-BinopInt types addition of integers. Rule T-Malloc produces checked pointers. Rule T-Unchecked introduces unchecked mode, relaxing access rules. Rule T-Cast enforces that checked pointers cannot be cast targets in checked mode.

Rules T-Deref and T-Assign type pointer accesses. These rules require unchecked pointers only be dereferenced in unchecked mode. Rule T-Index permits reading a computed pointer to an array, and rule T-IndAssign permits writing to one. These rules are not strong enough to permit updating a pointer to an array after performing arithmetic on it. (This limitation can be overcome in unchecked code by casting the array to an `int`, adjusting the pointer, and then casting it back.)

3.4 Metatheory

Our goal is to show that well-typed programs will never fail due to a spatial safety violation so long as unchecked code completes its execution leaving the program in a well-formed state. In effect, any failure can be blamed on unchecked code. Our main result is formalized as two lemmas.

The first lemma, Progress, indicates that a well-typed program either is a value, can take a step (in either mode), or else is stuck (cannot evaluate) in unchecked code. The latter is true if e only type checks in mode u, or its (unique) context E has mode u.

Lemma 3.1 (Progress). *If $\cdot \vdash_m e : \tau$ (under heap H) then one of the following holds:*

- *e is an integer n^{τ}*
- *There exists H', m', and r such that $H; e \longrightarrow^{m'} H'; r$ where r is either some e', `Null`, or `Bounds`.*
- *$m = u$ or $e = E[e'']$ and $mode(E) = u$ for some E and e''.*

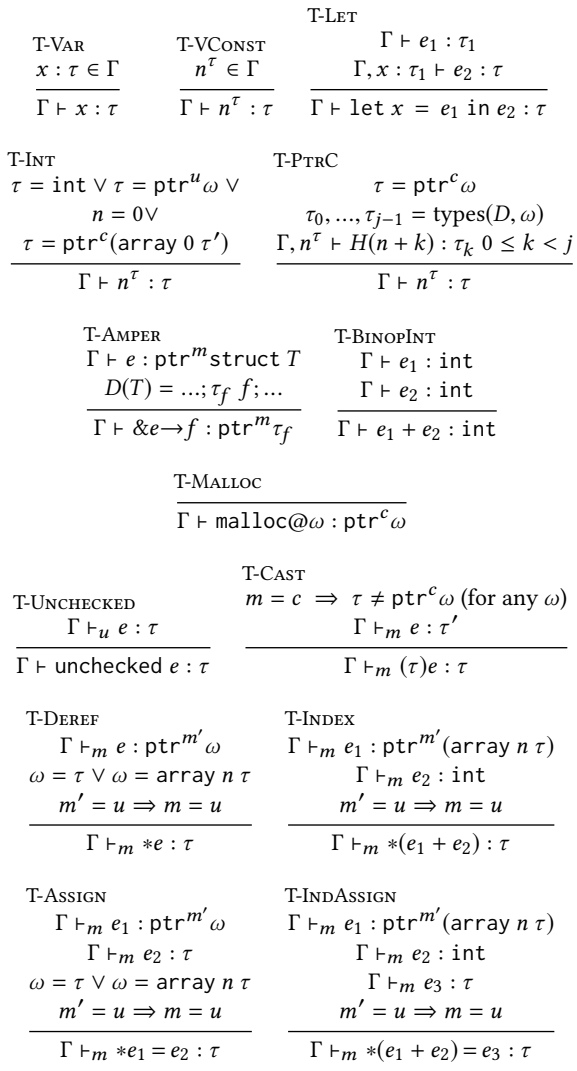

Figure 8: Typing judgment

The second lemma, Preservation, implies that if a well-typed program takes a step in checked mode then the resulting program is also well-typed.

Lemma 3.2 (Preservation). *If $\Gamma \vdash_m e : \tau$ (under a heap H), and $\vdash \Gamma$, and $H; e \longrightarrow^{m'} H'; r$ (for some H', m', r), then $m' = c$ and $r = e'$ implies $H' \vdash H$ and $\Gamma \vdash_m e' : \tau$ (under heap H').*

Here we write $\vdash \Gamma$ to mean $\nexists n^\tau \in \Gamma$, i.e., Γ just contains mappings of variables to types. We write $H' \vdash H$ to mean that for all i, if $H(i) = n^\tau$ such that $\cdot \vdash n^\tau : \tau$ under H then there exists n' such that $H'(i) = n'^\tau$ where $\cdot \vdash n'^\tau : \tau$ under H'. This says that H' basically agrees with H on the types of its well-defined locations.

The proofs of both lemmas are by induction on the typing derivation. The most delicate aspect of the proofs is ensuring that modifications to checked pointers in the heap preserve typing, despite the creation or modification of cyclic data structures.

Putting these two lemmas together guarantees that a well-typed program (that is not a value) can always take a step when in checked code, and when it does so the result is a well-typed program, or else it will fail gracefully due to a null or bounds check. Evaluating in unchecked code provides no guarantee. A well-typed program could fail in unchecked mode due to an attempt to dereference an unchecked pointer that points to undefined memory. Or, a well-typed program could take a step in unchecked mode that constructs a bogus checked pointer (e.g., that points to undefined memory), and thus fails to type check. A corollary of these observations is that if the program's evaluation is consistently well-typed (i.e., no bogus checked pointers are created at any point) the only possible failures (i.e., due to a stuck program) will be in unchecked code. In any case, if an unchecked block completes in a well-typed state, then checked code execution from thereon will be safe.

