Formally computer-verified protections against timing-based side-channel attacks

Swarn Priya

▶ **To cite this version:**

Swarn Priya. Formally computer-verified protections against timing-based side-channel attacks. Cryptography and Security [cs.CR]. Université Côte d'Azur, 2023. English. NNT : 2023COAZ4088 . tel-04331805v1

HAL Id: tel-04331805

https://hal.science/tel-04331805v1

Submitted on 8 Dec 2023 (v1), last revised 15 Dec 2023 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ CÔTE D'AZUR

ÉCOLE DOCTORALE
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THÈSE DE DOCTORAT

Protections vérifiées formellement par ordinateur contre les attaques par canaux auxiliaires basées sur le temps

Swarn PRIYA

Centre Inria d'Université Côte d'Azur STAMP

Présentée en vue de l'obtention du grade de docteur en Informatique **d'**Université Côte d'Azur.

Dirigée par : Yves BERTOT

Co-encadrée par :
Benjamin GRÉGOIRE

Soutenue le : 22/11/2023

Devant le jury, composé de :
Yves Bertot
Benjamin Grégoire
Sandrine Blazy
Karthikeyan Bhargavan
Étienne Lozes
Lesly-Ann Daniel

Protections vérifiées formellement par ordinateur contre les attaques par canaux auxiliaires basées sur le temps

Jury :

Rapporteurs:
Sandrine Blazy, Professeur et Chercheur, Université de Rennes 1 et IRISA
Karthikeyan Bhargavan, Directeur de recherche Inria, Paris

Examinateurs:
Étienne Lozes, Professeur des universités Université Cote d'Azur, I3S lab
Lesly-Ann Daniel, Docteur, KU Leuven

Directeurs
Yves Bertot, Directeur de recherche, Centre Inria d'Université Côte d'Azur
Benjamin Grégoire, Chargé de recherche, Centre Inria d'Université Côte d'Azur

Formally computer-verified protections against timing-based side-channel attacks

Jury :

Rapporteurs:
Sandrine Blazy, Professeur et Chercheur, Université de Rennes 1 et IRISA
Karthikeyan Bhargavan, Directeur de recherche Inria, Paris

Examinateurs:
Étienne Lozes, Professeur des universités Université Cote d'Azur, I3S lab
Lesly-Ann Daniel, Docteur, KU Leuven

Directeurs
Yves Bertot, Directeur de recherche, Centre Inria d'Université Côte d'Azur
Benjamin Grégoire, Chargé de recherche, Centre Inria d'Université Côte d'Azur

Résumé

Les développeurs de logiciels critiques cherchent à concevoir des logiciels fonctionnellement corrects, dotés de propriétés telles que la confidentialité. Cependant, la confidentialité n'est pas une conséquence directe de la correction fonctionnelle, car il peut y avoir des fuites par canaux auxiliaires, comme le temps d'exécution, qui peuvent aider l'attaquant à récupérer des données secrètes.

Par exemple, un algorithme de comparaison de chaînes de caractères qui compare deux chaînes caractère par caractère et se termine en cas de non-concordance ou renvoie un succès lorsque les chaînes sont égales est un algorithme fonctionnellement correct pour vérifier un mot de passe. Cependant, un attaquant peut deviner la longueur du mot de passe en mesurant le temps d'exécution de l'algorithme, ce qui montre qu'il n'est pas protégé contre les attaques par mesure de temps. Plus généralement, le branchement sur des secrets peut entraîner une fuite de données secrètes, car les deux branches peuvent avoir des temps d'exécution différents. Les processeurs modernes utilisent la prédiction de branchement pour maximiser la performance des programmes. Par exemple, si la destination d'une condition de branchement conduit à une lecture en mémoire, le processeur contourne la vérification de la condition et exécute l'opération de lecture en mémoire de manière spéculative. En cas d'erreur de prédiction, le processeur revient en arrière et recommence l'exécution avec le résultat correct de l'évaluation de la condition. Bien que le retour en arrière corrige les résultats spéculatifs, il laisse cependant une trace dans le cache, que l'attaquant peut exploiter pour accéder aux données secrètes (illustré dans les attaques Spectre). Ainsi, la spéculation améliore la performance globale au détriment de la sécurité. Les effets des canaux auxiliaires ne sont pas pris en compte dans la sémantique formelle classique des programmes et, par conséquent, cette sémantique ne peut pas être utilisée pour raisonner sur les hyperpropriétés telles que les attaques par canaux auxiliaires basées sur le temps. Sans la notion de modèles formels prenant en compte les effets secondaires produits au cours de l'exécution d'un programme, les développeurs ne peuvent que corriger manuellement le programme pour s'assurer qu'il prenne toujours le même temps pour s'exécuter et supposer que le compilateur ne casse pas la propriété lors de la compilation, sans garantie formelle, du programme vers l'assembleur. Traditionnellement, les cryptographes ont utilisé la propriété de temps constant pour se défendre contre les attaques par canaux auxiliaires basées sur le temps. Un programme à temps constant n'effectue pas d'accès mémoire ni de branchement dépendant du secret. Malheureusement, les optimisations effectuées par le compilateur pour accroître l'efficacité du programme ont tendance à casser la propriété de temps constant. Tout ceci plaide pour l'existence d'un modèle formel qui capture les effets secondaires produits pendant l'exécution d'un programme. Cette thèse vise à fournir des modèles formels prenant en compte les effets secondaires produits lors de l'exécution d'un programme, ainsi que des définitions formelles de propriétés de sécurité, comme le temps constant et le temps constant spéculatif. La thèse inclut le développement d'un compilateur sécurisé, formellement

vérifié, qui préserve la propriété de temps constant, et des méthodologies pour ajouter des protections contre différentes formes d'attaques Spectre. Elle définit des systèmes de types qui permettent de vérifier l'utilisation correcte de ces protections et illustre une méthode efficace de protection contre les attaques Spectre v1 appelée Selective Speculative Load Hardening. Tous ces travaux sont formellement vérifiés en utilisant l'assistant de preuve Coq, leur donnant la solidité des preuves vérifiées par ordinateur.

Keywords : Vérification formelle, Preuves sur ordinateur, Cryptographie vérifiée, Compilation

Abstract

Developers of high-assurance software aim to design functionally correct software with properties like confidentiality. However, confidentiality is not a direct consequence of functional correctness, as there could be unintentional side-channel leaks like execution time that might help the attacker to retrieve secret data.

For example, a string-comparison algorithm that compares two strings character by character and exits in case of a mismatch or return success when strings are equal is a functionally correct algorithm to verify a password. However, an attacker can guess the password's length by measuring the execution time of the algorithm, which means the latter is not secure against timing attacks. More generally, branching on secrets may leak secret data as the two branches may have different execution times. Modern processors use branch prediction to maximize the performance of the program. For example, if the destination of a branch condition leads to a memory read, the processor bypasses the condition check and executes the memory read operation speculatively. In case of misprediction, the processor backtracks and starts the execution again with the correct condition's result. Though backtracking corrects the misspeculated results, it still leaves a trace in the cache, which the attacker can exploit to get to the secret data (illustrated in Spectre attacks). Thus, the speculation improves overall efficiency at the expense of security.

Side-channel effects are not captured in the classic formal semantics of programs, and hence, these semantics cannot be used to reason about hyperproperties like timing-based side-channel attacks. Without the notion of formal models capturing side-effects produced during a program's execution, the developers can only manually fix the program to ensure it always takes the same time to execute and assume that the compiler does not break the property while compiling the program to assembly with no formal guarantees. Traditionally, to defend against timing-based side-channel attacks, cryptographers have used the constant-time property to develop programs that do not perform secret-dependent memory access and branching at the source level. Unfortunately, the optimizations performed by the compiler to increase the efficiency of the program tend to break constant-time property. This calls for the existence of a formal model that captures the side effects produced during a program's execution.

This thesis aims to provide formal models for capturing the side-effects produced during a program's execution together with a formal notion of security properties like constant-time and speculative constant-time. The thesis incorporates the development of a formally verified secure compiler that preserves the constant-time property and methodologies to add protections against various kinds of Spectre attacks. It defines type systems that help check the correct usage of these protections and also illustrates an efficient method to protect against Spectre v1 attacks called Selective Speculative Load Hardening. All these works are formally verified using the interactive theorem prover Coq, leading to machine-checked proofs rather than relying on human assurances.

Keywords : Formal verification, Computer-verified proofs, Verified cryptography, Compilation

Acknowledgements

This section is the hardest to write in my thesis as it brings all the emotions I experienced throughout my Ph.D. journey. First, I want to thank my advisors, Benjamin Grégoire and Yves Bertot. I came to France for an internship, where I worked with Benjamin. I was so intrigued by his way of mentoring me and his expertise in the domain that I decided to do a Ph.D. under him. Throughout the Ph.D., I used to talk to him every week and sometimes even during the weekend when I got stuck in some of the technical details. Benjamin was always ready to talk to and motivate me to improve professionally, irrespective of how busy he was. I was never reluctant to ask him any silly questions because he always made me feel that I was learning new things every day and never judged me based on my existing technical knowledge. Thank you, Benjamin, for having hope in me whenever I lost hope in myself during the journey and pushing me to work harder.

Talking about my other advisor, Yves Bertot, makes me very emotional because I not only shared an advisor and student relationship with him, but he is more like a father figure. Knowing that I don't speak French and am away from my parents, he ensured I was always safe and happy in this new country. Though we didn't work closely on the topic, he always provided expert advice on formal methods whenever I needed it. I would also like to thank Janet Bertot, whom I was introduced to through Yves, as Yves is her husband. Being around Janet was always enjoyable because she is a very cheerful person, and on the topics where Yves and I disagree, mostly Janet was on my side.

I would also like to thank my jury members, Sandrine Blazy, Karthikeyan Bhargavan, Étienne Lozes, and Lesly-Ann Daniel, for taking the time out of your busy schedule to attend my thesis defense. Thank you, Sandrine and Karthik, for providing valuable feedback to improve my thesis.

I would also like to thank my collaborators, Tamara Rezk, Vincent Laporte, Peter Schwabe, Tiago Oliveira, and Gilles Barthe. I want to thank two women researchers I met during the Ph.D. journey, Tamara Rezk and Nataliia Bielova, whom I admire as successful women researchers. Thank you, Tamara, for motivating me to apply for the "Young Talent Women in Science" award and always encouraging me to believe in myself. I will cherish my endless talks with Nataliia about excelling as a researcher. I greatly enjoyed all my conversations with Nataliia regarding clothes and makeup.

My Ph.D. journey would not have been this enjoyable without my best work buddies, Jean-Christophe Léchenet (JC), Pierre Boutry, and Santiago Arranz Olmos. Thank you, JC, for all your singing during the working hours; it made me feel like I was working while attending a concert. Thank you, Pierre, for introducing me to good wine and whiskey. Your existence in the room is unavoidable because you are a bully to me sometimes and extra nice to me the rest of the time. Thank you, JC and Pierre, for helping me with everything that needed to be done in French. Having JC and Pierre as my officemates was incredible, and I would like to take them both with me to my next workplace. Thank

you, Santiago, for being patient with all my stupid questions regarding Jasmin and always celebrating my achievements just like it was yours. All the crazy nights where we were drinking, singing, and dancing till 4 am will be dearly missed by me. You guys were technically my officemates, but I consider you guys "friends for life."

I would also like to thank the STAMP and SPLiTS team. Every team member was super friendly to me. Thank you, Laurent Théry, for taking time out of your busy schedule to help me practice my Ph.D. defense an endless number of times. All the talks during the lunch with Enrico, Thomas, and others helped me settle down comfortably in the new country.

Now, I would like to thank my family members, without whom nothing would be possible. I want to thank my brother, Amit Rajan, for proofreading my thesis and always supporting his little sister no matter what. I want to thank my dad for ensuring I had enough sleep and food to eat during my Ph.D. journey. I want to thank my mother and father-in-law for always supporting me in achieving my dreams and celebrating every little thing I achieved during my Ph.D. journey. I want to thank my dear husband, Basavesh, for being very patient with all my mood swings. This thesis would not be possible without you because you helped me grow technically and encouraged me to improve. Thank you for assuring me that I am enough to achieve anything in this world if I want to. You are my biggest cheerleader, and I am fortunate to have a partner like you.

I want to thank my beautiful babies, Perk and Chloe. Though they don't speak like us, their existence in my life was enough for me to get out of a bad day. Mommy loves you both more than she loves this thesis.

Last but not least, I would like to dedicate this thesis to my mother. Due to various society-biased stereotypes against women in India, my mom had to give up on her dreams. But she always ensured I would never face such a situation and worked hard throughout her life to help me fulfill my dreams. Thank you, maa, for everything you have done for me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Today's world is evolving to be increasingly dependent on using software. Software is used in every field, like medicine, the banking industry, embedded systems, mission-critical systems, cryptography, etc. Various characteristics must be considered during the development phase of a software, like functional correctness, usability, maintainability, reliability, portability, efficiency, integrity, etc. Out of all these characteristics, the functional correctness of software is the most important one because software is useless without it. Functional correctness refers to the ability of the software to behave exactly as intended in all scenarios. Imagine having a super-fast software producing the wrong result. Software's correctness helps support other characteristics like maintainability, reliability, etc. Software should only be trusted with evidence of satisfying specifications describing its behavior.

