Adding Spatial Memory Safety to EDK II through Checked C (Experience Paper)

SOURAG CHERUPATTAMOOLAYIL, Purdue University, USA
ARUNKUMAR BHATTAR, Purdue University, USA
CONNOR EVERETT GLOSNER, Purdue University, USA
ARAVIND MACHIRY, Purdue University, USA

Embedded software, predominantly written in C, is prone to memory corruption vulnerabilities due to spatial memory issues. Although various memory safety techniques exist, they are often unsuitable for embedded systems due to resource constraints and a lack of standardized OS support. Checked C, a backward-compatible, memory-safe C dialect, offers a potential solution by using pointer annotations for runtime checks to enhance spatial memory safety with minimal overhead. This paper provides the first experience report of porting EDK2 (an open-source UEFI implementation), an exemplary embedded codebase to Checked C, highlighting challenges and providing insights into applying Checked C to similar embedded systems. We also provide an enhanced automated annotation tool e3c, which improves the conversion rate by 25%, enabling easier conversion to Checked C.

CCS Concepts: • **Software and its engineering**; • **Security and privacy** → **Systems security**; *Software and application security*; *Embedded systems security*;

Additional Key Words and Phrases: Spatial Safety, UEFI, Checked C

ACM Reference Format:
Sourag Cherupattamoolayil, Arunkumar Bhattar, Connor Everett Glosner, and Aravind Machiry. 2025. Adding Spatial Memory Safety to EDK II through Checked C (Experience Paper). *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA054 (July 2025), 22 pages. https://doi.org/10.1145/3728929

1 Introduction

Most of the embedded software is in C because of its performance and ability to interact with hardware. Vulnerabilities due to memory corruption, especially spatial memory corruption, are still a major issue for C programs [6, 60, 69]. Embedded codebases are also riddled with memory safety issues [2, 18, 61, 63]. Although many mitigation and memory safety hardening techniques exist [51], they do not apply to embedded software because of their resource-constraint nature, close interaction with hardware, and the lack of standard Operating System (OS) abstraction.

Several industrial and research efforts, including CCured [39], Softbound [37], and ASAN [46], have investigated ways to better compile C programs with automatic safety enforcement. These approaches impose performance overheads deemed too high for deployment use, especially for

Authors' Contact Information: Sourag Cherupattamoolayil, Purdue University, West Lafayette, USA, scherupa@purdue.edu; Arunkumar Bhattar, Purdue University, West Lafayette, USA, bhattar1@purdue.edu; Connor Everett Glosner, Purdue University, West Lafayette, USA, cglosne@purdue.edu; Aravind Machiry, Purdue University, West Lafayette, USA, amachiry@purdue.edu.

ACM 2994-970X/2025/7-ARTISSTA054
https://doi.org/10.1145/3728929

resource-constrained embedded systems. One way to prevent such vulnerabilities is to use high-performant memory-safe languages, such as Rust [44]. However, as we explain in § 2.3.2, Rust has various issues in dealing with embedded system codebases. Furthermore, converting C to Rust is a tedious process. Although automated conversion tools exist, their effectiveness is very low.

Checked C [13], a backward compatible safe C dialect provides a feasible and practical alternative. Checked C uses special pointer annotations that capture additional information about the memory accessible through the pointer. The compiler will use these annotations to insert runtime checks that can prevent spatial safety vulnerabilities. Checked C uses compiler optimization techniques to reduce both memory and runtime overhead greatly. Recent work [12] shows that in practice, Checked C adds no overhead (neither memory nor runtime). However, a few aspects remain unclear. *How much effort does it take to convert a real embedded codebase to* Checked C*? What are the challenges?*

In this paper, we present our experience in converting EDK2 to Checked C. EDK2 is an open-source and well-maintained implementation of Unified Extensible Firmware Interface (UEFI) standard. EDK2 interacts with hardware and provides a standard interface for OS during boot and after boot to access system resources. EDK2 is a representative embedded codebase that closely interacts with hardware and has no standard OS interfaces. *We argue that our report will provide insights into converting large embedded codebases to* Checked C*.*

We observed that although Checked C is backward compatible, its annotations are not. Specifically, Checked C annotated code cannot be compiled with existing C compilers. We introduced backward-compatible annotations so that the converted code can be compiled using existing compilers by disabling the annotations with a preprocessor flag. We also made the necessary changes to the compiler front end.

(Methodology) We first tried 3c [31], an existing automated annotation tool, on EDK2 and identified various shortcomings. We created e3c, which is an enhanced variant 3c for EDK2 to improve the conversion rate (from 56% to 81%). We identified various EDK2 idioms that require semantic reasoning and are hard to convert to Checked C in an automated way. We manually converted such idioms by using appropriate Checked C features. We identified various interesting findings and lessons. For instance, (i) the interactive mode of 3c does not work because it produces incompilable code, various C idioms (common in embedded codebases) are not supported by Checked C, precise points-to information is still required for manual conversion.

In summary, the following are our contributions:

- We perform the first conversion of EDK2 codebase to Checked C and make it open-source.
- We identified various shortcomings of the existing automated annotation tool, 3c, and made enhancements to create e3c with better conversion effectiveness.
- We identified several code idioms that could not be handled through automation and had to be manually converted. Our findings indicate automation opportunities and the need for specialized techniques, indicating research opportunities.
- We identified various features in Checked C specification that still need to be implemented in the compiler. We also identified several code idioms unsupported in Checked C, indicating the need for new features. Our findings also shed light on challenges to consider for currently ongoing efforts of automatically converting C to safer variants, such as Rust.

All our findings are acknowledged by the Checked C team, and our enhancements to 3c (automated annotated tool) are accepted by the corresponding developers. We envision that our report provides insights into the benefits, challenges, and open problems in converting large C codebases to Checked C and, more generally, in converting C to other safe languages such as Rust.

2 Background and Motivation

We provide the necessary background to understand the rest of the paper and motivation for our work.

2.1 Checked C

Checked C [9, 13] extends C with support for *checked pointers*. Specifically, `ptr`<T> (*ptr*), `array_ptr`<T> (*arr*), and `nt_array_ptr`<T> (*ntarr*), which describe pointers to a single element, an array of elements, and a null-terminated array of elements of type T, respectively. Both *arr* and *ntarr* pointers have an associated *bound* that defines the range of memory referenced by the pointer. Here are the three different ways to specify the bounds for a pointer `p`; the corresponding memory region is on the right-hand side:

`array_ptr`<T> `p: count(`n`)`	[p, p+**sizeof**(T) $\times$ n)
`array_ptr`<T> `p: byte_count(`b`)`	[p, p+b)
`array_ptr`<T> `p: bounds(`x, y`)`	[x, y)

Bounds expressions, like the n in `count(`n`)` above, may refer to in-scope variables; **`struct`** members can refer to adjacent fields in bounds expressions. For instance, consider the following bounds annotation:

```
struct EFI_CAPSULE_BLOCK {
  array_ptr<FVBLOCK> BlockPtr : count(NumBlocks);
  int NumBlocks;
};
```

This indicates the bounds for all objects (`obj`) whose type is **`struct EFI_CAPSULE_BLOCK`**, the bounds of `obj.BlockPtr` are:

`[obj.BlockPtr, obj.BlockPtr + sizeof(FVBLOCK) * obj.NumBlocks)`

The interpretation of an *ntarr*'s bounds is similar, but the range can extend further to the right, until a `NULL` terminator is reached (*i.e.,* the `NULL` is not within the bounds). Checked C also supports interface types (`itypes`), flow-sensitive bounds, and various other constructs to make it easy to modify C code with Checked pointers [9].

Low Overhead Spatial Safety: The Checked C compiler will instrument the program at checked pointer dereferences (load and store) to confirm that (a) the pointer is *not NULL* and (b) that (if an *arr* or *ntarr*) *the dereference is within the range of the declared bounds.* For instance, in the code `if (n>0) a[n-1] = ...` the write is via address α = `a + sizeof(int)` $\times$ `(n-1)`. If the bounds of `a` are `count(u)`, the inserted check will confirm that prior to dereference `a` $\leq$ α $<$ `a + sizeof(int)` $\times$ `u`. Failed checks throw an exception. The vulnerability in Listing 1 shows a complex case of buffer-overflow vulnerability despite many checks that try to prevent it.

```
// void CopyMem(array_ptr<char> dst : byte_count(l),
//              ..., uint l);
....
FMBase = *MemoryBase; FMSize = *MemorySize;
NumBlocks = 0;
while(...blocks end..) {
  ...
  // Integer overflow of DSize
  ⚠DSize += CBlock->Length;
  NumBlocks++;
}
CSize = NumBlocks*sizeof(CAP_BLOCK);
// Integer overflow of: CSize + DSize
// Check can be bypassed because of integer overflow.
⚠if (FMSize <= (CSize + DSize)) return BUFF_SMALL;
// Copy all blocks into DestPtr
DestPtr = FMBase;
// array_ptr<char> DestPtr :
//   bounds(FMBase, FMBase + FMSIZE) = FMBase;

while (..blocks end..) {
 // Buffer overflow of memory in DestPtr
 🐞CopyMem (DestPtr, CBlock->Union.DataBlock,
          CBlock->Length);
 // Pointer can go out of bounds.
 DestPtr += CBlock->Length; }
```

Listing 1. Integer Overflow Leading to Buffer Overflow in EDK2 (VU-552286) and the Checked C Annotations (Highlighted) That Would Have Prevented It.