These results can be related to the *blame theorem* in the gradual typing literature [31, 46, 53]. In particular, the theorem "well-typed code can't be blamed" [53] indicates that the statically typed part of a mixed-typing program can always execute without error; only corruption by or execution of the dynamically typed component can produce a failure. For Checked C, the same situation holds, respectively, for checked and unchecked code.

4 IMPLEMENTATION

We have implemented Checked C as an extension to the Clang/LLVM compiler. This section describes the various changes we made. It also presents an evaluation of Checked C on a suite of standard benchmarks. We count the source code changes required to make these benchmarks checked, and measure the corresponding overhead to both compilation time and running time.

4.1 Compiler Implementation

The Clang and LLVM toolchain is organized with Clang being the C or C++ frontend that produces an assembly-like IR for LLVM, the backend. LLVM contains most of the analyses and optimizers.

We extended the C grammar [26] to support bounds declarations, the new `_Ptr<T>`, `_Array_ptr<T>`, and `T _Checked[N]` types, and adding `_Checked` or `_Unchecked` annotations to blocks and functions. We chose reserved identifiers so that they will not conflict with identifiers in existing code.

We include a set of *checked headers* which ascribe checked types and bounds to functions in the C standard library. These are used in Checked C programs as replacements for the standard C headers.

4.2 Type checking

In order to support Checked C's new types, we extended Clang's type representation and type checker. We added a pointer kind discriminator to Clang's pointer type representation, and a "checked" flag to Clang's array type representation.

C includes implicit conversions to allow use of values at assignments, function calls, and return statements where the types are not equal but are compatible. We additionally allow implicit conversions from `T*` to `_Ptr<T>` in unchecked code. The reverse is unsound, as that would not prevent performing arithmetic on `_Ptr<T>`s. We also allow implicit conversions from constant-sized unchecked arrays to constant-sized checked arrays of the same dimensions and types. We must know the runtime size of the data

pointed to by the unchecked pointer, which we do for address-taken locals and local arrays.

4.3 Bounds Expressions

Checked C's bounds expressions provide a static description of the bounds on a pointer. We check statically that the sub-expressions of a bounds expression are *non-modifying expressions*: they do not contain any assignment, increment or decrement operators, or function calls. This ensures that using the expressions at bounds checks does not cause unexpected side-effects.

We extended Clang's representation of variable, member, and function declarations to support adding bounds expressions, as well as related concepts such as redeclarations and type compatibility.

We extended Clang's representation of function types to represent the bounds expressions of the parameters and the return value in a position-agnostic way. This lets us use syntactic equivalence of bounds expressions to verify that bounds in function types are compatible with each other.

4.4 Bounds Inference

Checked C performs inference to compute a bounds expression that conservatively describes the bounds on a pointer. Inference uses bounds expressions normalized into bounds(l, u) form.

In C, expressions evaluate to either an lvalue or a value.[4] Lvalues represent the locations of objects in memory, and are used to access memory. Integers, floats and pointers are values. The expressions that evaluate to an lvalue are: variables; dereferences; array indexing; and member accesses including arrow expressions. All other expressions evaluate to a value. Lvalue expressions are required in the left-hand-side of assignment expressions; address-of expressions; and increment or decrement expressions. Lvalue expressions can be used where a value expression is required. In that case, if the lvalue expression has array type, it is converted to an array pointer value (*array conversion*). Otherwise, a read of memory using the lvalue produced by the lvalue expression is inserted (which C calls an *lvalue conversion*).

This means we need to infer bounds expressions for value expressions, (*value bounds*), bounds expressions for lvalue expressions, (*lvalue bounds*), and bounds expressions for the value result of an lvalue conversion, (*lvalue target bounds*).

In general, the value bounds of an expression are the value bounds of the pointer sub-expression with bounds; the lvalue bounds of an expression correspond to the bounds on the storage location of that lvalue; and the lvalue target bounds of an expression correspond to the bounds declared for that lvalue.

We perform narrowing of the inferred bounds in some places to ensure type safety. The rule is that we want to ensure that, if a bounds check succeeds, the user got back a value of the type they were expecting. This means that when we infer the bounds of member expressions, we narrow to the bounds of the individual struct member, as other struct members could have different types.

However, in the case of inner dimensions of multidimensional checked arrays, we do not narrow in from the outermost checked array, because every element of the array has the same type. This choice prevents us from having to do lots of dynamic checks inside array processing inner loops.

Narrowing can also be performed explicitly by the programmer by assigning to a variable with narrower declared bounds than the value being assigned to it.

4.5 Static Checking of Bounds Declarations

The Checked C compiler performs static checks to ensure that bounds declarations are valid and programs do not contain improper uses of checked pointers. One such check prevents performing pointer arithmetic on `_Ptr<T>` values.

The most important static check is called the *subsumption check*. This check ensures that when assigning to an lvalue expression with checked array pointer type, we infer the bounds for both sub-expressions, and then check the bounds of the value imply the bounds of the lvalue. This check allows assignment to narrow, but not to widen, the bounds of the assigned value. This requirement also applies at initialization and to function calls. We prove this statically so that we add no unexpected run-time overhead to variable assignment or function calls.