Let us discuss functional correctness and other properties related to software using a simple example present in Figure 1.1. Figure 1.1 presents a transaction *send* between two parties Alice and Bob. Execution of the *send[1, Alice, Bob]* transaction should produce the state where the balance corresponding to Bob should increase by 1 and the balance corresponding to Alice should decrease by 1.

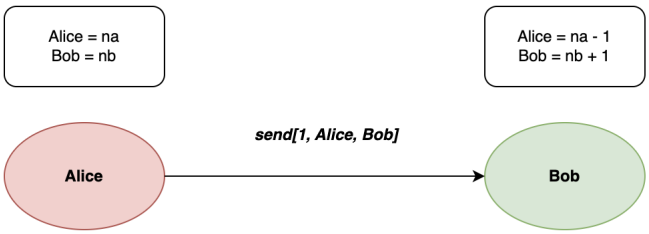


Figure 1.1 – A simple example showing a transaction between Alice and Bob

To ensure functional correctness, there is a need to provide a description of the functionality and behavior of the *send* transaction. This behavioral description is often modeled using formal mathematical notations, referred to as *specifications*. The goal is to capture the properties of interest through these specifications and provide evidence or proof that these specifications capture the properties of interest correctly.

The specifications corresponding to the *send* transaction can be as follows: (1) The balances of all parties not involved (Alice and Bob) remain unchanged. While Figure 1.1 presents two parties, assume the potential for more than two. (2) The transferred amount

is deducted from the sender's account (Alice), at least when this sender's account differs from the recipient's account (Bob), and the sender's balance is greater than or equal to the transferred amount. (3) The transferred amount is added to the recipient's account (Bob) when the sender's account differs. (4) The sender's account remains unchanged if it is the same as the recipient's account.

An additional property is that the overall balance in the chain remains unaltered after executing the *send* transaction. This property cross-validates the four previously defined specifications concerning the *send* transaction's behavior. These five specifications collectively describe the behavior expected from the *send* transaction to achieve functional correctness. Ensuring this functional correctness can be accomplished through various methods such as testing, auditing, fuzzing, program analysis, and formal techniques like theorem provers and automatic solvers. Formal method techniques, like theorem provers, provide more reliable guarantees due to formally verified certification about bug absence. This is discussed in greater detail in Section 1.2.

Functional correctness is a vital part but not a sufficient part of the software development process, as it does not give any guarantees about other security properties. For example, the functional correctness of the example presented in Figure 1.1 does not assure the data security of the *send* transaction, and neither the specifications defined above capture confidentiality.

Apart from functional correctness, one of the prime goals of any software system is to ensure data security and privacy. Cryptography is often used to enhance user privacy and secure the data transferred between parties over an unsecured network. For example, a cryptographic algorithm like RSA [Rivest et al., 1978] is widely used to secure data transfer over an insecure network. Let us talk about the RSA decryption scheme in detail. The RSA decryption algorithm computes $C^d \bmod n$ where C is the cipher-text obtained by applying the encryption algorithm on the data being sent, n is the public modulus number (product of two big prime numbers), and d is the secret key, also called as the RSA private key known only to the receiver at the other end.

A functionally correct implementation of modular exponentiation (computes $C^d \bmod n$) is present in Figure 1.2. It is functionally correct but its execution time depends on whether the bit j of d (dj) is 1 or 0. As we can see in lines 4-5, the algorithm performs extra computation when dj = 1, resulting in longer computation time. The attacker can compute $R = C^d \bmod n$ for different values of C and by precisely measuring and analyzing the time the algorithm takes to execute, the bits of secret key d can be recovered. Hence, even though the RSA algorithm is functionally correct, it was feasible to recover the secret key without breaking RSA [Kocher, 1996a].

```
1 R = C;
2 for { j = 0; j <= w -1; j++ } {
3    R = mod(R^2, n);
4    if dj == 1
5       { R = mod(R * C, n); }
6 }
7 return R;
```

Figure 1.2 – RSA decryption algorithm

A similar security vulnerability based on timing variation was seen in the implementation of zero-knowledge proof [Goldwasser et al., 1985]. Cryptosystems like Zcash or

Monero use *zero knowledge proof* technique to protect the secret information involved during a transaction. The *zero knowledge proof* is a method of proving the validity of a statement without revealing any information about it. For example, in the transaction between Alice and Bob, Bob can authenticate the amount received from Alice by using the *zero-knowledge proof*, even if the transaction does not reveal the amount sent or Alice and Bob's identities. It was possible to exploit the time taken (side-effect produced by the implementation) to generate a *zero-knowledge proof* as it varies depending on the secret transaction data [Tramèr et al., 2020]. These attacks are possible because of how a cryptographic algorithm is implemented, even though it is functionally correct. This shows that functionally correct and efficient software is still vulnerable and cannot be trusted for its confidentiality.

From the preceding discussions, it is evident that the confidentiality of any software system can be compromised via effects produced during its execution. When any software is executed, it produces measurable physical effects that can leak secret information. For example, via execution time, power consumption, noise, etc., the outside world can extract information about secret data if the software produces variations in these physical effects for different secret inputs. To avoid these attacks, the development of software systems (especially cryptographic systems, whose prime goal is security and privacy) must follow specific programming disciplines to design the algorithm which produces the same physical effects when executed on different inputs. In a nutshell, protecting secret information in any software system is hard, and guarantees like functional correctness, memory safety, etc., do not extend easily to give protection against security vulnerabilities.

Lastly, the safety property is another desired stability requirement of any software system. Safety properties provide that a program execution does not reach a "bad state". These properties often deal with reasoning about a single execution trace. However, other security properties exist, like protecting the software from attackers, which cannot be expressed using a single execution trace. These come in the category of *hyperproperties* [Clarkson and Schneider, 2008] and need to be expressed using a set of traces. The traditional compilers are not designed to preserve these hyperproperties and need to be adapted to support them. The compiler responsible for producing the binaries should not break these hyperproperties during the compilation process.

This thesis aims to provide a methodology for developing a formally verified secure compiler (preserving hyperproperties representing mitigation against timing-based side-channel attacks). The later sections in this chapter give an overview of timing-based side-channel attacks, possible mitigations against them, and the importance of formal verification in ensuring security.

1.1 Side-channel attacks

Side-channel attacks extract secret information from a program without targeting the program itself but by exploiting characteristics of the execution environment. These attacks are done by measuring or analyzing various side-effects produced during program execution, like execution time (cache hit/miss), power consumption, etc. One of the most common attacks of this kind is *timing attacks* [Kocher, 1996b]. In timing attacks, the attacker analyzes an algorithm's time to execute. Another type is *electromagnetic attacks* [Sayakkara et al., 2019] that measure (by performing some signal analysis) the electromagnetic radiations emitted from a device. Another kind is *simple power analysis*, in which the attacker studies the power consumption of a machine. Similarly, other types

of side-channel attacks exist, like profiling attacks [Picek et al., 2021], acoustic cryptanalysis [Genkin et al., 2016], etc. This thesis focuses on timing-based attacks. Timing attacks cover a broad category of side-channel attacks. For example, it includes attacks like cache-based attacks, Spectre attacks, and attacks based solely on timing variations without involving cache exploitation.

1.1.1 Cache-based attacks

Cache-based attacks are carried out by monitoring the time variation in accessing data from cache or physical memory. A cache is an essential part of modern architecture as it greatly improves the program's performance, and the cache's effectiveness is based on the cache hit or cache miss. In the occurrence of a cache miss, the data needs to be fetched from the main memory, reducing the program's performance. Cache-based attacks exploit cache-hits and cache-misses to get to secret data.

There are several kinds of cache-based attacks discussed in prior works [Ge et al., 2018]. In the PRIME+PROBE [Tromer et al., 2010] attack, the attacker primes the cache by filling one or more cache lines. When the victim program is executed, the attacker tries to determine which cache lines were evicted by the victim. By noticing the difference in the time to access the memory address that the victim evicted in the cache set, the attacker can get to the memory index accessed by the victim. Another attack is FLUSH+RELOAD [Yarom and Falkner, 2014], which uses shared memory (such as shared libraries) between the attacker and the victim. First, the attacker flushes a cache line of interest to the attacker and the victim using dedicated instruction like cflush and later measures the execution time of reloading the data. Based on the time variation, the attacker comes to know whether the victim has reloaded a memory address or not. Another kind of attack is FLUSH+FLUSH [Gruss et al., 2016], which exploits the execution time of the cflush instruction. The time taken by cflush will be shorter if the data is not brought to the cache, and it will take longer if it has been brought to the cache by the victim program. Another kind of attack is EVICT+TIME [Osvik et al., 2006]. In this attack, the attacker waits for the victim program to run, which might bring some data into the cache. The attacker then evicts a cache line (due to the limited size of the cache) and reruns the victim's program. The difference in the execution time helps the attacker to get to the memory address being accessed by the victim.

```
1 x = 4;
2 if secret {
3     x = x + 1;
4 }
5 else
6     while (x != 0) { x = x - 1; }
```

Figure 1.3 – Branch on secret

All the attacks discussed in the previous paragraph perform cache-based exploitation. There are also timing-based attacks that can be possible without involving a cache. Figure 1.3 presents a simple program that gives room for performing timing-based attacks without involving cache. The program has a conditional instruction that branches on a secret. The true branch consists of only an assignment instruction, and the false branch

consists of a while loop (which takes longer to execute than a single assignment instruction). Depending on the value of secret passed to the branch, its execution time will vary; that, in turn, helps the attacker to predict the secret.

A more realistic example discussed by Kocher in his paper [Kocher, 1996b] is: by measuring the time required to perform the private key operations in the RSA decryption, the attacker can get to the private key. There is various recent work in literature like [Canella et al., 2019b] [Bernstein, 2005] [Osvik et al., 2006] [Lipp et al., 2018] [Schwarz et al., 2019] [van Schaik et al., 2019] [Bulck et al., 2018] [Canella et al., 2019a] etc., that explains how timing-based attacks have helped the attacker to get to the secret data.

1.1.2 Spectre attacks

Spectre attack [Kocher et al., 2019a] is also a timing-based side-channel attack. Modern processors are designed to perform speculative execution to improve performance. Modern processors aggressively use branch, address, and value prediction to process the execution without awaiting the result of the prior computation. Once the prior computation is done, the real value is computed and must be checked against the guessed value. If the predicted path is correct, the microprocessor commits speculatively computed results to the architectural state, increasing the overall performance. If the predicted path is incorrect, the microprocessor backtracks to the last correct state by discarding all speculatively computed results like resetting the registers involved, resetting the program counter, etc. In this case, the results of misspeculation are never committed to the micro-architectural state, but they still leave traces such as leaving a trace in the cache, modifying the content in the cache, etc.

Figure 1.4 presents two code snippets, which show how an attacker can exploit branch misprediction to leak the data from the secret part of memory.

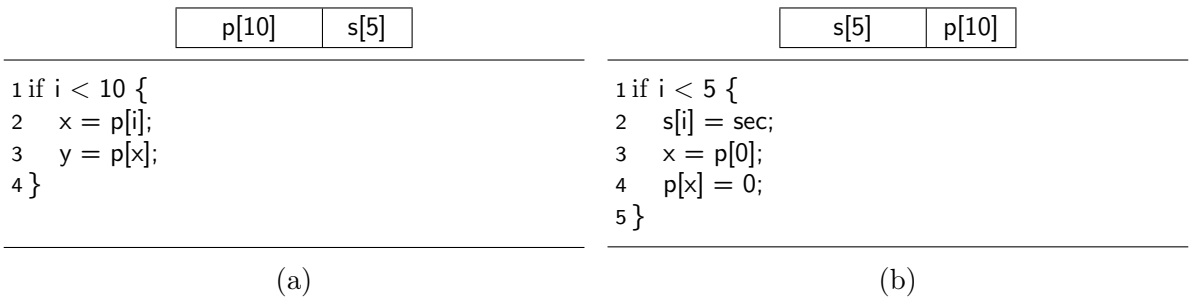
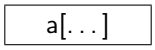


Figure 1.4 – Array p is public and array s is secret.

Figure 1.4a presents a consecutive fragment of memory where the memory cells till index 9 (represented as p[10]) are public parts, and the following five cells (represented as s[5]) are secret parts. The attacker first primes the branch to predict that the condition i < 10 is true by executing the code several times with a value of i less than 10. Then the attacker provides the value of i as 13. Based on the old heuristics, the processor mispredicts the branch in line 1 and proceeds to execute the true branch; hence loads data from memory at index 13, i.e. s[2] into the variable x. In other words, variable x becomes transient (speculatively depends on the secret). On line 3, the attacker performs a read operation based on the value of x, which brings x to the cache. Once the processor

evaluates the condition of the branch, it rolls back the computations, but the cache persists. The attacker can later exploit the cache to infer the value of s[2].

