Investigating the Security of the Smaller C Compiler

Anonymous Author(s)

ABSTRACT

Programs written in C and C++ can be vulnerable to a wide range of memory safety issues. From out-of-bounds reads and writes to use after free errors, developers have to be careful to ensure that they release code without critical security flaws. Some of the most common vulnerabilities include out-of-bounds (OOB) read and writes, use after free (UAF), and improper use of shared resources. Often, compilers have tools built in that help developers catch these memory safety violations. While it is not possible to catch or prevent all memory safety vulnerabilities in a compiler, especially in languages where there is unsafe behavior explicitly allowed by the standard like C, there are many approaches that can be applied to increase memory safety.

This paper investigates a few methods to catch memory safety violations to see if they could be applied to a single-pass C compiler called "Smaller C". This includes exploring a basic Canary implementation to prevent stack-smashing attacks and an implementation of poisoning memory addresses to catch UAF errors. Along with this, the research covers built-in address sanitation and investigation into compile-time checks for potential race conditions, which are effective, but potentially cumbersome– and by extension difficult to implement in the Smaller C compiler with traditional methods. The investigation into security methods that can be used in a variety of compilers also sheds light on what is viable for lighter systems, and what techniques can avoid losing out on hard-earned performance optimizations. Overall, the Smaller C compiler is not able to catch a significant number of memory safety violations and should not be used to compile code for any important applications.

KEYWORDS

Compiler, Security, Single-Pass Compiler, C, C++, Use-After-Free, Out-Of-Bounds, Memory Safety, Stack-Smashing

1 OVERVIEW

Memory safety vulnerabilities are a constant security threat to programs written in C and C++. As recently as this year, major consumer software companies still uncover security vulnerabilities that cause privacy concerns for users. Large tech companies such as Apple, Google, and Microsoft still regularly find seemingly basic vulnerabilities regarding incorrect memory usage [2].

The Common Weakness Enumeration [6] describes many of the most common memory safety issues. Of these vulnerabilities, only a few can theoretically be caught by compilers.

The first of these bugs is out-of-bounds (OOB) read and writes, which occur when memory outside of an allocated space is accessed in the program. This can lead to undefined behavior, and can be exploited by malicious actors to change program functionality. Ideally, static out-of-bounds references should be caught by the compiler and flagged as an error, but dynamic accesses can still pass through. Some compilers implement a memory tagging procedure to detect OOB errors at runtime [5]. In this procedure, memory blocks are locked with a tag that is stored in the end bits of a pointer. When a memory access is made, the tag in the pointer is checked against the lock on the memory block. If they do not match, that means that the pointer being used is meant to access a different data block, and is trying to access data that it should not be able to access. In this scenario, program execution is terminated to prevent any security breaches.

Memory tagging procedures generally impose a large overhead and can be difficult to implement, so many compilers focus on preventing OOB errors where they are most important: on the stack. Overwriting data on the stack can allow the function return address to be overwritten, allowing malicious actors to execute arbitrary code by inserting a faulty address. David Monniaux details one method for catching these attacks, in which a randomly generated "canary" value is inserted before the location that the return address is stored [13]. This value is checked at the end of function execution, before a value is returned. If the value was changed, then much like a canary in a coal mine, it signifies that something has gone wrong. In this case, it means that the return address was likely changed too, so the program stops execution to prevent any undesired behavior.

A second issue is use after free (UAF), in which memory that was allocated for use is freed, then accessed later. This can be used to cause undefined behavior when exploited. In sequential programs, it should be possible to prevent this, but it would also be ideal to ensure that memory is freed in the same scope it is declared, and only after it has finished being used. Most C compilers do not address this, but using an address sanitation procedure to detect memory errors in implementation can reduce this vulnerability [14]. Another possible vulnerability is null-pointer dereferencing. Null-pointer dereferencing refers to attempting to read memory from a null-pointer, which could be a pointer that has not yet been assigned, or a pointer that was freed and reset. Checks like those described above could help with this, but these tasks will likely be harder due to the single-pass nature of Smaller C. Since there is only a single pass through each compilation unit, and no intermediate steps or representation, there are potential security flaws caused by one portion of a non-externalized unit that has emergent insecure behavior due to another subsequent portion. This means that the source code may need to be reviewed semantically for behaviors normally caught through the use of comparative or sequential compilation.

```
int useAfterFree(int count) {
    // Allocate a buffer on the heap
    int* someInts = calloc(intSize, count);
    for (int i = 0; i < count; i++)
        someInts[i] = 8 * i;

    // Check what's on the heap
    for (int i = 0; i < count; ++i)
        printf("\t\t%d\n", someInts[i]);
    printf("\n");

    // Free the buffer
```

```
    free(someInts);

    // This is our data after freeing
    for (int i = 0; i < count; ++i)
        printf("\t\t%d\n", someInts[i]);
    printf("\n");
}
```