This static subsumption check is not implemented yet in our compiler. In the benchmarks below, we used a set of simpler, unsound checks to catch trivial errors and manual code review to ensure this subsumption property holds.

4.6 Run-time Checks

The Checked C compiler inserts run-time checks into the evaluation of lvalue expressions that will access memory through a checked pointer. The code for these checks is handed to LLVM, which we allow to remove checks if it can prove they will always pass. In general, such checks are the only source of Checked C overhead (aside from programmer use of `_Dynamic_check`).

Before any `_Ptr<T>` accesses the compiler inserts a check that the pointer is non-null. Before any `_Array_ptr<T>` accesses the compiler inserts a non-null check followed by the required bounds check computed from the inferred bounds. The compiler does not perform any range checks during pointer arithmetic, unless the arithmetic accesses memory.

In some cases, such as a nested dereference expression like `**p`, we may have to emit more than one set of dynamic checks: the first for the outer pointer dereference and another for the inner pointer dereference. Similarly, these checks can be emitted during the calculation of the upper or lower bounds for a bounds check.

4.7 Evaluation

We converted small existing benchmarks from C as an initial evaluation of the consequences of porting code to Checked C, including evaluating both the changes required for the code to become checked, and the performance overhead of the run-time checks.

We chose 10 benchmarks from the Olden [42] and Ptrdist [4] benchmark suites, because these are suites specifically designed to test pointer-intensive applications, and they are the same benchmarks used to evaluate both Deputy and CCured. We chose to use only the benchmarks that were under 1 KLOC. We omitted bh and voroni from Olden and bc, ft, and yacr2 from Ptrdist. The benchmarks used are described in Table 1.

[4] Values are more commonly called rvalues in other language descriptions.

Name	LOC	Description
bisort	350	Sorts using two disjoint bitonic sequences
em3d	688	Simulates electromagnetic waves in 3D
health	504	Simulates Columbian health-care system
mst	428	Computes minimum spanning tre
perimeter	484	Computes perimeter of a set of quad-tree encoded images
power	622	The Power System Optimization problem
treadd	245	Computes the sum of values in a tree
tsp	582	Estimates solution for the Traveling-salesman problem
anagram	657	Generates anagrams from a list of words
ks	783	Schweikert-Kernighan graph partitioning

Table 1: Compiler Benchmarks. Top group is the Olden suite, bottom group is the Ptrdist suite. LOC includes all comments and blank lines in benchmark source files. Descriptions are from [4, 42].

	Code Changes			Observed Overheads		
Name	*LM %*	*EM %*	*LU %*	*RT ±%*	*CT ±%*	*ES ±%*
bisort	16.0	85.3	5.7	+ 0.1	- 1.3	+ 6.1
em3d	25.3	65.6	12.0	+ 43.0	+ 11.0	+ 0.7
health	15.7	97.8	8.7	+ 7.1	+ 10.0	- 1.6
mst	27.6	75.3	15.9	+ 0.4	- 0.2	- 16.6
perimeter	9.9	93.2	5.6	- 0.1	+ 0.4	+ 0.8
power	12.2	70.1	6.6	0.0	+ 16.7	+ 6.0
treeadd	14.7	92.9	13.5	+ 2.2	+ 49.9	+ 7.0
tsp	8.2	95.2	15.3	0.0	+ 38.0	+ 1.1
anagram	16.1	77.5	11.7	+ 32.8	+ 28.6	+ 27.5
ks	10.7	93.9	21.0	+ 7.0	+ 20.3	+ 21.5
Mean:	14.6	83.9	10.5	+ 8.4	+ 16.2	+ 4.6

Table 2: Benchmark Results. Key: *LM %*: Percentage of Source LOC Modified, including Additions; *EM %*: Percentage of Code Modifications deemed to be Easy (see 4.7.1); *LU %*: Percentage of Lines remaining Unchecked; *RT ±%*: Percentage Change in Run Time; *CT ±%*: Percentage Change in Compile Time; *ES ±%*: Percentage Change in Executable Size (`.text` section only). *Mean*: Geometric Mean.

We evaluate Checked C using these benchmarks in two ways. First, we quantify the number and type of source code changes required to convert these benchmarks from C to Checked C. Second, we quantify the overhead of the run-time checks on benchmark run time, compile time, and executable size. The evaluation results are presented in Table 2.

We ran these benchmarks on a 12-Core Intel Xeon X5650 2.66GHz, with 24GB of RAM, running Red Hat Enterprise Linux 6. All compilation and benchmarking was done without parallelism. We ran each benchmark 100 times with and without the Checked C changes using the test sizes from the LLVM benchmarks. We averaged the 100 runs and compared the arithmetic means.

4.7.1 Code Changes. On average, the changes modified around 15% of benchmark lines of code. Most of these changes were in declarations, initializers, and type definitions rather than in the program logic. In the evaluation of Deputy [13], the reported figure of lines changed ranges between 0.5% and 11% for the same benchmarks, showing they have a lower annotation burden than Checked C.

We modified the benchmarks to use checked blocks and the top-level checked pragma. We placed code that could not be checked because it used unchecked ponters in unchecked blocks. On average, about 10.5% of the code remained unchecked after conversion, with a minimum and maximum of 5.7% and 21%. The causes were almost entirely the use of strings and variable argument functions for printing.