The highlighted Checked C annotations (in
Listing 1) would have prevented that vulnerability (even in the absence of explicit checks). Specifically, the annotations cause the compiler to check at runtime that `DestPtr` passed to `CopyMem` is within its bounds and has at least `CBlock->Length` bytes, thus preventing the buffer overflow vulnerability. Checked C achieves low overhead by optimizing most of the bounds checks through smart static analyses. In fact, *a Checked C port of FreeBSD's UDP and IP stack was found to impose no overhead at all [12].*

Restrictions: Checked C achieves its soundness guarantees by restricting the usage of Checked pointers and variables used in bounds expressions. For instance, only non-modifying expressions can be used in bounds declaration. Consequently, bounds declaration such as: `byte_count(func(ptr))` is invalid because the function call (`func(ptr)`) is not non-modifying expression.

Similarly, direct assignment between checked and non-checked (or regular) pointers is not allowed, and the Checked C compiler will reject the corresponding program.

2.1.1 Automated Conversion using 3c*:* The Checked C annotations need to be explicitly added to the codebase, which might require considerable manual effort. Recently, Machiry *et al.,* created 3c [31], an open-source tool to interactively add Checked C annotations to a given codebase. 3c uses a set of inference techniques (*i.e.,* type-inference and bounds-inference) to infer Checked C types for pointers in the code and adds annotations through source code rewriting. Furthermore, 3c claims to support interactive conversion where manual refactoring could aid in conversion.

2.2 Unified Extensible Firmware Interface (UEFI)

UEFI is a specification that defines interfaces for different firmware drivers and components, which is used for generalizing the interaction between different drivers or with higher-level software(*e.g.,* Operating System (OS) or hypervisors)[20]. UEFI standardizes the boot process and enables bootloader-independent OS design. Specifically, UEFI enables a standard interface for OS to access system resources in the pre-boot environment. This enables OS designers to be agnostic to the internal workings of the bootloader, provided it exposes the UEFI interface.

UEFI is being increasingly adopted in both traditional and embedded devices. Almost all the latest desktop-class and server-class devices from popular vendors, such as Dell, Hewlett Packard, Microsoft, and Apple, use a UEFI-compatible bootloader. UEFI standard [19] is maintained by a board of directors from various hardware and software companies, such as Intel, ARM, Insyde, Lenovo, AMD, AMI, Apple, Dell, Hewlett Packard Enterprise, HP Inc, Microsoft, and Phoenix.

2.3 EDK2

Extensible Framework Interface **D**evelopment **K**it - version **2** (EDK2) is the most popular open-source implementation of the UEFI that was first developed by Intel and then, later, was re-organized under the TianoCore community, which is responsible for many UEFI related projects (e.g. `edk2-platforms`[57], `edk2-pytool-library`[58]). EDK2 is maintained by roughly 19 companies (Microsoft, Intel, 9 Elements, StarLabs, AMD, Red Hat, ARM, AMI, NVIDIA, Loongson, Ventana Micro, Xen Project, Oracle, Suse, Google, freebsd, bsdio, Byosoft, Apple), along with many independent developers, which emphasizes its footprint throughout the UEFI community[56].

Software Architecture: EDK2 codebase is modularly organized and is composed of a set of modules or packages, *e.g.,* `MdePkg`, `NetworkPkg`. Each package takes care of certain functionality. For example, all of the security-related features are found in `SecurityPkg`, while network boot and other network features can be found in `NetworkPkg`. EDK2 can be built using several configurations, each composed of a set of modules. For instance, building EDK2 for emulator includes `EmulatorPkg` module. Building for a real target (*e.g.,* x64) does not include this module. There are some modules, such as `MdePkg` and `MdeModulePkg`, that implement certain base functionalities (*e.g.,* core memory management) and are available in all configurations.

There are many popular chip manufacturers that support mainline EDK2[56] (*e.g.,* AMD, Ampere, ARM, BeagleBoard, etc.[57]) or have custom forks of mainline (*e.g.,* ARM[4], NVIDIA[40], Microsoft[35]). EDK2 is the first piece of software that runs when the system boots and plays an important role in establishing the root of trust. Vulnerabilities in EDK2 could compromise the whole system's security. Furthermore, the large deployment of EDK2 makes such vulnerabilities highly impactful.

2.3.1 *Importance of Memory Safety in* EDK2.

The current EDK2 codebase [56] is written in C and contains over ∼1.2M lines of C code. However, a previous study [62] shows that code written in C is prone to vulnerabilities, especially memory corruption vulnerabilities. Recent works [21, 33, 52] have shown the prevalence of severe security vulnerabilities in EDK2 components. For instance, recently, Binarly found 24 severe security vulnerabilities in image parsing components of EDK2, dubbed LogoFail [52].

Prevalence of Memory Safety Issues: We performed a simple vulnerability study of all previously reported public vulnerabilities. Specifically, we searched the CVE database for all bugs related to EDK2 (through a keyword-based search). We found a total of 155 vulnerabilities and manually analyzed them to find their root causes. *We found that the majority (58%) of these previously known vulnerabilities in* EDK2 *are spatial memory safety vulnerabilities*, *i.e.,* buffer-overflow, out-of-bounds access, `NULL`-ptr dereference. Listing 1 shows one such example.

Lack of Detection and Mitigation Techniques: Despite various efforts [17, 64], the official EDK2 does not have a robust implementation of security hardening techniques, such as Address Space Layout Randomization (ASLR) [47], stack canaries [25, 65], deployable on real devices. On the other hand, there are no specialized vulnerability detection techniques targeting EDK2. Most of the existing works [66, 67] are mainly for System Management Interrupt (SMI) handlers, which are a special set of handlers installed by EDK2 and can be invoked by OS through SMI interrupts. Recently, Shafiuzzaman *et al.,* [48] designed a specialized static analysis technique for EDK2, but this work requires designing vulnerability patterns — a known tedious task.

2.3.2 *Ensuring Memory Safety.*

There are several ways to prevent memory safety vulnerabilities. Memory safety retrofitting techniques, such as CETS [38], prevent vulnerabilities through compiler instrumentation. Specifically, for all memory objects, these techniques track additional metadata that can be used to check and prevent invalid memory accesses. However, these techniques have a very high overhead, which is inapplicable for EDK2 with the tight memory layout. Furthermore, these techniques require certain OS abstractions, such as virtual memory, which do not exist in the boot environment.

High-performant memory-safe languages, such as Rust, can be used to ensure memory safety. However, this requires significant manual rewriting effort, and automated C to Rust is a hard problem [16, 29, 70]. Furthermore, Rust, with its complex lifetime semantics [41], has a steep learning curve[59] and raises maintainability issues. Finally, recent work [49] shows that the current state of Rust has various shortcomings to be used in embedded (*i.e.,* non-OS dependent) environment. Note that although there exists a Rust-based EDK2 [1, 53, 55], it is just a wrapper on top of the original EDK2.

Checked C (§ 2.1), with its C semantics, backward compatibility, and low overhead, provides the best alternative to add memory safety. As shown by our vulnerability study (§ 2.3.1), most of the vulnerabilities are spatial safety issues, and Checked C enables spatial memory safety with almost no overhead. Second, backward compatibility also enables modular conversion. Specifically, only certain modules (*e.g.,* high-risk) can be annotated with Checked C annotations, which will prevent spatial memory safety issues in those modules. Finally, the automated annotation tool 3C could help in automated rewriting, thus reducing manual effort.

Existing works on converting to Checked C are evaluated on regular (non-embedded) codebases. However, the complex software architecture and event-driven nature of EDK2 codebase pose unique challenges. *What are the challenges in converting such a large embedded (i.e., OS-independent) codebase to* Checked C*?* We argue that the insights from our experience report will serve as guidance for future conversion efforts of embedded codebases and ongoing efforts to convert C to Rust.

3 Enhancing Backward Compatibility of Checked C Annotations

Although Checked C is backward compatible, its type annotations are not syntactically backward compatible. Specifically, Checked C annotated C code cannot be compiled with existing C compilers, such as GCC. This limitation heavily restricts portability and poses a significant challenge for adoption, especially in modular conversion scenarios. Developers may want to use other C compilers on Checked C annotated code for various reasons. For instance, as we discuss in § 5.1, Checked C compiler might have bugs, or the compiler may not support certain non-standard C features (*e.g.*, variable-length arrays as `struct` fields), which are common in embedded codebases [50].