useafterfree.c

A particularly difficult to diagnose vulnerability that could be addressed is the use of shared resources with improper synchronization. This refers to accessing any shared data or resource in a multi-threaded system in a way that either interferes with other programs trying to use the same resource, or prevents other programs from being able to access that resource. If a resource is declared as "critical," which means that it should not be accessed by multiple sources during an operation, it should always be surrounded by a semaphore, lock, or other equivalent security measure that ensures exclusive access to it. A compiler could check for unexpected behavior when entering and exiting critical sections of code [11], so we will see if Smaller C implements any such behavior.

Smaller C has some unique limitations that may, if improperly implemented, lead to security vulnerabilities. First, there is no stack overflow check, and very few compile-time checks compared to most multi-pass compilers. This might cause a program to be able to be frozen by putting it into an infinite loop. Secondly, large integer constants are ambiguously assigned to types of different sizes unless they are explicitly marked. This can cause different amounts of space to be allocated, which can cause issues in extreme cases.

This paper documents security issues present in smaller C and puts forward a number of improvements that would reduce the vulnerabilities present in code compiled with Smaller C.

2 RELATED WORK

2.1 Smaller C Function

It is important to understand the steps taken by Smaller C to compile code. Before any compilation happens, there is typically a preprocessing step. In GCC, the preprocessor performs a few simple tasks, such as breaking the program into lines, removing comments, and merging continued lines [1]. Ultimately, this step tends to not have much impact on the finished code, as long as it functions properly. The code for Smaller C's preprocessor was not included in the github, only the executable files. In the compilation step, a traditional compiler might begin by using a lexical analyzer to divide the characters in the program into tokens, which are distinct symbols with meaning in the program. This includes math symbols, semicolons, file names, and all other atomic elements of code. From there, this stream of tokens would be syntactically analyzed, verifying that the tokens appear in a valid order. Then, the function of the tokens would be determined using a semantic analyzer, and an intermediary code file would be generated. This intermediary code is more suited to optimization and modification. The optimization step involves reordering code blocks and adjusting code so that it runs faster. Finally, machine code is generated. Most compilers perform each step in sequence, not moving onto the next until the previous has been completed. In contrast, Smaller C parses each token, verifies it as being properly syntactically ordered and adds it to a block of symbols. As soon as a block of code can be understood, it is immediately converted into machine code. This means that any improvements made to Smaller C cannot come from code analysis, since there is no period of time in which more than a single line of code is analyzed. As such, improvements to Smaller C have to be made primarily in the code generation step, padding unsafe actions written in C into memory safe interactions in machine code.

2.1.1 Code Optimization. Code Optimization in compilers can often unintentionally remove security checks or make code less secure. This can be especially true when compiling on hardware different than the intended program location. We will explore how Smaller C handles common optimization techniques, such as dead store elimination. D'Silva et al. laid out how dead store elimination in GCC removed a necessary security feature from the function `crypt()` [7]. Another common error with optimization is how the compiler handles undefined behavior within the C standard. There have been noted cases, by Wang et al. where various security checks are removed due to it being undefined by the C standard [16]. After looking into the codebase of Smaller C, no evidence of code optimization was found, although it seems as though an intermediary called RetroBSD may reorder some instructions. RetroBSD was not investigated during this project. Smaller C performs very little in the form of code modifications or optimizations, so it is extremely unlikely that it could create a security threat or remove a security check from the code.