We manually inspected changes and divided them into *easy* changes and *hard* changes. Easy changes include: replacing included headers with their checked versions; converting a `T*` to a `_Ptr<T>`; adding the `_Checked` keyword to an array declaration; introducing a `_Checked` or `_Unchecked` region; adding an initializer; and replacing a call to `malloc` with a call to `calloc`. Hard changes are all other changes, including changing a `T*` to a `_Array_ptr<T>` and adding a bounds declaration, adding structs, struct members, and local variables to represent run-time bounds information, and code modernization.

We distinguish between the two because we believe easy changes can be automated (as with our automated `_Ptr<T>` conversion tool in Section 5) or made unnecessary in the future by relaxing requirements such as the additions of initializers.

In two benchmarks, em3d and mst, we had to add intermediate structs so that we could represent the bounds on an `_Array_ptr<T>`s nested inside arrays. In mst we also had to add a member to a struct to represent the bounds on an `_Array_ptr<T>`. In the first case, this is because we cannot represent the bounds on nested `_Array_ptr<T>`s, in the second case this is because we only allow bounds on members to reference other members in the same struct. In em3d and anagram we also added local temporary variables to represent bounds information.

In all of our benchmarks, we found the majority of changes were easy. In six of the benchmarks, the only hard changes were adding bounds annotations relating to the parameters of `main`.

4.7.2 Observed Overheads. An important concern about run-time checking for C is the effect on performance and compile time. The average run-time overhead introduced by adding dynamic checks was 8.4%. In half of the benchmarks the overhead was less than 1%, including one case where no overhead was added at all. We believe this to be an acceptably low overhead that better static analysis may reduce even further.

In all but three benchmarks, the added overhead is not more than the overhead added by Deputy. In the outlier results, em3d and anagram, we have 12 and 7 percentage points respectively less overhead than Deputy. In all the benchmarks, the added overhead is not more than the overhead reported by CCured.

On average, the compile-time overhead introduced by using Checked C is 16.2%. The maximum overhead introduced is 50%, and the minimum is 1% faster than compiling with C.

We also evaluated code size overhead, by looking at the change in the size of `.text` section of the executable. This excludes data that might be stripped, like debugging information. Across the benchmarks, there is an average 5% code size overhead from the

introduction of dynamic checks. Eight of the benchmarks have a code size increase of less than 7%, including a reduction in code size in two benchmarks.

4.8 Work In Progress

As we have mentioned before, the development of the compiler is still in progress. There are three kinds of dynamic checks that we have not yet implemented. The first is the dynamic version of our subsumption check, performed by an explicit bounds cast operator. The second is checking that checked pointer arithmetic is not done on the null pointer (to prevent forging of checked pointers). The third is checking for overflow on checked pointer arithmetic.

5 AUTOMATIC PORTING

Porting legacy code to use checked pointers and regions can be time consuming. To assist the process, we developed a source-to-source translation tool called *checked-c-convert* that discovers safely-used pointers and rewrites them to be checked. This section presents the design and implementation of the tool and evaluates its effectiveness on a series of benchmark programs.

5.1 Conversion tool design and overview

checked-c-convert aims to be sound while also producing edits that are minimal and unsurprising. A rewritten program should be recognizable by the author and it should be usable as a starting point for both the development of new features and additional porting. A particular challenge is to preserve syntactic structure of the program. Previous, similar analyses have often been defined on code produced after preprocessing. These analyses also sometimes work by combining multiple source files into one file, prior to analysis. These choices are problematic for us: We need to rewrite one file at a time (perhaps taking into account whole-program knowledge), and preserve the definition and use of macros, and other formatting, in the source code.

The *checked-c-convert* tool is implemented as a clang libtooling application that traverses the AST to generate constraints based on pointer usage, and solves those constraints. With a solution, the tool will rewrite some declared pointer types to be checked, and may insert some casts. The tool operates on post-preprocessed code, but has sufficient location information to be able to rewrite the original source files. Moreover, for macro expansions that have parameters, it considers all expansions of those parameters together, so as to be able to rewrite the original macro's definition. In effect, this produces a context- and flow-insensitive rewriting of macros, just as would occur for functions.

5.2 Constraint logic and solving

The basic idea of the tool is to infer a *qualifier* q_i for each defined pointer variable i. Inspired by the approach taken by CCured [35], qualifiers can be either *PTR*, *ARR* and UNK, which are organized as a lattice (*PTR* is the lowest, UNK is the highest). Those variables with inferred qualifier *PTR* can be rewritten into `_Ptr<T>` types, while those with UNK are left as is. Those with the *ARR* qualifier are eligible to have `_Array_ptr<T>` type. For the moment we only signal this fact in a comment and do not rewrite because we cannot always infer proper bounds expressions. In addition to local and global variables, constraint variables are associated with individual struct fields, return values from functions, and parameters to functions, assuming each contain a pointer. Constraint variables for return values and parameters are globally unique, so no calling context being considered when analyzing constraints on functions.