We want to have Checked C *annotations that are syntactically backward compatible, i.e., the annotated code should be compilable with existing compilers (albeit with no spatial safety guarantees). But should have the expected spatial safety guarantees when compiled with* Checked C *compiler.* We take inspiration from EDK2 parameter annotations, *i.e.*, `IN`, and `OUT`, which are used to mark input and output parameters [54]. During compilation, these annotations will be erased through pre-processor directives, *e.g.*, `#define IN`. A similar technique is used for `__user` and `__kernel` annotations [27] in Linux Kernel code.

We came up with similar *erasable* annotations for Checked C as illustrated in Listing 3, where the original annotations are shown in Listing 2. The new annotations will be erased when non-Checked C compilers is used, *i.e.*, the symbol `CHECKED_C` is not defined.

```
int main(void) {
    int data[5] = {1, 2, 3, 4, 5};
    void *raw_ptr = data;

        _Ptr<char> star = '*';

    _Array_ptr<int> arr : count(5) =
        _Assume_bounds_cast<_Array_ptr<int>>(raw_ptr,
    count(5));

    process_array(arr, 5, star);
    return 0;
}
```

Listing 2. Standard Checked C Annotations.

```
#ifdef CHECKED_C
    #define _Assume_bounds_cast_M(T, e1, ... ) \
        _Assume_bounds_cast<T>(e1, __VA_ARGS__)
#else
    #define _Single
    #define _Array
    #define _Assume_bounds_cast_M(T, e1, ... ) (e1)
    #define _Count(e)
#endif

int main(void) {
    int data[5] = {1, 2, 3, 4, 5};
    void *raw_ptr = data;

        char* _Single star = '*';

    int* _Array arr _Count(5) =
        _Assume_bounds_cast_M(int* _Array, raw_ptr,
    _Count(5));

    process_array(arr, 5, star);
    return 0;
}
```

Listing 3. Macro-Based Erasable Checked C Syntax.

3.1 Compiler Frontend Modifications

We extended the grammar (in frontend) of Checked C compiler (which is implemented in LLVM [28]) to recognize the new macro-based annotations. We made a total of 385 SLoC (Source Lines of

Code) modifications. We also recreated existing compiler test cases with new syntax to ensure a robust implementation. Our changes are also integrated by the Checked C development team into their main repository.

3.2 Integration with 3c

To align with the enhancements in Checked C, we also updated 3c's rewriter (210 SLoC modifications) to support the new macro-based syntax. This enhancement allows users to choose between the old and new syntax formats when converting their codebases. By supporting the macro-based syntax, 3c facilitates seamless integration of Checked C features into existing projects while maintaining compatibility with standard C compilers.

4 Methodology

Our goal is to have EDK2 source code properly annotated with Checked C types, adhering to its typing rules and avoiding (or rather minimizing) *wild* (*i.e.,* non-Checked C or regular) pointers and unsafe casts. The Figure 1 shows the overview of our methodology.

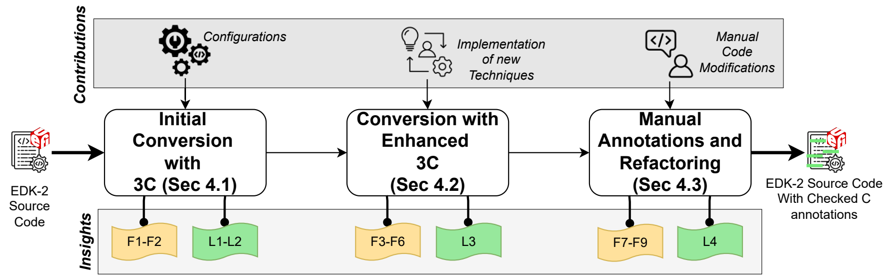

Contributions
Configurations
Implementation of new Techniques
Manual Code Modifications
EDK-2 Source Code
Initial Conversion with 3C (Sec 4.1)
Conversion with Enhanced 3C (Sec 4.2)
Manual Annotations and Refactoring (Sec 4.3)
EDK-2 Source Code With Checked C annotations
Insights
F1-F2
L1-L2
F3-F6
L3
F7-F9
L4

Fig. 1. Overview of Our Methodology.

First, we tried to automatically annotate EDK2 by using the existing 3c tool and identified various shortcomings and robustness issues in handling EDK2 codebase or, in general, embedded codebases. Second, we fixed various robustness issues and, added enhancements, and created e3c, which improved the conversion rate.

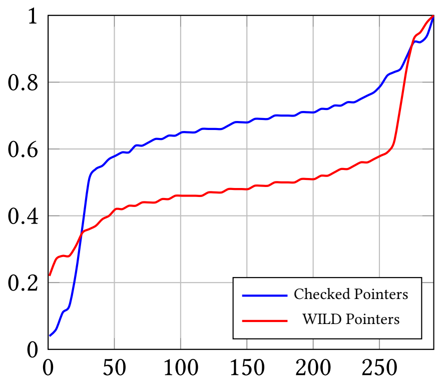
We found various EDK2 coding idioms that are hard to convert automatically. Finally, we tried to manually annotate the rest of the code by refactoring and using appropriate Checked C annotations. To balance time and effort, we picked 120 (out of the 293) commonly used modules from EDK2, with a total of 48,003 SLoC. These modules cover different categories of coding idioms and are representative of EDK2 codebase. In each of the above phases, we summarize our findings and learnings, which will serve as guidelines for future conversion efforts and action items for Checked C maintainers.

4.1 Initial Conversion using 3c

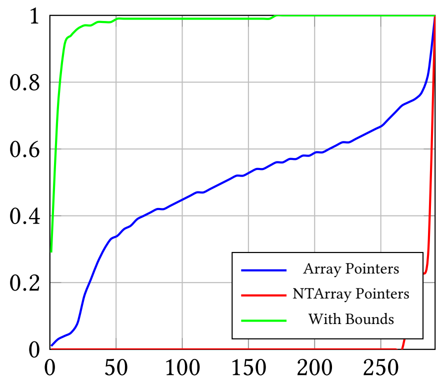
As mentioned before, recently, Machiry *et al.,* developed 3c, an automated C to Checked C conversion tool. In this initial phase, we describe our methodology and experience of using 3c on EDK2.

4.1.1 Configuration. The official documentation [7] of 3c mentions that we just need to provide `compile_command.json` of the entire codebase.

However, we couldn't run 3c on the entire EDK2 at once. This is because EDK2's codebase is organized into various modules, and functions with the same name might exist in different modules.



(a) Wild vs. Safe Pointers.



(b) Array vs. NTArray Pointers.

Fig. 2. Cumulative Distribution Function of Conversion Rate across all Modules after Initial Conversion.

For instance, the function `ProcessLibraryConstructorList` is present in both `AcpiPlatformDxe` and `PeiMain` with different signatures. Trying to compile multiple modules results in multiple definitions of a function. 3c can only handle cases where the function with multiple declarations has compatible types for parameters, which is not the case here. To circumvent this, we modified the build script of EDK2 so it generates the compilation database for each module separately. After this, we could run 3c on each module separately.

4.1.2 Robustness Issues. While working on improving 3c, we also identified a few issues that either caused 3c to crash or resulted in wrong inference and output.

The listing on the right shows an example code snippet causing 3c to crash. We also identified that 3c fails to properly re-write typedef to function pointer having variadic arguments. 3c also had a bounds inference issue when there are multiple conflicting bounds coming from the neighbors of a pointer that needed bounds.

```
int main() {
    char *ptr = 0;
    int size = ptr;
     ptr++;
}
```

We added a new class that resolves this conflict and infers the correct bounds. We made minor code-level fixes (~160 SLoC) to handle all these robustness issues.

Experience with Interactivity: The authors of 3c promise an interactive approach in using their tool. They claim that we can run 3c once, get the output, fix a few pointers causing the major issue, and then rerun 3c to generate code with even more annotations. However, this does not work in practice as 3c could generate partially annotated EDK2 code that fails to compile. Consequently, it becomes hard to perform any manual annotations that require compiler interaction. Despite the tool's claims, we were not able to use the interactivity feature of 3c.

```
BOOLEAN
EFIAPI
IsValidMicrocode (
  IN CPU_MICROCODE_HEADER *Microcode
       _Itype(CPU_MICROCODE_HEADER *_Array)
       _Count(MicrocodeLength),
  IN UINTN MicrocodeLength,
  IN UINT32 MinimumRevision,
  IN EDKII_PEI_MICROCODE_CPU_ID *_Array MicrocodeCpuIds
       _Count(MicrocodeCpuIdCount),
  IN UINTN MicrocodeCpuIdCount,
  IN BOOLEAN VerifyChecksum)
{
  UINT32 TotalSize;
...
  TotalSize = GetMicrocodeLength(Microcode);
...
}
```

Listing 4. Example Where 3c Inferred Correct Bounds and Type.