Figure 1.4b presents a consecutive fragment of memory where the memory cell till index 4 (represented as s[5]) are secret parts, and the following ten cells are public. The processor can speculate for the value of i as 6. The processor mispredicts and will write secret data sec in the public part of the memory at p[0]. In line 3, the load operation will load sec in the variable x. Line 4 leaks sec by performing a store operation at index x.




```
1 fun STL(int sec, int pub) {
2     int i r;
3     a[0] = sec;       Store secret value
4     a[0] = pub;       Store public value
5     i = a[0];         Speculatively load secret value
6     r = a[i];         Secret-dependent access
7     return r;
8 }
```

Figure 1.5 – Store bypass load

Figure 1.5 presents another example where the store operation on the memory fragment a is speculatively bypassed. In this example, the attacker exploits the memory disambiguator, which predicts Store To Load (STL) data dependencies and gets to the secret data. The function STL takes arguments sec and pub and allocates them on memory array a. Line 3 allocates sec at index 0, and line 4 assigns pub at the same location. Line 5 loads the value at index 0 into i, and then at line 6, the value of the array at index i is accessed. Without speculation, at line 5, the variable i is supposed to get the value pub. However, i can be equal to sec if the memory disambiguator incorrectly predicts that the store to a[0] at line 4 is unrelated to the load into i at line 5 and hence ends up loading the secret-dependent memory location a[sec] into the cache at line 6.

Spectre attacks can be of different types. An overview of different kinds of Spectre attacks is as follows:

Spectre v1 Spectre v1 is triggered by exploiting conditional branches. Attackers mistrain the CPU's branch predictor to proceed in the wrong direction, which helps the attacker read secret information. Figure 1.4 presents program vulnerable to Spectre v1. These kinds of attacks are also called Spectre-PHT (pattern history table) attacks.

Spectre v2 Spectre v2 is triggered by exploiting indirect branches. The indirect branch predictor's job is to guess the target of indirect branches. In the case of direct branches, the instruction includes the jump destination address (either stored as full or an offset that can help in calculating the destination address). In the case of indirect branches, the instruction includes a pointer to a memory address, and this location in memory stores the destination address. The attacker can mistrain the branch predictor with malicious destinations, making the speculative execution lead to these malicious memory addresses. This influences the target of the indirect branches, causing the processor to carry on with the speculated indirect branch prediction and exposing the secret data. Eventually, the attacker can exploit the traces left in the cache.

Spectre v4 Specter v4 is also called Speculative Store Bypass. In this attack, speculatively, data is read from the memory address, which was not ready. For example, if a store operation precedes a load operation, the result of the store operation depends on the prior instructions. The load operation can be executed speculatively (ignoring or bypassing the store), reading the data at the memory location, which does not contain the valid data yet as the store has yet to be executed. This leads to the exposure of the secret data. Eventually, the processor recovers from the misspeculation, but the exposed secret data is still present in the cache which the attacker can further exploit. The program present in Figure 1.5 that is discussed earlier is one of the examples of Spectre v4.

Spectre v3/Meltdown Meltdown works differently than Spectre attacks. It does not involve branch prediction; instead, it observes when an instruction causes a "trap". A trap occurs when the processor encounters an undefined instruction—for example, an error in instructions like division by zero or performing an illegal memory access. An attacker can train a speculatively executed instruction to bypass memory protection. This might even help the attacker to access kernel memory from the user space. This scenario will create a trap, but before the trap is issued, the instruction that caused the trap will leak the accessed memory address.

1.1.3 Possible mitigation against side-channel attacks

An essential goal of any software system, especially the cryptographic system, is to ensure data security and privacy with efficient implementation. Timing attacks are a significant hurdle in ensuring data security and privacy. The most reliable and popular way to deal with timing-based side-channel attacks is via *constant-time* programming. For example, to deal with Spectre attacks, the program should be written so that it does not leak secret information, even in case of speculation. The terminology often used to describe this is *speculative constant-time* programming discipline, which ensures that a constant-time program should not leak secret information even when the processor speculates.

Constant-time

Constant-time programming is a programming discipline used in many cryptographic systems and algorithms to make them secure against timing-based attacks. Timing-based attacks can result due to exploitation of control flow or exploitation due to memory access. A constant-time program should not branch on secrets and should not perform any secret-dependent memory accesses.

In today's world, most libraries are public, meaning anyone has access to various cryptographic implementations. The developer's goal is to write those implementations so that secret information does not influence the execution time of those implementations. The exploitation due to control flow can be mitigated by doing a manual or a tool-based transformation of a branching instruction into a straight-line code. This can be achieved by replacing conditional instructions with constant-time bit-masking operations. Similarly, exploitation due to memory accesses should be avoided.

```
1 fun isnotCT(int x, int y, bool secret) {
2    if secret
3    then return x;
4    else return y;
5 }
6
```

(a) Non constant-time version

```
1 fun isCT(int x, int y, bool secret) {
2    return ((x ∧ (0 - secret))
3          ∨ (y ∧ ∼(0 - secret))) }
4
```

(b) Constant-time version

Figure 1.6 – From non constant-time to constant-time version.

Figure 1.6 presents a function isnotCT on the left, which is prone to exploitation due to control flow. The function returns x or y depending on the secret bit. As it performs a secret-dependent branching, the program is not constant-time. The function isCT on the right in Figure 1.6 replaces the branching with logical bitwise operations to select x or y. The two functions are functionally equivalent and the function isCT is constant-time.

Writing a constant-time program requires complex thinking, as one needs to think about how the program will behave in terms of its execution time, which in turn brings a need to think about the low-level details. Before and after transforming the program to be constant-time, the program should be functionally correct and produce the same desired behavior. Another major issue when relying on manual transformation to produce a constant-time program is the false belief that the program is constant-time. These facts give rise to the need for some sound mechanism or a formally verified tool that offers a more robust guarantee that a program is a constant-time program and hence does not leak secrets.

There are specific pre-existing tools [Barthe et al., 2013] [Cauligi et al., 2017], guaranteeing that a program follows constant-time programming discipline and is secure against timing-based attacks. Another way to prevent exploitation due to control flow or memory access is by designing a *information-flow based type system* to derive facts about the security types of the program. An information-flow-based type system can be used to derive facts about the security types of the program. The variables in the program should be annotated with security types like secret (variable storing secret information) and public (variables storing public information). The typing rules for each instruction should ensure that when the instruction is executed, it does not leak any secret information. For example, the index of the memory access should be annotated as public so that the attacker can not infer any information about the secret data by exploiting the cache.

Speculative constant-time

As discussed in Section 1.1.2, modern processors generally optimize the execution of the code by relying on speculation. Hence more than constant-time programming is needed in the world of speculation, where the execution can go out of order. The constant-time programming is applicable to provide mitigation against timing-based attacks only when the instructions are executed in order. A constant-time program can still reach the secret part of the memory if executed speculatively. Speculative constant-time programming goes beyond constant-time programming discipline and ensures that the secret data is not leaked even in case of speculation. There are various existing countermeasures for writing speculative constant-time programs. They are broadly divided into two categories:

- **Hardware countermeasures**: A straightforward solution to avoid cache-based side-channel attacks in case of speculation is to stop the speculation. Processors

provide some hardware features to stop the speculation. One of them is lfence instruction. An lfence instruction at a program point forces the processor to evaluate all the prior instructions before that program point. For example, putting a lfence instruction right after the conditional guard (or before an indirect branch), the processor must evaluate the guard before proceeding to one of its branches. The lfence instruction is supported by architectures like AMD [AMD, 2018] and Intel [Intel, 2018]. Though current processors do not provide any in-built solution to stop the speculation completely, it is the programmer's job to correctly insert lfence instruction. This solution has a significant impact on performance as it stops speculation.

As discussed in the work [Intel, 2018] by Intel, a better approach would be to use static analysis to reduce the number of lfence instructions to be inserted because many paths (including speculation) in the program do not have potential threats to leak the secret data. As discussed in Microsoft's C compiler, MSVC [Spe, 2018] misses many vulnerable places in the code post static analysis. The reason is that it is hard for the compiler to statically determine the faulty spots that might be vulnerable to timing attacks. It is necessary to avoid unreliable mitigation as even a single exploitable code might leak the entire memory content.

- **Software countermeasures**: Speculative load hardening is a compiler-based software countermeasure for Spectre v1 attacks. The program's execution keeps track of misspeculation by maintaining a predicate indicating whether a program is misspeculating. The compiler uses this predicate to check whether the load operation is part of a speculating or non-speculating path. When data is being loaded from a speculated source, the output of the load operation (loaded value) or the input of the load operation (memory address) are masked to a default value. Both of these variants are supported by the LLVM compiler [Chandler Carruth, 2021].

```
1 fun withoutSLH(int* address) {
2   if condition {
3       ...
4       x = a[address]; }
5 }
6
```

(a) Without SLH

```
1 fun withSLH(int* address) {
2   all_zero_mask = 0;
3   all_one_mask = 1 ... 1; size of address
4   if condition {
5       predicate = !condition ? all_zero_mask : all_one_mask;
6       ...
7       address = address & predicate;
8       x = a[address]; }
9 }
10
```

(b) With SLH

Figure 1.7 – Speculative load hardening

Figure 1.7 presents two code snippets describing speculative load hardening. Figure 1.7a presents a simple function withoutSLH that might leak the address speculatively. Depending on the condition, the load operation in line 4 might lead to out-of-bound access and leak the secret address. Figure 1.7b presents the same program that adds a predicate to check the speculation. The variable predicate gets the value all_zero_mask in case of misspeculation (!condition) and all_one_mask in case of no misspeculation (when condition is satisfied). all_one_mask is a sequence of size

address pointer containing all 1s. Line 7 calculates the address based on the predicate. It uses a bit-wise and operator (&) to compute the address. In misspeculation, the address will be masked to 0. In case of no speculation, the address will remain the same. Hence the safe masked value of the address will be leaked in line 8 instead of the secret address.

1.1.4 Verification of constant-time and speculative constant-time

Verifying a property at the source level is often much easier than at the low level. A lot of information gets lost during the compilation process, and it is much easier to reason with high-level code syntax than at assembly. A wide range of tools are available in literature and practice to verify the constant-time property. These tools can be a dynamic tool or a static tool.

Dynamic tools observe the run-time behavior with various inputs to evaluate whether different run-time executions produce the same or different visible observations for the outside world. Static tools usually use formal method techniques like static analysis, program transformation, etc., to reason about the program against some abstract model. This abstract model represents the visible effects produced during the program's execution. Static analysis tools based on symbolic execution, abstract interpretation, taint analysis, etc., are often designed to find program bugs, do not aim to provide completeness, and often need help with state-explosion issues. Formal method techniques like formally verifying software against some mathematical description of its behavior aim to justify the absence of specific categories of bugs. They are more complete and sound than static analysis tools.

Another point to consider before choosing any tools is the kind of input they need, i.e., whether they work at the source or target level. Several tools like FlowTracker [Rodrigues et al., 2016b], FaCT [Cauligi et al., 2019a], etc. work on the source level. Some tools work on low-level language like ct-verif [Almeida et al., 2016a], Binsec/Rel [Daniel et al., 2020], etc.

The tools working at the source-level language do not provide an end-to-end guarantee against timing-based side-channel attacks. The source-level language is present above in the compilation chain to the assembly/binaries. The secure source-level code is still vulnerable to leaking secret information at the lower level because the compiler can introduce timing attacks. This is because general-purpose compilers like LLVM do not provide side-channel resistance. This makes the tools working directly at the low-level language to have a stronger guarantee against side-channel attacks.

Another concern about these tools are regarding their usability and versatility. For example, tools like EasyCrypt [Barthe et al., 2011a] take Jasmin (a language designed to write highly-efficient cryptography programs that target x86 architecture) as input. To verify using EasyCrypt (proving constant-time property), the user needs to develop the program in Jasmin or directly in EasyCrypt domain-specific language, which requires rewriting the existing libraries.

State-of-art of some of the existing tools This paragraph discusses some of the tools available in the literature and in practice, with their usability and soundness to reason about constant-time property

- ct-verif[Almeida et al., 2016a]: A static tool that verifies the constant-time property and works on the level of LLVM-IR with source-code annotation. It ensures that the code does not perform secret-dependent branching, secret-dependent memory access, or timing-variable operations. The approach is proved to be sound and complete using Coq proof assistant. The tool's usability depends on many other tools like Boogie, Z3, etc.
- FlowTracker[Rodrigues et al., 2016b]: A static tool that ensures the constant-timeness by analyzing the Program Dependence Graph at the target level (LLVM-IR form). The approach does not guarantee completeness.
- FaCT[Cauligi et al., 2019a]: FaCT provides a domain-specific language for writing a constant-time program that removes the possibilities of leakage by program transformation. The language is close to C and is compiled to LLVM-IR. The program needs to be annotated with `secret` keyword that triggers the compiler to perform the transformation, which makes the program constant time. The soundness of the FaCT framework is proved on paper.
- Binsec/Rel[Daniel et al., 2020]: It is a symbolic verification analysis tool that works on a binary level. It targets x86 and ARM architecture and does not rely on the source code.

All these tools have been used to verify constant-time properties for several Cryptographic algorithms. Yet, this still leaves an open question: Can we trust these tools? For example, EasyCrypt also takes translated Jasmin code as input, but the translation is still a trusted base. Similarly, the translation used in ct-verif and other tools is also a trusted base. For the tools like Binsec/Rel, the state explosion problem is always a concern. The FaCT gives no machine-checked guarantee about the correctness of its framework. An ideal situation would be to formally verify these tools using some machine-checked proofs. But as it is well-known that formal verification is expensive, it is often skipped.

