Compiler Annotation Solutions for Concurrent Information Flow Security

Alexander Blyth

September 2020

***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 citemantel2014noninterference[28][30]. 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. [32] 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 [31] [26] [19], however, these tools focus on either generic solutions for propagating properties or to assist the static analysis of the *Worst Case Execution Time*. We look to use these tools or similar approaches to maintain compiler annotations for analysis of information flow security in concurrent programs.

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 [17] a security flaw called Heartbleed was discovered. Secure secrets such as financial data, encryption keys, or anything else stored in the server's

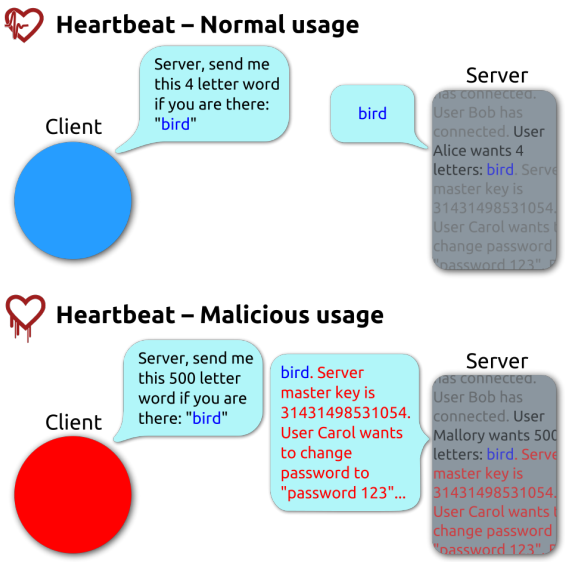

Figure 1: The Heartbleed bug. [13]

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 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 [16]. 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 [29]. 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 [25]. 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,

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

Figure 2: Implicit flow of data to a public variable

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, 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].

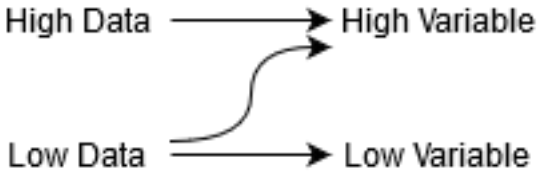

Figure 3: Permitted flow of data

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 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 [32]. 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 [20][28][30]. Research has been conducted into concurrent programs to explore ways the security of concurrent programs can be verified. Mantel et al. [21] 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] [22] 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 [24].

To overcome information flow security in non-blocking concurrent threads, Winter et al. [32] 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.

```
crypt() {
    key := 0xC0DE // Read key
    ... // Work with the key
    key := 0x0 // Clear memory
}
```

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

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.

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 [23]. Additionally, dead store elimination has been proven to be functionally correct [3][18].

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 [27]. 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. [32]. 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.

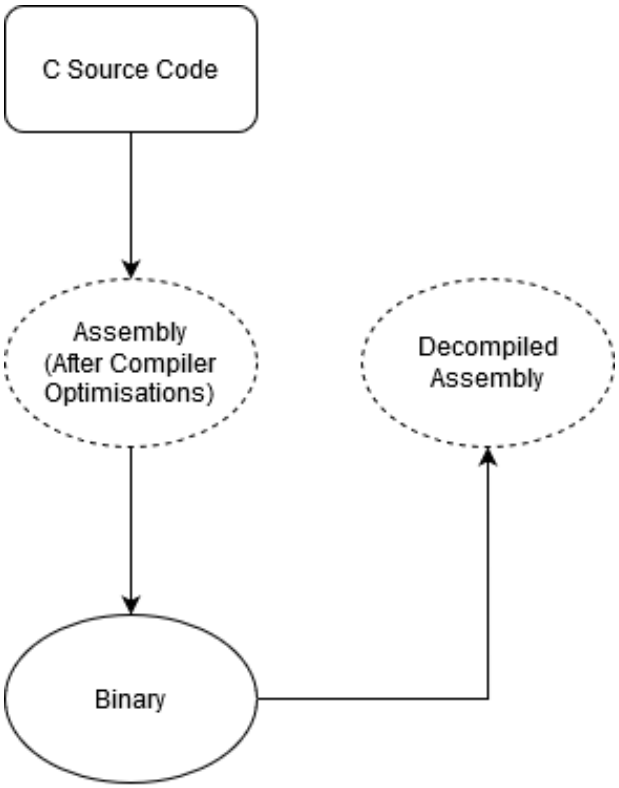

Figure 5: The static analysis options after compilation.

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 [26]. 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][19]. 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.

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. [31] 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.

Also worth mentioning are GCC plugins [15]. Compiler plugins can be constructed to add new features to a compiler without modifying the compiler itself. Support for this is available from GCC 4.5.0 and onwards.

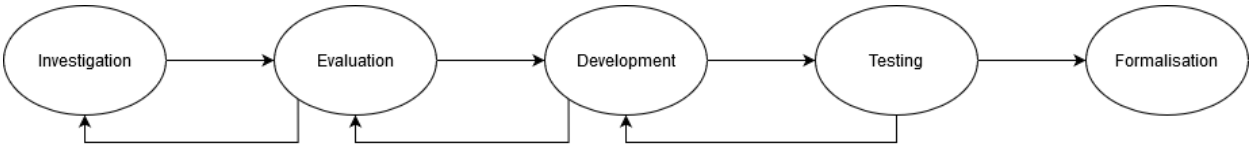

Figure 6: Project Plan Overview

3 Project Plan

There are several key milestones associated with this project. These are:

1. Investigation into existing C compiler annotation solutions and approaches,
2. Evaluation of approaches and techniques,
3. Development of tool,
4. Testing for correctness and performance, and
5. Formalisation of results.

