C Annotations for Concurrent Information Flow Security

Alexander Blyth

June 2021

Abstract **- Information flow security in concurrency is difficult due to the increasing complexity introduced with multiple threads. Additionally, compiler optimisations can break security guarantees that have been verified in source code. In this paper, we propose a thesis to explore these issues through providing annotations in C source code that propagate through to the binary or assembly. These annotations could then be used to guide a static analysis of information flow security in concurrency. This approach involves (1) capturing C source code annotations provided by the user about the security policy of data and variables and (2) passing these annotations down to lower representations where static analysis tools can be utilised to identify security vulnerabilities in the produced binary.**

1 Topic Definition

This paper describes the motivation, background knowledge and plan for the proposed thesis *Compiler Annotation Solutions for Concurrent Information Flow Security.*

There is a high degree of complexity in verifying security guarantees in concurrent programs [19][27][29]. Additionally, aggressive compiler optimisations can modify the binary output in unexpected ways [7]. To preserve the security of a program, the flow of sensitive information must be protected to avoid flowing in to untrusted sources [2]. This is where static analysis tools can be used to verify the integrity of security guarantees and the flow of sensitive information. In this thesis, we look to explore a solution to information flow security in concurrent programs through analysing the output after aggressive compiler optimisations.

We propose a tool to analyse C programs to detect security violations in information flow control. This tool will preserve annotations provided by the programmer in source code through lowering passes and aggressive compiler optimisations. The tool will work alongside the *Weakest Precondition for Information Flow* (wpif) transformer described by Winter et al. [31] to allow the programmer to assess the security of information flow in their concurrent programs.

Similar tools for propagating annotations and properties through compiler optimisations have been explored [30] [25] [18], however, these tools focus on either generic solutions for propagating properties or to assist the static analysis of the *Worst Case Execution Time.*

2 Background

Vulnerabilities in software can lead to catastrophic consequences when manipulated by attackers. In an open-source cryptographic software library (OpenSSL) used by an estimated two-thirds of web servers [16] a security flaw called Heartbleed was discovered. Secure secrets such as financial data, encryption keys, or anything else stored in the server's memory could be leaked. Normally, one would send a Heartbeat request with a text string payload and the length of the payload. For example, a message of "hello" could be sent with the length of the message, 5. However, due to a improper input validation (buffer over-read), one could send a length longer

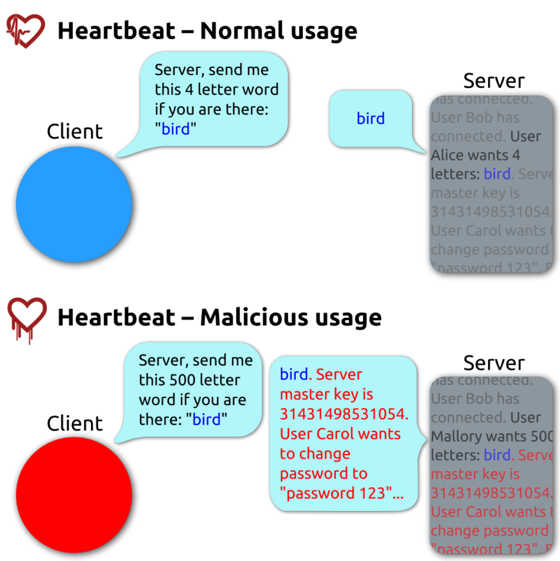

Figure 1: The Heartbleed bug. [13]

that the string they actually sent. This would cause the server to respond with the original message and anything that was in the allocated memory at the time, including any potentially sensitive information. An example of this is shown in figure 1 [13].

Heartbleed was one of the most dangerous security bugs ever, and calls for major reflection by everyone in industry and research [2].

2.1 Information Security

Computer security is defined as a preservation of **integrity**, **availability** and **confidentiality** of information, and extends to include not only software but hardware, firmware, information, data and telecommunications [15]. Confidentiality requires that data is not available to unauthorised users, and that individuals can control what information can be collected and disclosed to others. Data integrity requires that only authorised sources can modify data, and that the system can perform tasks without interference from outside sources. Finally, availability of a system requires that service is not denied to authorised users. Together, these principles create the CIA triad [28]. To enforce a secure system, all three principles must be upheld.

Modern programs are becoming increasingly complex with potential for networking, multi-threading and storage permissions and more. As such, security mechanisms must be put in place to verify and enforce the information security requirements. The adequacy of a security mechanism depends on the adversary model. The adversary model is a formal definition of the attacker and their abilities in a system, and defines who we are protecting against [10]. Ideally we would like to design a system to protect against the strongest adversary or attacker, however, this is often not required or even possible. Instead, we must consider the security policy, security mechanism and strongest adversary model to make a system secure [2].

Standard security processes that handle access control such as a firewall or antivirus software can fail as they do not constrain where information is allowed to flow, meaning that once access is granted it can propagate to insecure processes where it can be accessed by attackers. Where a large system is being used, it is often the case that not all components of the codebase can be trusted, often containing potentially malicious code [24]. Take for example your modern-day web project. Where a package manager such as Node Package Manager (npm) could be used to utilise open-source packages to speed up development progress, it could also inadvertently introduce security vulnerabilities. Rewriting all packages used to ensure security would be time-consuming and expensive and is not a viable option. Instead, controlling where information can flow and preventing secure data from flowing into untrusted sources or packages can maintain confidentiality of a system.

One may suggest runtime monitoring the flow of data to prevent leakage of secure data. Aside from the obvious computational and memory overhead, this method can have its own issues. Although it can detect an *explicit* flow of data from a secure variable to a public variable, it is unable to detect *implicit* data flow, where the state of secure data can be inferred from the state of public data or a public variable [9]. Take for example figure 2. In this example,

```
secret := 0xC0DE mod 2
public := 1
if secret = 1
    public := 0

```

Figure 2: Implicit flow of data to a public variable

a public, readable variable is initially set to the value of 1. There is also a secret variable which may contain a key, password or some other secret that must be kept secure from any attackers. Depending on the value of the secret variable an attacker can infer information about this variable depending on whether the value of the public variable is updated to a value of 0. Assuming that the inner workings of the system is known by the attacker, information about the secret variable can be leaked *implicitly* and inferred by the state of public variables.

Security concerns do not only exist at the application level. In a huge codebase such as an OS, different low-level bugs can be exploited to gain access to data, such as by using buffer overflows to inject viruses or trojans [1].

2.2 Information Flow Control

As seen by the issues that can be introduced via implicit and explicit flow of data, there is room to improve on the existing techniques imposed by current security measures. To protect confidentiality, secure or sensitive information must be prevented from flowing into public on insecure variables. Additionally, to protect integrity, untrusted data from public sources must be prevented from flowing into secure or trusted destinations [2]. An information flow security policy can be introduced to classify or label data, or more formally, a set of *security levels* to which each object is bound by across a multi-level security lattice [8]. In this thesis, we will focus primarily on preserving confidentiality.

Many security levels can be identified to classify different classes of objects, however, for now we will consider two security levels: high and low. Data labelled as high signifies that the data is secret, and

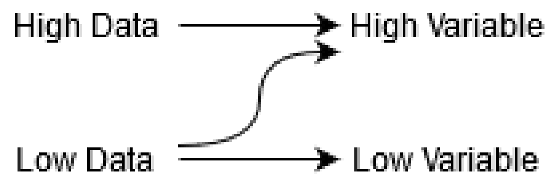

Figure 3: Permitted flow of data

low data is classified as non-sensitive data, such that it does not need to be protected from an attacker or adversary. Variables that can hold data in a program can additionally be classified as high or low as a *security classification*. A variable's security classification shows the highest classification of data it can safely contain [31]. A high variable can hold both high and low data, whereas a low variable which is visible to an attacker can only safely hold low data. As mentioned previously, confidentiality must be upheld by preventing high or secret data from flowing to low or public variables where an attacker can observe it. The permitted flow of data can be observed in 3. Note that high data is not allowed to flow into low variables.

2.3 Information Flow Security in Concurrency

Controlling the flow of information is a difficult problem, however, this is only exacerbated in concurrent programs, which are a well known source of security issues [19][27][29]. Research has been conducted into concurrent programs to explore ways the security of concurrent programs can be verified. Mantel et al. [20] introduced the concept of assumption and guarantee conditions, where assumptions are made about how concurrent threads access shared memory and guarantees are made about how an individual thread access shared memory that other threads may rely upon. Each thread can be observed individually using assumptions over the environment behaviour of other threads that can be then used to prove a guarantee about that individual thread. As two concurrent threads can interleave their steps and behaviour, there is a lot of complexity and possibilities for the overall behaviour. This concept of assumptions (or rely) and guarantee conditions can reduce the complexity of understanding interleaving behaviour in threads and assist in verifying the correctness of information flow security in concurrency. However, this approach is limited in the types of assumptions and guarantees it supports. Building on this, Murray et al. [12] [21] provide information flow logic on how to handle dynamic, value-dependent security levels in concurrent programs. In this case, the security level of a particular variable may depend on one or more other variables in the program. As such, the variable's security level can change as the state of the program changes. This logic is essential where the security level of data depends on its source. However, this approach is not sufficient when analysing non-blocking programs. The approach relies heavily on locks which block particular threads from executing. This in turn leads to slower processing due to blocked threads [23].

To overcome information flow security in non-blocking concurrent threads, Winter et al. [31] explores verifying security properties such as non-interference through the use of general rely/guarantee conditions using backwards, weakest precondition reasoning. Such an analysis would additionally handle implicit flows as shown in figure 3. Ideally a tool could be created to verify security policies required for sensitive processes. Users of this system could provide rely/guarantee conditions for each thread as well as security levels for data and variables i.e. high or low data and variables. Working backwards through the execution of the program, violations of the security policy will be detected. Detected violations could be due to an incorrect assumption of the rely and guarantee conditions or a failure to uphold the security policy. This thesis will focus on the compilation stage of this tool.

2.4 Compilers and Security

Compilers are well known to be a weak link between source code and the hardware executing it. Source code that has been verified to provide a security guarantee, potentially using formal techniques, may not hold those security guarantees when being executed. This is caused by compiler optimisations that may be technically correct, however, a compiler has no notion of timing behaviour or on the expected state of memory after executing a statement [7]. This problem is known as the *correctness security gap*. One example of the correctness security gap is caused by an optimisation called dead store elimination. Figure 4 was derived from CWE-14 [6] and CWE-733 [5] and used by D'Silva et al. [7]. Here a secret key was retrieved and stored in a local variable to perform some work. After completing the work, and to prevent sensitive data from flowing into untrusted sources, the key is wiped from memory by assigning it the value 0x0.

```
crypt() {
    key := 0xCODE // Read key
    ... // Work with the key
    key := 0x0 // Clear memory
}

```

Figure 4: Implicit flow of data to a public variable [7]

From the perspective of the source code, a programmer would expect the sensitive data from key to be scrubbed after exiting the function. However, key is a variable local to the function. As key is not read after exiting the function, the statement that assigns key to a value of 0x0 will be removed as part of dead store elimination. This results in lingering memory that could be exploited by an attacker. In GCC, with compiler optimisations on, dead store elimination is performed by default [22]. Additionally, dead store elimination has been proven to be functionally correct [3][17].