2.2 Related Memory Security Improvements

This subsection discusses several memory safety improvements that have been introduced in other compilers that likely will not work in Smaller C. Memory tagging, like described above, is a process in which sections of memory are locked so that they can only be accessed by certain pointers. Chen et al. describes an implementation that works to secure memory on the heap /citeChen:HeMate. During compilation, the author's tool, HeMate, finds the type, base address, and size, and uses it to generate tags for each allocated block of memory. HeMate then stores the tags in the ending bits of pointers such that they contain type information, as well as a randomly generated unique tag. This requires every memory allocation to have a concrete type at compile time, which, despite being a standard style for C, is not assured by the Smaller C compiler. This requirement makes storing types in memory tags all but impossible in Smaller C. Unfortunately, memory tagging in general does not seem to be possible either. While it would be relatively easy to add a lock to a block of memory with a simple modification to code generation, editing the pointers to include a tag would require a significant refactoring of smaller C. Since it is a single pass system as well, it would also be extremely difficult to ensure that pointers to the same location in memory have the same tag, as is typical in memory tagging procedures. Memory tagging does not appear feasible in Smaller C.

Stack Smashing is a common attack that exploits a memory safety violation. Stack smashing occurs when a malicious attacker overflows the allocated bounds of an array or other memory structure in such a way that it overwrites a return address on the stack.

This overwritten address can then jump to code that the attacker loaded onto the stack, likely somewhere in the data used to overflow the memory structure. From there, this code will be executed, more or less allowing the attacker to do anything they want. Many other compilers have safeguards against this. One of the simplest modifications that could be made is to add a canary to protect the function return value. A typical implementation of a canary starts by saving a value on the stack directly next to the return address. When an overflow occurs, the canary is overwritten. Right before the function returns, the canary value is checked. If the value is different than expected, that means that an overflow occurred, and the return address has likely been changed as well. As such, the program would be terminated to prevent any potential attacks. Unfortunately, canary values could not be implemented in Smaller C due to some limitations in its design.

In lieu of this, there's another consideration to be made about a similar memory tagging technique, pointer tagging. However, Smaller C is implemented for only 32-bit and 16-bit systems, and does not have enough unused bits in pointers to facilitate pointer tagging. Pointer tagging reduces the number of bits available for addressing memory by storing additional information in specific bits of a pointer. This reduces the maximum addressable memory below the theoretical 4 GB limit, posing a challenge to memory-constrained programs trying to run using a 32-bit executable. However, if Smaller C were to implement features taking advantage of data alignment inherent to the C language, pointer tagging could be implemented without compromising usable address space.

One of the most vulnerable parts of memory management is the heap. The dynamic nature allows for improved efficiency, when implemented right. That is where the issue lies, C and C++ giving control to the programmer, while allowing greater efficiency, allows the programmer to make errors when dynamically allocating and freeing memory. As stated above there are numerous errors that can occur, such as UAF, dangling pointers, out-of-bounds, etc. One solution proposed to this is Lin et al. Compiler and Allocator-based Heap Memory Protection, or CAMP [12]. While many methods already exist to combat these errors, one such being address sanitization, they can cause up to 26 times overhead. Their proposed solution has minimal overhead and is fully in software, not relying on any hardware. CAMP can protect against out-of-bounds memory access and dangling pointers, which negates any UAF errors. This is done by adding checks to the program while it is compiling.

3 DESIGN AND IMPLEMENTATION

Smaller C is vulnerable to stack smashing attacks, since it has no protections in place to stop them. To prevent this, the program was examined to see if implementing a canary check would be feasible. Unfortunately, this did not end up being possible. In the function `GenFxnEpilog` in `genfxnepilog.c`, we can see that the return address is stored on the stack in a spot four bytes after the frame pointer. Interestingly enough, when the function doesn't make a further function call, it actually stores its return address in a register. This means that stack smashing is actually impossible in functions that don't call further functions, so there is no need to implement a canary in those cases. Of course, very few functions make no further function calls, so programs compiled in Smaller C are still vulnerable. In scenarios where the return address is stored on the stack, we can add assembly instructions to check a canary. If the value differs, a branch error is thrown and the program stops execution. For this to work however, we would need to ensure that the value of the canary on the stack and `CanaryStorageLocation` will not be overwritten during normal program execution. As previously mentioned, some parts of the code generation process are performed outside of the actual repository in other tools like RetroBSD. One of these parts is unfortunately the segment that copies over local variable registers, including the return address. This means that the location the canary is written into is being overwritten by a process that cannot be modified. As a result, virtually all programs, including those that do not perform overflows or stack smashing attacks will overwrite the canary. Even if it could be guaranteed that the canary would be restored afterwards, there is no suitable location to store the canary in memory. While we can avoid the issue of having to store the canary in a register by putting it at a fixed address, there is no guarantee that it will not be overwritten during program execution. Theoretically, a fixed value could be loaded in and tested against, but this provides so little in the way of security that it might as well not exist.