1.2 Formal verification

Software is correct if it behaves exactly as intended; hence, while designing any software, understanding the specification is crucial. A program specification is a mathematical description of the program's behavior, and it should specify the properties of interest related to the software concerned. A situation can arise where the correct software is unreliable if the specification does not capture the behavior correctly and vice-versa. Hence, a formal mathematical demonstration is needed to write these specifications, and it is necessary to capture the properties of interest through those specifications correctly. After coming up with the correct sets of specifications, the software needs to be proven or tested to satisfy those specifications.

Developer uses various methods to verify the correctness of their software. The classic approach to ensure software's correctness is to test them. This involves developing software, running it on sample data to generate the result, and verifying these results to see

if they are as expected. But it is well-known fact that the real-world use cases are so vast that it is almost infeasible to test the software for all possible scenarios. Hence, the testing mechanism does not guarantee the software's correctness; it needs more completeness. Another approach is to use auditing or perform a manual reviewing the software to find if there are any bugs in its implementation. The auditing process might involve automated techniques like fuzzing that cover broader test cases. Automated testing, like fuzzing, is widely used to help find the presence of bugs rather than their absence. It might help in reporting that the software did not fail for a broader set of samples generated using some automated tool. Still, it also does not guarantee the software's correctness, as it needs more completeness.

Formal method-based techniques can be essential in achieving soundness and completeness in software correction, security, and privacy. The formal method helps us to reason about software or hardware by mathematically specifying the model and its specifications and eventually verifying that these specifications are valid for all sample data sets. Functional correctness is one of the essential properties of any software application, but security and privacy play an equally important role, especially in any cryptographic system. As an application of this thesis is cryptography, it is essential to discuss the significance of formal methods in cryptographic systems.

Correct software is a basic requirement in the world of cryptography. Cryptography software should satisfy three properties: confidentiality, integrity, and authenticity. It is a challenging task to carry out formal verification in cryptography. Cryptography implementations are designed to be highly optimized code, making the formal verification much more difficult because the implementation is often written in a mixture of high-level language and assembly. This means that the verification tool should model the high-level language, assembly, and their interaction to reason about the program written using both. Also, each optimization needs a separate verification proof showing the process is sound. Cryptographic implementations often work on inputs and outputs of enormous size; hence the state space can be too ample for program verification tools, which makes the tool questionable about their performance and usability in real-world cryptography.

This thesis focuses on reasoning about non-interference security properties like constant-time and speculative constant-time for cryptography algorithms.

A taste of formal model for verifying constant-time property To prove constant-timeness, there is a need to establish an abstract model of the system. A specification representing the constant-time property can be stated formally as "a program executing from two indistinguishable (differ only in secret parts) states always produces the same visible effects". Since the visible effects will be the same for the external observer, the attacker cannot differentiate between the two traces. To model these visible effects, a formal model needs to be developed. Traditionally, the semantics of a program is described to capture the program's input-output (functional) behavior.

$$p : s \Downarrow s'$$

A program starting from state s produces state s' where the state captures the memory updates done by the program. The semantics needs to be instrumented to capture the non-functional properties like constant-time.

$$p : s \Downarrow^{\ell} s'$$

The notation ℓ (also often called leakages) represents the visible effects produced by the program. The outside world can see ℓ when the program p executes. To reason about cache-based attacks, ℓ represents a language describing the program effects that might lead to timing-variation attacks. For example, in case of conditional instruction $\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2$, ℓ will consist of b and effects produced during the execution of c_1 or c_2. As discussed in Section 1.1.3, a constant-time program should not branch on secrets; hence ℓ contains b in case of conditional instruction, making it public and visible to the outside world. Similarly, the memory accesses leak the index at which memory is accessed.

The formal model helps establish an abstract model for reasoning about the constant-time property. It is better compared to manual reviewing, testing, or automated testing methods because it helps in giving a complete solution of representing all possible situations that may give rise to timing-variation attacks. After establishing the formal model of the program's effects (leakages), the goal is to verify that the program is indeed constant-time formally. Formally it is stated as follows:

$$p : s_1 \Downarrow^{\ell_1} s_1' \wedge p : s_2 \Downarrow^{\ell_2} s_2' \wedge s_1 \sim s_2 \implies \ell_1 = \ell_2.$$

The above statement presents that two executions of program p produce the same visible effects i.e. $\ell_1 = \ell_2$ when they start their execution from two indistinguishable $\sim$ states s_1 and s_2. The indistinguishability relation $\sim$ induces an equivalence relation on states, ensuring that the two states only differ in their secret parts.

1.2.1 Preservation

The preservation is a property about the semantics of the language and compiler, which helps establish that if we evaluate a program satisfying a set of specific properties of interest, all the intermediate evaluation steps yield a valid program satisfying those properties as well. The end goal of a software development process is to build a bridge that resembles a connection between the high-level specification and the actual machine code implementation of the software that gets executed. The compiler is a complex part of the whole software framework as it takes care of many features like transforming source code into executable, optimizations, etc. Hence, a buggy compiler might produce an executable that does not produce the same output as the source. However, bugs introduced by the compiler are negligible compared to the bugs issued in the source program by the developer. However, the situation differs for safety-critical or mission-critical systems where human lives and information are at stake. Hence, the software used in these areas must be carefully reviewed and should leave no room for bugs. Unfortunately, most formal method tools operate at the source code level and do not provide formal guarantees for correctness. A buggy compiler still has the opportunity to invalidate the properties proved at the source and produce executable code that no longer guarantees those properties.

1.2.2 Certified compiler

We need to switch to a certified compiler to develop more trust in the compiler. A certified compiler guarantees that if a property holds for the source program and the source program compiles successfully to the target program, then the target program also satisfies those properties. A formally verified compiler has machine-checked proof for its correctness. In other words, the executable code it produces is formally proved to behave exactly as specified by the source-level code. These formally verified proofs help achieve

```
1 int sprog(int b, int x, int y) {
2   int result;
3   result = b ? x : y;
4   return result;
5 }
```

Figure 1.8 – Example program: source program

high assurance compared to manual testing. The formally verified compiler has been in the literature for a long time , and there exists formally verified compilers like CompCert [Leroy, 2009], Jasmin [Almeida et al., 2017] etc., that work for real-world programming languages. Their correctness proof shows that the compiler preserves the safety and behavior of the source-level programs. A certified compiler guarantees the preservation of functional correctness or memory safety but does not guarantee the preservation of security properties. Functional correctness or safety properties are represented using a single execution trace. To reason about the preservation of functional correctness or safety, two executions must be considered (one at the source and one at the target level). But to reason about security properties like constant-time, a set of traces is required at the source and target levels. Often the source and target states are not the same; hence the reasoning about security properties like constant time requires deriving indistinguishable relations between the target states from the source. Due to these limitations, traditional compiler proofs need to be adapted to reason about security properties.

1.2.3 Certified secure compiler

A compiler is designed to transform the source-level program into the executable program and also performs a range of optimizations on the source code to improve overall efficiency. In the process, it may transform a source program that satisfies the constant-time property to a target program that is not constant-time.

Figure 1.8 presents a simple program, which chooses between x and y based on b using the ternary operator. The value is assigned to the variable result using a ternary operator in line 3. The ternary operator b ? x : y is a constant-time operator as it does not leak the condition (assuming it will be compiled to a conditional move). Hence source program sprog is constant-time. Unfortunately, the compiled code in Figure 1.9 is not constant-time. The assembly program is generated using the x86-64 clang 15.0.0 compiler without any compiler option. The compiled program used a jmp instruction in line 11, which transfers the program control to a different point in the set of instructions based on the condition b. jmp is not a constant-time instruction as it leaks the condition and the instruction pointer. Hence the compiled code is not constant-time, even though the source program was constant-time. Another example is compiling a program that performs equality operations on strings. In programming languages like C, equality of string is performed using strcmp() build-in function provided in the String library. The strcmp() function is considered constant-time at the source-level, but it is compiled to a code that compares the strings character by character. The compiled semantics will not be constant-time as it uses a loop that exits early when two characters of input strings differ. The low-level compiled program may reveal the first index where the two characters of strings differ; hence, it is not constant-time if the input strings are secret.

Another example is presented in the paper [Kaufmann et al., 2016], where they build

```
1 sprog(int, int, int):
2     push rbp
3     mov rbp, rsp
4     mov dword ptr [rbp - 4], edi
5     mov dword ptr [rbp - 8], esi
6     mov dword ptr [rbp - 12], edx
7     cmp dword ptr [rbp - 4], 0
8     je .LBB0_2
9     mov eax, dword ptr [rbp - 8]
10    mov dword ptr [rbp - 20], eax
11    jmp .LBB0_3
12    .LBB0_2:
13    mov eax, dword ptr [rbp - 12]
14    mov dword ptr [rbp - 20], eax
15    .LBB0_3:
16    mov eax, dword ptr [rbp - 20]
17    mov dword ptr [rbp - 16], eax
18    mov eax, dword ptr [rbp - 16]
19    pop rbp
20    ret
```

Figure 1.9 – Example program: compiled program

a timing attack against an implementation of a scalar product on an elliptical curve. A 64-bit multiplication is assumed to be constant-time in an architecture like x86 (it does not leak information about its operands). But in an architecture like ARM, the compiler optimizes the multiplication to take less execution time when operated on smaller values than large ones and leaks the information about the operands.

The program present in the left-hand side in Figure 1.6 is compiled to a program using the x86-64 gcc compiler (without any optimization enabled) that contains a jmp instruction (clearly not a constant-time program), while the program present in the right-hand side in Figure 1.6 is compiled to a constant-time assembly that contains the cmov instruction.

These examples show a need to prove or justify that a compiler does not break the constant-time property while transforming the source-level code into binaries. It is harder to reason about these properties at the low-level code due to their complexity and loss of information during the transformation from the high-level to the low-level code. Hence it is essential to preserve these properties from the high-level code to the low-level code that guarantees end-to-end protection against timing-based side-channel attacks.

To make the code immune to timing-based side-channel attacks, it should be ensured that no unauthorized memory reads occur during a speculative execution or if it occurs. The values read are only utilized by safe operations (that do not leak any information). In addition, it should also be ensured that these guarantees are not broken by compiler optimizations and are carried out at the assembly level. A classic formally verified compiler must be extended to reason about non-functional properties like preservation of constant-timeness, etc. As the definition of preservation of functional correctness is based on semantics preservation (a notion of classic simulation), this can be further extended to discuss the preservation of security properties without breaking the functional correctness. A supporting machine-checked proof for preserving these properties will give a high assurance against timing-based attacks.

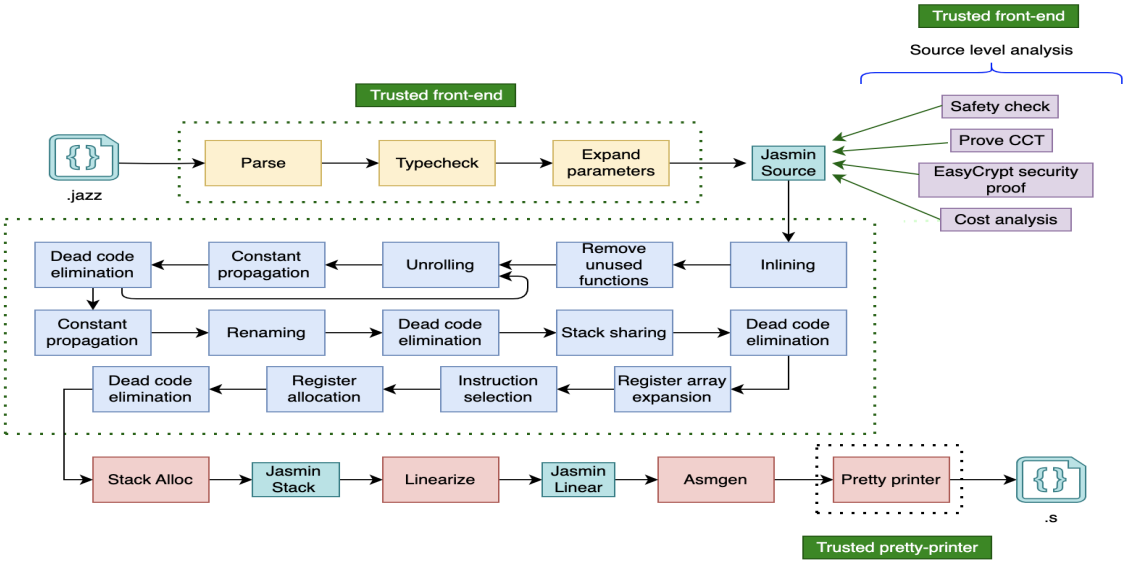


Figure 1.10 – Jasmin architecture

1.3 Background on Jasmin

As discussed in Section 1.2, the formal method plays an essential role in producing high-assurance cryptographic software that will be functionally correct and secure against timing-based side-channel attacks. On the practical side, there should be a sound connection between formal verification methods and a framework that can be used to produce efficient cryptographic implementations. In practice, it is often seen that a low-level programming language is used to write cryptographic algorithms as it gives programmers the freedom to choose between low-level details that, in turn, provide knowledge about where the data flows and help produce more efficient code. But it is not very easy to implement big complex algorithms in low-level language.