This leads to the question, *what security guarantees in source code are being violated by compiler optimisations?* Although one could analyse each individual compiler optimisation to check for potential security violations in source code, defensively programming against the compiler can be counter-initiative. Additionally, compilers are getting better at optimising away tricks programmers write to work against the compiler, and thus is not a future-proof solution [26]. One might also suggest turning compiler optimisations off, however, this leads to slower code. In a concurrent system where execution time is critical, turning compiler optimisations off is not a viable option. Instead an alternative solution is to perform a static analysis on binary or assembly for security violations. As compilation has already been executed, such analysis would reveal security guarantee violations that result due to compiler optimisations.

2.5 Annotations

This project can take two routes; the proposed tool will be required to run an analysis on either binary or assembly. For either route, annotations used to guide a static security analysis will need to be provided by the user in the C programs they write. The tool will then be required to propagate these annotations down to compiled forms, i.e. binary or assembly. From here, a static analysis can be conducted as described by Winter et al. [31]. Ideally these annotations can be propagated through with little to no modification of the C Compiler being used as to reduce complexity and increase modularity and reusability of such a a tool. However, it is unclear as to whether passing annotations down with no modification to the compiler is currently possible. In this thesis, this issue will be explored.

Running a static analysis on a binary can be difficult due to the low level nature of a binary file. As such, to sufficiently perform such an analysis, the binary would be required to be decompiled to a higher-level form, such as an assembly file. From here a static analysis could be conducted. The alternative approach would be to perform the analysis directly on the compiled assembly output files rather than reducing these to binary. Currently, it is unclear as to what compiler optimisations are made when reducing an assembly file to binary, and will be explored further throughout the lifetime of this thesis. The flow of information can be viewed in Figure 5, where formats a static analysis can be performed are outlined in a dashed line. In GCC, “temporary” intermediate files can be stored using the flag *save-temps* [14]. These stored files can then be used for analysis.

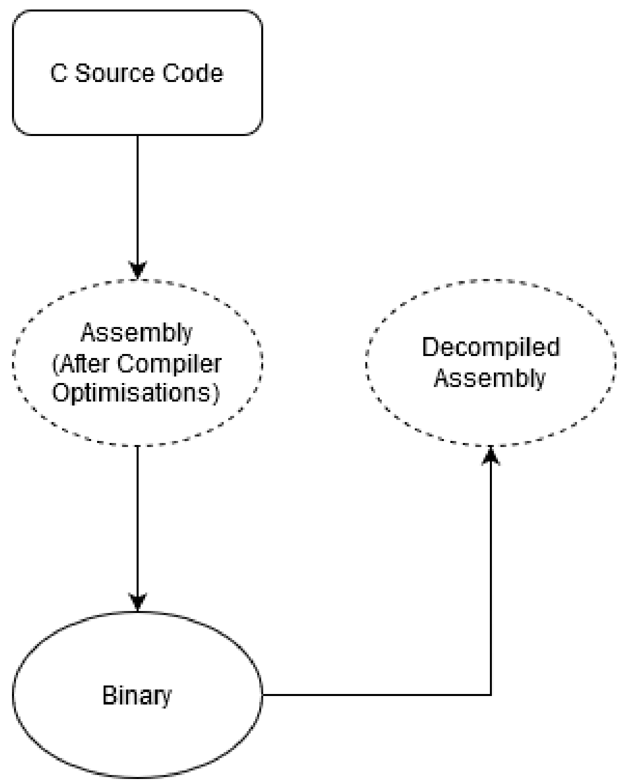

Figure 5: The static analysis options after compilation.

2.6 Related Work

In safety-critical real-time software such as flight control systems, it is required to analyse the *Worst Case Execution Time* (WCET). This kind of analysis can be conducted using static analysis tools to estimate safe upper bounds. In the case of AbsInt's aiT tool this analysis is conducted alongside compiler annotations to assist where loop bounds cannot be computed statically. In these cases, the user can provide annotations to guide the analysis tools [25]. This tool builds on an existing annotation mechanism that exist in CompCert, a C compiler that has been formally verified for use in life-critical and mission-critical software [4][18]. CompCert annotations are not limited to WCET analysis. A general mechanism for attaching free-form annotations that propagate through to assembly files can be achieved with CompCert. This approach is able to reliably transmit compiler annotations through to binary through method calls which are carried through compilation and the linked executable without using external annotation files. CompCert prints annotation strings as a comment in the generated assembly code, and an additional tool is used to parse these comments and generate annotations. However, due to its treatment as an external function, annotations cannot be placed at the top level of a compilation unit, unlike a variable declaration. Compiler optimisations can additionally cause further issues when trying to preserve annotations through compilation. If dead code is eliminated, annotations associated with that code can be lost as well. Extra care needs to be taken to avoid these optimisations destroying links between properties and the code they refer to during such transformations.

TODO: Include further documentation on how to use Compiler & inline assembly

A similar approach to CompCert is used by The ENTRA (Whole-Systems ENergy TRAnsparency). As part of providing a common assertion language, pragmas are used to propagate information through to comments in the assembler files. Information is retained in LLVM IR and ISA representations. However, these annotations are not stored in the final binary and thus comments must be extracted from assembler files [11].

Vu et al. [30] explore capturing and propagating properties from the source code level though though lowering passes and intermediate representations. Their goal was to maintain these properties to binary through aggressive compiler optimisations. As compilers only care about functional correctness, they have no notion of the link between properties and the code it refers to. Thus, there is no way to constrain transformations to preserve this link or to update these properties after the transformation. As such, they approached the problem to create a generic solution, modifying a LLVM compiler with virtually no optimisation changes. This was done by creating a library in LLVM. The properties were stored in strings, and these strings were parsed to build a list of observed variables and memory location. A LLVM pass was inserted to store all these properties in metadata. After each optimisation pass, a verification pass was inserted to check the presence of metadata representing the properties, variables amd memory locations. If an optimisation pass had cased the verification to fail the programmer would then be notified, to which they could annotate differently or disable the optimisation.

3 Approach

The approach was set out by first analysing existing methods of preserving annotations through intermediate representations. These include the:

- compCert Verified C Compiler,
- GNU Extension for Extended Inline Assembly, and
- Modifying the LLVM compiler to preserve annotations throughout intermediate representations.

Each of these approaches will be analysed individually for viability across each of the test cases outlined in section 3.1. For approaches that pass all necessary test cases, a further analysis will be conducted into its suitability and development of any necessary tools to assist in the preservation technique, as outlined in sections 3.2 and 3.4. Finally, an analysis on the runtime efficiency of the program will be conducted to assess the success of the annotations with various levels of optimisation. The approach for this analysis is outlined in 3.3.

3.1 Test Cases

A suite of test C programs (See Appendix A) were created to assist in guiding the process of evaluating each approach as a possible means of preserving annotations. Each program has inline comments documenting the annotation that should be preserved and its location within the program. Additionally, each program aims to test a separate element required to perform a static wpif analysis. Namely, these are to preserve the following through to the assembly output:

1. comments,
2. simple and complex variables (e.g. struct elements and volatile global variables),
3. security policies,
4. predicates on the initial state, and
5. loop invariants.

Each test was conducted to assess the viability of each approach of preserving annotations. If the approach cannot preserve all the required annotations described in the aforementioned list, then it is not viable for a wpif analysis and another technique must be explored.

The justification for each of the test files are as follows:

comment.c
This test case is primarily a stepping stone to testing more complex scenarios. Here we have a generic comment "critical comment" and we are looking to preserve it through to the assembly. As well as preserving the comment itself, the location of the comment within the source code is to be preserved.

variable.c
The test file *variable.c* builds off *comment.c*, however, we are additionally looking to preserve annotations about local variables within the program. Here multiple variable types are tested:

- int,
- char,
- unsigned int,
- short,
- long,
- float, and
- double.

With each of these variables their type data is included as an annotation. This test is particularly interesting as with higher levels of optimisation we can observe how the annotations behave when a variable is optimised out.

volatile.c
This test program looks at how the technique handles volatile variables. A variable declared as volatile tells the program its value could change unexpectedly. This is especially important when dealing with concurrent programs. If the technique cannot handle

volatile variables it is unable to be used for a wpif analysis.

loop.c
This test program tests how the annotator handles loops and loop invariants. It contains security policies, predicates on the initial state and loop invariants.

rooster.c
The test program, *rooster.c* delves into a more complex program, combining several features of the previous tests. It contains annotations within functions and global variables.

password.c
This program tests how annotations are preserved within structs, a user-defined data type. Additionally, *password.c* is a more complex program with multiple functions.

deadStoreElimination.c
Testing dead store elimination is a bit more complex, as it requires comparing the compiled output before and after compiler optimisations are turned on. Here, the test program simulates the program described in section 2.4.

pread.c
The program *pread.c* is a culmination of all the previous test cases, and is similar to *loop.c*, however, the global variables within it are volatile. It requires all the necessary components for a wpif analysis.

3.2 Quality Analysis

Although a method of preserving c annotations may be able to successfully pass all the test cases, it is important to avoid modifying the assembly instructions. The reason for performing a static analysis on the compiled output is due to the optimisations performed by the compiler. As such, it is important to ensure that preserved annotations do not remove or undo any optimisations that may have been performed by the compiler.

The methodology for testing quality in this manner is to compare the compiled assembly output for a program with annotations on to the compiled assembly output for the same program without annotations. If unnecessary assembly instructions have been added, it is indicative that the annotations has modified the program in unintended ways.

3.3 Efficiency and Optimisation

In the case that the annotations have introduced additional statements into the compiled assembly output, understanding the extent of these changes is important. Here, a efficiency analysis can be conducted on the assembly. Using big O notation, an upper bound can be placed on the program. Doing so allows for a comparison of the efficiency of the annotated and non-annotated assembly.

Let $A(n)$ be the function describing the annotated assembly, and $B(n)$ be the function describing the non-annotated assembly. Then,

$$A(n) \in \Theta(g(n))$$

$$B(n) \in \Theta(h(n))$$

If the non-annotated assembly has a lower bound than the annotated assembly, such that

$$h(n) \in O(g(n)), and$$

$$g(n) \notin O(h(n))$$

then the annotations have modified the program in a way that reduces runtime efficiency. It is important to detect when this has happened, as it indicates the annotations have reversed the intended compiler optimisations.

In the case where the annotation process has resulted in additional assembly instructions inserted into the compiled output, however, they do not reduce runtime efficiency in terms of big O notation, a empirical analysis of the runtime duration of a program can be conducted to assess the disadvantage of the annotated program.

3.4 Tool Development

In cases where it is appropriate, a tool may be developed to assist in the annotating process. This tool may either:

- assist in the annotating process,

- verify the correctness of the annotations, or
- perform additional analysis on the compiled output.

If the approach of modifying the LLVM compiler is pursued, developing such a tool to assist the annotating process will be necessary.

4 Execution

Experimentation began with the CompCert compiler and the provided assembly annotation tools, outlined in section 4.1. It was found that the CompCert compiler could not handle all cases necessary for the wpif analysis, specifically volatile variables. As a result, the testing moved on to other techniques. Following this, the GNU C extension for inline assembly was explored as a possibility to preserve annotations in C in section 4.2. This technique prevailed and was found to be excellent in handling assembly annotations by injecting comments in to the compiled assembly output. This technique was enhanced by developing a python program to inject inline assembly into the source C files to allow for enhanced analysis and furthermore avoids restricting the program to GNU extension supporting compilers. As a result of the success, modifying the compiler was not explored due to success documented in other research such as the work conducted by Vu et al. [30]. This allowed for further development and improvement of the inline assembly method.

4.1 CompCert AIS Annotations

CompCert is unfortunately not a free tool, however, for research purposes it can be used freely. The specifications of the CompCert install can be seen in Table 1.