Another idea would be to modify the compilation step of Smaller C to artificially generate and insert a canary into code. When it parses the start of a function, it could add code to initialize a variable that holds a random value and copy that value into a global array. When Smaller C parses the end of a function, before adding the end of function code, it could compare the value in the global array to the canary. In the case of a change, the program would be terminated. Unfortunately, there is no way to guarantee that this variable would be anywhere near the return address, making it rather useless. Additionally, this would necessarily add a reserved variable name, which would mean that programs that used that name could not compile. Finally, requiring every program to reserve space for an array equal to the maximum function call depth seems undesirable. While this would theoretically work and have a chance to prevent some stack smashing attacks, it would also prevent a number of valid programs from compiling. In conclusion, while canaries can be implemented in a single pass compiler, they are not able to be implemented in a useful way in Smaller C.

```
void GenFxnEpilog(void) {
  GenUpdateFrameSize();
  // Check if the function called another function
  if (!GenLeaf) {
    // Load Canary from Stack into temp register 1
    GenPrintInstr2Operands(
      MipsInstrLW, 0,
      MipsOpRegT1, 0,
      MipsOpRegFp, 12
    );
    // Load Canary from Storage Location into
        temp register 2
    GenPrintInstr2Operands(
      MipsInstrLW, 0,
      MipsOpRegT2, 0,
      CanaryStorageLocation, 0
```

```
    );
    // Compare the two values, if they aren't
        equal, crash.
    GenPrintInstr3Operands(
      MipsInstrBNE, 0,
      MipsOpRegT1, 0,
      MipsOpRegT2, 0,
      MipsInstrExit, 0
    );
    // Load the return address we stored on the
        stack into the return address register
    GenPrintInstr2Operands(
      MipsInstrLW, 0,
      MipsOpRegRa, 0,
      MipsOpIndRegFp, 4
    );
  }
  // Load the previous frame pointer back into a
      register
  GenPrintInstr2Operands(
    MipsInstrLW, 0,
    MipsOpRegFp, 0,
    MipsOpIndRegFp, 0
  );
  // Move the stack pointer past the function
      frame
  GenPrintInstr3Operands(
    MipsInstrAddU, 0,
    MipsOpRegSp, 0,
    MipsOpRegSp, 0,
    MipsOpConst, 12 - CurFxnMinLocalOfs
  );  //12 = RA + FP + Canary
  // Jump to the return address and continue
      execution
  GenPrintInstr1Operand(
    MipsInstrJ, 0,
    MipsOpRegRa, 0
  );
}
```