Another norm followed while developing cryptographic software is that the developers first produce an implementation that is very close to the mathematical specification (called *reference* implementation). The *reference* implementation can be compiled to be executed on different architectures and is considered reasonably efficient. A more optimized, architecture-specific, or generic implementation is produced later, taking the reference implementation as the base. This optimized implementation can be produced by directly implementing the algorithm in assembly or using low-level instructions directly at the source code level. Unfortunately, this approach is much harder in practice as it takes a lot of effort to directly implement any algorithm in assembly. Due to this, the optimized implementation is prone to more bugs as compared to reference implementations and hence receives efficiency at the expense of security. All these factors lead us to the need for a framework that makes sure that all the properties, such as functional correctness, efficiency, memory safety, security against timing-based side-channel attacks, etc. of cryptographic software is satisfied, with an ease of implementation in a high-level language for developers.

Jasmin was developed to meet all these requirements. Jasmin [Almeida et al., 2017] [Almeida et al., 2020] is a framework that consists of a Jasmin language, a formally-verified compiler (proved to be functionally correct using Coq proof assistant), and support of verification tools for proving memory safety, constant-time, etc. at source-level. The

formally verified Jasmin compiler currently supports AMD64 architecture and AVX/AVX2 (support for ARM is in progress). The source-level language's verification features rely on a tool called EasyCrypt. The Jasmin source-level program is translated to the EasyCrypt program (a trusted base), and various security proofs like constant-time properties can be carried out on the translated EasyCrypt programs (a set of modules). EasyCrypt program is a one-to-one mapping of the Jasmin program; hence any property satisfied by the EasyCrypt program is also valid for the Jasmin program. The proof script in EasyCrypt depends on the complexity of the specification (specifying the property) and implementation to be verified against that specification.

1.3.1 Jasmin language

The Jasmin language is a mixture of high-level constructs and low-level features. It is a verification-friendly programming language that supports "assembly in head" design. "Assembly in head" design means that the programmer writing a program in Jasmin knows what assembly instructions will be generated after the compilation. Jasmin language can be used to write highly efficient implementations due to the level of control it provides to the developers. In Jasmin, the developer gets to decide which variables are placed on registers or in memory. Jasmin also supports high-level language constructs like function calls, control flow, and arrays. Jasmin supports three different types of functions: inline, local, and export functions. A function annotated with inline keyword is fully inlined at the caller site. Export functions can be called from external implementations (at this moment, System V AMD64 ABI calling convention is supported). In Jasmin, it is not possible to call externally defined functions. The reason behind this is various proofs related to security are needed for cryptographic algorithms, and it becomes messy if we call untrusted functions from outside. In terms of control-flow structures, Jasmin supports if, while, and for loops.

A variable in Jasmin is declared with annotations specifying the storage class and type. The storage class (stack, reg, inline, global) resembles whether the variable will be stored in the stack or register. For example, if the storage class of a variable is stack, it will be stored in the program's stack frame, and the compiler calculates its relative address with respect to the stack pointer and the type of the variable. A variable allocated on the register is always present in the register. It is the job of the stack allocation and register allocation pass to share the stack or register between the live variables. This means a Jasmin program that is semantically valid might fail if there are more live variables whose storage class is reg than the number of registers. The developer must allocate some variables to the stack to compile these kinds of programs. The inline variable is always declared with statically known values that can be used as immediate values in the program while performing some computation. The global variable is placed in the .data section and is also declared with statically known values. These design choices are important in the case of implementing Cryptographic algorithms as it greatly improves performance. For example, accessing the variables in registers is always faster than accessing them from memory. The basic types Jasmin supports are words, boolean, and integers. Words range from 8 to 256 bits and are represented as u8, u16, u32, u64, u128 and u256 (u stands for unsigned). Boolean type is represented as bool in Jasmin and can be used to handle operations related to flags (the developer cannot directly manipulate these flags, but the compiler updates them). An integer is represented as int, and it represents an unbounded integer that should be statically known. Jasmin supports arrays that can be declared of

$e \in \mathsf{Expr}$	::=	x	variable
	|	b	boolean
	|	$a[n]$	array init
	|	c	constant
	|	$a[e]$	array access
	|	$[e]$	memory load
	|	$\mathsf{op}(e, \ldots, e)$	operator
	|	$\mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e$	conditional
$d \in \mathsf{Lval}$	::=	x	variable
	|	$a[e]$	array store
	|	$[e]$	memory store
$i \in \mathsf{Instr}$	::=	$d := e$	assignment
	|	$\mathsf{if}\ e\ \mathsf{then}\ i\ \mathsf{else}\ i$	conditional
	|	$\mathsf{while}\ i\ e\ i$	while loop
	|	$\mathsf{for}\ x\ r\ i$	for loop
	|	$\mathsf{call}\ ii\ xs\ fn\ e$	call
	|	$\{i; \ldots; i\}$	sequencing

$a \in \mathcal{A}$ ranges over array variables; $x \in \mathcal{X}$ ranges over scalar variables

Figure 1.11 – Syntax of programs

storage class reg and stack, and they are treated as first-class values: functions can take them as arguments and return them as results. Arguments are passed by value: an array passed to a function is not modified unless this function also returns it. This considerably simplifies the reasoning about program behaviors.

The syntax of the Jasmin language is present in Figure 1.11 that is later used for formal definitions and reasoning in the later chapters. It is a grammar describing the Jasmin source-level language. The Jasmin semantics is formally specified using Coq proof assistant. Hence, given a Jasmin program and initial memory, we can run the program and investigate the resulting memory. This gives more freedom to test and validate any specification related to the program.

1.3.2 Jasmin compiler

The Jasmin compiler compiles the high-level language into assembly (x86_64). The Jasmin compiler is mainly written using the Coq programming language; however, some parts are written in OCaml. The overall compilation chain is present in Figure 1.10 and is formally verified for its correctness in Coq, except the front-end (parsing, type checking, and expansion of parameters) and the assembly pretty-printer that are trusted. Throughout the compilation, five different intermediate representations (IR) are used. At the highest level, the Jasmin source language is verification-friendly: it is structured and has clean semantics. Formal verification of Jasmin programs is done on this intermediate representation. The middle end manipulates the Jasmin IR. The Jasmin IR has the same syntax (presented in Figure 1.11) as the Jasmin source but more flexible semantics that allow more optimizations. The last pass of the middle-end uses Jasmin-stack as output: this IR again has the same structured syntax but also features an explicit stack pointer. The

back-end outputs unstructured IR: Jasmin-linear with labels and gotos after linearization and assembly at the end.

The first two passes parse the source program and type-checks them. Then, the *expand params* pass replaces parameters by their values.

Inlining compiler pass replaces the function call annotated with keyword inline with its body (assigning their corresponding arguments and result variables between the caller and the inlined callee).

Dead call elimination (also called Remove unused functions) eliminates the functions that are never called.

Unrolling compiler pass thoroughly unrolls for-loops. The bounds are statically determined for the loops. There might be a case where the range of the loop is not constant and can only be known after *constant propagation*. Hence this pass iterates a sequence of *unrolling*, *constant propagation*, and *dead-code elimination* until the program stops changing or the maximum number of iterations is reached. It also introduces new assignments in the program to set the values of the variables involved in the range of the loop.

The *constant propagation* compiler pass replaces the variables and parameters with their constant values and propagates these values to expressions that contain them.

Dead code elimination pass removes all the unused variables; all the variables that are assigned some values but are not read/used afterward in the program.

Stack sharing pass optimizes the memory layout of local variables. This pass allocates the variables allocated on the stack but not alive simultaneously to overlapping stack regions.

The *register array expansion* pass translates arrays into register variables (collection of registers represented as arrays) or stack (array representing contiguous memory addresses) variables. It ensures that all array accesses are done through statically determined indices.

Lowering/Instruction selection pass translates the high-level Jasmin instructions into low-level architecture-specific instructions. For example, assignment is translated into low-level operations (for instance, x = x + 1 is replaced by x = #INC(x)), conditional expressions are translated into architecture-specific flag-based conditional instructions, etc.

The *register allocation* pass allocates the register variables on the architecture registers. This pass does not spill any variable to memory if there are insufficient registers for the live register variables. It infers the mapping from all variables annotated with reg storage class to the available architecture registers. The compiler returns an error if there are not enough registers. It also considers the architecture-based constraints; for example, some architecture-specific instructions require that the output register is the same as one of its arguments, precise handling of flags, and so on. It is based on an algorithm derived from linear scan algorithm [Poletto and Sarkar, 1999].

Stack allocation pass assigns all local stack variables into a single memory region at a function's entry and frees those regions after their exit. All arrays are removed at the end of this compiler pass, and memory operations are introduced. The correctness of the pass ensures that a program can only be executed correctly and safely if there are enough memory regions to be allocated to the stack variables. This phase also marks the end of the middle IR, its syntax is the same as that of the source level, but it introduces the notion of stacks.

Next comes the *linearization* pass that transforms the program into an unstructured list of instructions where high-level control structures are translated to instructions with goto and labels.

The last pass is *assembly generation*; it generates assembly programs and enforces architecture-level constraints. Finally, the assembly can be pretty-printed using *pretty printer* that can be used by any assembler or inlined into programs written in other languages like C, Rust, etc.

Discussion on trusted base of the Jasmin framework A few passes like typing, parsing, parameter expansion, and assembly pretty-printer are trusted base and is not formally verified in Coq. The extraction from the Jasmin source program to the EasyCrypt program is also a trusted base. The safety checker is also a trusted base. The source-level cost analysis later explained in this thesis involves using the safety checker to perform the static analysis on the source code, which again falls in the category of the trusted base.

Discussion The significant contribution of this thesis is to provide mitigation against timing-based side-channel attacks. This kind of attack is mostly seen in the domain of Cryptography, which is also one of the applications of the findings of this thesis. Though cryptography may represent a small part of the software world in general, it is one of the critical parts. All methodology, evaluation, and development are done on the Jasmin framework.

1.4 Contributions of this thesis

This section describes the contribution of this thesis, chapter by chapter, stating related publications. This thesis aims to produce a formally-verified secure compiler against timing-based side-channel attacks.

Chapter 2 Chapter 2 presents a formally-verified secure Jasmin framework for writing sequential constant-time code and producing constant-time assembly. The Jasmin compiler is formally verified using Coq proof assistant to preserve the constant-time property till the end. The semantics of the Jasmin language and compiler are instrumented to produce observable behavior (called “leakages") and functions to transform the source-level leakages to the target-level leakages (called “leakage transformers"). A novel design of data structure resembling the leakages closely aligned with the operational semantics of programs and a novel approach of explicitly producing the leakage transformers during the compilation phases helped in reasoning about other non-functional properties like the program cost described in Chapter 4.

My contribution: I designed the structured notion of leakages and leakage transformers. I instrumented the Jasmin compiler with leakages and leakage transformers and formally verified that the Jasmin compiler preserves constant-time property. Hence, it sums up that I did all the work. Paper writing was a joint effort.

Published in ACM SIGSAC Conference on Computer and Communications Security (CCS), 2021

Chapter 3 Chapter 3 presents a fine-grained version of the leakage model discussed in Chapter 2. The leakage model in Chapter 2 provides mitigation against timing-based attacks where the operators are assumed to be constant-time irrespective of their actual behavior in various architectures. And also, in terms of memory accesses, the complete address is leaked instead of the cache line of the address being accessed. The fine-grained

leakage model makes the analysis challenging and error-prone because of the modular arithmetic reasoning required for proving the equality of leakages. Chapter 3 provides a mechanized proof in Coq proof assistant that the Jasmin compiler preserves a class of fine-grained constant-time policies. This class extends the baseline constant-time policies discussed in Chapter 2 to provide mitigation against the timing-based attacks caused due to time-variable instructions and cache-line conflicts.

My contribution: I extended the baseline constant-time preservation method to reason about fine-grained constant-time policies. I also contributed to patching the extraction mechanism from Jasmin to EasyCrypt to incorporate reasoning about fine-grained constant-time policies. Paper writing was a joint effort.

Published in ACM SIGSAC Conference on Computer and Communications Security (CCS), 2022

Chapter 4 Chapter 4 presents a sound-certified algorithm to transform the source-level cost analysis of a program to the target level. It is a byproduct of the novel approach taken to prove the preservation of constant-time property discussed in chapter 2.

My contribution: I contributed to designing the function computing the cost at source and target levels and deducing the cost transformers from leakage transformers. Paper writing was a joint effort.

Published in ACM SIGSAC Conference on Computer and Communications Security (CCS), 2021

Chapter 5 The work explained in Chapter 2 and Chapter 3 guarantee protection against timing attacks if the execution is sequential. But in reality, we want to provide security guarantees where processors speculate. Chapter 5 explains a sound, formally verified verification method (paper-based proofs) for speculative constant-time. It defines a framework to analyze Jasmin's programs in the speculative context and establishes that reasoning about semantics that terminates during misspeculation is enough to reason about semantics that backtracks. This drastically reduces the number of execution paths to be considered.