Testing was initially conducted using the *comment.c* test file. The goal is to propagate the comment down to assembly where it can be used and interpreted. To do so, the comment in the source code needs to be replaced with a call to generate an annotation in the compiled assembly. Fortunately, with the CompCert compiler, this functionality is

OS Name	Ubuntu 20.04.2 LTS
OS Type	64-bit
Processor	Intel® Core™ i7-6700K CPU @ 4.00GHz × 8
Instruction Set	x86-64
CompCert Version	The CompCert C verified compiler, version 3.7

Table 1: CompCert install specifications

builtin. This assembly annotation is created through the use of the `__builtin_annot` function described in 2.6. The following builtin annotation was placed in line 2, within the main function in *comment.c*.

```
2    __builtin_ais_annot("%here Critical
     Comment");
```

Listing 1: comment.c

Within this annotation, `%here` is used to represent the location within the program. If the location is not important, `%here` can be omitted. The comment, `"Critical Comment"`, has been included to represent some kind of critical comment that is required to conduct a static analysis on the output. To compile the source to assembler only the following command was used:

```
$ ccomp comment.c -O0 -S
```

Here -O0 is used to specify to perform no optimisations during compilation. The full compiled output can be seen in Appendix C.16. Below is a snippet of the compiled assembly.

```
  .cfi_endproc
  .type main, @function
  .size main, . - main
  .section
    "__compcert_ais_annotations","",@note
  .ascii "# file:comment.c line:2
    function:main\n"
  .byte 7,8
  .quad .L100
  .ascii " Critical Comment\n"
```

Listing 2: comment-O0.s

The annotation is stored within assembler directives. Assembler directives are not a part of the processor instruction set, however, are a part of the assembler syntax. Assembler directives all start a period (.). On line 19 a new section has been created, named `"__compcert_ais_annotations"`. Following the declaration of the section is an ascii string, locating the source of the annotation within the source program *comment.c*. Line 23 provides the comment we aimed to preserve with our annotation. Thus, CompCert has shown an initial success in preserving annotations in the form of comments.

Additionally, one major benefit of compCert annotations is that they do not modify the source program, as they are inserted at the end of the program as an assembler directive metadata.

When experimenting with annotated variables, the first issues began to arise. The test file *variable.c* contains several variables with their types to preserve to assembly. The annotations behaved as expected for the types:

- int,
- char,
- short,
- long, and
- any signed or unsigned variations of the above mentioned types.

However, the CompCert annotations does not support floating point types. Upon compiling *variable.c* the following errors were generated.

```
variable.c:13: error: floating point types
    for parameter '%e1' are not supported
    in ais annotations
variable.c:15: error: floating point types
    for parameter '%e1' are not supported
    in ais annotations
2 errors detected.
```

This result shows that it is impossible to use the CompCert embedded program annotations for floating point types, vastly restricting its potential use as a technique for a wpif analysis.

It was discovered soon after that the CompCert annotations are unable to handle volatile variables, generating the follow error upon compiling *volatile.c*.

```
volatile.c:4: error: access to volatile
    variable 'x' for parameter '%e1' is not
    supported in ais annotations
1 error detected.
```

Unfortunately, this result shows that the CompCert AIS annotations approach is not suitable for wpif analysis. The wpif analysis requires use of volatile variables. This is because the primary purpose of the wpif technique is to verify security policy across concurrent programs. Shared variables within concurrent programs can change at any time, and as such it is imperative that shared variables are marked as volatile. As the CompCert AIS annotations cannot handle volatile variables, annotations required for wpif analysis cannot be generated.

Aside from the aforementioned issues, the CompCert AIS annotations performed excellently in generating annotations. The location of global variables in memory are easily identified, as shown in *rooster.c*. The CompCert AIS annotations must be placed within a method and called as if it was its own function. This creates some confusion when dealing with global variables. However, placing annotations on global variables at the start of main is a perfectly valid method of preserving these annotations. As the location of the annotation within the program is no longer important, the `%here` format specifier can be omitted.

```
  .cfi_endproc
  .type main, @function
  .size main, . - main
  .section
    "__compcert_ais_annotations","",@note
  .ascii "# file:rooster.c line:6
    function:fun\n"
  .byte 7,8
  .quad .L100
  .ascii " CRITICAL COMMENT\n"
  .ascii "# file:rooster.c line:26
    function:main\n"
  .byte 7,8
  .quad .L107
  .ascii " L(mem("
  .byte 7,8
  .quad goose
  .ascii ", 4)) = medium\n"
  .ascii "# file:rooster.c line:27
    function:main\n"
  .byte 7,8
  .quad .L108
```

```
102   .ascii " EXCEPTIONAL\n"
```

Listing 3: rooster-O0.s

From *rooster.c*, the comment "CRITICAL COMMENT" has been annotated from lines 88 to 91, and the comment "EXCEPTIONAL" has been annotated from lines 99 to 102. Most notably, the global variable `goose` has been annotated from lines 92 to 98. Reconstructed, the string `"L(mem(goose, 4)) = medium"` has been preserved. Thus, the CompCert annotations can successfully preserve annotations on global variables.

Another interesting problem faced when working with CompCert AIS annotations is found when working with structs. If the programmer wants to annotate a member of a struct for all structs of that type, each instance of that type of struct must be annotated when using CompCert AIS annotations. This is because CompCert treats `__builtin_ais_annot()` as a call to an external function. As such, an annotation cannot be created from outside a method, similar to when dealing with global variables. An example of this process can be seen in *password.c*. Within the program, each instantiation of the struct `user_t` requires another annotation.

```
    user_t* user_admin = malloc(sizeof(user_t));
    strcpy(user_admin->name, "admin");
    strcpy(user_admin->password, "4dm1n__4eva");
    __builtin_ais_annot("%here L(%e1) = high", user_admin->password);
    user_admin->balance = 1000000;

    user_t* user_alice = malloc(sizeof(user_t));
    strcpy(user_alice->name, "alice");
    strcpy(user_alice->password, "!alice12!_veuje@@hak");
    __builtin_ais_annot("%here L(%e1) = high", user_alice->password);
    user_alice->balance = 783;

    user_t* user_abdul = malloc(sizeof(user_t));
    strcpy(user_abdul->name, "abdul");
    strcpy(user_abdul->password, "passw0rd123");
    __builtin_ais_annot("%here L(%e1) = high", user_abdul->password);
    user_abdul->balance = 2;
```

Listing 4: password.c

The compiled output is as expected, with an annotation within the assembly for each of the annotations created within the source file.

```
320  .section "__compcert_ais_annotations","",@note
321  .ascii "# file:password.c line:20 function:setup_users\n"
322  .byte 7,8
323  .quad .L100
324  .ascii " L((reg(\"rbp\") + 264)) = high\n"
325  .ascii "# file:password.c line:26 function:setup_users\n"
326  .byte 7,8
327  .quad .L101
328  .ascii " L((reg(\"r12\") + 264)) = high\n"
329  .ascii "# file:password.c line:32 function:setup_users\n"
330  .byte 7,8
331  .quad .L102
332  .ascii " L((reg(\"rbx\") + 264)) = high\n"
```

Listing 5: password-O0.s

As seen in the assembly annotations, the location of the struct members have been preserved. Line 324 contains the annotation `L((reg("rbp") + 264)) = high`. This annotation notifies that the variable stored in register rbp with an offset of 264 has a security classification of high. Thus, another success for CompCert AIS annotations.

4.1.1 Quality Analysis

- Look at comparison with optimisation turned on

4.2 Inline Assembly

TODO: Inline assembly techniques

4.3 CompCert Builtin Annotations

4.4 LLVM Compiler Modification

the final technique of modifying the LLVM compiler was not experimented on. This was primarily due to two reasons. To begin with, the primary objective of this thesis is to explore techniques that do not modify the compiler, and instead work alongside the functionality of the compiler to preserve annotations. It is well known and documented that modifying the compiler to preserve annotations is possible and successful, as in the case of Vu et al. [30] Additionally, earlier success through the technique of using inline assembly allowed for more time to be allocated to exploring and improving this technique, as seen in 4.2. Therefore, evaluating compiler modification for static analysis purposes was not performed in this research.

References

[1] Pieter Agten et al. "Recent developments in low-level software security". In: *IFIP International Workshop on Information Security Theory and Practice*. Springer. 2012, pp. 1–16.

[2] Musard Balliu. "Logics for information flow security: from specification to verification". PhD thesis. KTH Royal Institute of Technology, 2014.

[3] Nick Benton. "Simple relational correctness proofs for static analyses and program transformations". In: *ACM SIGPLAN Notices* 39.1 (2004), pp. 14–25.

[4] *CompCert - The CompCert C compiler*. Accessed: 2020-09-01. 2020. URL: `http://compcert.inria.fr/compcert-C.html`.

[5] *Compiler Optimization Removal or Modification of Security-critical Code*. Accessed: 2020-09-01. 2008. URL: `https://cwe.mitre.org/data/definitions/733.html`.

[6] *Compiler Removal of Code to Clear Buffers*. Accessed: 2020-09-01. 2006. URL: `https://cwe.mitre.org/data/definitions/14.html`.

[7] Vijay D'Silva, Mathias Payer, and Dawn Song. "The correctness-security gap in compiler optimization". In: *2015 IEEE Security and Privacy Workshops*. IEEE. 2015, pp. 73–87.

[8] Dorothy E Denning. "A lattice model of secure information flow". In: *Communications of the ACM* 19.5 (1976), pp. 236–243.

[9] Dorothy E Denning and Peter J Denning. "Certification of programs for secure information flow". In: *Communications of the ACM* 20.7 (1977), pp. 504–513.

[10] Quang Do, Ben Martini, and Kim-Kwang Raymond Choo. "The role of the adversary model in applied security research". In: *Computers & Security* 81 (2019), pp. 156–181.

[11] K Eder, K Georgiou, and N Grech. *Common Assertion Language. ENTRA Project: Whole-Systems Energy Transparency (FET project 318337). Deliverable 2.1.* 2013.

[12] Gidon Ernst and Toby Murray. "SecCSL: Security concurrent separation logic". In: *International Conference on Computer Aided Verification*. Springer. 2019, pp. 208–230.

[13] *File:Simplified Heartbleed explanation.svg - Wikimedia Commons*. Accessed: 2020-09-02. 2014. URL: `https://commons.wikimedia.org/wiki/File:Simplified_Heartbleed_explanation.svg#mediaviewer/File:Simplified_Heartbleed_explanation.svg`.

[14] *GCC Developer Options*. Accessed: 2020-09-02. URL: `https://gcc.gnu.org/onlinedocs/gcc/Developer-Options.html`.

[15] Barbara Guttman and Edward A Roback. *An introduction to computer security: the NIST handbook*. Diane Publishing, 1995.

[16] *Heartbleed Bug*. Accessed: 2020-09-02. 2020. URL: `https://heartbleed.com/`.

[17] Xavier Leroy. "Formal certification of a compiler back-end or: programming a compiler with a proof assistant". In: *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2006, pp. 42–54.

[18] Xavier Leroy et al. "CompCert-a formally verified optimizing compiler". In: 2016.

[19] Heiko Mantel, Matthias Perner, and Jens Sauer. "Noninterference under weak memory models". In: *2014 IEEE 27th Computer Security Foundations Symposium*. IEEE. 2014, pp. 80–94.

[20] Heiko Mantel, David Sands, and Henning Sudbrock. "Assumptions and guarantees for compositional noninterference". In: *2011 IEEE 24th Computer Security Foundations Symposium*. IEEE. 2011, pp. 218–232.