genfxnepilog.c

The next fix proposed for the Smaller C compiler is adding memory address sanitization. This is a method of checking code on compile time for errors like buffer overflow and UAF. This is generally done by adding marks to memory addresses. These are either adding 32 byte aligned poisoned "redzones" before and after contiguous areas of memory, and poison tags when a memory address has been freed [3]. The redzone allows checks to be made if any access to memory is in one of these zones, and an error can be thrown and the compiler stopped. The poison tags for freeing equate to an if statement before any dereference of memory to check if that address has been poisoned. Both of these implementations heavily make use of the ShadowStack [14]. To implement these poison checks for Smaller C there need to be changes made to how it evaluates expressions. When a call to free() has been made, it can be assumed that the next symbol evaluated will be the name of the pointer that is freed. So a flag is raised that will log the pointer name in a table for poisoned pointers. This is currently implemented as a fixed length 2D array, where one dimension is the parse level, or scope, and the other dimension holds the poisoned pointers. Later implementations can make this also dynamically allocated to conserve memory. Then also during evaluation when a dereference is made it will check the name of the pointer to the table, with the matching parse level to account for variables that would be out of scope. If the pointer is in the poisoned table, then the compiler errors out and stops execution.

```
char* identGlobal = NULL;
char* poisonedIdents[32][1024];
int poisonedIndex[32];
char prevUnaryStar = 0;

...

char toPoison = 0;
if(identGlobal != NULL && strcmp(identGlobal,
    "free") == 0) {
  toPoison = 1;
}
// DONE: support __func__
char* ident = IdentTable + s;
if(toPoison) {
  poisonedIdents[ParseLevel][poisonedIndex[ParseLevel]]
      = ident;
  poisonedIndex[ParseLevel]++;
}
if(prevUnaryStar) {
  for(int j = ParseLevel; j >= 0; j--) {
    for(int i = 0; i < poisonedIndex[ParseLevel];
        i++) {
      if(strcmp(ident,
          poisonedIdents[ParseLevel][i]) == 0) {
        error("ident: %s is poisoned\n", ident);
      }
    }
  }
}
identGlobal = ident;
```

asanpoison.c

4 ANALYSIS

This section presents our findings for this project. First, we document a number of vulnerabilities and memory safety issues that are present in Smaller C. Second, we present a number of lessons learned from these issues and the process of implementing various improvements to Smaller C. Finally, we make some brief recommendations for areas of future research.

4.1 Smaller C Compiler Memory Safety Issues

While examining Smaller C, we were able to document a number of security vulnerabilities present. Firstly, there are a number of errors present in the compiler, where expressions that should be syntactically or semantically invalid are compiled and able to be executed.

One of these issues is related to the fact that the compiler treats structs as lvalues, which allows code to compile where structures are being used to assign to arrays of structures or function pointers. This should cause crashes on its own, and definitely can lead to unintended behaviors in programs. Another bug is that extern and static variables can have the same name, which should not be allowed since it produces ambiguous behavior due to shadowing.

One concerning property of the Smaller C compiler is that it does not check for pointer types when dereferencing, it just checks that the types have the same size. Technically, this should not lead to any memory safety issues on its own, but it could cause unintended behavior, which could lead to a memory safety violation.

Bugs related to improper synchronization and race conditions that are baked into programs can often be caught through the use of techniques that test the code using several optimization levels and monitoring critical sections of code [11]. However, since Smaller C uses one-pass compilation, it isn't possible to do any comparative or sequential analysis of various sections of code together, since the compile units are differentiated and converted to machine code procedurally. In our investigation we have found no way of mitigating a race condition in any way in a single pass, save for potentially scanning code semantically to attempt to find a race– but this would produce at best a warning.

Interestingly, Smaller C does implement a single common technique for reducing the damage done by overflow errors. Smaller C, like many other systems, saves return addresses at the start of the stack frame for functions. What this means is that, in the case of a simple overflow, the return address for that function is not affected. Of course, this only safeguards against the simplest of overflow-based stack smashing attacks, but it is nice to see some decisions in Smaller C being made with security in mind.

In all, Smaller C is clearly not a reasonable choice of compiler for any program meant to be distributed. Single-pass compilers are inherently insecure, since they cannot implement a significant number of sophisticated memory safety checks that multi-pass compilers can. Smaller C in particular is built in such a way that makes it difficult to modify and add security features to. Thankfully, the industry does not use Smaller C as a default compiler, instead opting for more secure compilers like GCC and Clang. Notably, while Smaller C presents itself as a "simple and small" compiler, it makes no obvious mentions of its complete lack of security capabilities. An unknowing developer could download, install, and begin using Smaller C under without realizing that it has zero security features, making their programs vulnerable to all sorts of errors or attacks. While it is debatable whether or not users should expect security features from a random compiler that they find online, it is still at least slightly irresponsible not to add a disclaimer in the readme stating "due to a lack of security features, this should not be used for any serious applications."