Table 1. Overall Pointer Stats after Each Conversion Stage.

Total Pointers	Checked Pointers (% of Pointers)	Ptrs (% of Checked)	Arrays Total (% of Checked)	Arrays With bounds (% of Total)	NT Arrays Total (% of Checked)	NT Arrays With bounds (% of Total)
112,410	**3c**					
	62,860 (56%)	26,985 (43%)	35,874 (57%)	35,668 (99%)	1 (0.002%)	0 (0%))
	e3c					
	90,798 (81%)	54,851 (60%)	35,944 (40%)	35,712 (99%)	3 (0.003%)	0 (0%)
	Manual					
	99,582 (89%)	62,225(62%)	37,334 (37%)	36,968 (99%)	23 (0.023%))	9 (39%)

4.1.3 Conversion Results. The top row of Table 1 shows the conversion results. The overall conversion rate is 56% (10% lower than reported numbers in the original 3c paper). The Figure 2a shows the CDF of the conversion rate across all modules. The relatively flat blue line indicates that the conversion rate is almost the same across all modules. The percentage of *arr* and *ntarr* pointers are 57% (47% higher than the reported numbers) and 0.02% (9% lower than the reported numbers), respectively. The bounds detection is at 99%, much better than the reported numbers. Listing 4 shows an instance where 3c was able to accurately infer *arr* pointer and its bounds. The Figure 2b shows the CDF of *arr* and *ntarr* detection across all modules. The relatively diagonal *arr* line indicates a lot of diversity in the number of array pointers across all modules. The almost vertical *ntarr* line indicates that only a few modules contain *ntarr*s. Similarly, the almost flat bounds lines indicate that the performance of bounds inference is almost the same across all modules, and it is high. In summary, the performance of 3c on EDK2 differs significantly from that of the reported numbers (on traditional codebases). For instance, the total conversion rate dropped by 10%, but the bounds inference rate increased by 30%.

Conversion Failures: In total, 3c failed to convert 48.14% of pointers. Table 2 shows the top 5 reasons and their contribution to the failure percentage. Note that these percentages are not exclusive. There can be a single pointer that failed conversion because of multiple reasons.

The interface-driven nature of EDK2 results in the prevalent use of `void*` pointers. Although Checked C supports generic pointers, 3c does not properly annotate `void*` with generic types, resulting in the highest impact on failed conversions.

Finding 1 (F1): It is hard to use the claimed interactive mode of 3c because of its generation of incompilable code.
Finding 2 (F2): Effectiveness of 3c on EDK2 differs from the reported numbers. 3c suffers in overall conversion, but performs much better on bounds inference.

Learning 1 (L1): Special attention should be paid while using 3c on multi-module projects. One possible way is to use 3c on a per-module basis and manually merge the results.
Learning 2 (L2): Robustness fixes might be required when trying to apply 3c on large projects.

4.2 e3c: Enhanced 3c for EDK2

Some of the root causes are fundamental limitations of source code refactoring techniques, *e.g.*, rewriting pointers in macro, which is not feasible as frontend techniques work on post-preprocessed files. However, we found that other causes can be fixed by improving 3c. We focused on the top three root causes and designed techniques to handle them. We call this improved variant *Enhanced* 3c *(*e3c*)*.

Table 2. Wildness Reason, Corresponding Contribution, and Example Code.

Reason for WILDness	Percentage	Count	Example Code
Default void* type	41.36%	90,754	`EFI_STATUS` `FvBufPackageFreeformRawFile(` `IN VOID* RawData,` `OUT VOID** FfsFile` `) { ... }`
Unchecked pointer in parameter of undefined function	36.91%	80,987	`UINT32` `EFIAPI` `ParseAcpiBitFields (` `IN CONST CHAR8 *AsciiName OPTIONAL,` `IN UINT8 *Ptr,` `IN UINT32 Length` `);` `// No definition.`
Invalid Cast	15.61%	34,250	`struct A {` `SHORT a;` `};` `struct B {` `CHAR b;` `};` `INT main() {` `struct A *a;` `struct B *b = (struct B *)a;` `return 0;` `}`
Pointer in Macro	2.63%	5,783	`#define MACRO(ptr) \` `CHAR* p = ptr; \` `FUNC(p);`
Assigning from 1 depth pointer to 2 depth pointer	1.02%	2,259	`EFI_STATUS` `GetBistInfoFromPpi (` `OUT VOID **BistInformationData` `) { ... }` `Status = GetBistInfoFromPpi (` `(VOID *)&SecPlatformInformation2` `);`

All the improvements made to 3C excluding *Handling directional qualifiers* from § 4.2.4 can be generalized to other code bases as well.

4.2.1 `Void *` Parameters and Returns. 3C by default considers all `void*` pointers as WILD, *i.e.,* will add $p \sqsubseteq WILD$ constraint to the corresponding constraint variable (*i.e.,* p). However, automatically identifying accurate and safe Checked C annotations for general `void*` pointers is infeasible. But, we can use generic types for function parameters and returns if they are used safely. To handle this, we first removed the default WILD constraint on `void*` parameters and returns. Next, we split the external and internal constraints used by 3C for function parameters and returns. This way, even if the internal constraint is solved to WILD, we can insert an `itype` to a generic type. Since 3C's rewriter already supported generic type rewriting as 3C can run on partially converted files, we left the rewriter as it is. The top row of Table 3 shows an example of the improvement. The last column shows the improved annotation (E3C) while the middle row shows the failed annotation by 3C.

4.2.2 Invalid Casts. 3C only handles casts between built-in types, considers all other casts unsafe, and makes the corresponding pointers WILD. However, this is overly conservative as the casts between composite (*i.e.,* `struct`) types, *i.e.,* from larger to smaller types, could also be valid. We added support for struct types. Given D_t (destination `struct` type) and S_t (source `struct` type), we aim to check *if less memory can be accessed using D_t compared to S_t*. Specifically, we consider cast from S_t to D_t is safe for the following cases:

- If D_t has no pointer members, then sizeof(D_t) should be less than sizeof(S_t).
- If both D_t and S_t have pointer members, then we check that the number of pointer members in D_t is less than or equal to those in S_t. We also check if the offset of all pointer members in D_t matches with that of S_t.

For the rest of the cases, we consider the cast to be unsafe, *e.g.,* if D_t has pointer members but S_t doesn't. We handle nested `struct`s by extending the above comparison to the inner `struct` types as well. The second row of Table 3 shows an example of this enhancement.

4.2.3 Adding Annotations to EDK2 *Headers.* As mentioned in § 2.1, we can use `itype` for external functions to enable passing Checked pointers as arguments without violating typing rules. Checked C team provided `itype`-annotated headers for commonly used external functions, *e.g.,* those in standard C library (`libc`). However, EDK2 uses custom standard library functions without `itype`s. Consequently, 3c treats any pointer passed as an argument to such functions to be WILD, resulting in a high failure conversion rate as shown in the second row of Table 2. To handle this, we created an alternate `itype` annotated variants of all the custom library functions. Given a large number of functions, we leveraged Large Language Models (LLMs). Specifically, Claude [3] (We also tried other models, but Claude seemed to give the best results) in one-shot mode. We gave examples of existing `itype` annotated `libc` functions along with the Checked C specifications document and asked the LLM to annotate EDK2 library functions. We manually verified that the annotations were valid. The third row of Table 3 shows an example of such an annotated function. This saved us a considerable amount of manual annotation effort.

4.2.4 Other Enhancements. We also made the following enhancements to 3c to improve its bounds inferences and source rewriting.

- **Handling Single Element Arrays (*e.g.,* `fld[1]`):** 3c's bounds inference works by finding seed bounds (*e.g.,* declared in source code or inferred through malloc) and propagating them across pointers. However, the inference algorithm aggressively propagates bounds, resulting in incorrect bounds in the presence of *elastic* `structs`, *i.e.,* structures with the last member being an array of size 1 (one). Consider the code in the fourth row of Table 3 where `func` has two call sites one with `ptr` (without bounds) and the other with `test.buf` (last member of an elastic structure). In this case, 3c will use 1 as the seed bounds and propagate it to both the call-sites, resulting in incorrect bounds for `ptr` (as shown in the third column). We fixed the inference logic so that the propagation would not occur from elastic members (*i.e.,* arrays of size 1) to other pointers. The last column of the fourth row shows the result after our fix.
- **Handling Directional Qualifiers:** EDK2 uses directional qualifiers for its function parameters, *i.e.,* `IN` (for input parameters) and `OUT` (for output parameters), which gets removed by the pre-processor. 3c works on post-preprocessed code and does not see these qualifiers. Consequently, 3c fails to restore these qualifiers while adding Checked C annotations to function parameters. We fix this by recording these qualifiers through the source range of parameters and then using it while annotating them.
- **Minor Robustness Fixes:** We also made several robustness fixes to add support for C features. For instance, 3c had issues handling alias attributes (*i.e.,*`__attribute__ ((__alias__ (...)))`), where it considered multiple aliases as a single function.