Qualifiers are determined based on constraints introduced based on how pointer variables are used. Constraints are written in a simple logic that can express equality, inequality, and implication, and represent flow and path insensitive program facts and are solved using a unification based algorithm. They are generated over the whole program, to include multiple compilation units and header files, potentially including system header files. An expression that performs arithmetic on a pointer value, either via `+` or `[]`, introduces a constraint on that pointer value $q_i = ARR$. Assignments between pointers introduce aliasing constraints of the form $q_i = q_j$. Existing Checked C annotations introduce negation constraints to fix a constraint variable to the constraint represented by the Checked C type. Casts introduce implication constraints based on the relationship between the sizes of the two types. If the sizes are not comparable, then both constraint variables in an assignment based cast are constrained to UNK via an equality constraint. Constraints are generated for each file individually, at first. Then these constraints are "linked" together when it can be determined that they refer to the same global definition. Once all constraints are connected they are solved, producing a final assignment of qualifiers to variables. After solving, the individual files are rewritten according to their solutions. If a conflict arises due to an equality constraint between any variable and the parameter to the `free` function, an explicit cast is inserted.

Turning to the question of solving, the algorithm resembles the approach taken by CCured. Generated constraints are limited to be of one of the following forms:

(1) $q_i = PTR \mid ARR \mid UNK \mid q_j$
(2) $\neg(q_i = PTR \mid ARR \mid UNK)$
(3) $q_i = ARR \longrightarrow q_j = ARR$
(4) $q_i = UNK \longrightarrow q_j = UNK$

During solving, each constraint variable begins with an initial value of *PTR*. Solving this system of constraints works iteratively by propagating aliasing and equality constraints to UNK first, then aliasing, equality and implication constraints involving UNK, then aliasing, implication and equality constraints to *ARR*. The constraint language differs slightly from CCured due to the modular nature of Checked C: an existing portion of the program could specify that a variable is a `_Ptr<T>` and *checked-c-convert* should not regress the program, so it is not always acceptable to constrain variables to UNK. These conflicts can be resolved with the insertion of explicit casts, as described above. Like CCured, this algorithm runs in linear time proportional to the number of pointer variables in the program.

5.3 Evaluation

To evaluate the rewriter, we ran it on six programs and libraries and recorded how many pointer types the rewriter converted. The rewriter was executed on an Amazon AWS `c3.4xlarge` (a Xeon E5-2680 v2 2.80GHz processor with 32GB of RAM) instance running Ubuntu 16.04. We chose these programs as they represent legacy,

Program	# of *	%/# _Ptr	Arr.	Unk.	Time (s)	LOC
zlib 1.2.8	897	42%/376	10%/94	48%/427	19	6220
sqlite 3.18.1	36269	23%/8161	3%/952	75%/27156	94	134788
libarchive 3.3.1	19461	31%/6023	2%/459	67%/12979	596	80182
lua 5.3.4	4291	25%/1060	2%/78	73%/3153	28	14585
libtiff 4.0.6	7609	24%/1791	4%/267	73%/5551	90	57091
vsftpd 3.0.3	2037	42%/861	2%/33	56%/1143	15	15048

Table 3: Number of pointer types converted. The # of * column represents the number of pointer types in the program. The Arr and Unk columns represent constraints where the rewriter determined that the access into the pointer was via indexing (Arr) or that the constraints can't be captured by the rewriter (Unk) due to casts, assignment to a non-zero literal, or some other operation.

low level libraries that are used in commodity systems and frequently in security-sensitive contexts. Table 3 contains the results. The value in the `_Ptr<T>` column indicates the number of `_Ptr<T>` added to the program that replace standard C pointers. These are re-written at the location they are declared. After investigation, there are usually two reasons that a pointer cannot be replaced with a `_Ptr<T>`: either some arithmetic is performed on the pointer, or it is passed as a parameter to a library function for which a bounds-safe interface does not exist.

6 RELATED WORK

There has been extensive research addressing out-of-bounds memory accesses in C [49]. The research falls into 4 categories: languages, implementations, static analysis, and security mitigations.

Safe languages. Cyclone [27] and Deputy [13, 57] are type-safe dialects of C. Cyclone's key novelty is its support for GC-free temporal safety [22, 48]. Checked C differs from Cyclone by being backward compatible (Cyclone disallowed many legacy idioms) and avoiding pointer format changes (e.g., Cyclone used "fat" pointers to support arithmetic). Deputy keeps pointer layout unchanged by allowing a programmer to describe the bounds using other program expressions. Checked C builds on this, but make bounds checking a first-class part of the language. Deputy incorporates the bounds information into the types of pointers by using dependent types. This makes type checking hard to understand . Deputy requires that values of all pointers stay in bounds so that they match their types. To enforce this invariant (and make type checking decidable), it inserts runtime checks before pointer arithmetic. Checked C uses separate annotations that describe bounds invariants instead of incorporating bounds into pointer types and inserts runtime checks only at memory accesses.

Like Cyclone, programming languages like D [17] and Rust [43] aim to support safe, low-level systems-oriented programming without requiring GC. Go [21] and C# [33] target a similar domain. Legacy programs would need to be ported wholesale to take advantage of these languages, which could be a costly affair.

Safe C implementations. Rather than use a new language, several projects have looked at new ways to implement legacy C programs so as to make them spatially safe. The *bcc* source-to-source translator [29] and the *rtcc* compiler [47] changed the representations of pointers to include bounds. The *rtcc*-generated code was 3 times larger and about 10 times slower. Fail-Safe C [38] changed the representation of pointers and integers to be pairs. Benchmarks were 2 to 4 times slower. CCured [35] employed a whole-program analysis for transforming programs to be safe. Its transformation involved changes to data layout (e.g., fat and "wild" pointers), which could cause interoperation headaches. Compilation was all-or-nothing: unhandled code idioms in one compilation unit could inhibit compilation of the entire program. Our rewriting algorithm is inspired by CCured's analysis with the important differences that (a) not every pointer need be made safe, and (b) the output is not a step in compilation, but programmer-maintainable source code.

