Compiler Annotation Solutions to the Correctness Security Gap

Alexander Blyth

August 2020

1 Introduction

- Analysis at application level for the C language.

Come back to the introduction 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. Morbi consequat tellus est, ut commodo orci condimentum ac. Donec dapibus efficitur eros sed iaculis. Cras viverra consequat nisi nec dictum. Quisque mattis ultricies tincidunt. Sed vitae mattis erat, vitae condimentum odio. Suspendisse sed efficitur odio, vel iaculis nunc. Sed sodales pharetra metus, vel ultricies nisi tincidunt vitae.

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 [6]. 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 [11]. 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 [5]. 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 [9]. 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 develop-

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

Figure 1: Implicit flow of data to a public variable

ment 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 [4]. Take for example figure 1.1. 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 [7]. Therefore, this thesis will focus primarily on security concerns at the high level.

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 [3].

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 the 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. 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 1.2. Note that high data is not allowed to flow into low variables.

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 [8][10][12]. Research has been conducted into concurrent programs

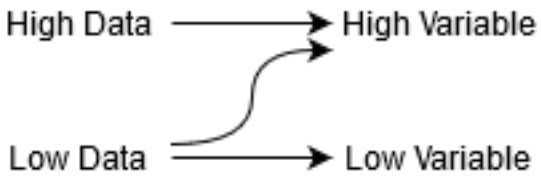

Figure 2: Permitted flow of data

[13]

1.4 TODO

2 Existing Solutions

- Mandatory Access control

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

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

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