[21] Toby Murray, Robert Sison, and Kai Engelhardt. "COVERN: A logic for compositional verification of information flow control". In: *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2018, pp. 16–30.

[22] *Options That Control Optimization*. Accessed: 2020-09-01. URL: `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`.

[23] Sundeep Prakash, Yann-Hang Lee, and Theodore Johnson. "Non-blocking algorithms for concurrent data structures". MA thesis. Citeseer, 1991.

[24] Andrei Sabelfeld and Andrew C Myers. "Language-based information-flow security". In: *IEEE Journal on selected areas in communications* 21.1 (2003), pp. 5–19.

[25] Bernhard Schommer et al. "Embedded program annotations for WCET analysis". In: *18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2018.

[26] Laurent Simon, David Chisnall, and Ross Anderson. "What you get is what you C: Controlling side effects in mainstream C compilers". In: *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2018, pp. 1–15.

[27] Graeme Smith, Nicholas Coughlin, and Toby Murray. "Value-Dependent Information-Flow Security on Weak Memory Models". In: *International Symposium on Formal Methods*. Springer. 2019, pp. 539–555.

[28] William Stallings et al. *Computer security: principles and practice*. Pearson Education Upper Saddle River, NJ, USA, 2012.

[29] Jeffrey A Vaughan and Todd Millstein. "Secure information flow for concurrent programs under total store order". In: *2012 IEEE 25th Computer Security Foundations Symposium*. IEEE. 2012, pp. 19–29.

[30] Son Tuan Vu et al. "Secure delivery of program properties through optimizing compilation". In: *Proceedings of the 29th International Conference on Compiler Construction*. 2020, pp. 14–26.

[31] Kirsten Winter, Graeme Smith, and Nicholas Coughlin. "Information flow security in the presence of fine-grained concurrency". Unpublished. Aug. 2020.

Appendices

A Test C Programs

A.1 comment.c

```c
int main() {
    // Critical Comment
    return 0;
}
```

A.2 variable.c

```c
int main(int argc, char* argv[]) {
    // a = int
    // b = char
    // c = unsigned int
    // d = short
    // e = long
    // x = float
    // y = double
    int a = -10;
    char b = 'b';
    unsigned int c = -b;
    short d = 0x1;
    long e = 4294967296;
    float x = 3.141592653589793;
    double y = x / 2.3784;
    return (int)(e / 32) + (int)a + (int)c + (int)d + (int) x + (int) y + argc;
}
```

A.3 volatile.c

```c
volatile int x;

int main() {
    // L(x) = High
    return x + 1;
}
```

A.4 loop.c

```c
int z;
int x;

// security policies
// {L(z)=true}
// {L(x)=z % 2 == 0}

// predicates on initial state
// {_P_0: r1 % 2 == 0}
// {_Gamma_0: r1 -> LOW, r2 -> LOW}

int main() {
```

```
    int r1 = 0;
    // {L(r2)=False}
    int r2 = 0;

    while(1) {
    do {
        // {_invariant: r1 % 2 == 0 /\ r1 <= z}
        // {_Gamma: r1 -> LOW, r2 -> (r1 == z), z -> LOW}
        do {
            // {_invariant: r1 <= z}
            // {_Gamma: r1 -> LOW}
            r1 = z;
        } while (r1 %2 != 0);
            r2 = x;
        } while (z != r1);
    }
    return r2;
}
```

A.5 rooster.c

```
int rooster;
int drake;
// MEDIUM
int goose;

int fun(int a, int b, int c) {
    // CRITICAL COMMENT
    static int count = 0;
    int sum = a + b + c;
    if (sum < 0) {
        return sum;
    }
    if (a < b && b < c) {
        while (a != b) {
            a++;
            count++;
            while (b != c) {
                c--;
                count++;
            }
        }
    }
    return count;
}

int main(void) {
    // EXCEPTIONAL
    rooster = 1;
    drake = 5;
    goose = 10;
    int result;
    result = fun(rooster,drake,goose);
    return 0;
}
```

A.6 password.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define BUFF_LEN 256

typedef struct user_t user_t;

struct user_t {
    user_t* next;
    char name[BUFF_LEN];
    // L(password) = High
    char password[BUFF_LEN];
    size_t balance;
};

user_t* setup_users() {
    user_t* user_admin = malloc(sizeof(user_t));
    strcpy(user_admin->name, "admin");
    strcpy(user_admin->password, "4dm1n__4eva");
    user_admin->balance = 1000000;

    user_t* user_alice = malloc(sizeof(user_t));
    strcpy(user_alice->name, "alice");
    strcpy(user_alice->password, "!alice12!_veuje@@hak");
    user_alice->balance = 783;

    user_t* user_abdul = malloc(sizeof(user_t));
    strcpy(user_abdul->name, "abdul");
    strcpy(user_abdul->password, "passw0rd123");
    user_abdul->balance = 2;

    user_admin->next = user_alice;
    user_alice->next = user_abdul;
    user_abdul->next = NULL;

    return user_admin;
}

void print_users(user_t* users) {
    printf("--- USERS ---\n");
    size_t count = 0;
    while (users != NULL) {
        printf(" %02ld. %s\n", ++count, users->name);
        users = users->next;
    }
    printf("\n");
}

user_t* getUser(user_t* user_list, char* name) {
    while (user_list != NULL) {
        if (strcmp(user_list->name, name) == 0) {
            return user_list;
        }
        user_list = user_list->next;
    }
    return NULL;
```

```c
}

int main() {
    user_t* users = setup_users();

    printf("Welcome to BigBank Australia!\n");

    char username[BUFF_LEN];
    printf("Username: ");
    scanf("%255s", username);

    user_t* user = getUser(users, username);
    if (user == NULL) {
        printf("User < %s > does not exist.\n", username);
        return 0;
    }

    char password[BUFF_LEN];
    printf("Password: ");
    scanf("%255s", password);
    if (strcmp(user->password, password) != 0) {
        printf("ERROR: incorrect password\n");
        return 0;
    }

    printf("Logged in as < %s >!\n", user->name);
    printf("\n");
    printf("Welcome, %s!\n", user->name);
    printf("Your balance: $%ld\n", user->balance);
}
```

A.7 deadStoreElimination.c

```c
int deadStore(int i, int n) {
    int key = 0xabcd;
    // L(key) = high

    // do some work
    int result = 0;
    while (i > n) {
        result += key;
        i--;
    }

    // clear out our secret key
    key = 0;
    return i + n;
}

int main(int argc, char *argv[]) {
    deadStore(argc, 2);
}
```

A.8 pread.c

```c
volatile int z;
volatile int x;
```

```

// security policies
// {L(z)=true}
// {L(x)=z % 2 == 0}

// predicates on initial state
// {_P_0: r1 % 2 == 0}
// {_Gamma_0: r1 -> LOW, r2 -> LOW}

int main() {
    int r1 = 0;
    // {L(r2)=False}
    int r2 = 0;

    while(1) {
    do {
        // {_invariant: r1 % 2 == 0 /\ r1 <= z}
        // {_Gamma: r1 -> LOW, r2 -> (r1 == z), z -> LOW}
        do {
            // {_invariant: r1 <= z}
            // {_Gamma: r1 -> LOW}
            r1 = z;
        } while (r1 %2 != 0);
            r2 = x;
        } while (z != r1);
    }
    return r2;
}
```

B CompCert Annotated C Programs

B.1 comment.c

```
int main() {
    __builtin_ais_annot("%here Critical Comment");
    return 0;
}
```

B.2 variable.c

```
int main(int argc, char* argv[]) {
    int a = -10;
    __builtin_ais_annot("%here %e1 = int", a);
    char b = 'b';
    __builtin_ais_annot("%here %e1 = char", b);
    unsigned int c = -b;
    __builtin_ais_annot("%here %e1 = unsigned int", c);
    short d = 0x1;
    __builtin_ais_annot("%here %e1 = short", d);
    long e = 4294967296;
    __builtin_ais_annot("%here %e1 = long", e);
    float x = 3.141592653589793;
    __builtin_ais_annot("%here %e1 = float", x);
    double y = x / 2.3784;
    __builtin_ais_annot("%here %e1 = double", y);
    return (int)(e / 32) + (int)a + (int)c + (int)d + (int) x + (int) y + argc;
```

```
}
```

B.3 volatile.c

```
volatile int x;

int main() {
    __builtin_ais_annot("%here L(%e1)= false", x);
    return x + 1;
}
```

B.4 loop.c

```

int z;
int x;

int main() {
    // Security Policies
    __builtin_ais_annot("%here L(%e1) = true", z);
    __builtin_ais_annot("%here L(%e1)= %e2 %% 2 == 0", x, z);
    int r1 = 0;
    int r2 = 0;
    __builtin_ais_annot("%here L(%e1)= false", r2);

    // Predicates on initial state
    __builtin_ais_annot("%here _P_0: %e1 %% 2 == 0", r1);
    __builtin_ais_annot("%here _Gamma_0: %e1 -> LOW, %e2 -> LOW", r1, r2);

    while(1) {
    do {
        __builtin_ais_annot("%here _invariant: %e1 %% 2 == 0 & %e1 <= %e2", r1, z);
        __builtin_ais_annot("%here _Gamma: %e1 -> LOW, %e2 -> (%e1 == %e3), %e3 -> LOW",
    r1, r2, z);
        do {
            __builtin_ais_annot("%here _invariant: %e1 <= %e2", r1, z);
            __builtin_ais_annot("%here _Gamma: %e1 -> LOW", r1);
            r1 = z;
        } while (r1 %2 != 0);
            r2 = x;
        } while (z != r1);
    }
    return r2;
}
```

B.5 rooster.c

```
int rooster;
int drake;
int goose;

int fun(int a, int b, int c) {
    __builtin_ais_annot("%here CRITICAL COMMENT");
    static int count = 0;
    int sum = a + b + c;
    if (sum < 0) {
        return sum;
```

```
    }
    if (a < b && b < c) {
        while (a != b) {
            a++;
            count++;
            while (b != c) {
                c--;
                count++;
            }
        }
    }
    return count;
}

int main(void) {
    __builtin_ais_annot("%here L(%e1) = medium", goose);
    __builtin_ais_annot("%here EXCEPTIONAL");
    rooster = 1;
    drake = 5;
    goose = 10;
    int result;
    result = fun(rooster,drake,goose);
    return 0;
}
```

B.6 password.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define BUFF_LEN 256

typedef struct user_t user_t;

struct user_t {
    user_t* next;
    char name[BUFF_LEN];
    char password[BUFF_LEN];          //           { L(password) = High }
    size_t balance;
};

user_t* setup_users() {
    user_t* user_admin = malloc(sizeof(user_t));
    strcpy(user_admin->name, "admin");
    strcpy(user_admin->password, "4dm1n__4eva");
    __builtin_ais_annot("%here L(%e1) = high", user_admin->password);
    user_admin->balance = 1000000;

    user_t* user_alice = malloc(sizeof(user_t));
    strcpy(user_alice->name, "alice");
    strcpy(user_alice->password, "!alice12!_veuje@@hak");
    __builtin_ais_annot("%here L(%e1) = high", user_alice->password);
    user_alice->balance = 783;

    user_t* user_abdul = malloc(sizeof(user_t));
    strcpy(user_abdul->name, "abdul");
```

```c
    strcpy(user_abdul->password, "passw0rd123");
    __builtin_ais_annot("%here L(%e1) = high", user_abdul->password);
    user_abdul->balance = 2;

    user_admin->next = user_alice;
    user_alice->next = user_abdul;
    user_abdul->next = NULL;

    return user_admin;
}

void print_users(user_t* users) {
    printf("--- USERS ---\n");
    size_t count = 0;
    while (users != NULL) {
        printf(" %02ld. %s\n", ++count, users->name);
        users = users->next;
    }
    printf("\n");
}

user_t* getUser(user_t* user_list, char* name) {
    while (user_list != NULL) {
        if (strcmp(user_list->name, name) == 0) {
            return user_list;
        }
        user_list = user_list->next;
    }
    return NULL;
}

int main() {
    user_t* users = setup_users();

    printf("Welcome to BigBank Australia!\n");

    char username[BUFF_LEN];
    printf("Username: ");
    scanf("%255s", username);

    user_t* user = getUser(users, username);
    if (user == NULL) {
        printf("User < %s > does not exist.\n", username);
        return 0;
    }

    char password[BUFF_LEN];
    printf("Password: ");
    scanf("%255s", password);
    if (strcmp(user->password, password) != 0) {
        printf("ERROR: incorrect password\n");
        return 0;
    }

    printf("Logged in as < %s >!\n", user->name);
    printf("\n");
    printf("Welcome, %s!\n", user->name);
```

```
    printf("Your balance: $%ld\n", user->balance);
}
```