Safety can also be offered by the loader and run-time system. "Red zones", used by Purify [25, 51] are inserted before and after dynamically-allocated object and between statically-allocated objects, where bytes in the red zone are marked as inaccessible (at a cost of 2 bits per protected byte). Red-zone approaches cannot detect out-of-bounds accesses that occur entirely within valid memory for other objects or stack frames or intra-object buffer overruns (a write to an array in a struct that overwrites another member of the struct). Checked C detects accesses to unrelated objects and intra-object overruns.

Similar tools include Bounds Checker [32], Dr. Memory [9, 18], Intel Inspector [14], Oracle Solaris Studio Code Analyzer [39], Valgrind Memcheck [36, 52], Insure++ [40], and AddressSanitizer (ASAN) [45]. ASAN is incorporated into the LLVM and GCC compilers. It tracks the state of 8-byte chunks in memory. It increases SPEC CPU program execution time by 73% when checking reads and writes and 26% when only checking writes. SPEC CPU2006 average memory usage is 3.37 times larger. Light-weight Bounds Checking [24] uses a two-level table to reduce memory overhead.

Checking that accesses are to the proper objects can be done using richer side data structures that track object bounds and by checking that pointer arithmetic stays in bounds [3, 19, 28, 34, 41, 44, 56]. Baggy Bounds Checking [3] provides a fast implementation of object bounds by reserving 1/n of the virtual address space for a table, where n is the smallest allowed object size and requiring object sizes be powers of 2. It increases SPECINT 2000 execution time by 60% and memory usage by 20%. SoftBound [34] tracks bounds information by using a hash table or a shadow copy of memory. It increases execution time for a set of benchmarks by 67% and average memory footprint by 64%. SoftBound can check only writes, in which case execution time increases by 22%.

There is also work on adding temporal safety with different memory allocation implementations, e.g., via conservative garbage collection [8] or regions [22, 48]. Checked C focuses on spatial safety both due to its importance at stopping code injection style attacks as well as information disclosure attacks, though temporal safety is important and we plan to investigate it in the future.

Static analysis. Static analysis tools take source or binary code and attempt to find possible bugs, such as out-of-bounds array accesses, by analyzing the code. Commercial tools include CodeSonar, Coverity Static Analysis, HP Fortify, IBM Security AppScan, Klocwork, Microsoft Visual Studio Code Analysis for C/C++, and Polyspace Static Analysis [6, 10, 20]. Static analysis tools have difficulty balancing precision and performance. To be precise, they may not scale to large programs. While imprecision can aid scalability, it can result in false positives, i.e., error reports that do not correspond to real bugs. False positives are a significant problem [6]. As a result, tools may make unsound assumptions (e.g., inspecting only a limited number of paths through function [10]) but the result is they may also miss genuine bugs (false negatives). Alternatively, they may focus on supporting coding styles that avoid problematic code constructs, e.g., pointer arithmetic and dynamic memory allocation [2, 7, 16, 30]. Or, they may require sophisticated side conditions on specifications, i.e., as pre- and post-conditions at function boundaries, so that the analysis can be modular, and thus more scalable [23].

Checked C occupies a different design point than static analysis tools. It avoids problems with false positives by deferring bounds checks to runtime—in essence, it trades run-time overhead for soundness and coding flexibility. In addition, Checked C avoids complicated specifications on functions. For example, a modular static analysis might have required the code in Figure 2 to include that `src_count` $\leq$ `dst_count` as a function pre-condition. While this constraint is not particularly onerous, some specifications can be. In Checked C, such side conditions are unnecessary; instead, soundness ensured by occasional dynamic checks.

Security mitigations. Security mitigations employ runtime-only mechanisms that detect whether memory has been corrupted or prevent an attacker from taking control of a system after such corruption. They include data execution prevention (DEP), software fault isolation (SFI) [54] , address-space layout randomization (ASLR) [50, 55], stack canaries [15], shadow stacks [5, 12], and control-flow integrity (CFI) [1]. DEP, ASLR, and CFI focus on preventing execution of arbitrary code and control-flow modification. Stack protection mechanisms focus on protecting data or return addresses on the stack.

Checked C provides protection against data modification and data disclosure attacks, which the other approaches do not. For example, ASLR does not protect against data modification or data disclosure attacks. Data may be located on the stack adjacent to a variable that is subject to a buffer overrun; the buffer overrun can be be used reliably to overwrite or read the data. Shadow stacks do not protect stack-allocated buffers or arrays, heap data, and statically-allocated data. Chen et al. [11] show that data modification attacks that do not alter control-flow pose a serious long-term threat. The Heartbleed attack illustrates the damage possible.

7 SUMMARY

In this paper, we presented the Checked C extension to C. Checked C's design is focused on interoperability with legacy C, usability, and high performance. To assist in incrementally strengthening legacy code, we have provided an automated porting tool for rewriting code to use checked pointers. Checked C's *checked regions* help ensure the non-culpability of ported code in any safety violation. Our implementation of Checked C as an LLVM extension enjoys good performance, with relatively low run-time overheads.

Checked C is an ongoing project, with code freely available on the Internet. We are working to improve our support for pointer arithmetic, to add support for zero-terminated pointers, and to extend the capabilities of the rewriting tool. In the longer term we plan to support temporal safety checking as well.