4.2 Lessons Learned

The most important lesson learned from this is to use compilers that are verifiably secure, and that implement robust security measures. It is difficult to verify or rigorously check the security of any compiler, even if you know what you are doing. Code that would be secure and free from error when compiled with GCC may be riddled with vulnerabilities if compiled with Smaller C. As such, it is probably best to use well-established tools that have been thoroughly examined by cybersecurity professionals and not random compilers found online. In general, all compilers should be far more upfront about their security capabilities and any vulnerabilities that they fail to catch. It would be nice to imagine that developers know and understand the risks associated with their programming language and compiler of choice, but this is obviously not the case. Developers using insecure tools must be made aware that they are doing so, and provided steps to mitigate any vulnerabilities that might cause those.

Another lesson learned from examining Smaller C is just how much assembly code is needed to make assembly code secure. Smaller C adds almost no security checks, and as a result, is able to translate most simple functions into a few lines of machine code. The theoretical implementation for canaries would have required at least twenty lines of additional machine code for function initialization and conclusion, and that is a relatively small modification. Most lines of machine code in a completed program likely have nothing to do with the program itself, and were generated by a compiler to ensure memory safety. This is probably why so much thought and effort is put into compiler modifications that prevent insecure actions or undefined behaviors as opposed to generating code to make those actions secure. The best way to ensure that all programs are secure is to not allow insecure ones to compile.

A third lesson learned from this project is that it is important to understand the limitations of a codebase. Canaries being almost impossible to implement in Smaller C is almost solely due to the fact that Smaller C does not handle transferring and backing up registers, instead relegating that task to a separate program. Had this fact been known at the start of this project, perhaps another memory safety feature could have been researched and actually implemented. It is always a good idea to thoroughly document code since it may be used or modified at a later date. Good documentation can prevent developers from wasting significant time and effort trying to fix something in one program, only to learn that the issue is due to a completely separate program. Spending so much time attempting to implement canaries was not necessarily a mistake, since it did allow for a deep and thorough understanding of Smaller C's code generation process, but time would have been better spent looking into other potential fixes.

4.3 Future Work

Developers should not spend too much time attempting to add security features to Smaller C, or any other single pass compiler. Smaller C was written by a single developer as a learning experience, and should not be used or examined further. There is already much work being done to improve the security of other C compilers, as there should be. More effort should be spent improving the memory safety of those tools. Another interesting area for research would be thoroughly reviewing and analyzing the memory safety features of GCC or Clang. In particular, a comprehensive report that clearly and concisely describes the limitations, vulnerabilities, and necessary compiler flags for security would be very interesting, and likely helpful for most developers.

Another possible area for research would be reducing the overhead for currently existing compiler memory safety features. Many compiler flags that add significant memory security are disabled by default in GCC and Clang because they impose a significant overhead on the program. This is very similar to the address sanitization procedure that this project implemented into Smaller C. By reducing the overhead of these security features, it increases the likelihood that they would be adopted for use by more developers, improving the security of their code. In fact, if the overhead was reduced enough, these features could be enabled by default, so that all code compiled would be more secure.

5 LEGAL CONSIDERATIONS

As laws regarding computing security and secure coding generally become more prevalent, it is imperative that organizations that distribute software ensure that their programs are secure and minimize common security vulnerabilities. Making informed decisions about which compiler to use and leveraging the security features inherent to that compiler are essential considerations that are often overlooked. Software integrated into vital parts of society, including healthcare and finance, are subject to increasingly strict regulations by bodies such as the European Union, which implemented the General Data Protection Regulation in 2016[8] and came into effect two years later. Due to the privacy concerns for users that sparked GDPR, secure coding is a necessary extension of the conclusions present in the law.

The GDPR emphasizes the protection of personal data and mandates organizations to implement appropriate technical measures to ensure data protection and security. This includes the need for secure coding practices to prevent data breaches that compromise personal information. Compilers have the capacity to improve or worsen the security of a given piece of software, through checking or lack thereof, and should be considered essential to the understanding of secure code as defined by regulations such as the GDPR. By enforcing strict penalties on non-compliance, GDPR incentivizes organizations to prioritize security in development. In the United States, the introduction of the American Privacy Rights Act (APRA) in 2024 signifies a greater move towards regulating and strengthening laws surrounding data protection and cybersecurity[15]. APRA aims to provide comprehensive protection to individuals and imposes strict obligations on how organizations are allowed to handle personal data, including mandates for robust security measures. This development emphasizes the importance of secure coding practices in preventing data breaches and unauthorized access.