My contribution: I presented the formal notion of speculative execution (with and without backtracking). I also provided the formal proofs for all the claims made in the chapter. Paper writing was a joint effort.

Published in IEEE Symposium on Security and Privacy (S&P), 2021.

Chapter 6 Chapter 6 discusses an information flow-based security type system approach that guarantees protection against timing-based side-channel attacks. The type system tracks security levels, ensuring no secret-dependent branching or secret-dependent memory accesses for a well-typed program. It also provides the programmer with extra primitives that can help them write more efficient constant-time programs compared to the work explained in Chapter 5 and includes soundness proof of the type system (done using Coq). This approach is carried out for a toy language and later extended in Chapter 7 for Jasmin.

My contribution: All the work in this chapter was completely done by me.

Chapter 7 Chapter 5 provides protection against Spectre attacks but suffers from performance overhead and has some limitations, like the requirement of memory safety. Chapter 7 explains how Jasmin is extended with new primitives that the programmer

can use to add protections against Spectre v1 attacks with significantly less performance overhead. It allows the programmer to add targeted protections like Speculative Load Hardening and Selective Load Hardening, which minimizes the performance overhead. An information flow-based security type system, as explained in Chapter 6, is extended for Jasmin, and it guarantees the correct usage of these primitives.

My contribution: I contributed to extending the idea from Chapter 6 for Jasmin. I also worked on the paper-based formal proofs for proving the soundness of the type system and the correctness of the new primitives. Paper writing was a joint effort.

Published in IEEE Symposium on Security and Privacy (S&P), 2023.

Chapter 2

Enforcing constant-time policies

2.1 Introduction

Timing-based side-channel attacks are a class of side-channel attacks that is a primary source of security vulnerabilities in cryptographic implementations as discussed in Section 1.1 of Chapter 1. An approach to minimize these attacks is ensuring that observable behavior during a program's execution does not depend on secrets, using an idealized leakage model. A leakage model defines a language describing the visible effects produced during a program's execution. Many cryptographic libraries adopt this approach under the generic umbrella of constant-time cryptography. For instance, a memory access ([i]) at an address i leaks the accessed memory address (i). A conditional instruction if e then c_1 else c_2 produces the leakage that includes the visible effects produced during the evaluation of the condition (e), the value of the condition itself, and the visible effects produced during the evaluation of then (c_1) or else (c_2) branch. To ensure constant-time property, a program's memory access should not depend on the secrets, and conditional instruction should not branch on secrets. Hence, the leakage model ensures that the memory address is always produced as visible effects during the execution of the memory access, and the condition is always produced as visible effects during the execution of conditional instruction. Having the address as leakage during the execution of memory access means the address is ensured to be public. Having the condition as part of the leakage produced during the execution of conditional instruction implies the condition is ensured to be public.

Modern compilers are designed to transform the source-level program into the target-level program and also to carry out aggressive program optimization while respecting the input-output behavior of programs. In simpler settings, where behaviors are modeled as execution traces, compiler correctness is thus stated as inclusion between the set of traces of the target program and the set of traces of source programs. As discussed in Section 1.2.1, this approach needs to be revised in a security context, as the inclusion of instrumented traces fails for most common compiler optimization. To address these shortcomings, the researchers have developed the foundations of secure compilation, where compilers are required to preserve the functional behavior and security properties like constant-timeness. Certified compilers come with machine-checkable proof that the compiler is correct, i.e., preserve the behavior of programs. The statement of compiler correctness relies on operational semantics, which formalizes the execution of source and assembly programs. The semantics are expressed as judgments of the form: $p : s \Downarrow s'$ (resp. $\overline{p} : s \Downarrow s'$), stating that execution of source program p (resp. target program $\overline{p}$)

on initial state s terminates with final state s'. Compiler correctness is informally stated using this notation: for all source programs p with the compilation $\overline{p}$, and all states s and s'.

$$p : s \Downarrow s' \implies \overline{p} : s \Downarrow s'.$$

The formal notion above assumes that the source and target programs operate over the same state space. But in reality, the source and target programs operate over different state spaces and is considered while developing these kinds of theorems in Coq. The Coq development for this work also considers different state spaces concerning the source and target programs.

Security properties like constant-time are expressed relative to instrumented semantics, which tracks visible effects of program execution (called leakages). The instrumented semantics is based on the leakage model describing what is leaked during program execution, leading to a judgment of the form: $p : s \Downarrow_\ell s'$, stating that executing program p on initial state s yields a final state s' and leaks ℓ.

2.1.1 Possible design decisions for leakage model

There is various methodology to design a leakage model.

- Sequence of atomic leakages: One kind is where the leakages of a program P are defined as a sequence of atomic leakages where each leakage is generated by single-step execution of the program. For example, an expression of the form $(0 + \mathsf{e}) + 1$ produces leakage of the form $[] +\!\!+ \ell_\mathsf{e} +\!\!+ []$ that result to ℓ_e because concatenation with empty sequence results in the original sequence.

- Set of atomic leakages: One kind is where a program P leakages are defined as a set of atomic leakages. For example, an expression of the form $(0 + \mathsf{e}) + 1$ produces leakage of the form $\{\bullet; \ell_\mathsf{e}; \bullet\}$.

- Structured leakage: The word structured resembles the fact that the syntax of leakage is closely related to the operational semantics of the programs. The expression $(0 + \mathsf{e}) + 1$ produces leakage of the form $((\bullet, \ell_e), \bullet)$.

  Discussion: Different notions of leakage models have various advantages and disadvantages over others.

  - Sequence of atomic leakages: In the case of leakages defined as a sequence of atomic leakages, it is hard to identify the leakages belonging to the sub-components of an expression or an instruction from the flattened sequence. For example, an expression of the form $(0 + \mathsf{e}) + 1$ produces the final leakage as ℓ_e because concatenation with empty sequence results in the original sequence. The overall goal is to transform the source leakage into the target leakage using some function, and a flattened sequence makes it hard to construct such a function. But also, there is a possibility of using all the existing libraries defined for the *sequence* data structure in Coq, and there is no need to redefine the various functions operating on the sequence.
  - Set of atomic leakages: In the case of leakages defined as a set of atomic leakages, it is hard to identify the leakages belonging to the sub-components

of an expression or an instruction from the set because a set is an unordered collection of elements. Again, it will be hard to define the transformation functions. But also, from the usability point of view, a set data structure is more powerful as many libraries in Coq defined for sets exist.

– Structured leakage: Structured notation of leakages presents a notion closely related to the syntax and semantics of the instructions. By looking at the leakage produced during the program execution, we can extract a lot of information about the operational semantics of the program. Here, in the case of leakage $((\bullet, \ell_e), \bullet)$, $\bullet$ resembles that the expression's execution produces empty leakage; hence, the expression is either a constant or a variable. Hence, the intuitive nature of the leakage helps us to reason about the program without looking at the actual program. Also, it is easier to design a set of transformation functions. For example, the compiler passes like constant-propagation will transform the expression $(\mathsf{0} + \mathsf{e}) + \mathsf{1}$ to $\mathsf{e} + \mathsf{1}$. The transformation function should remove the leakage corresponding to 0. The notion of leakage is so intuitive that designing a function that eliminates $\bullet$ (leakage corresponding to 0) is very straightforward.

But in the end, the choice of the leakage model depends on the purpose for which it is used. In this work, the structured notion is used because it helped in reasoning about various properties in a structured manner.

2.1.2 Constant-time

The constant-time (CT) is an instance of observational non-interference that is formalized as a leakage model such that the control-flow instructions leak the condition and memory accessing instructions leak the address being accessed. The constant-time property is based on the indistinguishability relation, stating that the indistinguishable states only differ in their private parts. The constant-time property for a program p is stated as: for all initial state s_1 and s_2,

$$\left.\begin{array}{l} p : s_1 \Downarrow_{\ell_1} s_1' \\ p : s_2 \Downarrow_{\ell_2} s_2' \end{array}\right\} \implies s_1 \sim s_2 \implies \ell_1 = \ell_2.$$

Under this formalization, preservation of constant-time for a program p with compilation $\overline{p}$ is stated as: for all initial states s_1 and s_2,

$$\left.\begin{array}{l} p : s_1 \Downarrow_{\ell_1} s_1' \\ p : s_2 \Downarrow_{\ell_2} s_2' \\ \overline{p} : s_1 \Downarrow_{\overline{\ell}_1} s_1' \\ \overline{p} : s_2 \Downarrow_{\overline{\ell}_2} s_2' \end{array}\right\} \implies s_1 \sim s_2 \implies \ell_1 = \ell_2 \implies \overline{\ell}_1 = \overline{\ell}_2.$$

2.1.3 Function to transform source leakage to target leakage

As seen in Section 2.1.2, preserving constant-time requires proving equality between the target leakages. A compiler consists of various passes that transform the source to the target program. For most compiler passes, source and target programs have different leakages because the source and target programs differ. The various compiler passes

introduce, remove, or reorder instructions according to the compiled program. Hence, the target leakage should also correspond to these transformations.

There is a need for a transformation function that transforms the source leakage into the target leakage. The compiler correctness should also ensure that this transformation function converts the source leakage correctly. There are various ways to exhibit such function F. One approach is to say that there exists such function F and while doing the correctness proof of the compiler, an instance of such a function is provided.

$$\forall s\ s'\ \ell, p : s \Downarrow_\ell s' \implies \exists\ F, \overline{p} : s \Downarrow_{F(\ell)} s'$$

Another approach is that the compiler produces the function F.

$$\forall s\ s'\ \ell\ F, p : s \Downarrow_\ell s' \implies \overline{p} : s \Downarrow_{F(\ell)} s'$$

Discussion Both methods have their benefits; for example, the first approach does not affect the compilation time. As the compiler does not do any extra computation to produce the transformation function F, compilation time does not change. The second approach is more applicable as the function generated by the compiler can be used to reason about other properties and makes reasoning about constant-time property much more straightforward. The transformation function gives an intuition about the changes made to the source leakage to produce the target leakage. Hence, it provides a recipe to transform the source-level information into target-level information. This recipe can be used to reason about other properties of the program.

Focus on preservation of constant-time considering both approaches To preserve constant-time property, we must deal with various compiler passes that transform the source into the target program. When the compiler is not obliged to produce the transformation function F, there is no information (even at the abstract level) about how the program is transformed according to that particular compiler pass.

Carrying out the proof of preservation becomes harder in such cases, as discussed in [Barthe et al., 2020] and [Barthe et al., 2018]. The compiler pass that preserves the trace and leakage can easily be extended to include the reasoning about the preservation of constant time, as the leakage is the same at the source and target level. The compiler pass-like dead-code elimination that might erase some leakage at the target level needs some extra attention because there is a need to establish a relationship that predicts the number of steps at the target level as shown in Figure 2.1a. The proof of preservation for such passes needs to establish judgments about the correctness of this relationship. But when the compiler produces the leakage transformer, the transformer gives information about how the program is transformed. For example, the source-level instruction $i; c$ is transformed to c during the dead-code elimination. The dead-code elimination compiler pass produces the leakage transformer of the form $remove; id$ that gives information that the leakage corresponding to i is removed and the leakage associated with c is preserved. There is no need to establish a relation between the source and target states; the assumption that the source-level program is constant-time will show the constant-time property after the dead-code elimination at the target level. As explained in [Barthe et al., 2020], they need to establish a cube for reasoning about the constant-time property for compiler pass-like linearization. The cube diagram presented in Figure 2.1b shows a need to develop a counting simulation between source and target states, and reasoning about four traces was required to prove the preservation of constant time.

Overall, the disadvantage of not explicitly producing the transformation function as a byproduct of compilation makes the proof harder as it can only use the correctness proof of the compiler, but it cannot be extended easily to reason about the number of possible steps at the target level. For example, in the case of linearization, it is more difficult as the high-level control flow is transformed to linear instructions full of gotos and labels, and it makes it harder to reason about the number of steps at the target level in relation with the source; hence the cube diagram (present in Figure 2.1b) is necessary. In the case where the compiler pass explicitly produces the transformation function, the transformation function gives an abstract overview of the transformation. Also, the proof methodology varies for different compiler passes; hence, the preservation of constant-time needs to be done separately for each compiler pass as one proof methodology is not enough for reasoning about all compiler passes, which is unlikely the case where the compiler explicitly produces the transformation function.

(a) Leakage Preservation/Erasure (b) CT cube diagram

Figure 2.1 – CT simulation diagrams.

In this work, function F is generated by the compiler and is called a leakage transformer. The proof of preservation of constant-time becomes straightforward and only involves reasoning about the two traces at the target level (assuming that the source program is constant-time). It did not require generating a relationship between the source and target states as the leakage transformer captures the information about the transformation, and there is no need to establish the connection between the number of steps at the source and target. The proof is a consequence of the correctness proof of the leakage transformers (later explained in Section 2.5).