ACKNOWLEDGMENTS

We would like to thank Jijoong Moon and Wonsub Kim from Samsung for their assistance with the implementation of the Checked C compiler and the manual conversion of several benchmarks.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS '05)*. ACM, New York, NY, USA, 340–353. https://doi.org/10.1145/1102120.1102165

[2] AbsOmt. 2016. Astrée: Fast and sound runtime error analysis. http://www.absint.com/astree/index.htm. (2016). Accessed May 12, 2016.

[3] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. 2008. Preventing Memory Error Exploits with WIT. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy (SP '08)*. IEEE Computer Society, Washington, DC, USA, 263–277. https://doi.org/10.1109/SP.2008.30

[4] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. 1994. Efficient Detection of All Pointer and Array Access Errors. *SIGPLAN Not.* 29, 6 (June 1994), 290–301. https://doi.org/10.1145/773473.178446

[5] Arash Baratloo, Navjot Singh, and Timothy Tsai. 2000. Transparent Run-time Defense Against Stack Smashing Attacks. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '00)*. USENIX Association, Berkeley, CA, USA, 21–21. http://dl.acm.org/citation.cfm?id=1267724.1267745

[6] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM* 53, 2 (Feb. 2010), 66–75. https://doi.org/10.1145/1646353.1646374

[7] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2003. A Static Analyzer for Large Safety-critical Software. *SIGPLAN Not.* 38, 5 (May 2003), 196–207. https://doi.org/10.1145/780822.781153

[8] Hans-Juergen Boehm and Mark Weiser. 1988. Garbage Collection in an Uncooperative Environment. *Softw. Pract. Exper.* 18, 9 (Sept. 1988), 807–820.

[9] Derek Bruening and Qin Zhao. 2011. Practical Memory Checking with Dr. Memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11)*. IEEE Computer Society, Washington, DC, USA, 213–223. http://dl.acm.org/citation.cfm?id=2190025.2190067

[10] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. 2000. A Static Analyzer for Finding Dynamic Programming Errors. *Softw. Pract. Exper.* 30, 7 (June 2000), 775–802. https://doi.org/10.1002/(SICI)1097-024X(200006)30:7<775::AID-SPE309>3.0.CO;2-H

[11] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. 2005. Non-control-data Attacks Are Realistic Threats. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14 (SSYM'05)*. USENIX Association, Berkeley, CA, USA, 12–12. http://dl.acm.org/citation.cfm?id=1251398.1251410

[12] Tzi-cker Chiueh and Fu-Hau Hsu. 2001. RAD: A Compile-Time Solution to Buffer Overflow Attacks. In *Proceedings of the The 21st International Conference on Distributed Computing Systems (ICDCS '01)*. IEEE Computer Society, Washington, DC, USA. http://dl.acm.org/citation.cfm?id=876878.879316

[13] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. 2007. Dependent Types for Low-Level Programming. In *Proceedings of European Symposium on Programming (ESOP '07) (Lecture Notes in Computer Science)*, Vol. 4421. Springer-Verlag, Heidelberg, 520–535.

[14] Intel Corporation. 2016. Intel Inspector. https://software.intel.com/en-us/intel-inspector-xe. (2016). Accessed May 6, 2016.

[15] Crispin Cowan, Calton Pu, Dave Maiere, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. 1998. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7 (SSYM'98)*. USENIX Association, Berkeley, CA, USA, 5–5. http://dl.acm.org/citation.cfm?id=1267549.1267554

[16] David Delmas and Jean Souyris. 2007. Astrée: From Research to Industry. In *Proceedings of the 14th International Conference on Static Analysis (SAS'07)*. Springer-Verlag, Berlin, Heidelberg, 437–451. http://dl.acm.org/citation.cfm?id=2391451.2391480

[17] dlang.org. 2016. D. http://dlang.org/. (2016). Accessed May 13, 2016.

[18] Dr. Memory. 2016. Dr. Memory: Memory Debugger for Windows, Linux, and Mac. http://www.drmemory.org/. (2016). Accessed May 6, 2016.

[19] Gregory J. Duck and Roland H. C. Yap. 2016. Heap Bounds Protection with Low Fat Pointers. In *Proceedings of the 25th International Conference on Compiler Construction (CC 2016)*. ACM, New York, NY, USA, 132–142. https://doi.org/10.1145/2892208.2892212

[20] Pär Emanuelsson and Ulf Nilsson. 2008. A Comparative Study of Industrial Static Analysis Tools. *Electron. Notes Theor. Comput. Sci.* 217 (July 2008), 5–21. https://doi.org/10.1016/j.entcs.2008.06.039

[21] golang.org. 2016. The Go Programming Language. https://golang.org/. (2016). Accessed May 13, 2016.

[22] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. 2002. Region-based Memory Management in Cyclone. In *PLDI*.

[23] Brian Hackett, Manuvir Das, Daniel Wang, and Zhe Yang. 2006. Modular Checking for Buffer Overflows in the Large. In *ICSE*.

[24] Niranjan Hasabnis, Ashish Misra, and R. Sekar. 2012. Light-weight Bounds Checking. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO '12)*. ACM, New York, NY, USA, 135–144. https://doi.org/10.1145/2259016.2259034

[25] Reed Hastings and Bob Joyce. 1992. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter 1992 USENIX Conference*. USENIX Association, Berkeley, CA, USA, 125–138.