More recent laws have also been passed in response to increasingly sophisticated cyber-attacks across the world, including ransomware in critical sectors of society. Similarly in the EU, the NIS2 Directive mandates a basic level of security required by software developers in all member states[9]. Addressing the surge of cyber-attacks, it urges member states to pass swift and harsh legislation on organizations that violate security practices, and encourage the adoption of secure practices in their software development processes.

These regulatory developments highlight a global trend toward enforcing higher standards of cybersecurity and data protection. However, mandating the use or adoption of specific compilers could be fraught, as organizations with a vested interest in undermining the security of software could theoretically implement backdoors or security flaws into compilers without the knowledge of users. Therefore, it is important to understand what limitations should exist for regulations and legislation passed by governing bodies when it comes to limiting the freedom of organizations to develop without undue burdens, while also ensuring the code those organizations create is suitably protected from cyber-attacks and with penalties for non-cooperation.

6 ETHICAL CONSIDERATIONS

The ACM code of ethics has a few relevant sections when it comes to memory safety vulnerabilities in compilers [4].

- **Code 1.2: "Avoid harm"** This is relevant to the designer of the compiler, as well as software developers who use it. Memory security vulnerabilities can cause software to stop working suddenly, which could cause loss of life if a hospital suddenly was unable to care for patients. Additionally, hackers can take control of programs with memory security vulnerabilities. This could be dangerous, if for example, they got control of an application that controlled a defensive military installation and prevented it from intercepting missiles. As such, a compiler writer wants to ensure that their compiler will prevent memory vulnerabilities, and developers want to write code that prevents them.
- **Code 1.3: "Be honest and trustworthy"** A compiler should not misrepresent the extent to which it prevents or detects memory safety vulnerabilities, or a developer might unintentionally release a program that is unsafe.
- **Code 2.7: "Foster public awareness and understanding of computing, related technologies, and their consequences"** This section similarly mandates that a compiler vendor explain the importance of vulnerabilities that may not be caught, and should also put forward methods to prevent issues that could make it through to compile time.
- **Code 2.9: "Design and implement systems that are robustly and usably secure"** As discussed previously, some compilers may optimize code in a way that introduces security vulnerabilities, which violates this principle. Developers should also be wary of using compilers that do not guarantee memory safety, since releasing any insecure product is a violation of this code.
- **Code 3.7: "Recognize and take special care of systems that become integrated into the infrastructure of society"** This can be pertinent for any compilers that aim to be used by an IDE. It is the responsibility of large developer tools producers to ship compilers that come packaged with their code environments, such as Visual Studio by Microsoft, that are memory safe by default, since many developers use that compiler without ever thinking about it. Since a compiler is a tool used by other developers, creating an insecure compiler can have a knock on effect that results in widespread vulnerabilities. Thus, it is very important for compilers to enforce memory safety wherever possible as an ethical duty to the developers who use them.

A similar code of conduct is the IEEE CS Code of Ethics, which focuses on the role of software engineers in the world [10]. The role of this code of ethics mirrors much of what the ACM code of ethics aims to achieve. The 8 general principles are public, client and employer, product, judgment, management, profession, colleagues, and self.

- **Code 1.03 "Approve software only if they have a well-founded belief that it is safe, meets specifications, passes appropriate tests, and does not diminish quality of life, diminish privacy or harm the environment. The ultimate effect of the work should be to the public good"** covers much of what Code 1.2 does for the ACM Code of Ethics. A memory vulnerability in a critical piece of software can cause immeasurable harm, as stated above possibly can even cost a life. That's why it is imperative to uphold this code.
- **Code 3.10 "Ensure adequate testing, debugging, and review of software and related documents on which they work"** This code is important when it comes to the security of any piece of software, but even more so when it comes to compilers. A compiler is something programmers depend on to build their code, and it's paramount that a compiler is thoroughly tested to find any insecure parts to fix.
- **Code 6.08 "Take responsibility for detecting, correcting, and reporting errors in software and associated documents on which they work"** This code is also important when it comes to the security of compilers. Unchecked errors, such as UAF or out-of-bounds, can cause important systems to fail. Engineers who rely on compilers in order to prevent these types of errors from occurring can be put at risk by a compiler developer who neglects this duty.