3.1 Why not, Waterfall?

As with all software projects, the project specifications and the associated plan are likely to change throughout the course of the project. In this project, it is likely that roadblocks will be faced when assessing particular approaches which will result in having to go back and reassess from the investigation stage once again. As such, following a waterfall-style plan does not suit this type of project, and creating waterfall-style plan such as a Gantt chart will be avoided. Instead, approaches and techniques will be discussed rather than a rigid plan.

3.2 Milestones and Phases

Here each milestone is discussed in further detail. Here we use 'milestones' and 'phases' interchangeably. An estimated time has been associated with each phase and subsections of each milestone. An overview of this plan can be seen in Figure 6. At each stage of the plan, it is possible to return back to previous phase.

3.2.1 Investigation Phase

Estimated time: 30 hours

As part of the investigation phase, the compiler annotation tools that are currently in use will be explored. These are the:

- AbsInt aiT tool *Estimated time: 10 hours*,
- CompCert annotations *Estimated time: 10 hours*, and
- Vu et al. LLVM compiler properties *Estimated time: 10 hours*.

For each of these, the source code will be inspected and experimented with to understand the methodology and assess the applicability to wpif analysis. If new approaches or tools are discovered throughout the progress of this thesis this phase will need to be returned to.

The resources required for this phase include:

- books,
- UQ library,
- Google Scholar,
- computer,
- internet, and
- code inspection tools (VS Code).

3.2.2 Evaluation Phase

Estimated time: 10 hours
This phase is expected to be revisited several times over the course of the project. In this phase each option will be evaluated before beginning development. Different aspects of the development approach will need to be evaluated in this stage. This includes the evaluation of: assembly or binary approaches:

- an assembly or binary approach,
- standalone or existing tool approach.

The existing tool approaches are listed in section 3.2.1. A standale tool could be devleoped using GCC plugins or some other method.

The resources required for this phase include:

- books,
- UQ library,
- Google Scholar,
- computer, and
- internet.

3.2.3 Development Phase

Estimated time: 100 hours
In this phase, the proposed tool will be developed from a home workstation. The specifics of what needs to be completed in this phase depends heavily on what approach is being used. As mentioned previously, if a roadblock is met, a revaluation will need to be made to reassess the approach being used.

The resources required for this phase include:

- a computer,
- internet, and
- code development tools (VS Code, Command line).

3.2.4 Testing Phase

Estimated time: 30 hours
In this phase, the developed tool will be tested for correctness as well as performance and efficiency. These results will then be documented before returning to the development phase to fix bugs or issues with the tool. The development and testing phase will be revisited several times throughout the course of this thesis.

The resources required for this phase include:

- a computer,
- internet, and
- code development tools (VS Code, Command line).

3.2.5 Formalisation

Estimated time: 30 hours
The formalisation of the thesis results will be done after development and testing of the tool. In this stage, the thesis content and report will be written and the poster produced.

The resources required for this phase include:

- books,
- UQ library,
- Google Scholar,
- computer,
- internet, and
- code inspection tools (VS Code).

4 Risk Assessment

Risks associated with this thesis have been listed in table 3. Each risk has an associated level of low, moderate, high or catastrophic. A description of each risk level has been listed in table 1. Additionally, a likelihood rating of each risk has been assigned. These likelihood ratings have been listed in table 2.

4.1 Occupational Health and Safety Risk Assessment

This thesis will be conducted from a home workstation. This environment is a *low risk laboratory* and as such, work is covered by general OHS laboratory rules.

Level	Description
Low	Minimal disruption to work. Loss of maximum one day to a week of thesis-related work or a setback of similar duration.
Moderate	Moderate disruption to work. Loss of a week to a month of thesis-related work or a setback of similar duration.
High	Major disruption to work. Loss of a month or more of thesis-related work or a setback of similar duration.
Catastrophic	Loss of life, criminal charges or inability to complete thesis.

Table 1: Risk level descriptions

Likelihood	Description
Improbable	Hazard is unlikely to occur during the life of the project.
Possible	Hazard is likely to occur once during the lifetime of this project.
Likely	Hazard is likely to occur once in 3 months.
Extremely Likely	Hazard is likely to occur once in a month.

Table 2: Risk likelihood descriptions

Risk	Level	Likelihood	Mitigation Plan
Loss of data	Moderate	Possible	Data will be constantly backed up through the use of GitHub and Google Drive. Code and LaTeX documents will be backed up to GitHub with regular commits, with all other documents stored on Google Drive.
Hardware damage	Moderate	Improbable	Keep food, water and other hazardous materials away from the workstation. Maintain a backup PC in case of damage to the main computer.
Pandemic-related lockdown	Low	Likely	Plans will be put in place to prepare a home PC for external work in case the UQ St Lucia campus is unavailable due to COVID-19. Meetings can be conducted online over Zoom if required.
Out of scope work	Moderate	Possible	Maintain regular, weekly meetings with thesis supervisors to maintain scope of work conducted.
Illness preventing thesis work	Moderate	Possible	Maintain regular work progression of a minimal 10 hours per week such that if illness does occur, thesis work is not set back far.

Table 3: Risk likelihood descriptions

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] *GCC Plugins*. Accessed: 2020-09-02. URL: `https://gcc.gnu.org/wiki/plugins`.

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

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

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

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

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

[30] 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.

[31] 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.

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