2.1.4 Contributions

This chapter proposes a methodology for formally verifying the preservation of constant-time property. All the work is being carried out around the Jasmin compiler, and all the proofs are machine-checked using theorem prover Coq. The development is present here: https://github.com/jasmin-lang/jasmin/tree/constant-time. Technical contributions in nut-shell:

- The definitions of structured leakage and leakage transformers;
- Formal proofs of correctness of leakage transformers for all the passes of Jasmin compiler;
- A mechanized proof showing that the Jasmin compiler preserves Cryptographic constant-time

2.1.5 Illustrative example

This section presents a few examples to give an intuition about structured leakages and leakage transformers.

Expressions Figure 2.2 introduces two code snippets representing the source and target code for addition operations and their associated leakages. Figure 2.2 also presents the leakage transformer produced during this transformation. The first addition operation adds 0 to the value present at index 0 in the array a, and the second addition adds 1 to the result obtained from the first addition. The compiler knows statically that the result of the first operation will be $a[0]$; hence, the target code is just one addition operation with operands $a[0]$ and 1. The leakage for $(0 + a[0]) + 1$ is $((\bullet, (\bullet, [0])), \bullet)$ representing that evaluation of a constant produces no leakage, and array access leaks the index accessed. The leakage for the compiled expression $a[0] + 1$ is $((\bullet, [0]), \bullet)$. The compiler produces leakage transformer (π_2, id) where π_2 projects the leakage at index 2 from the source leakage $(\bullet, (\bullet, [0]))$ and id preserves the leakage. If the leakage produced by the addition operation was represented as a concatenation of its subpart's leakages, it would be difficult to project at the corresponding index, as concatenating with an empty leakage returns the original leakage. The flattened source leakage of the above example will be the concatenation of $\bullet$, $\bullet$, $[0]$ and $\bullet$, which will get reduced to $[0]$. From the flattened list, it is hard to identify the leakages belonging to the sub-parts.

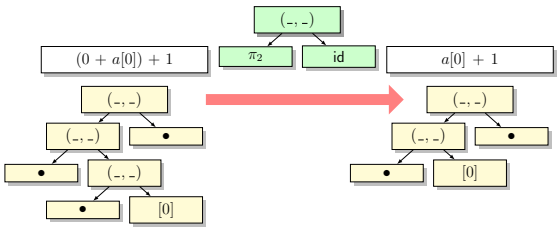


Figure 2.2 – Example: Structured leakage and leakage transformer for expression

Instructions Figure 2.3 presents a conditional instruction with a $\#t$ guard, that is reduced to its then branch after compilation. This kind of transformation is carried out when the compiler statically knows the value of the boolean condition. The leakage for conditional instruction at the source level is $\mathsf{if}_{\#t}(\bullet, \mathsf{op}_{\mathsf{l}}((\bullet, [0]); \bullet))$. The structure of the source leakage is closely aligned with the structure of the conditional instruction, with $\#t$ indicating that the boolean condition is satisfied and $\mathsf{op}_{\mathsf{l}}((\bullet, [0]); \bullet)$ (op_{l} is represented as $:=$ in the figure for better readability) indicating that the then branch is an assignment instruction. The target leakage is $\mathsf{op}_{\mathsf{l}}((\bullet, [0]); \bullet)$, which gives us information that the conditional instruction is reduced to an assignment instruction. The leakage transformer produced during this transformation is $\mathsf{ceval}_{\#t}\ (\mathsf{op}(\mathsf{id}, \mathsf{id}))$, where $\#t$ indicates the branch taken and $\mathsf{op}(\mathsf{id}, \mathsf{id})$ transforms the leakage for the then branch from the source. Designing the leakage transformer was straightforward because of the structured notion of leakages. If the leakage generated from conditional instruction were a concatenation of its sub-parts, then it would be hard to detect which part in the list belongs to the then or else branch.

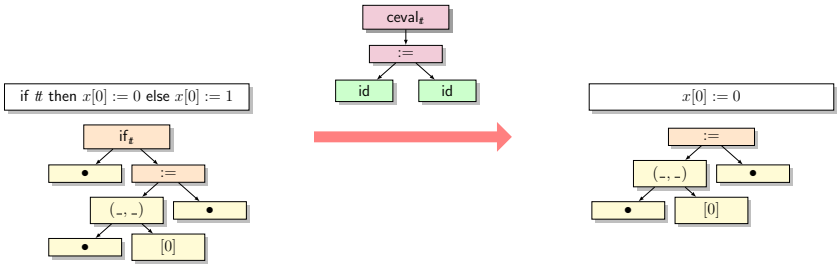


Figure 2.3 – Example: Structured leakage and leakage transformer for instruction

2.2 Stuctured leakage and instrumented semantics

This section presents the formal notion of structured leakages and how they are used in instrumented operational semantics of the Jasmin language shown in Figure 1.11.

2.2.1 Instrumented semantics

The instrumented semantics is produced from the original semantics by annotating the judgments with leakage. The semantic uses three judgments. The first, $e \downarrow^s_{\ell_e} v$, provides the semantic of the expression e in the state s, it produces a leakage ℓ_e and a value v. The second, $d := v \downarrow^s_{\ell_e} s'$ provides semantics of assigning a value v to a destination d in a state s; it generates a new state s' and some leakage ℓ_e. The last judgment, $i, s \Downarrow_\ell s'$, provides semantics of instructions; it takes an instruction i and a state s and returns an instruction's leakage ℓ and a state s'. The three judgments are presented in Figure 2.4.

A state is a pair of a memory (a mapping from address to values) and a valuation for variables (a mapping from variables to values). $s(x)$ is the value associated to x in s (which can be an array). $s[p]$ loads the value stored in memory at an address p. When a value v is an array, $v[i]$ denotes the value at index i in this array. $s\{x \leftarrow v\}$ updates the value associated to x with v and $s\{p \leftarrow v\}$ writes v in memory at address p.

2.2.2 Structured leakage

The leakage model is an abstract mechanism to capture visible effects produced during a program's execution. To enforce the preservation of constant-time policies, one important goal is to relate source programs' leakage and their compilation. There are four kinds of leakages: two for the high-level Jasmin language: ℓ_e and ℓ, one for the intermediate-level ℓ_i and one for the assembly-level ℓ_a.

Structured leakage for the source level

The syntax of leakage is closely related to programs' syntax and semantics. Figure 2.5 presents the leakages for expressions and instructions. In the case of expressions, ℓ_e can be $\bullet$, an array index $[z]$, a memory address $*p$ or a tuple of leakage $(\ell^1_e, \ldots, \ell^n_e)$. In the case of instructions, there is one constructor per semantic rule.

Expression semantics:

$$\frac{}{c \downarrow^s_\bullet c} \qquad \frac{}{b \downarrow^s_\bullet b} \qquad \frac{}{a[n] \downarrow^s_\bullet a[n]} \qquad \frac{v = s(x)}{x \downarrow^s_\bullet v} \qquad \frac{v = gd(g)}{g \downarrow^s_\bullet v}$$

$$\frac{e \downarrow^s_{\ell_e} z \quad s(a) = t}{a[e] \downarrow^s_{(\ell_e,[\mathsf{z}])} t[z]} \qquad \frac{e \downarrow^s_{\ell_e} p}{*e \downarrow^s_{(\ell_e,*\mathsf{p})} s[p]}$$

$$\frac{e_i \downarrow^s_{\ell^i_e} v_i \quad v = \mathsf{op}(v_1, \ldots, v_n)}{\mathsf{op}(e_1, \ldots, e_n) \downarrow^s_{(\ell^1_e,\ldots,\ell^n_e)} v} \qquad \frac{e \downarrow^s_{\ell_e} b \quad e_t \downarrow^s_{\ell^{tt}_e} v_{tt} \quad e \downarrow^s_{\ell^{ff}_e} v_{ff}}{\mathsf{if}\ e\ \mathsf{then}\ e_{tt}\ \mathsf{else}\ e_{ff} \downarrow^s_{(\ell_e,\ell^{tt}_e,\ell^{ff}_e)} \mathsf{if}\ b\ \mathsf{then}\ v_{tt}\ \mathsf{else}\ v_{ff}}$$

Assignment semantics:

$$\frac{}{x := v \downarrow^s_\bullet s\{x \leftarrow v\}}$$

$$\frac{e \downarrow^s_{\ell_e} z \quad s(a) = t \quad t' = t\{z \leftarrow v\}}{a[e] := v \downarrow^s_{(\ell_e,[\mathsf{z}])} s\{a \leftarrow t'\}} \qquad \frac{e \downarrow^s_{\ell_e} p}{*e := v \downarrow^s_{(\ell_e,*\mathsf{p})} s\{p \leftarrow v\}}$$

Instruction semantics:

$$\frac{}{\{\} : s \Downarrow_{\{\}} s} \qquad \frac{i : s \Downarrow_{\ell_i} s_1 \quad \{c\} : s_1 \Downarrow_{\{\ell_c\}} s_2}{\{i; c\} : s \Downarrow_{\{\ell_i;\ell_c\}} s_2}$$

$$\frac{e \downarrow^s_{\ell_e} v \quad d := v \downarrow^s_{\ell_d} s'}{d := e : s \Downarrow_{\mathsf{op_l}(\ell_d;\ell_e)} s'}$$

$$\frac{e \downarrow^s_{\ell_e} b \quad c_b : s \Downarrow_{\ell_c} s'}{\mathsf{if}\ e\ \mathsf{then}\ c_{tt}\ \mathsf{else}\ c_{ff} : s \Downarrow_{\mathsf{if}_b(\ell_e,\ell_c)} s'} \qquad \frac{c, s \Downarrow_{\ell_c} s_1 \quad e \downarrow^{s_1}_{\ell_e} ff}{\mathsf{while}\ c\ e\ c' : s \Downarrow_{\mathsf{while_f}(\ell_c,\ell_e)} s}$$

$$\frac{c, s \Downarrow_{\ell_c} s_1 \quad e \downarrow^{s_1}_{\ell_e} tt \quad c', s_1 \Downarrow_{\ell_{c'}} s_2 \quad \mathsf{while}\ c\ e\ c' : s_2 \Downarrow_{\ell_w} s_3}{\mathsf{while}\ c\ e\ c' : s \Downarrow_{\mathsf{while_t}(\ell_c,\ell_e,\ell'_c,\ell_w)} s_3}$$

$$\frac{r \downarrow^s_{\ell_r} wr \quad \mathsf{for_{sem}}\ i\ wr\ c : s \Downarrow_{\ell_f} s_1}{\mathsf{for}\ i\ r\ c : s \Downarrow_{\mathsf{for}\ \ell_r\ \ell_f} s_1}$$

$$\frac{args \downarrow^s_{\ell_e;\ldots;\ell_e} vs \quad \mathsf{call_{sem}}\ fn\ vs : s \Downarrow_{\ell_f} s_1 \quad xs := vs \downarrow^{s_1}_{\ell_w;\ldots;\ell_w} s_2}{\mathsf{call}\ ii\ xs\ fn\ args : s \Downarrow_{\mathsf{call}\ (\ell_e;\ldots;\ell_e)\ \ell_f\ (\ell_w;\ldots;\ell_w)} s_2}$$

For semantics:

$$\frac{}{\mathsf{for_{sem}}\ i\ \{\}\ c : s \Downarrow_{\{\}} s} \qquad \frac{s\{i \leftarrow w\} = s_1 \quad c, s_1 \downarrow_{\ell_c} s_2 \quad \mathsf{for_{sem}}\ i\ ws\ c \downarrow_{\ell_w} s_3}{\mathsf{for_{sem}}\ i\ (w :: ws)\ c : s \Downarrow_{(\ell_c,\ell_w)} s_3}$$

Call semantics:

$$\frac{fd(P, fn) = f \quad s\{fparams(f) \leftarrow args\} = s_1 \quad fbody(f), s_1 \Downarrow_{\ell_c} s_2}{\mathsf{call_{sem}}\ fn\ args : s \Downarrow_{(fn,\ell_c)} s_2}$$

Figure 2.4 – Instrumented semantics.

$\ell_e ::=$	$\bullet$	empty	$\ell ::=$	$\mathsf{op}_\mathsf{l}(\ell_e;\ell_e)$	assignment
$\mid$	$[z]$	index	$\mid$	$\mathsf{if}_b(\ell_e,\ell)$	conditional
$\mid$	$*p$	address	$\mid$	$\mathsf{while}_\mathsf{t}(\ell,\ell_e,\ell,\ell)$	iteration
$\mid$	$(\ell_e,\ldots,\ell_e)$	sub-leakage	$\mid$	$\mathsf{while}_\mathsf{f}(\ell,\ell_e)$	loop end
			$\mid$	for ℓ_e $\{\{\ell;\ldots;\ell\};\ldots;\{\ell;\ldots;\ell\}\}$	for
			$\mid$	call $(\ell_e,\ldots,\ell_e)$ $(fn,\{\ell;\ldots;\ell\})$ $(\ell_e,\ldots,\ell_e)$	call
			$\mid$	$\{\ell;\ldots;\ell\}$	sequence

Figure 2.5 – Syntax of structured leakages

The instrumented semantics of expressions is presented in Figure 2.4. Variables, constants, booleans, array initialization, and global variables leak $\bullet$, i.e., a mark indicating that a variable has been evaluated. The evaluation of an array access $a[e]$ leaks a pair $(\ell_e,[z])$ where ℓ_e is the leakage corresponding to the evaluation of the index e and z is the value of e. The evaluation of memory access $[e]$ leaks a pair $(\ell_e,*p)$ where ℓ_e is the leakage corresponding to the evaluation of the address e and p is the value of e. Operators leak the tuple composed by the leakages of their arguments. The evaluation of the conditional expression $\mathsf{if}\ e\ \mathsf{then}\ e_{tt}\ \mathsf{else}\ e_{ff}$ leaks a sequence consisting of ℓ_e, ℓ_e^{tt} and ℓ_e^{ff} where ℓ_e is the leakage corresponding to the evaluation of the boolean expression e, ℓ_e^{tt} is the leakage corresponding to the evaluation of the then branch and ℓ_e^{ff} is the leakage corresponding to the evaluation of the else branch.