B.7 deadStoreElimination.c

```
int deadStore(int i, int n) {
    int key = 0xabcd;
    __builtin_ais_annot("%here L(%e1) = high", key);

    // do some work
    int result = 0;
    while (i > n) {
        result += key;
        i--;
    }

    // clear out our secret key
    key = 0;
    return i + n;
}

int main(int argc, char *argv[]) {
    deadStore(argc, 2);
}
```

B.8 pread.c

```
volatile int z;
volatile int x;

int main() {
    // Security Policies
    __builtin_ais_annot("%here L(%e1) = true", z);
    __builtin_ais_annot("%here L(%e1)= %e2 %% 2 == 0", x, z);

    int r1 = 0;
    int r2 = 0;          //             {L(r2)=False}
    __builtin_ais_annot("%here L(%e1)= false", r2);

    // Predicates on initial state
    __builtin_ais_annot("%here _P_0: %e1 %% 2 == 0", r1);
    __builtin_ais_annot("%here _Gamma_0: %e1 -> LOW, %e2 -> LOW", r1, r2);


    while(1) {
        do {
            __builtin_ais_annot("%here _invariant: %e1 %% 2 == 0 & %e1 <= %e2", r1, z);
            __builtin_ais_annot("%here _Gamma: %e1 -> LOW, %e2 -> (%e1 == %e3), %e3 ->
    LOW", r1, r2, z);
            do {
                __builtin_ais_annot("%here _invariant: %e1 <= %e2", r1, z);
                __builtin_ais_annot("%here _Gamma: %e1 -> LOW", r1);
                r1 = z;
            } while (r1 %2 != 0);
                r2 = x;
        } while (z != r1);
    }
```

```
    return r2;
}
```

C CompCert Annotated Assembly Output

C.1 comment-O0.s

```
# File generated by CompCert 3.7
# Command line: comment.c -O0 -S
  .text
  .align  16
  .globl main
main:
  .cfi_startproc
  subq  $8, %rsp
  .cfi_adjust_cfa_offset  8
  leaq  16(%rsp), %rax
  movq  %rax, 0(%rsp)
.L100:
  xorl  %eax, %eax
  addq  $8, %rsp
  ret
  .cfi_endproc
  .type main, @function
  .size main, . - main
  .section  "__compcert_ais_annotations","",@note
  .ascii "# file:comment.c line:2 function:main\n"
  .byte 7,8
  .quad .L100
  .ascii " Critical Comment\n"
```

C.2 comment-O3.s

```
# File generated by CompCert 3.7
# Command line: comment.c -O3 -S
  .text
  .align  16
  .globl main
main:
  .cfi_startproc
  subq  $8, %rsp
  .cfi_adjust_cfa_offset  8
  leaq  16(%rsp), %rax
  movq  %rax, 0(%rsp)
.L100:
  xorl  %eax, %eax
  addq  $8, %rsp
  ret
  .cfi_endproc
  .type main, @function
  .size main, . - main
  .section  "__compcert_ais_annotations","",@note
  .ascii "# file:comment.c line:2 function:main\n"
  .byte 7,8
  .quad .L100
  .ascii " Critical Comment\n"
```

C.3 variable-O0.s

```
# File generated by CompCert 3.7
# Command line: variable.c -S -O0
	.text
	.align	16
	.globl main
main:
	.cfi_startproc
	subq	$8, %rsp
	.cfi_adjust_cfa_offset	8
	leaq	16(%rsp), %rax
	movq	%rax, 0(%rsp)
	movl	$-10, %esi
.L100:
	movl	$98, %ecx
.L101:
	negl	%ecx
.L102:
	movl	$1, %r10d
.L103:
	movabsq $4294967296, %rax
.L104:
	movsd .L105(%rip), %xmm1 # 3.14159265358979312
	cvtsd2ss %xmm1, %xmm3
	cvtss2sd %xmm3, %xmm0
	movsd .L106(%rip), %xmm2 # 2.37840000000000007
	divsd %xmm2, %xmm0
	cqto
	shrq	$59, %rdx
	leaq	0(%rax,%rdx,1), %rax
	sarq	$5, %rax
	leal	0(%eax,%esi,1), %r9d
	leal	0(%r9d,%ecx,1), %r8d
	leal	0(%r8d,%r10d,1), %r10d
	cvttss2si %xmm3, %edx
	leal	0(%r10d,%edx,1), %r11d
	cvttsd2si %xmm0, %r8d
	leal	0(%r11d,%r8d,1), %ecx
	leal	0(%ecx,%edi,1), %eax
	addq	$8, %rsp
	ret
	.cfi_endproc
	.type main, @function
	.size main, . - main
	.section	.rodata.cst8,"aM",@progbits,8
	.align	8
.L105:	.quad 0x400921fb54442d18
.L106:	.quad 0x400306f694467382
	.section	"__compcert_ais_annotations","",@note
	.ascii "# file:variable.c line:3 function:main\n"
	.byte 7,8
	.quad .L100
	.ascii " reg(\"rsi\") = int\n"
	.ascii "# file:variable.c line:5 function:main\n"
	.byte 7,8
	.quad .L101
```

```
  .ascii " reg(\"rcx\") = char\n"
  .ascii "# file:variable.c line:7 function:main\n"
  .byte 7,8
  .quad .L102
  .ascii " reg(\"rcx\") = unsigned int\n"
  .ascii "# file:variable.c line:9 function:main\n"
  .byte 7,8
  .quad .L103
  .ascii " reg(\"r10\") = short\n"
  .ascii "# file:variable.c line:11 function:main\n"
  .byte 7,8
  .quad .L104
  .ascii " reg(\"rax\") = long\n"
```

C.4 variable-O3.s

```
# File generated by CompCert 3.7
# Command line: variable.c -S -O3
  .text
  .align  16
  .globl main
main:
  .cfi_startproc
  subq  $8, %rsp
  .cfi_adjust_cfa_offset  8
  leaq  16(%rsp), %rax
  movq  %rax, 0(%rsp)
.L100:
.L101:
.L102:
.L103:
.L104:
  leal  134217625(%edi), %eax
  addq  $8, %rsp
  ret
  .cfi_endproc
  .type main, @function
  .size main, . - main
  .section  "__compcert_ais_annotations","",@note
  .ascii "# file:variable.c line:3 function:main\n"
  .byte 7,8
  .quad .L100
  .ascii " -10 = int\n"
  .ascii "# file:variable.c line:5 function:main\n"
  .byte 7,8
  .quad .L101
  .ascii " 98 = char\n"
  .ascii "# file:variable.c line:7 function:main\n"
  .byte 7,8
  .quad .L102
  .ascii " -98 = unsigned int\n"
  .ascii "# file:variable.c line:9 function:main\n"
  .byte 7,8
  .quad .L103
  .ascii " 1 = short\n"
  .ascii "# file:variable.c line:11 function:main\n"
  .byte 7,8
```

```
  .quad .L104
  .ascii " 4294967296 = long\n"
```

C.5 volatile-O0.s

```
volatile.c:4: error: access to volatile variable 'x' for parameter '%e1' is not supported
    in ais annotations
1 error detected.
```

C.6 volatile-O3.s

```
volatile.c:4: error: access to volatile variable 'x' for parameter '%e1' is not supported
    in ais annotations
1 error detected.
```

C.7 loop-O0.s

```
# File generated by CompCert 3.7
# Command line: loop.c -O0 -S
  .comm z, 4, 4
  .comm x, 4, 4
  .text
  .align  16
  .globl main
main:
  .cfi_startproc
  subq  $8, %rsp
  .cfi_adjust_cfa_offset  8
  leaq  16(%rsp), %rax
  movq  %rax, 0(%rsp)
.L100:
.L101:
  xorl  %edx, %edx
  xorl  %edi, %edi
.L102:
.L103:
.L104:
  nop
.L105:
.L106:
.L107:
  nop
.L108:
.L109:
.L110:
  movl  z(%rip), %edx
  movq  %rdx, %rax
  testl %eax, %eax
  leal  1(%eax), %ecx
  cmovl %rcx, %rax
  sarl  $1, %eax
  leal  0(,%eax,2), %edi
  movq  %rdx, %rcx
  subl  %edi, %ecx
  testl %ecx, %ecx
  jne .L108
  movl  x(%rip), %edi
```

```
movl  z(%rip), %esi
jmp .L105
.cfi_endproc
.type main, @function
.size main, . - main
.section  "__compcert_ais_annotations","",@note
.ascii "# file:loop.c line:7 function:main\n"
.byte 7,8
.quad .L100
.ascii " L(mem("
.byte 7,8
.quad z
.ascii ", 4)) = true\n"
.ascii "# file:loop.c line:8 function:main\n"
.byte 7,8
.quad .L101
.ascii " L(mem("
.byte 7,8
.quad x
.ascii ", 4))= mem("
.byte 7,8
.quad z
.ascii ", 4) % 2 == 0\n"
.ascii "# file:loop.c line:11 function:main\n"
.byte 7,8
.quad .L102
.ascii " L(reg(\"rdi\"))= false\n"
.ascii "# file:loop.c line:14 function:main\n"
.byte 7,8
.quad .L103
.ascii " _P_0: reg(\"rdx\") % 2 == 0\n"
.ascii "# file:loop.c line:15 function:main\n"
.byte 7,8
.quad .L104
.ascii " _Gamma_0: reg(\"rdx\") -> LOW, reg(\"rdi\") -> LOW\n"
.ascii "# file:loop.c line:19 function:main\n"
.byte 7,8
.quad .L106
.ascii " _invariant: reg(\"rdx\") % 2 == 0 & reg(\"rdx\") <= mem("
.byte 7,8
.quad z
.ascii ", 4)\n"
.ascii "# file:loop.c line:20 function:main\n"
.byte 7,8
.quad .L107
.ascii " _Gamma: reg(\"rdx\") -> LOW, reg(\"rdi\") -> (reg(\"rdx\") == mem("
.byte 7,8
.quad z
.ascii ", 4)), mem("
.byte 7,8
.quad z
.ascii ", 4) -> LOW\n"
.ascii "# file:loop.c line:22 function:main\n"
.byte 7,8
.quad .L109
.ascii " _invariant: reg(\"rdx\") <= mem("
.byte 7,8
```

```
  .quad z
  .ascii ", 4)\n"
  .ascii "# file:loop.c line:23 function:main\n"
  .byte 7,8
  .quad .L110
  .ascii " _Gamma: reg(\"rdx\") -> LOW\n"
```