Table 3. Summary of the Improvements (with Examples) Made in E3C.

Improvement	**Original code**	**3c**	**e3c**
`void *` Parameters and Returns (48 SLoC)	`EFI_STATUS InitializeCpu( IN VOID* ImageHandle, IN EFI_SYSTEM_TABLE* SystemTable) { ... }`	`EFI_STATUS InitializeCpu( IN VOID* ImageHandle, IN EFI_SYSTEM_TABLE* SystemTable) { ... }`	`_For_any(T) EFI_STATUS InitializeCpu( IN T* _Single ImageHandle, IN EFI_SYSTEM_TABLE* _Single SystemTable) { ... }`
Invalid Casts (68 SLoC)	`struct A { INT *b; CHAR c; CHAR d; }; struct B { INT *b; SHORT c; }; INT main() { struct A *a; struct B *b = (struct B *)a; return 0; }`	`struct A { INT *_Single b; CHAR c; CHAR d; }; struct B { INT *_Single b; SHORT c; }; INT main() { struct A *a; struct B *b = (struct B *)a; return 0; }`	`struct A { INT *_Single b; CHAR c; CHAR d; }; struct B { INT *_Single b; SHORT c; }; INT main() { struct A *_Single a = ((VOID *)0); struct B *_Single b = (struct B *_Single)a; return 0; }`
Adding Annotations to EDK2 Headers	`VOID AsmCpucfg ( IN UINT32 Index, OUT UINT32 *Data );`	`VOID AsmCpucfg ( IN UINT32 Index, OUT UINT32 *Data );`	`VOID AsmCpucfg ( IN UINT32 Index, OUT UINT32 *Data _Itype(UINT32 * _Single) );`
Handling single element arrays (12 SLoC)	`typedef struct { char buf[1]; }test; void func(char *ptr) { int l; ptr[l]; } int main() { char *ptr; char arr[1]; int n; test test; func(ptr); func(test.buf); // use ptr }`	`typedef struct { char buf _Checked[1]; }test; void func(char *_Array ptr _Count(1)) { int l; ptr[l]; } int main() { char *_Array ptr _Count(1) = 0; char arr _Checked[1]; test test = {}; func(ptr); func(test.buf); // use ptr }`	`typedef struct { char buf _Checked[1]; }test; void func(char *_Array ptr) { int l; ptr[l]; } int main() { char *_Array ptr = 0; char arr _Checked[1]; test test = {}; func(ptr); func(test.buf); // use ptr }`
Correcting `Assume_Bounds _Cast` (80 SLoC)	`BasePrintLibSPrint ( ValueBuffer, MAXIMUM_VALUE_CHARACTERS);`	`BasePrintLibSPrint ( _Assume_bounds_cast_M( CHAR8* _Array, ValueBuffer, bounds(unknown)), MAXIMUM_VALUE_CHARACTERS);`	`BasePrintLibSPrint ( _Assume_bounds_cast_M( CHAR8* _Array, ValueBuffer, count(MAXIMUM_VALUE_CHARACTERS)), MAXIMUM_VALUE_CHARACTERS)`

- **Correcting `Assume_Bounds_Cast`:** Checked C allows calls to Checked functions from within an Unchecked function. This allows a partially converted code base to have security assurances whenever execution is in Checked scope. However, the calls should use `_Assume_bounds_cast` to cast unchecked types into appropriate Checked types. The cast expression should also contain the bounds of the target type. However, 3c fails to add proper bounds and just adds `bounds(unknown)`. This results in compilation failure as we are trying to pass an argument of unknown bounds. We fixed this by adding appropriate bounds to the `_Assume_bounds_cast` expression. Specifically, we check if the callee argument has some known bounds. If yes, and if it is a constant bounds, we directly use it. If the bounds expression includes one of the parameters, we get the index of that parameter from the function definition and then get the corresponding argument at the call site. We will create the bounds expression using appropriate argument values. The fifth row of Table 3 shows this issue and correct bounds after our enhancement.

We are in the process of merging all our improvements to the mainline version of 3c.

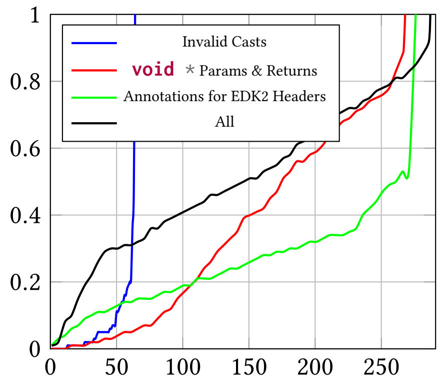


Fig. 3. CDF of improvements across all modules per enhancement

```
// With 3c
VOID
UiCustomizeFrontPage (
  IN EFI_HII_HANDLE  HiiHandle,
  IN VOID           *StartOpCodeHandle
  )
{
...
  UiCreateEmptyLine (HiiHandle, StartOpCodeHandle);
...
}

// With e3c
_For_any(T,U)
VOID
UiCustomizeFrontPage (
  IN T *_Single HiiHandle,
  IN U *_Single StartOpCodeHandle
  )
{
...
  UiCreateEmptyLine _TyArgs(T,U) (HiiHandle,
       StartOpCodeHandle);
...
}
```

Listing 5. Example code where 3c failed but E3c succeeded

4.2.5 *Conversion Results.*

The number under the E3c column in Table 1 shows the results of executing E3c on EDK2. *The overall conversion rate increased to 81% (from 56%) with an improvement of 25%.* Specifically, we were able to infer more single pointers (*i.e., ptrs*). Listing 5 shows an instance of failed conversion by 3c, but E3c was able to infer the types of both parameters accurately. The Figure 3 shows the CDF of the improvements achieved by each of our enhancements and combined (*i.e.,* All) improvements across all modules. We can see that 50% of the modules have 40% or more improved conversion.

4.2.6 *Residual* Wild *Pointers.*

Although E3c improved the conversion rate, it is not 100%. There are still *wild* pointers. The Table 4 shows the top three reasons, and the percentage reduction shows the decrease in its impact because of our enhancements. Even with our `void*` enhancement, the `void*` pointers still remain the top reason for failed conversion. The main reason for this is `void*` pointers inside structures (illustrated in the top row of Table 5). Checked C supports generic `struct` types; however, rewriting requires reasoning about the corresponding member's usages in

Table 4. Wildness Reason and Corresponding Reduction in Contribution after Our Enhancements (E3C).

Reason for WILDness	Percentage Reduction	Count
Default void* type	⬇ **2.53%**	88,449
Unchecked pointer in parameter of undefined function	⬇ **35.08%**	52,576
Invalid Cast	⬇ **15.52%**	28,932

a context-sensitive manner — an intractable task. This gets even more complicated when `struct` contains multiple `void*` fields. The second major reason is pointers in macro. As discussed before, CLANG-rewrite (on which E3C and 3C is built on) currently doesn't support the rewriting of macros.

Hence, any pointer in macro is considered *wild*. One possible way to tackle this is to infer the macro pointers just like any other pointers and then, during the rewrite stage, dump these into a file as hints, and developers can manually annotate them.

Finding 3 (F3): There exists codebase-specific improvement opportunities for 3C.
Finding 4 (F4): LLMs show promise in making simple CHECKED C annotations to external functions.
Finding 5 (F5): Simple enhancements to 3C can result in significant improvements (+25%) in conversion rate.
Finding 6 (F6): It is beneficial to systematically investigate 3C failures and devise piecewise solutions.

Learning 3 (L3): Despite the automation efforts, certain C code idioms (*e.g.*, macros) are inherently hard to automatically rewrite.

4.3 Manual Annotations

For the rest of the pointers, we resorted to manual annotations. The manual annotations can be categorized into mainly 3 categories. Table 5 summarizes these categories along with examples. We believe further enhancements can be made to E3C to help automate § 4.3.1 and § 4.3.2.

4.3.1 Generic `Structs`*.* Currently, 3C doesn't have support to handle `void *` inside structures. When it encounters one, it directly marks them as *wild*. But since CHECKED C supports generic structures, we had to manually convert all possible structures with `void *` to generic.

4.3.2 Local `Void *` *Variables.* Handling `void *` in a function body is a bit more complicated since there are no generic types for local variables in CHECKED C, and we had to check the variable's usage to figure out which type we should convert it into. After this, we had to update the type argument passed to the function that used the local `void *` variables as arguments.

4.3.3 Unary Operations on Bounded Pointers. Unary operations of pointers that use itself in its bounds declaration are not allowed in CHECKED C. This is because it is not possible for the compiler to reason whether the bounds will still hold after the unary operation. It is a pretty common practice in the EDK2 code to do unary operations on pointers whenever there is some kind of string manipulation required. As illustrated in the last row of Table 5, we fix these issues by first assigning the pointer to a temporary variable and using that temporary variable in the bounds expression of the original pointer.