7 CONCLUSIONS

Reading over and understanding a program like Smaller C, in which all functions are static and have hundreds of side effects is difficult. As such, a majority of progress up until this point has been spent analyzing the program execution and figuring out how everything interacts. In the future, we intend to implement more programs to show vulnerabilities that pass, and finish implementing more fixes to reduce the prevalence of memory vulnerabilities in Smaller C.

We do not recommend using Smaller C to compile any program intended for serious use: Smaller C was a hobby project and is not able to perform any of the security or optimization tasks that we expect a compiler to do. Its lack of disclosure about security vulnerabilities should be an indicator to any user, as with any other comparable compiler, that code that may otherwise be memory safe or checked for its memory safety will not have the same properties or expectations with this compiler.

REFERENCES

[1] [n. d.]. *GNU Manual: CPP.* https://gcc.gnu.org/onlinedocs/cpp/

[2] 2024. *About the security content of iOS 18 and iPadOS 18.* https://support.apple.com/en-us/121250

[3] Mitch Phillips Alexander Potapenko. 2017. *AddressSanitizerAlgorithm.* https://github.com/google/sanitizers/wiki/AddressSanitizerAlgorithm

[4] Association for Computing Machinery. 2018. ACM Code of Ethics and Professional Conduct. ACM, New York. https://www.acm.org/code-of-ethics.

[5] Yu-Chang Chen and Shih-Wei Li. 2024. HeMate: Enhancing Heap Security through Isolating Primitive Types with Arm Memory Tagging Extension. In *Proceedings of the 19th International Conference on Availability, Reliability and Security* (Vienna, Austria) *(ARES '24)*. Association for Computing Machinery, New York, NY, USA, Article 30, 11 pages. https://doi.org/10.1145/3664476.3664492

[6] CWE. 2023. https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html

[7] Vijay D'Silva, Mathias Payer, and Dawn Song. 2015. The Correctness-Security Gap in Compiler Optimization. In *2015 IEEE Security and Privacy Workshops*. IEEE, IEEE, New York, NY, 73–87.

[8] European Parliament and Council of the European Union. 2016. General Data Protection Regulation (GDPR). Official Journal of the European Union. https://eur-lex.europa.eu/eli/reg/2016/679/oj Regulation (EU) 2016/679.

[9] European Union. 2022. Directive (EU) 2022/2555 of the European Parliament and of the Council of 14 December 2022 on measures for a high common level of cybersecurity across the Union (NIS2 Directive). Official Journal of the European Union, L 333, pp. 80–152. https://eur-lex.europa.eu/eli/dir/2022/2555/oj

[10] IEEE Computer Society. 1999. Code of Ethics. IEEE. https://www.computer.org/education/code-of-ethics.

[11] Shaohua Li and Zhendong Su. 2023. Finding Unstable Code via Compiler-Driven Differential Testing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 238–251. https://doi.org/10.1145/3582016.3582053

[12] Zhenpeng Lin, Zheng Yu, Ziyi Guo, Simone Campanoni, Peter Dinda, and Xinyu Xing. 2024. CAMP: Compiler and Allocator-based Heap Memory Protection. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 4015–4032. https://www.usenix.org/conference/usenixsecurity24/presentation/lin-zhenpeng

[13] David Monniaux. 2024. Memory Simulations, Security and Optimization in a Verified Compiler. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs*. Association for Computing Machinery, Association for Computing Machinery, London, UK, 103–117.

[14] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX Association, Boston, MA, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[15] U.S. Congress. 2024. American Privacy Rights Act of 2024. H.R.8818, 118th Congress, 2nd session. https://www.congress.gov/bill/118th-congress/house-bill/8818 Introduced in House on June 25, 2024.

[16] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. 2013. Towards optimization-safe systems: analyzing the impact of undefined behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. Association for Computing Machinery, Association for Computing Machinery, New York, NY, 260–275.