C.8 loop-O3.s

```
# File generated by CompCert 3.7
# Command line: loop.c -O3 -S
  .comm z, 4, 4
  .comm x, 4, 4
  .text
  .align  16
  .globl main
main:
  .cfi_startproc
  subq  $8, %rsp
  .cfi_adjust_cfa_offset  8
  leaq  16(%rsp), %rax
  movq  %rax, 0(%rsp)
.L100:
.L101:
  xorl  %edx, %edx
  xorl  %edi, %edi
.L102:
.L103:
.L104:
  nop
.L105:
.L106:
.L107:
  nop
.L108:
.L109:
.L110:
  movl  z(%rip), %edx
  movq  %rdx, %rax
  testl %eax, %eax
  leal  1(%eax), %ecx
  cmovl %rcx, %rax
  sarl  $1, %eax
  leal  0(,%eax,2), %edi
  movq  %rdx, %rcx
  subl  %edi, %ecx
  testl %ecx, %ecx
  jne .L108
  movl  x(%rip), %edi
  movq  %rdx, %rsi
  jmp .L105
  .cfi_endproc
  .type main, @function
  .size main, . - main
  .section  "__compcert_ais_annotations","",@note
  .ascii "# file:loop.c line:7 function:main\n"
  .byte 7,8
```

```
.quad .L100
.ascii " L(mem("
.byte 7,8
.quad z
.ascii ", 4)) = true\n"
.ascii "# file:loop.c line:8 function:main\n"
.byte 7,8
.quad .L101
.ascii " L(mem("
.byte 7,8
.quad x
.ascii ", 4))= mem("
.byte 7,8
.quad z
.ascii ", 4) % 2 == 0\n"
.ascii "# file:loop.c line:11 function:main\n"
.byte 7,8
.quad .L102
.ascii " L(0)= false\n"
.ascii "# file:loop.c line:14 function:main\n"
.byte 7,8
.quad .L103
.ascii " _P_0: 0 % 2 == 0\n"
.ascii "# file:loop.c line:15 function:main\n"
.byte 7,8
.quad .L104
.ascii " _Gamma_0: 0 -> LOW, 0 -> LOW\n"
.ascii "# file:loop.c line:19 function:main\n"
.byte 7,8
.quad .L106
.ascii " _invariant: reg(\"rdx\") % 2 == 0 & reg(\"rdx\") <= mem("
.byte 7,8
.quad z
.ascii ", 4)\n"
.ascii "# file:loop.c line:20 function:main\n"
.byte 7,8
.quad .L107
.ascii " _Gamma: reg(\"rdx\") -> LOW, reg(\"rdi\") -> (reg(\"rdx\") == mem("
.byte 7,8
.quad z
.ascii ", 4)), mem("
.byte 7,8
.quad z
.ascii ", 4) -> LOW\n"
.ascii "# file:loop.c line:22 function:main\n"
.byte 7,8
.quad .L109
.ascii " _invariant: reg(\"rdx\") <= mem("
.byte 7,8
.quad z
.ascii ", 4)\n"
.ascii "# file:loop.c line:23 function:main\n"
.byte 7,8
.quad .L110
.ascii " _Gamma: reg(\"rdx\") -> LOW\n"
```

C.9 rooster-O0.s

```
# File generated by CompCert 3.7
# Command line: rooster.c -O0 -S
  .comm rooster, 4, 4
  .comm drake, 4, 4
  .comm goose, 4, 4
  .data
  .align  4
count:
  .long 0
  .type count, @object
  .size count, . - count
  .text
  .align  16
  .globl fun
fun:
  .cfi_startproc
  subq  $8, %rsp
  .cfi_adjust_cfa_offset  8
  leaq  16(%rsp), %rax
  movq  %rax, 0(%rsp)
.L100:
  leal  0(%edi,%esi,1), %r8d
  leal  0(%r8d,%edx,1), %eax
  testl %eax, %eax
  jl   .L101
  cmpl  %esi, %edi
  jl   .L102
  xorl  %r8d, %r8d
  jmp .L103
.L102:
  cmpl  %edx, %esi
  setl  %r8b
  movzbl  %r8b, %r8d
.L103:
  cmpl  $0, %r8d
  je   .L104
.L105:
  cmpl  %esi, %edi
  je   .L104
  leal  1(%edi), %edi
  movl  count(%rip), %eax
  leal  1(%eax), %ecx
  movl  %ecx, count(%rip)
.L106:
  cmpl  %edx, %esi
  je   .L105
  leal  -1(%edx), %edx
  movl  count(%rip), %r9d
  leal  1(%r9d), %r8d
  movl  %r8d, count(%rip)
  jmp .L106
.L104:
  movl  count(%rip), %eax
.L101:
  addq  $8, %rsp
  ret
  .cfi_endproc
```

```
  .type fun, @function
  .size fun, . - fun
  .text
  .align  16
  .globl main
main:
  .cfi_startproc
  subq  $8, %rsp
  .cfi_adjust_cfa_offset  8
  leaq  16(%rsp), %rax
  movq  %rax, 0(%rsp)
.L107:
.L108:
  movl  $1, %eax
  movl  %eax, rooster(%rip)
  movl  $5, %eax
  movl  %eax, drake(%rip)
  movl  $10, %eax
  movl  %eax, goose(%rip)
  movl  rooster(%rip), %edi
  movl  drake(%rip), %esi
  movl  goose(%rip), %edx
  call  fun
  xorl  %eax, %eax
  addq  $8, %rsp
  ret
  .cfi_endproc
  .type main, @function
  .size main, . - main
  .section  "__compcert_ais_annotations","",@note
  .ascii "# file:rooster.c line:6 function:fun\n"
  .byte 7,8
  .quad .L100
  .ascii " CRITICAL COMMENT\n"
  .ascii "# file:rooster.c line:26 function:main\n"
  .byte 7,8
  .quad .L107
  .ascii " L(mem("
  .byte 7,8
  .quad goose
  .ascii ", 4)) = medium\n"
  .ascii "# file:rooster.c line:27 function:main\n"
  .byte 7,8
  .quad .L108
  .ascii " EXCEPTIONAL\n"
```

C.10 rooster-O3.s

```
# File generated by CompCert 3.7
# Command line: rooster.c -O3 -S
  .comm rooster, 4, 4
  .comm drake, 4, 4
  .comm goose, 4, 4
  .data
  .align  4
count:
  .long 0
```

```
  .type count, @object
  .size count, . - count
  .text
  .align  16
  .globl fun
fun:
  .cfi_startproc
  subq  $8, %rsp
  .cfi_adjust_cfa_offset  8
  leaq  16(%rsp), %rax
  movq  %rax, 0(%rsp)
.L100:
  leal  0(%edi,%esi,1), %r9d
  leal  0(%r9d,%edx,1), %eax
  testl %eax, %eax
  jl   .L101
  cmpl  %edx, %esi
  setl  %al
  movzbl  %al, %eax
  xorl  %r8d, %r8d
  cmpl  %esi, %edi
  cmovge  %r8, %rax
  cmpl  $0, %eax
  je   .L102
.L103:
  cmpl  %esi, %edi
  je   .L102
  leal  1(%edi), %edi
  movl  count(%rip), %ecx
  leal  1(%ecx), %r8d
  movl  %r8d, count(%rip)
.L104:
  cmpl  %edx, %esi
  je   .L103
  leal  -1(%edx), %edx
  movl  count(%rip), %r10d
  leal  1(%r10d), %r8d
  movl  %r8d, count(%rip)
  jmp .L104
.L102:
  movl  count(%rip), %eax
.L101:
  addq  $8, %rsp
  ret
  .cfi_endproc
  .type fun, @function
  .size fun, . - fun
  .text
  .align  16
  .globl main
main:
  .cfi_startproc
  subq  $8, %rsp
  .cfi_adjust_cfa_offset  8
  leaq  16(%rsp), %rax
  movq  %rax, 0(%rsp)
.L105:
```

```
.L106:
  movl   $1, %eax
  movl   %eax, rooster(%rip)
  movl   $5, %eax
  movl   %eax, drake(%rip)
  movl   $10, %eax
  movl   %eax, goose(%rip)
  movl   $1, %edi
  movl   $5, %esi
  movl   $10, %edx
  call   fun
  xorl   %eax, %eax
  addq   $8, %rsp
  ret
  .cfi_endproc
  .type main, @function
  .size main, . - main
  .section  "__compcert_ais_annotations","",@note
  .ascii "# file:rooster.c line:6 function:fun\n"
  .byte 7,8
  .quad .L100
  .ascii " CRITICAL COMMENT\n"
  .ascii "# file:rooster.c line:26 function:main\n"
  .byte 7,8
  .quad .L105
  .ascii " L(mem("
  .byte 7,8
  .quad goose
  .ascii ", 4)) = medium\n"
  .ascii "# file:rooster.c line:27 function:main\n"
  .byte 7,8
  .quad .L106
  .ascii " EXCEPTIONAL\n"
```