Table 5. Manual Conversion Categories and Examples.

Category	**Original code**	**Manual Conversion**
Generic `struct`s	`struct st {` `void *ptr;` `int len;` `};` `int main() {` `char *p = "hello";` `struct st s = {.ptr = p, .len = 5};` `return 0;` `}`	`struct st _For_any(T) {` `T*_Array ptr _Byte_count(len);` `int len;` `};` `int main() {` `char *_Array p = "hello";` `struct st<char> s = {.ptr = p, .len = 5};` `return 0;` `}`
Local `void`* variables	`void func(void *ptr, int len) {` `char buf[10];` `strcpy(buf, ptr);` `}` `int main() {` `void *buf = malloc(20);` `func(buf, 20);` `}`	`void func(char *_Array ptr _Byte_count(len), int len) {` `char buf[10];` `strcpy(buf, ptr);` `}` `int main() {` `char * _Array buf _Byte_count(20) = malloc<char>(20);` `func(buf, 20);` `}`
Unary operations on bounded pointers	`void func() {` `char *_Array ptr _Byte_count(10) = malloc<char>(10);` `ptr++;` `}`	`void func() {` `char *_Array tmp _Byte_count(10) = malloc<char>(10);` `char *_Array ptr _Bounds(tmp, tmp + 10) = tmp;` `ptr++;` `}`

4.3.4 Auxiliary Fixes. As mentioned in § 4.1.2, 3c (and e3c) misses certain annotations resulting in Checked C type errors. One of the main reasons is missing bounds. For instance, there might be a *arr* pointer without bounds, and Checked C compiler does not allow dereferencing such pointers. We had to manually fix all these errors by finding the appropriate bounds and adding corresponding annotations. There were also some other errors, which are actually compiler warnings, but since EDK2 has -*Werror* enabled by default, these warnings become errors.

4.3.5 Annotation Effort. On average, it took around 20 minutes per module to fix all the compilation errors from the files that e3c generated. This was the first step in doing the manual conversion. *After this, we spend around one hour per module to convert as many* wild *pointers as possible.* The residual wild pointers are because of the cases that are impossible to convert to Checked C because of compiler errors or unsupported features, as will be discussed in § 4.3.7.

4.3.6 Final Conversion Results. The last column of Table 1 shows the results after manual conversion. In total, we annotated an additional 8,784 pointers across all modules, resulting in a total of ~13,000 SLoC modification. However, there are still pointers that we could not convert.

4.3.7 Remaining wild *Pointers.* There were cases that were hard to convert for the following three reasons:

Function Parameters with No Bounds: There are functions that has *arr* parameters but do not have bounds, *e.g.,* `func(int *p _Array)`. These functions have their address taken, *i.e.,* can be invoked indirectly through function pointers, *e.g.,* `= &func;`. We cannot just change the function signature to explicitly pass bounds, *i.e.,* `func(int *p _Array _Count(n), unsigned n)`. *This requires changing all call sites, including indirect calls (i.e., through function pointers), which is impossible without precise points-to-information — an intractable problem [42]* Avoiding this requires major refactoring of EDK2 codebase.

Unimplemented Checked C **Features:** Although in typical programs, performing arithmetic on `void *` is not common, it becomes logical to allow arithmetic on them once they are converted to generic types, without needing to cast them to another type. Even though this is mentioned as a feature in the Checked C spec, it is not yet implemented in the compiler. Once implemented, this will be a useful feature.

Unsupported C Idioms: Checked C has several unsupported C idioms. One such important idiom is the use of `void *` parameters in `typedef` of function pointers, *e.g.,*`typedef void (*func)(void *p);`. Since Checked C doesn't support having generic types of `typedef`s, all these function declarations couldn't be made generic. Subsequently, all functions that we assign to this `typedef` also needed to remain unchecked. We propose the addition of this feature, which will be useful in codebases like EDK2, where this pattern is quite common.

Finding 7 (F7): Despite automation tools, we still need significant manual effort to annotate a real-world codebase with Checked C annotations.
Finding 8 (F8): Several Checked C features are not implemented in the current version of the compiler.
Finding 9 (F9): A few C idioms are not supported by Checked C. Developers should pay attention to the prevalence of these unsupported idioms before endeavoring into conversion efforts.

Learning 4 (L4): Precise points-to information is a linchpin problem even for manual conversion of EDK2 codebase to safe C dialects.

5 Discussion

5.1 Compiler Bugs

During our effort to convert EDK2 into Checked C, we also discovered several issues with the compiler. We had to use the old syntax in some parts of the manual conversion because the compiler could not handle some cases with the new syntax. For *e.g.,* new syntax for generic `struct`s is not yet implemented in the compiler. We also discovered that the compiler is not able to tell `m*n` is the same as `n*m` when this expression is used in bounds expression thereby leading to errors. These issues show that the type checker of the compiler needs some fixing.

5.2 Performance

Even with 120 modules converted to Checked C, the boot-up time is virtually the same as that of the original code. The size of the final firmware image and the individual `.efi` files remained the same after adding the annotations. This is because the efi files are padded to a multiple of 512 bytes and the checks added by Checked C are always less than that size. This shows how good Checked C is at minimal time and space overhead.

5.3 Security Impact

There were 9 bugs out of the 155 we collected which were from the open source part of EDK2. Of these, all spatial safety vulnerabilities were prevented by the annotations added by 3c, e3c, or by manual conversion. We believe that even other bugs that we couldn't verify due to it being proprietary could have been prevented.

5.4 Implications on C to Rust Conversion

Recently, there has been an increased interest in automated C to Rust conversion techniques [11]. Although Rust is not exactly the same as Checked C, we believe that the observations and findings in this work will serve as insights into current Rust conversion efforts. First, the examples in Table 2 and Table 3 illustrate the hard-to-handle C idioms that should be considered while converting to Rust. Second, the complex C constructs (§ 4.3) that require manual refactoring provide challenging cases for Rust's conversion. Finally, we envision that our findings will serve as motivation and examples to create C to Rust evaluation datasets, as there exists no such dataset.

5.4.1 Checked C as an Intermediate Step. Converting C to Rust is a known hard problem. There are several interrelated technical challenges in converting C to Equivalent and Idiomatic Rust code, such as identifying array pointers, lifetimes [41], ownership, and typing issues such as generics. Existing techniques [16, 24, 70] try to tackle individual challenges in the presence of all challenges. However, interdependency between these challenges (Figure 4) results in known hard problems, such as precise points-to-analysis[15]. For instance, using rich Rust types, *e.g.,* `vec`, requires ensuring that all aliases to the corresponding C pointer are appropriately updated. Consequently, existing techniques result in only minor improvements.

We argue that Checked C provides a practical intermediate stage to enable more effective Rust conversion techniques. Specifically, we envision a staged conversion (Figure 4) where we will first convert C to Checked C and then convert the resulting code to Rust. In principle, such a staged approach decouples the challenges, enabling us to develop effective techniques to solve them.

From a technical standpoint, first, Checked C types can be directly converted to rich Rust types. Specifically, all *arr* and *ntarr* pointers can be converted directly to `Vec` or fixed-length arrays depending on the bounds. For instance, `int *p _Array _Count(i+1);...; *(p+i) =` will be converted to `let mut p = Vec::with_capacity(i+1); ...; p[i] =`. Similarly, Checked C generic types (*i.e.,* `structs` and `function`) can be converted to the corresponding Rust generic types.

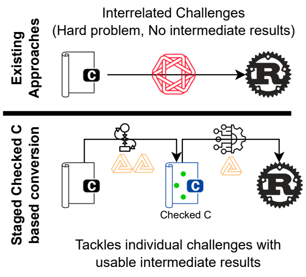

Interrelated Challenges
(Hard problem, No intermediate results)
Existing Approaches
C
Staged Checked C based conversion
C
C
Checked C
Tackles individual challenges with usable intermediate results

Fig. 4. Existing v/s Staged Conversion

Second, Checked C typing rules (§ 2.1) restrict certain aliases, which will reduce the imprecision of alias analysis, enabling more accurate points-to sets.

From a practical standpoint, Checked C is both syntactically and semantically backward compatible. Our new syntax (§ 3.1) makes Checked C annotations *erasable* (*i.e.,* can be disabled through pre-processor directives), enabling the code to be compiled with existing compilers. After converting to Checked C, developers can continue maintaining Checked C code without requiring any changes to their other build infrastructure, and the backward compatibility enables modular conversion (*i.e.,* individual functions, files, etc). Developers interested in spatial safety can just replace their compiler with Checked C compiler (*e.g.,* by a symlink) — again without changing anything else.

