Compiler Annotation Solutions for Concurrent Information Flow Security

Alexander Blyth

August 2020

***Abstract* - Come back to the abstract later, since I can't think of the words to write right now, so instead, have some Lorem Ipsum.**

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas vestibulum ultricies euismod. Duis turpis augue, tempor eget ligula in, malesuada pulvinar mi. Maecenas elementum dolor diam, ut condimentum est dapibus ac. Morbi sed molestie ipsum, vel varius mi. Mauris quis ex sed massa efficitur varius. Curabitur elementum, odio at faucibus rutrum, velit erat ullamcorper mauris, non maximus magna orci a mi. Donec sed condimentum ipsum. Interdum et malesuada fames ac ante ipsum primis in faucibus.

1 Background

TODO: Brief overview of the background of the topic

- provide a tool to analyse C programs to detect security violations created through information flow control through wpif.

- Analysis done after compilation to a binary to detect information flow violations introduced by compiler optimisations

- Explore ways to preserve annotations in C through compilation to binary with minimal modification to the compiler

1.1 Information Security

Vulnerabilities in software can lead to catastrophic consequences when manipulated by attackers. *Example in here?*

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 [11]. 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 [21]. 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 [9]. 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 [19]. 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 [8]. Take for example figure 1. 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.

Do we care about implicit flow in this thesis? Or are we only focussing on explicit flow through concurrency?

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]. However, most security failures are due to security violations introduced at the application level [12]. Therefore, this thesis will focus primarily on security concerns at the high level.

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

Figure 1: Implicit flow of data to a public variable

1.2 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 *From my understanding, we don't care about this in this thesis?* [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 [7].

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 [23]. 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 2. Note that high data is not allowed to flow into low variables.

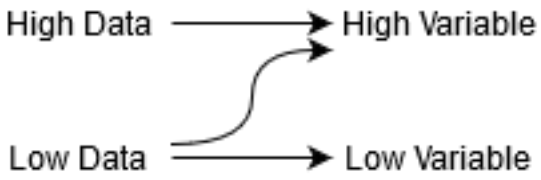

Figure 2: Permitted flow of data

1.3 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 [14][20][22]. Research has been conducted into concurrent programs to explore ways the security of concurrent programs can be verified. Mantel et al. [15] 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 that can be then used to prove a guarantee about that individual thread. This concept of assumptions (or rely) and guarantee conditions can reduce the complexity of dealing with concurrency 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. [10] [16] 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 [18].

To overcome information flow security in non-blocking concurrent threads, Winter et al. [23] explores verifying security properties such as non-interference through the use of general rely/guarantee conditions using backwards, weakest precondition reasoning. 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.

1.4 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, subvert the expectations a programmer may have about the execution of their program [6]. 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 3 was derived from CWE-14 [5] and CWE-733 [4] and used by D'Silva et al. [6]. 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

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

Figure 3: Implicit flow of data to a public variable

performed by default [17]. Additionally, dead store elimination has been proven to be functionally correct [3][13].

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. 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 a binary for security violations. As compilation has already been executed, such analysis would reveal security guarantee violations that result due to compiler optimisations.

2 Similar Solutions

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 - Safety-critical, real-time software such as flight control systems require worst-case execution time analysis to prove correctness. - AbsInt analyses worst case execution time using static analysis tools alongside compiler annotations. - Using CompCert: Supports annotation via __builtin_annot() - places comment in generated assembly code. Additional tool can be used to parse comments and generate annotations - Optimisations can remove annotations - need to verify annotations propagated through - Downfall: treatment as a call to an external function, cannot be placed at top-level of compilation unit unlike a var declaration.

- The ENTRA (Whole-Systems ENergy TRAnsparency) Deliverable describe mechanism to transfer properties from the source to the machine level. - Comments need to be extracted from assembler files, not stored in final binary

- Li, Puat and Rohou present framework where source code annotations are transformed into annotations on the binary level - Focus on loop count annotations and preservation through classic loop optimisations

2.1 TODO

Still a lot of goodness to discuss:

1. the correctness security gap (compiler violations of security guarantees and why it is not the compiler's role to ensure security)
2. How compiler annotations can be used to assist analysis using WPIF
3. Existing solutions for compiler annotations
4. Discussion about modifying / not modifying compiler - approach
5. GCC plugins: https://gcc.gnu.org/wiki/plugins

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

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

[6] 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.

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

[8] 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.

[9] 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.

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

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

[12] Dongseok Jang et al. "An empirical study of privacy-violating information flows in JavaScript web applications". In: *Proceedings of the 17th ACM conference on Computer and communications security*. 2010, pp. 270–283.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

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

[19] 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.

[20] 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.

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

[22] 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.

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