C.11 password-O0.s

```
# File generated by CompCert 3.7
# Command line: password.c -O0 -S
  .section  .rodata
  .align  1
__stringlit_7:
  .ascii  "--- USERS ---\012\000"
  .type __stringlit_7, @object
  .size __stringlit_7, . - __stringlit_7
  .section  .rodata
  .align  1
__stringlit_6:
  .ascii  "passw0rd123\000"
  .type __stringlit_6, @object
  .size __stringlit_6, . - __stringlit_6
  .section  .rodata
  .align  1
__stringlit_4:
  .ascii  "!alice12!_veuje@@hak\000"
  .type __stringlit_4, @object
  .size __stringlit_4, . - __stringlit_4
  .section  .rodata
```

```
  .align  1
__stringlit_14:
  .ascii  "Password: \000"
  .type __stringlit_14, @object
  .size __stringlit_14, . - __stringlit_14
  .section  .rodata
  .align  1
__stringlit_18:
  .ascii  "Your balance: $%ld\012\000"
  .type __stringlit_18, @object
  .size __stringlit_18, . - __stringlit_18
  .section  .rodata
  .align  1
__stringlit_13:
  .ascii  "User < %s > does not exist.\012\000"
  .type __stringlit_13, @object
  .size __stringlit_13, . - __stringlit_13
  .section  .rodata
  .align  1
__stringlit_8:
  .ascii  " %02ld. %s\012\000"
  .type __stringlit_8, @object
  .size __stringlit_8, . - __stringlit_8
  .section  .rodata
  .align  1
__stringlit_1:
  .ascii  "admin\000"
  .type __stringlit_1, @object
  .size __stringlit_1, . - __stringlit_1
  .section  .rodata
  .align  1
__stringlit_2:
  .ascii  "4dm1n__4eva\000"
  .type __stringlit_2, @object
  .size __stringlit_2, . - __stringlit_2
  .section  .rodata
  .align  1
__stringlit_3:
  .ascii  "alice\000"
  .type __stringlit_3, @object
  .size __stringlit_3, . - __stringlit_3
  .section  .rodata
  .align  1
__stringlit_11:
  .ascii  "Username: \000"
  .type __stringlit_11, @object
  .size __stringlit_11, . - __stringlit_11
  .section  .rodata
  .align  1
__stringlit_5:
  .ascii  "abdul\000"
  .type __stringlit_5, @object
  .size __stringlit_5, . - __stringlit_5
  .section  .rodata
  .align  1
__stringlit_17:
  .ascii  "Welcome, %s!\012\000"
```

```
  .type __stringlit_17, @object
  .size __stringlit_17, . - __stringlit_17
  .section  .rodata
  .align  1
__stringlit_12:
  .ascii  "%255s\000"
  .type __stringlit_12, @object
  .size __stringlit_12, . - __stringlit_12
  .section  .rodata
  .align  1
__stringlit_9:
  .ascii  "\012\000"
  .type __stringlit_9, @object
  .size __stringlit_9, . - __stringlit_9
  .section  .rodata
  .align  1
__stringlit_15:
  .ascii  "ERROR: incorrect password\012\000"
  .type __stringlit_15, @object
  .size __stringlit_15, . - __stringlit_15
  .section  .rodata
  .align  1
__stringlit_10:
  .ascii  "Welcome to BigBank Australia!\012\000"
  .type __stringlit_10, @object
  .size __stringlit_10, . - __stringlit_10
  .section  .rodata
  .align  1
__stringlit_16:
  .ascii  "Logged in as < %s >!\012\000"
  .type __stringlit_16, @object
  .size __stringlit_16, . - __stringlit_16
  .text
  .align  16
  .globl setup_users
setup_users:
  .cfi_startproc
  subq  $40, %rsp
  .cfi_adjust_cfa_offset  40
  leaq  48(%rsp), %rax
  movq  %rax, 0(%rsp)
  movq  %rbx, 8(%rsp)
  movq  %rbp, 16(%rsp)
  movq  %r12, 24(%rsp)
  movq  $528, %rdi
  call  malloc
  movq  %rax, %rbp
  leaq  8(%rbp), %rdi
  leaq  __stringlit_1(%rip), %rsi
  call  strcpy
  leaq  264(%rbp), %rdi
  leaq  __stringlit_2(%rip), %rsi
  call  strcpy
.L100:
  movq  $1000000, %r10
  movq  %r10, 520(%rbp)
  movq  $528, %rdi
```

```
  call  malloc
  movq  %rax, %r12
  leaq  8(%r12), %rdi
  leaq  __stringlit_3(%rip), %rsi
  call  strcpy
  leaq  264(%r12), %rdi
  leaq  __stringlit_4(%rip), %rsi
  call  strcpy
.L101:
  movq  $783, %r9
  movq  %r9, 520(%r12)
  movq  $528, %rdi
  call  malloc
  movq  %rax, %rbx
  leaq  8(%rbx), %rdi
  leaq  __stringlit_5(%rip), %rsi
  call  strcpy
  leaq  264(%rbx), %rdi
  leaq  __stringlit_6(%rip), %rsi
  call  strcpy
.L102:
  movq  $2, %r11
  movq  %r11, 520(%rbx)
  movq  %r12, 0(%rbp)
  movq  %rbx, 0(%r12)
  xorq  %r8, %r8
  movq  %r8, 0(%rbx)
  movq  %rbp, %rax
  movq  8(%rsp), %rbx
  movq  16(%rsp), %rbp
  movq  24(%rsp), %r12
  addq  $40, %rsp
  ret
  .cfi_endproc
  .type setup_users, @function
  .size setup_users, . - setup_users
  .text
  .align  16
  .globl print_users
print_users:
  .cfi_startproc
  subq  $24, %rsp
  .cfi_adjust_cfa_offset  24
  leaq  32(%rsp), %rax
  movq  %rax, 0(%rsp)
  movq  %rbx, 8(%rsp)
  movq  %rbp, 16(%rsp)
  movq  %rdi, %rbx
  leaq  __stringlit_7(%rip), %rdi
  movl  $0, %eax
  call  printf
  xorq  %rbp, %rbp
.L103:
  cmpq  $0, %rbx
  je  .L104
  leaq  1(%rbp), %rbp
  leaq  __stringlit_8(%rip), %rdi
```

```
  leaq  8(%rbx), %rdx
  movq  %rbp, %rsi
  movl  $0, %eax
  call  printf
  movq  0(%rbx), %rbx
  jmp .L103
.L104:
  leaq  __stringlit_9(%rip), %rdi
  movl  $0, %eax
  call  printf
  movq  8(%rsp), %rbx
  movq  16(%rsp), %rbp
  addq  $24, %rsp
  ret
  .cfi_endproc
  .type print_users, @function
  .size print_users, . - print_users
  .text
  .align  16
  .globl getUser
getUser:
  .cfi_startproc
  subq  $24, %rsp
  .cfi_adjust_cfa_offset  24
  leaq  32(%rsp), %rax
  movq  %rax, 0(%rsp)
  movq  %rbx, 8(%rsp)
  movq  %rbp, 16(%rsp)
  movq  %rsi, %rbp
  movq  %rdi, %rbx
.L105:
  cmpq  $0, %rbx
  je  .L106
  leaq  8(%rbx), %rdi
  movq  %rbp, %rsi
  call  strcmp
  testl %eax, %eax
  je  .L107
  movq  0(%rbx), %rbx
  jmp .L105
.L106:
  xorq  %rbx, %rbx
.L107:
  movq  %rbx, %rax
  movq  8(%rsp), %rbx
  movq  16(%rsp), %rbp
  addq  $24, %rsp
  ret
  .cfi_endproc
  .type getUser, @function
  .size getUser, . - getUser
  .text
  .align  16
  .globl main
main:
  .cfi_startproc
  subq  $536, %rsp
```

```
  .cfi_adjust_cfa_offset  536
  leaq  544(%rsp), %rax
  movq  %rax, 0(%rsp)
  movq  %rbx, 8(%rsp)
  call  setup_users
  movq  %rax, %rbx
  leaq  __stringlit_10(%rip), %rdi
  movl  $0, %eax
  call  printf
  leaq  __stringlit_11(%rip), %rdi
  movl  $0, %eax
  call  printf
  leaq  __stringlit_12(%rip), %rdi
  leaq  16(%rsp), %rsi
  movl  $0, %eax
  call  __isoc99_scanf
  leaq  16(%rsp), %rsi
  movq  %rbx, %rdi
  call  getUser
  movq  %rax, %rbx
  cmpq  $0, %rbx
  jne .L108
  leaq  __stringlit_13(%rip), %rdi
  leaq  16(%rsp), %rsi
  movl  $0, %eax
  call  printf
  xorl  %eax, %eax
  jmp .L109
.L108:
  leaq  __stringlit_14(%rip), %rdi
  movl  $0, %eax
  call  printf
  leaq  __stringlit_12(%rip), %rdi
  leaq  272(%rsp), %rsi
  movl  $0, %eax
  call  __isoc99_scanf
  leaq  264(%rbx), %rdi
  leaq  272(%rsp), %rsi
  call  strcmp
  testl %eax, %eax
  je  .L110
  leaq  __stringlit_15(%rip), %rdi
  movl  $0, %eax
  call  printf
  xorl  %eax, %eax
  jmp .L109
.L110:
  leaq  __stringlit_16(%rip), %rdi
  leaq  8(%rbx), %rsi
  movl  $0, %eax
  call  printf
  leaq  __stringlit_9(%rip), %rdi
  movl  $0, %eax
  call  printf
  leaq  __stringlit_17(%rip), %rdi
  leaq  8(%rbx), %rsi
  movl  $0, %eax
```

```
  call  printf
  leaq  __stringlit_18(%rip), %rdi
  movq  520(%rbx), %rsi
  movl  $0, %eax
  call  printf
  xorl  %eax, %eax
.L109:
  movq  8(%rsp), %rbx
  addq  $536, %rsp
  ret
  .cfi_endproc
  .type main, @function
  .size main, . - main
  .section  "__compcert_ais_annotations","",@note
  .ascii "# file:password.c line:20 function:setup_users\n"
  .byte 7,8
  .quad .L100
  .ascii " L((reg(\"rbp\") + 264)) = high\n"
  .ascii "# file:password.c line:26 function:setup_users\n"
  .byte 7,8
  .quad .L101
  .ascii " L((reg(\"r12\") + 264)) = high\n"
  .ascii "# file:password.c line:32 function:setup_users\n"
  .byte 7,8
  .quad .L102
  .ascii " L((reg(\"rbx\") + 264)) = high\n"
```