6 Limitations

We note the following limitations of our study:

- **Generalizability.** Given that 3C and Checked C involve code-level modification, our findings on EDK2 may not generalize to other system projects (*e.g.,* Linux kernel) as they might have different coding styles and idioms. However, we believe that the information about certain fundamental issues (*e.g.,* `void*` pointers) is still valuable for such conversion efforts.
- **LLMs Assisted Conversion.** Recent work [36] showed that LLM-assisted C to Checked C conversion slightly improves the conversion rate of 3C. However, our preliminary experiments with ChatGPT (LLM) on EDK2 showed poor results. But, there could be other fine-tuning or re-training approaches that could improve the results.
- **Temporal Safety.** Recently, Zhou *et al.,* [71] added automated temporal safety retrofitting support to Checked C. We did not use it as it is not backward compatible, and the features are not yet integrated into the Checked C mainline repository.

7 Related Work

C is one of the most widely used languages in the open source community [30]. However, it also inherently has the greatest potential for memory safety issues. Among all the issues found or reported in any C project, memory safety problems are the most prevalent [62]. Static [14, 23, 68] and Dynamic analysis [5, 8, 43] tools can be used to detect memory corruption issues, but these either have high false-positive rates [22] or are not scalable to large code bases.

Rust is a popular alternative to C, but its adoption is slow due to the significant amount of work needed to port large C projects. Even with the help of C-to-Rust tools, getting idiomatic Rust equivalent to C is hard as mentioned in § 2.3.2. It is not just the porting that is difficult in using Rust, its ownership system and borrow checker [34, 45] represent a significant paradigm shift from C's memory model. This requires the developer to restructure code extensively. It is also possible that some C idioms might not be directly supported in Rust, which also requires code restructuring.

Rather than porting C to Rust, developers can also switch to safer dialects of C. This would require much less effort than converting to Rust. Several approaches have emerged over the years. CCured [39] combines type inference with runtime checks to enforce memory safety, but requires whole-program analysis and changes data representations. AddressSanitizer [46] and SoftBound [37] can detect memory errors but incur substantial performance overhead. Cyclone [26] was built to be a good extension for C, but as discovered in [32], it is not a good candidate. Another extension to C is Checked C [13] which is shown to have good performance that is close to C as well as being backward compatible. Developers can do incremental upgrades, unlike other extensions that can't work with C.

Previous work [12] has shown that refactoring a large code base gets tedious if no automation tools are used. 3C [31] is a tool that helps in the automated conversion of C to Checked C. We use 3C to automatically annotate most of the EDK2 source. Mohammed *et al.,* [36] uses 3C as their base and uses LLMs to fix parts of code that 3C was not able to convert. Since this work builds on the output generated by 3C, the improvements we made to the tool are essential for achieving better results.

8 Conclusion

This paper presents the first experience report of converting EDK2 (a C-based embedded codebase) to Checked C. We use a tool-assisted method to perform the conversion. We use findings from our experience to create an enhanced conversion tool, E3C, which improved the conversion rate by 25%. We also identified several C idioms that are hard to convert automatically. We made manual annotations and identified that certain C language features are not supported in Checked C. We believe that our findings will provide insights into future conversion efforts.

9 Data Availability

Our enhanced E3C can be found at our GitHub repository. All the scripts that can be used to help in conversion, insert the converted modules into the firmware, and instructions on how to use those scripts can be found on Zenodo [10].

Acknowledgments

This research was supported by the National Science Foundation (NSF) under Grant CNS-2340548 and the Defense Advanced Research Projects Agency (DARPA) under contract number N660012224037. The U.S. government is authorized to reproduce and distribute reprints for Governmental purposes, notwithstanding any copyright notation thereon. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF or the United States Government.

References

[1] Hussein Aitlahcen. 2022. TianoCore EDK2 bindings for Rust UEFI Applications. https://github.com/hussein-aitlahcen/edk2-rs
[2] Abdullah Al-Boghdady, Khaled Wassif, and Mohammad El-Ramly. 2021. The Presence, Trends, and Causes of Security Vulnerabilities in Operating Systems of IoT's Low-End Devices. *Sensors* 21, 7 (2021). doi:10.3390/s21072329
[3] Anthropic. 2025. AI research and products. https://www.anthropic.com/
[4] ARM. 2020. EDK-2. https://gitlab.arm.com/arm-reference-solutions/edk2-platforms
[5] Thoms Ball. 1999. The concept of dynamic analysis. *SIGSOFT Softw. Eng. Notes* 24, 6 (Oct. 1999), 216–234. doi:10.1145/318774.318944
[6] BlueHat. 2019. Memory corruption is still the most prevalent security vulnerability. https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/
[7] Checked C. 2024. 3C: Semi-automated conversion of C code to Checked C. https://github.com/checkedc/checkedc-clang/blob/main/clang/docs/checkedc/3C/README.md
[8] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 209–224.
[9] Checked C. 2024. The Checked C. https://github.com/checkedc/checkedc
[10] Sourag Cherupattamoolayil. 2025. Adding Spatial Memory Safety to EDK II through Checked C (Artifact). doi:10.5281/zenodo.15450480
[11] DARPA. 2024. Translating All C to Rust. https://www.darpa.mil/program/translating-all-c-to-rust
[12] Junhan Duan, Yudi Yang, Jie Zhou, and John Criswell. 2020. Refactoring the FreeBSD Kernel with Checked C. In *Proceedings of the 2020 IEEE Cybersecurity Development Conference (SecDev)*. 15–22. doi:10.1109/SecDev45635.2020.00018
[13] Archibald Samuel Elliott, Andrew Ruef, Michael Hicks, and David Tarditi. 2018. Checked C: Making C Safe by Extension. In *2018 IEEE Cybersecurity Development (SecDev)*. 53–60. doi:10.1109/SecDev.2018.00015
[14] Pär Emanuelsson and Ulf Nilsson. 2008. A Comparative Study of Industrial Static Analysis Tools. *Electronic Notes in Theoretical Computer Science* 217 (2008), 5–21. doi:10.1016/j.entcs.2008.06.039 Proceedings of the 3rd International Workshop on Systems Software Verification (SSV 2008).
[15] Mehmet Emre, Peter Boyland, Aesha Parekh, Ryan Schroeder, Kyle Dewey, and Ben Hardekopf. 2023. Aliasing Limits on Translating C to Safe Rust. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 94 (April 2023), 29 pages. doi:10.1145/3586046
[16] Mehmet Emre, Ryan Schroeder, Kyle Dewey, and Ben Hardekopf. 2021. Translating C to safer Rust. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 121 (Oct. 2021), 29 pages. doi:10.1145/3485498
[17] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. doi:10.1109/MICRO.2016.7783743
[18] Forescout. 2020. AMNESIA:33 – Foresout Research Labs Finds 33 New Vulnerabilities in Open Source TCP/IP Stacks. https://www.forescout.com/blog/amnesia33-forescout-research-labs-finds-33-new-vulnerabilities-in-open-source-tcp-ip-stacks/
[19] Unified EFI Forum. 2025. Unified Extensible Firmware Interface (UEFI) Board. https://uefi.org/board
[20] Unified EFI Forum. 2025. Unified Extensible Firmware Interface (UEFI) Specification. https://uefi.org/specifications
[21] Iván Arce Francisco Falcon. 2024. PixieFail: Nine vulnerabilities in Tianocore's EDK II IPv6 network stack. https://blog.quarkslab.com/pixiefail-nine-vulnerabilities-in-tianocores-edk-ii-ipv6-network-stack.html

[22] Zhaoqiang Guo, Tingting Tan, Shiran Liu, Xutong Liu, Wei Lai, Yibiao Yang, Yanhui Li, Lin Chen, Wei Dong, and Yuming Zhou. 2023. Mitigating False Positive Static Analysis Warnings: Progress, Challenges, and Opportunities. *IEEE Transactions on Software Engineering* 49, 12 (2023), 5154–5188. doi:10.1109/TSE.2023.3329667

[23] Dominik Harmim, Vladimır Marcin, and Ondrej Pavela. 2019. Scalable static analysis using facebook infer. *I, VI-B* (2019).

[24] Jaemin Hong and Sukyoung Ryu. 2023. Concrat: An Automatic C-to-Rust Lock API Translator for Concurrent Programs. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE).* 716–728. doi:10.1109/ICSE48619.2023.00069

[25] Jiewen Yao, Vincent J. Zimmer, and Jian Wang. 2020. A Tour Beyond BIOS - Security Enhancement to Mitigate Buffer Overflow in Unified Extensible Interface (UEFI). https://tianocore-docs.github.io/ATBB-Mitigate_Buffer_Overflow_in_UEFI/draft/

[26] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference (ATEC '02).* USENIX Association, USA, 275–288.

[27] Rob Johnson and David Wagner. 2004. Finding user/kernel pointer bugs with type inference. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13* (San Diego, CA) *(SSYM'04).* USENIX Association, USA, 9.