[26] ISO. 2011. ISO/IEC 9899:2011 - Information Technology - Programming Languages - C (C11 standard). (2011).

[27] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, , and Yanling Wang. 2002. Cyclone: A Safe Dialect of C. In *USENIX Annual Technical Conference*. USENIX, Monterey, CA, 275–288.

[28] Richard W. M. Jones and Paul H. J. Kelly. 1997. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Third International Workshop on Automated Debugging (Linkoping Electronic Conference Proceedings)*, Miriam Kamkar and D. Byers (Eds.). Linkoping University Electronic Press. "http://www.ep.liu.se/ea/cis/1997/009/".

[29] Samuel C. Kendall. 1983. Bcc: runtime checking for C programs. In *USENIX Toronto 1983 Summer Conference*. USENIX Association, Berkeley, CA, USA.

[30] Mathworks. 2016. Polyspace Code Prover: prove the absence of run-time errors in software. http://www.mathworks.com/products/polyspace-code-prover/index.html. (2016). Accessed May 12, 2016.

[31] Jacob Matthews and Robert Bruce Findler. 2007. Operational Semantics for Multi-language Programs. In *POPL*.

[32] MicroFocus. 2016. DevPartner. http://www.borland.com/en-GB/Products/Software-Testing/Automated-Testing/Devpartner-Studio. (2016). Accessed May 6, 2016.

[33] Microsoft Corporation. 2016. C# Programming Guide. https://msdn.microsoft.com/en-us/library/67ef8sbd.aspx. (2016). Accessed May 13, 2016.

[34] 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 (PLDI '09)*. ACM, New York, NY, USA, 245–258. https://doi.org/10.1145/1542476.1542504

[35] George C Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 27, 3 (2005), 477–526.

[36] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 89–100. https://doi.org/10.1145/1250734.1250746

[37] NVDB. NIST vulnerability database. https://nvd.nist.gov. (????). Accessed May 17, 2017.

[38] Yutaka Oiwa. 2009. Implementation of the Memory-safe Full ANSI-C Compiler. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 259–269. https://doi.org/10.1145/1542476.1542505

[39] Oracle Corporation. 2016. Oracle Solaris Studio. http://www.oracle.com/technetwork/server-storage/solarisstudio/overview/index.html. (2016). Accessed May 6, 2016.

[40] Parasoft. 2016. Memory Error Detection. https://www.parasoft.com/capability/memory-error-detection/. (2016). Accessed May 6, 2016.

[41] Harish Patil and Charles Fischer. 1997. Low-cost, Concurrent Checking of Pointer and Array Accesses in C Programs. *Software: Practice & Experience* 27, 1 (Jan. 1997), 87–110.

[42] 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

[43] Rust-lang.org. 2016. Rust Documentation. https://www.rust-lang.org/documentation.html. (2016). Accessed May 13, 2016.

[44] Olatunji Ruwase and Monica S. Lam. 2004. A Practical Dynamic Buffer Overflow Detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*. Internet Society, Reston, VA, USA, 159–169. http://www.internetsociety.org/doc/practical-dynamic-buffer-overflow-detector.

[45] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC'12)*. USENIX Association, Berkeley, CA, USA, 28–28. http://dl.acm.org/citation.cfm?id=2342821.2342849

[46] Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages.

[47] Joseph L. Steffen. 1992. Adding Run-time Checking to the Portable C Compiler. *Softw. Pract. Exper.* 22, 4 (April 1992), 305–316. https://doi.org/10.1002/spe.4380220403

[48] Nikhil Swamy, Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. 2006. Safe Manual Memory Management in Cyclone. *Sci. of Comp. Programming* 62, 2 (Oct. 2006), 122–144. Special issue on memory management. Expands ISMM conference paper of the same name.

[49] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP '13)*. IEEE Computer Society, Washington, DC, USA, 48–62. https://doi.org/10.1109/SP.2013.13

[50] PaX Team. 2001. http://pax.grsecurity.net/docs/aslr.txt. (2001).

[51] Inc. Unicom Systems. 2016. PurifyPlus. http://unicomsi.com/products/purifyplus/. (2016). Accessed May 6, 2016.

[52] Valgrind. 2016. Valgrind. http://valgrind.org/. (2016). Accessed May 6, 2016.

[53] Philip Wadler and Robert Bruce Findler. 2009. Well-Typed Programs Can'T Be Blamed. In *ESOP*.

[54] Robert Wahbe, Steven Lucco, Thoma E. Anderson, and Susan L. Graham. 1993. Efficient Software-based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SOSP '93)*. ACM, New York, NY, USA, 203–216. https://doi.org/10.1145/168619.168635

[55] Wikipedia. 2016. Address space layout randomization. https://en.wikipedia.org/wiki/Address_space_layout_randomization. (2016). Accessed April 25, 2016.

[56] Yves Younan, Pieter Philippaerts, Lorenzo Cavallaro, R. Sekar, Frank Piessens, and Wouter Joosen. 2010. PAriCheck: An Efficient Pointer Arithmetic Checker for C Programs. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security (ASIACCS '10)*. ACM, New York, NY, USA, 145–156. https://doi.org/10.1145/1755688.1755707

[57] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. 2006. SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques. In *7th Symposium on Operating System Design and Implementation (OSDI'06)*. USENIX Association, Seattle, Washington.