C.12 password-O3.s

```
# File generated by CompCert 3.7
# Command line: password.c -O3 -S
  .section  .rodata
  .align  1
__stringlit_7:
  .ascii  "--- USERS ---\012\000"
  .type __stringlit_7, @object
  .size __stringlit_7, . - __stringlit_7
  .section  .rodata
  .align  1
__stringlit_6:
  .ascii  "passw0rd123\000"
  .type __stringlit_6, @object
  .size __stringlit_6, . - __stringlit_6
  .section  .rodata
  .align  1
__stringlit_4:
  .ascii  "!alice12!_veuje@@hak\000"
  .type __stringlit_4, @object
  .size __stringlit_4, . - __stringlit_4
  .section  .rodata
  .align  1
__stringlit_14:
  .ascii  "Password: \000"
  .type __stringlit_14, @object
  .size __stringlit_14, . - __stringlit_14
  .section  .rodata
  .align  1
```

```
__stringlit_18:
  .ascii  "Your balance: $%ld\012\000"
  .type __stringlit_18, @object
  .size __stringlit_18, . - __stringlit_18
  .section  .rodata
  .align  1
__stringlit_13:
  .ascii  "User < %s > does not exist.\012\000"
  .type __stringlit_13, @object
  .size __stringlit_13, . - __stringlit_13
  .section  .rodata
  .align  1
__stringlit_8:
  .ascii  " %02ld. %s\012\000"
  .type __stringlit_8, @object
  .size __stringlit_8, . - __stringlit_8
  .section  .rodata
  .align  1
__stringlit_1:
  .ascii  "admin\000"
  .type __stringlit_1, @object
  .size __stringlit_1, . - __stringlit_1
  .section  .rodata
  .align  1
__stringlit_2:
  .ascii  "4dm1n__4eva\000"
  .type __stringlit_2, @object
  .size __stringlit_2, . - __stringlit_2
  .section  .rodata
  .align  1
__stringlit_3:
  .ascii  "alice\000"
  .type __stringlit_3, @object
  .size __stringlit_3, . - __stringlit_3
  .section  .rodata
  .align  1
__stringlit_11:
  .ascii  "Username: \000"
  .type __stringlit_11, @object
  .size __stringlit_11, . - __stringlit_11
  .section  .rodata
  .align  1
__stringlit_5:
  .ascii  "abdul\000"
  .type __stringlit_5, @object
  .size __stringlit_5, . - __stringlit_5
  .section  .rodata
  .align  1
__stringlit_17:
  .ascii  "Welcome, %s!\012\000"
  .type __stringlit_17, @object
  .size __stringlit_17, . - __stringlit_17
  .section  .rodata
  .align  1
__stringlit_12:
  .ascii  "%255s\000"
  .type __stringlit_12, @object
```

```
  .size __stringlit_12, . - __stringlit_12
  .section  .rodata
  .align  1
__stringlit_9:
  .ascii  "\012\000"
  .type __stringlit_9, @object
  .size __stringlit_9, . - __stringlit_9
  .section  .rodata
  .align  1
__stringlit_15:
  .ascii  "ERROR: incorrect password\012\000"
  .type __stringlit_15, @object
  .size __stringlit_15, . - __stringlit_15
  .section  .rodata
  .align  1
__stringlit_10:
  .ascii  "Welcome to BigBank Australia!\012\000"
  .type __stringlit_10, @object
  .size __stringlit_10, . - __stringlit_10
  .section  .rodata
  .align  1
__stringlit_16:
  .ascii  "Logged in as < %s >!\012\000"
  .type __stringlit_16, @object
  .size __stringlit_16, . - __stringlit_16
  .text
  .align  16
  .globl setup_users
setup_users:
  .cfi_startproc
  subq  $40, %rsp
  .cfi_adjust_cfa_offset  40
  leaq  48(%rsp), %rax
  movq  %rax, 0(%rsp)
  movq  %rbx, 8(%rsp)
  movq  %rbp, 16(%rsp)
  movq  %r12, 24(%rsp)
  movq  $528, %rdi
  call  malloc
  movq  %rax, %rbp
  leaq  8(%rbp), %rdi
  leaq  __stringlit_1(%rip), %rsi
  call  strcpy
  leaq  264(%rbp), %rdi
  leaq  __stringlit_2(%rip), %rsi
  call  strcpy
.L100:
  movq  $1000000, %r10
  movq  %r10, 520(%rbp)
  movq  $528, %rdi
  call  malloc
  movq  %rax, %r12
  leaq  8(%r12), %rdi
  leaq  __stringlit_3(%rip), %rsi
  call  strcpy
  leaq  264(%r12), %rdi
  leaq  __stringlit_4(%rip), %rsi
```

```
  call  strcpy
.L101:
  movq  $783, %r9
  movq  %r9, 520(%r12)
  movq  $528, %rdi
  call  malloc
  movq  %rax, %rbx
  leaq  8(%rbx), %rdi
  leaq  __stringlit_5(%rip), %rsi
  call  strcpy
  leaq  264(%rbx), %rdi
  leaq  __stringlit_6(%rip), %rsi
  call  strcpy
.L102:
  movq  $2, %r11
  movq  %r11, 520(%rbx)
  movq  %r12, 0(%rbp)
  movq  %rbx, 0(%r12)
  xorq  %r8, %r8
  movq  %r8, 0(%rbx)
  movq  %rbp, %rax
  movq  8(%rsp), %rbx
  movq  16(%rsp), %rbp
  movq  24(%rsp), %r12
  addq  $40, %rsp
  ret
  .cfi_endproc
  .type setup_users, @function
  .size setup_users, . - setup_users
  .text
  .align  16
  .globl print_users
print_users:
  .cfi_startproc
  subq  $24, %rsp
  .cfi_adjust_cfa_offset  24
  leaq  32(%rsp), %rax
  movq  %rax, 0(%rsp)
  movq  %rbx, 8(%rsp)
  movq  %rbp, 16(%rsp)
  movq  %rdi, %rbp
  leaq  __stringlit_7(%rip), %rdi
  movl  $0, %eax
  call  printf
  xorq  %rbx, %rbx
.L103:
  cmpq  $0, %rbp
  je  .L104
  leaq  1(%rbx), %rbx
  leaq  __stringlit_8(%rip), %rdi
  leaq  8(%rbp), %rdx
  movq  %rbx, %rsi
  movl  $0, %eax
  call  printf
  movq  0(%rbp), %rbp
  jmp .L103
.L104:
```

```
  leaq  __stringlit_9(%rip), %rdi
  movl  $0, %eax
  call  printf
  movq  8(%rsp), %rbx
  movq  16(%rsp), %rbp
  addq  $24, %rsp
  ret
  .cfi_endproc
  .type print_users, @function
  .size print_users, . - print_users
  .text
  .align  16
  .globl getUser
getUser:
  .cfi_startproc
  subq  $24, %rsp
  .cfi_adjust_cfa_offset  24
  leaq  32(%rsp), %rax
  movq  %rax, 0(%rsp)
  movq  %rbx, 8(%rsp)
  movq  %rbp, 16(%rsp)
  movq  %rsi, %rbp
  movq  %rdi, %rbx
.L105:
  cmpq  $0, %rbx
  je  .L106
  leaq  8(%rbx), %rdi
  movq  %rbp, %rsi
  call  strcmp
  testl %eax, %eax
  je  .L107
  movq  0(%rbx), %rbx
  jmp .L105
.L106:
  xorq  %rbx, %rbx
.L107:
  movq  %rbx, %rax
  movq  8(%rsp), %rbx
  movq  16(%rsp), %rbp
  addq  $24, %rsp
  ret
  .cfi_endproc
  .type getUser, @function
  .size getUser, . - getUser
  .text
  .align  16
  .globl main
main:
  .cfi_startproc
  subq  $536, %rsp
  .cfi_adjust_cfa_offset  536
  leaq  544(%rsp), %rax
  movq  %rax, 0(%rsp)
  movq  %rbx, 8(%rsp)
  call  setup_users
  movq  %rax, %rbx
  leaq  __stringlit_10(%rip), %rdi
```

```
  movl  $0, %eax
  call  printf
  leaq  __stringlit_11(%rip), %rdi
  movl  $0, %eax
  call  printf
  leaq  __stringlit_12(%rip), %rdi
  leaq  16(%rsp), %rsi
  movl  $0, %eax
  call  __isoc99_scanf
  leaq  16(%rsp), %rsi
  movq  %rbx, %rdi
  call  getUser
  movq  %rax, %rbx
  cmpq  $0, %rbx
  jne .L108
  leaq  __stringlit_13(%rip), %rdi
  leaq  16(%rsp), %rsi
  movl  $0, %eax
  call  printf
  xorl  %eax, %eax
  jmp .L109
.L108:
  leaq  __stringlit_14(%rip), %rdi
  movl  $0, %eax
  call  printf
  leaq  __stringlit_12(%rip), %rdi
  leaq  272(%rsp), %rsi
  movl  $0, %eax
  call  __isoc99_scanf
  leaq  264(%rbx), %rdi
  leaq  272(%rsp), %rsi
  call  strcmp
  testl %eax, %eax
  je   .L110
  leaq  __stringlit_15(%rip), %rdi
  movl  $0, %eax
  call  printf
  xorl  %eax, %eax
  jmp .L109
.L110:
  leaq  __stringlit_16(%rip), %rdi
  leaq  8(%rbx), %rsi
  movl  $0, %eax
  call  printf
  leaq  __stringlit_9(%rip), %rdi
  movl  $0, %eax
  call  printf
  leaq  __stringlit_17(%rip), %rdi
  leaq  8(%rbx), %rsi
  movl  $0, %eax
  call  printf
  leaq  __stringlit_18(%rip), %rdi
  movq  520(%rbx), %rsi
  movl  $0, %eax
  call  printf
  xorl  %eax, %eax
.L109:
```

```
  movq  8(%rsp), %rbx
  addq  $536, %rsp
  ret
  .cfi_endproc
  .type main, @function
  .size main, . - main
  .section  "__compcert_ais_annotations","",@note
  .ascii "# file:password.c line:20 function:setup_users\n"
  .byte 7,8
  .quad .L100
  .ascii " L((reg(\"rbp\") + 264)) = high\n"
  .ascii "# file:password.c line:26 function:setup_users\n"
  .byte 7,8
  .quad .L101
  .ascii " L((reg(\"r12\") + 264)) = high\n"
  .ascii "# file:password.c line:32 function:setup_users\n"
  .byte 7,8
  .quad .L102
  .ascii " L((reg(\"rbx\") + 264)) = high\n"
```

C.13 deadStoreElimination-O0.s

```
# File generated by CompCert 3.7
# Command line: deadStoreElimination.c -O0 -S
  .text
  .align  16
  .globl deadStore
deadStore:
  .cfi_startproc
  subq  $8, %rsp
  .cfi_adjust_cfa_offset  8
  leaq  16(%rsp), %rax
  movq  %rax, 0(%rsp)
  movl  $43981, %ecx
.L100:
  nop
.L101:
  cmpl  %esi, %edi
  jle .L102
  leal  -1(%edi), %edi
  jmp .L101
.L102:
  leal  0(%edi,%esi,1), %eax
  addq  $8, %rsp
  ret
  .cfi_endproc
  .type deadStore, @function
  .size deadStore, . - deadStore
  .text
  .align  16
  .globl main
main:
  .cfi_startproc
  subq  $8, %rsp
  .cfi_adjust_cfa_offset  8
  leaq  16(%rsp), %rax
  movq  %rax, 0(%rsp)
```

```
  movl  $2, %esi
  call  deadStore
  xorl  %eax, %eax
  addq  $8, %rsp
  ret
  .cfi_endproc
  .type main, @function
  .size main, . - main
  .section  "__compcert_ais_annotations","",@note
  .ascii "# file:deadStoreElimination.c line:3 function:deadStore\n"
  .byte 7,8
  .quad .L100
  .ascii " L(reg(\"rcx\")) = high\n"
```

C.14 deadStoreElimination-O3.s

```
# File generated by CompCert 3.7
# Command line: deadStoreElimination.c -O3 -S
  .text
  .align  16
  .globl deadStore
deadStore:
  .cfi_startproc
  subq  $8, %rsp
  .cfi_adjust_cfa_offset  8
  leaq  16(%rsp), %rax
  movq  %rax, 0(%rsp)
.L100:
  nop
.L101:
  cmpl  %esi, %edi
  jle .L102
  leal  -1(%edi), %edi
  jmp .L101
.L102:
  leal  0(%edi,%esi,1), %eax
  addq  $8, %rsp
  ret
  .cfi_endproc
  .type deadStore, @function
  .size deadStore, . - deadStore
  .text
  .align  16
  .globl main
main:
  .cfi_startproc
  subq  $8, %rsp
  .cfi_adjust_cfa_offset  8
  leaq  16(%rsp), %rax
  movq  %rax, 0(%rsp)
  movl  $2, %esi
  call  deadStore
  xorl  %eax, %eax
  addq  $8, %rsp
  ret
  .cfi_endproc
  .type main, @function
```

```
  .size main, . - main
  .section  "__compcert_ais_annotations","",@note
  .ascii "# file:deadStoreElimination.c line:3 function:deadStore\n"
  .byte 7,8
  .quad .L100
  .ascii " L(43981) = high\n"
```

C.15 pread-O0.s

```
pread.c:6: error: access to volatile variable 'z' for parameter '%e1' is not supported in
    ais annotations
pread.c:7: error: access to volatile variable 'x' for parameter '%e1' is not supported in
    ais annotations
pread.c:7: error: access to volatile variable 'z' for parameter '%e2' is not supported in
    ais annotations
pread.c:20: error: access to volatile variable 'z' for parameter '%e2' is not supported in
    ais annotations
pread.c:21: error: access to volatile variable 'z' for parameter '%e3' is not supported in
    ais annotations
pread.c:21: error: access to volatile variable 'z' for parameter '%e3' is not supported in
    ais annotations
pread.c:23: error: access to volatile variable 'z' for parameter '%e2' is not supported in
    ais annotations
7 errors detected.
```

C.16 pread-O3.s

```
pread.c:6: error: access to volatile variable 'z' for parameter '%e1' is not supported in
    ais annotations
pread.c:7: error: access to volatile variable 'x' for parameter '%e1' is not supported in
    ais annotations
pread.c:7: error: access to volatile variable 'z' for parameter '%e2' is not supported in
    ais annotations
pread.c:20: error: access to volatile variable 'z' for parameter '%e2' is not supported in
    ais annotations
pread.c:21: error: access to volatile variable 'z' for parameter '%e3' is not supported in
    ais annotations
pread.c:21: error: access to volatile variable 'z' for parameter '%e3' is not supported in
    ais annotations
pread.c:23: error: access to volatile variable 'z' for parameter '%e2' is not supported in
    ais annotations
7 errors detected.
```