[28] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. doi:10.1109/CGO.2004.1281665

[29] Michael Ling, Yijun Yu, Haitao Wu, Yuan Wang, James R. Cordy, and Ahmed E. Hassan. 2022. In rust we trust: a transpiler from unsafe C to safer rust. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings* (Pittsburgh, Pennsylvania) *(ICSE '22).* Association for Computing Machinery, New York, NY, USA, 354–355. doi:10.1145/3510454.3528640

[30] Dongdong Lu, Jie Wu, Yongxiang Sheng, Peng Liu, and Mengmeng Yang. 2020. Analysis of the popularity of programming languages in open source software communities. In *2020 International Conference on Big Data and Social Sciences (ICBDSS).* 111–114. doi:10.1109/ICBDSS51270.2020.00033

[31] Aravind Machiry, John Kastner, Matt McCutchen, Aaron Eline, Kyle Headley, and Michael Hicks. 2022. C to checked C by 3c. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 78 (April 2022), 29 pages. doi:10.1145/3527322

[32] Lloyd Markle. [n. d.]. Experiences in a Real World Application of Cyclone: A Type-safe Dialect of C. ([n. d.]).

[33] Alex Matrosov. 2023. The Untold Story of the BlackLotus UEFI Bootkit. https://www.binarly.io/posts/The_Untold_Story_of_the_BlackLotus_UEFI_Bootkit/index.html

[34] Medium. 2023. Ownership and Borrowing in Rust: A Comprehensive Guide. https://medium.com/@TechSavvyScribe/ownership-and-borrowing-in-rust-a-comprehensive-guide-1400d2bae02a

[35] Microsoft. 2025. EDK-2. https://github.com/microsoft/mu

[36] Nausheen Mohammed, Akash Lal, Aseem Rastogi, Rahul Sharma, and Subhajit Roy. 2025. LLM Assistance for Memory Safety . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE).* IEEE Computer Society, Los Alamitos, CA, USA, 280–291. https://doi.ieeecomputersociety.org/10.1109/ICSE55347.2025.00023

[37] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09).* Association for Computing Machinery, New York, NY, USA, 245–258. doi:10.1145/1542476.1542504

[38] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2010. CETS: compiler enforced temporal safety for C. *SIGPLAN Not.* 45, 8 (June 2010), 31–40. doi:10.1145/1837855.1806657

[39] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.* 27, 3 (May 2005), 477–526. doi:10.1145/1065887.1065892

[40] NVIDIA. 2022. EDK-2. https://github.com/NVIDIA/edk2

[41] David J. Pearce. 2021. A Lightweight Formalism for Reference Lifetimes and Borrowing in Rust. *ACM Trans. Program. Lang. Syst.* 43, 1, Article 3 (April 2021), 73 pages. doi:10.1145/3443420

[42] G. Ramalingam. 1994. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.* 16, 5 (Sept. 1994), 1467–1471. doi:10.1145/186025.186041

[43] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. 2010. An analysis of the dynamic behavior of JavaScript programs. *SIGPLAN Not.* 45, 6 (jun 2010), 1–12. doi:10.1145/1809028.1806598

[44] Rust. 2025. Rust Programming Language. https://www.rust-lang.org/

[45] Rust. 2025. Understanding Ownership. https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html

[46] 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* (Boston, MA) *(USENIX ATC'12).* USENIX Association, USA, 28.

[47] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security* (Washington DC, USA) *(CCS '04)*. Association for Computing Machinery, New York, NY, USA, 298–307. doi:10.1145/1030083.1030124

[48] Md Shafiuzzaman, Achintya Desai, Laboni Sarker, and Tevfik Bultan. 2024. UEFI Vulnerability Signature Generation using Static and Symbolic Analysis. (2024). doi:10.48550/arXiv.2407.07166 arXiv:arXiv:2407.07166

[49] Ayushi Sharma, Shashank Sharma, Sai Ritvik Tanksalkar, Santiago Torres-Arias, and Aravind Machiry. 2024. Rust for Embedded Systems: Current State and Open Problems. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 2296–2310. doi:10.1145/3658644.3690275

[50] Mingjie Shen, James C. Davis, and Aravind Machiry. 2023. Towards Automated Identification of Layering Violations in Embedded Applications (WIP). In *Proceedings of the 24th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems* (Orlando, FL, USA) *(LCTES 2023)*. Association for Computing Machinery, New York, NY, USA, 143–147. doi:10.1145/3589610.3596271

[51] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. 2019. SoK: Sanitizing for Security. In *2019 IEEE Symposium on Security and Privacy (SP)*. 1275–1295. doi:10.1109/SP.2019.00010

[52] Binarly Research Team. 2023. Finding LogoFAIL: The Dangers of Image Parsing During System Boot. https://binarly.io/posts/finding_logofail_the_dangers_of_image_parsing_during_system_boot/

[53] Tianocore. 2019. EDK II new feature staging. https://github.com/tianocore/edk2-staging/tree/edkii-rust

[54] Tianocore. 2022. Function Definition Layout. https://tianocore-docs.github.io/edk2-CCodingStandardsSpecification/draft/5_source_files/57_c_programming.html#57-c-programming

[55] Tianocore. 2022. Tasks Add Rust Support to EDK II. https://github.com/tianocore/tianocore.github.io/wiki/Tasks-Add-Rust-Support-to-EDK-II

[56] Tianocore. 2025. EDK-2. https://github.com/tianocore/edk2

[57] Tianocore. 2025. EDK-2 Platforms. https://github.com/tianocore/edk2-platforms

[58] Tianocore. 2025. EDK-2 pytool Library. https://github.com/tianocore/edk2-pytool-library

[59] Nicole Tietz-Sokolskaya. 2023. Why Rust's learning curve seems harsh, and ideas to reduce it. https://ntietz.com/blog/rust-resources-learning-curve/

[60] CVE Trends. 2021. CVE trends. https://www.cvedetails.com/vulnerabilities-by-types.php

[61] Margus Välja, Matus Korman, and Robert Lagerström. 2017. A Study on Software Vulnerabilities and Weaknesses of Embedded Systems in Power Networks. In *Proceedings of the 2nd Workshop on Cyber-Physical Security and Resilience in Smart Grids* (Pittsburgh, PA, USA) *(CPSR-SG'17)*. Association for Computing Machinery, New York, NY, USA, 47–52. doi:10.1145/3055386.3055397

[62] Paul C. van Oorschot. 2023. Memory Errors and Memory Safety: C as a Case Study. *IEEE Security & Privacy* 21, 2 (2023), 70–76. doi:10.1109/MSEC.2023.3236542

[63] Wikipedia. 2024. Ripple20. https://en.wikipedia.org/wiki/Ripple20

[64] Jiewen Yao. 2022. SecurityEx. https://github.com/jyao1/SecurityEx

[65] Jiewen Yao, Vincent J. Zimmer, and Matt Fleming. 2016. A Tour Beyond BIOS Memory Map and Practices in UEFI BIOS. https://raw.githubusercontent.com/tianocore-docs/Docs/master/White_Papers/A_Tour_Beyond_BIOS_Memory_Map_And_Practices_in_UEFI_BIOS_V2.pdf

[66] Jiawei Yin, Menghao Li, Yuekang Li, Yong Yu, Boru Lin, Yanyan Zou, Yang Liu, Wei Huo, and Jingling Xue. 2023. RSFuzzer: Discovering Deep SMI Handler Vulnerabilities in UEFI Firmware with Hybrid Fuzzing. In *2023 IEEE Symposium on Security and Privacy (SP)*. 2155–2169. doi:10.1109/SP46215.2023.10179421

[67] Jiawei Yin, Menghao Li, Wei Wu, Dandan Sun, Jianhua Zhou, Wei Huo, and Jingling Xue. 2022. Finding SMM Privilege-Escalation Vulnerabilities in UEFI Firmware with Protocol-Centric Static Analysis. In *2022 IEEE Symposium on Security and Privacy (SP)*. 1623–1637. doi:10.1109/SP46214.2022.9833723

[68] Dongjun Youn, Sungho Lee, and Sukyoung Ryu. 2023. Declarative static analysis for multilingual programs using CodeQL. *Software: Practice and Experience* 53, 7 (2023), 1472–1495. doi:10.1002/spe.3199 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.3199

[69] Bin Zeng, Gang Tan, and Úlfar Erlingsson. 2013. Strato: a retargetable framework for low-level inlined-reference monitors. In *Proceedings of the 22nd USENIX Conference on Security* (Washington, D.C.) *(SEC'13)*. USENIX Association, USA, 369–382.

[70] Hanliang Zhang, Cristina David, Yijun Yu, and Meng Wang. 2023. Ownership Guided C to Rust Translation. In *Computer Aided Verification*, Constantin Enea and Akash Lal (Eds.). Springer Nature Switzerland, Cham, 459–482.

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

Received 2024-10-31; accepted 2025-03-31