A destination can be either a variable, an array, or a memory destination. The semantic of assignment also generates leakages due to memory and array stores. When the destination is a variable, the leakage is $\bullet$. In case of an array destination $a[e]$, $(\ell_e,[z])$ is leaked where ℓ_e is the leakage corresponding to the evaluation of the index e and z is the value of e. Similarly, in the case of memory destination $[e]$, $(\ell_e,*p)$ is leaked where ℓ_e is the leakage corresponding to the evaluation of the address e and p is the value of e.

Except for the leakage, the non-instrumented rules are mostly standard; hence, the discussion focuses on the leakage part. The leakage of a sequence is composed of the leakage of each of its components. An assignment instruction $d := e$ produces a leakage $\mathsf{op}_\mathsf{l}(\ell_e;\ell_d)$ composed of the leakage generated during the evaluation of the expression e and the one generated during the evaluation of the assignment $d := v$ where v is the value obtained after the evaluation of e. For conditional $\mathsf{if}\ e\ \mathsf{then}\ c_{tt}\ \mathsf{else}\ c_{ff}$, the leakage is $\mathsf{if}_b(\ell_e,\ell_{c_b})$, so it contains the leakage ℓ_e generated by the evaluation of the condition e, the value b of the condition and the leakage ℓ_{c_b} generated by the evaluation of the taken branch. In Jasmin, the while loop is represented as $\mathsf{while}\ c\ e\ c'$ and works like a do-while loop in C language. If the condition evaluates to false, the loop exits and the leakage is $\mathsf{while}_\mathsf{f}(\ell_c,\ell_e)$ where ℓ_c is the leakage generated during the execution of c, and ℓ_e is the leakage generated during the evaluation of the condition. Otherwise the leakage is $\mathsf{while}_\mathsf{t}(\ell_c,\ell_e,\ell'_c,\ell_w)$, where ℓ_c is the leakage generated during the execution of c, ℓ_e is the leakage generated by the boolean condition, ℓ'_c is the leakage generated during the evaluation of c', and ℓ_w is the leakage obtained during the loop iteration. A for-loop $\mathsf{for}\ i\ r\ c$ produces a leakage of the form $\mathsf{for}\ \ell_r\ \ell_f$ where the leakage ℓ_r represents the leakage generated during the evaluation of the range of the loop and ℓ_f is obtained during the iteration of the for loop's body. A function call $\mathsf{call}\ ii\ xs\ fn\ args$ produces a leakage of the form $\mathsf{call}\ \ell_e\ \ell_f\ \ell_w$ composed of the leakage ℓ_e generated during the evaluation of the arguments $(args)$ passed to the function call, ℓ_f generated during the evaluation of the

function body and ℓ_w generated while assigning the results to corresponding destinations ($xs := [\![args]\!]_\rho$). The instrumented semantics are deterministic, both with respect to states and with respect to leakages.

Structured leakage for the intermediate level

The intermediate language consists of five kinds of instructions. $\mathsf{goto}\ n$ jumps to the location labeled as n. $\mathsf{label}\ n$ creates a label name n and align aligns the instruction. $\mathsf{op_i}(xs, o, es)$ represents the intermediate-level operations, and $\mathsf{if_i}(e, n)$ represents the intermediate-level conditional instruction where e represents the guard, and n represents the label of the instruction that will be executed when the guard evaluates to tt.

$$
\begin{array}{lll}
i_l ::= & \mathsf{align} & \text{align} \\
| & \mathsf{goto}\ n & \text{goto} \\
| & \mathsf{label}\ n & \text{label} \\
| & \mathsf{op_i}(xs, o, es) & \text{operation} \\
| & \mathsf{if_i}(e, n) & \text{cond}
\end{array}
\qquad
\begin{array}{lll}
\ell_i ::= & \bullet & \text{empty} \\
| & i & \text{pc} \\
| & \mathsf{op}\ \ell_e & \text{operation} \\
| & \mathsf{if_l}\ i\ \ell_e\ b & \text{conditional}
\end{array}
$$

Figure 2.6 – Syntax of intermediate language and leakages

Auxiliary functions:

$$\langle \rho, \mu, ic, n \rangle_{\mathsf{mem}} = \mu \qquad \langle \rho, \mu, ic, n \rangle_{\mathsf{rmap}} = \rho$$

$$\langle \rho, \mu, ic, n \rangle_{\mathsf{cmd}} = ic \qquad \langle \rho, \mu, ic, n \rangle_{\mathsf{pc}} = n$$

$$\mathsf{set_{pc}}(s, n) = \langle s_{\mathsf{mem}}, s_{\mathsf{rmap}}, s_{\mathsf{cmd}}, n \rangle$$

$$\mathsf{is_{lbl}}(n, i) = \begin{cases} true & i = \mathsf{label}\ \ \mathrm{n}' \ \wedge\ n' = n \\ false & i \neq \mathsf{label}\ \ \mathrm{n}' \end{cases}$$

$$\mathsf{find_{lbl}}(n, ic) = \begin{cases} ok\ idx & \exists idx : [ic]_{idx} = i \wedge \mathsf{is_{lbl}}(n, i) = true \wedge idx \leq |ic| \\ error & idx > |ic| \end{cases}$$

$$\mathsf{to_s}\langle \rho, \mu, ic, n \rangle = \langle \rho, \mu \rangle \qquad \mathsf{of_s}(\langle \rho, \mu \rangle, ic, n) = \langle \rho, \mu, ic, n \rangle$$

Figure 2.7 – Auxiliary functions used in intermediate level semantics

The high-level instruction is compiled to the intermediate-level instruction during the linearization compiler pass that transforms the program into an unstructured list of instructions. The linearization pass is explained in detail in Section 2.3.3. The instrumented intermediate-level semantics is produced from the original intermediate-level semantics by annotating the judgments with intermediate leakage ℓ_i present in the right side of Figure 2.6. The intermediate-level semantic uses the judgment of the form: $s_i \Downarrow_{\ell_i} s_i'$. The

intermediate state comprises a register map, memory, intermediate-level command (seq of intermediate-level instructions), and program counter. A set of auxiliary functions defined in Figure 2.7 are used in the instrumented semantics. For example, $\mathsf{find}_{\mathsf{lbl}}(n, s_{cmd})$ returns the position of instruction label n in a sequence of intermediate-level instructions s_{cmd}. $\mathsf{set}_{\mathsf{pc}}(s, pc')$ sets the program counter field of the state s to pc'. The function $\mathsf{is}_{\mathsf{lbl}}(n, i)$ checks if the instruction i is a label instruction and if it is equal to n. The function to_s and of_s define the transformation of the intermediate-level state to the source-level state and vice-versa. The intermediate-level instrumented semantics is present in Figure 2.8.

Intermediate-level instruction semantics:

$$\frac{s_{\mathsf{pc}} = pc}{\mathsf{align} : s \Downarrow_{\bullet} \mathsf{set}_{\mathsf{pc}}(s, pc+1)}\ [\textsc{Align}] \qquad \frac{s_{\mathsf{pc}} = pc}{\mathsf{label}\ : s \Downarrow_{\bullet} \mathsf{set}_{\mathsf{pc}}(s, pc+1)}\ [\textsc{Label}]$$

$$\frac{\mathsf{find}_{\mathsf{lbl}}(n, s_{\mathsf{cmd}}) = pc' \wedge s_{\mathsf{pc}} = pc}{\mathsf{goto}\ n : s \Downarrow_{(pc'+1)-pc} \mathsf{set}_{\mathsf{pc}}(s, pc'+1)}\ [\textsc{Goto}]$$

$$\frac{e \downarrow_{\ell_e}^{\mathsf{to_s}\, s}\ tt \wedge \mathsf{find}_{\mathsf{lbl}}(n, s_{\mathsf{cmd}}) = pc' \wedge s_{\mathsf{pc}} = pc}{\mathsf{if_i}(e, n) : s \Downarrow_{\mathsf{if_l}\ ((pc'+1)-pc)\ \ell_e\ tt} \mathsf{set}_{\mathsf{pc}}(s, pc'+1)}\ [\textsc{CondT}]$$

$$\frac{e \downarrow_{\ell_e}^{\mathsf{to_s}\, s}\ ff \wedge s_{\mathsf{pc}} = pc}{\mathsf{if_i}(e, n) : s \Downarrow_{\mathsf{if_l}\ 1\ \ell_e\ ff} \mathsf{set}_{\mathsf{pc}}(s, pc+1)}\ [\textsc{CondF}]$$

$$\frac{xs := o(es) \downarrow_{\ell_o}^{\mathsf{to_s}\, s}\ s' \wedge s_{\mathsf{pc}} = pc}{\mathsf{op_i}(xs, o, es) : s \Downarrow_{\mathsf{op}\ \ell_o} \mathsf{of_s}(s', s_{\mathsf{cmd}}, pc+1)}\ [\textsc{Op}]$$

Figure 2.8 – Instrumented semantics of intermediate language

Both align and label instruction increment the program counter by using the auxiliary function $\mathsf{set}_{\mathsf{pc}}(s, pc+1)$ (where s is the initial intermediate state and pc is the program counter stored in it) and does not leak anything. goto n finds the program counter of the instruction associated with the label n and sets the program counter to $\mathsf{pc}' + 1$ where pc' is the program counter of instruction with label n. Evaluation of goto n leaks $(\mathsf{pc}' + 1) - \mathsf{pc}$ where pc is the current program counter. $\mathsf{if_i}(e, n)$ sets the program counter to the instruction associated with label n when the guard e evaluates to tt else, it sets the program counter to the next instruction. It leaks $\mathsf{if_l}\ i\ \ell_e\ b$ where i is the difference between the new and old program counter, ℓ_e is the leakage obtained during the evaluation of the condition e, and also leaks the boolean (either tt or ff). $\mathsf{op_i}(xs, o, es)$ uses the high-level semantics of operations and produces the leakage ℓ_o that is produced during the evaluation of es and assigning it to xs.

Structured leakage for the assembly level

The assembly-level language consists of five different kinds of instructions presented in Figure 2.9. $\mathsf{label_a}\ n$ and $\mathsf{align_a}$ works in similar manner like intermediate-level instructions. jump n represents unconditional jump to the instruction with label n and $\mathsf{jcc}(n, cond)$ represents conditional jump to the instruction with label n. $\mathsf{asm_{op}}(o, es)$ represents the assembly-level operations.

$i_a ::=$	$\mathsf{align_a}$	align	$\ell_a ::=$	$\bullet$	empty
$\mid$	$\mathsf{jump}\ n$	jump	$\mid$	i	pc
$\mid$	$\mathsf{label_a}\ n$	label	$\mid$	$\mathsf{if_a}\ i\ b$	conditional
$\mid$	$\mathsf{jcc}(n, cond)$	jump on condition	$\mid$	$(*p, \ldots, *p)$	pointers
$\mid$	$\mathsf{asm_{op}}(o, es)$	assembly operation			

Figure 2.9 – Syntax of assembly-level instructions and leakages

The assembly-level instructions operate on a state consisting of assembly-level memory, a sequence of assembly-level instructions, and an instruction pointer. The state is of the form $\langle mem_a, ac, ip\rangle$ where mem_a is represented as $\{\rho, \mu, \rho_x, \rho_f\}$ (ρ_x maps the extra registers like mmx to their values and ρ_f maps the flag registers to their values). The state parameters are updated using the auxiliary functions defined in Figure 2.10, and their semantics are very similar to the auxiliary functions of the intermediate level.

Auxiliary functions:

$$\langle mem_a, ac, n\rangle_{\mathsf{ip}} = n \quad \langle mem_a, ac, n\rangle_{\mathsf{acmd}} = ac \quad \langle\{\rho, \mu, \rho_x, \rho_f\}, ac, n\rangle_{\mathsf{flag}} = \rho_f$$

$$\mathsf{set_{ip}}(s, n) = \langle mem_a, ac, n\rangle \quad \langle mem_a, ac, n\rangle_{\mathsf{mem}} = mem_a$$

$$\mathsf{is_{lbla}}(n, i) = \begin{cases} true & i = \mathsf{label_a}\ \mathrm{n}' \ \wedge\ n' = n \\ false & i \neq \mathsf{label_a}\ \mathrm{n}' \end{cases}$$

$$\mathsf{find_{lbla}}(n, ac) = \begin{cases} ok\ idx & \exists idx : [ac]_{idx} = i \wedge \mathsf{is_{lbla}}(n, i) = true \wedge idx \leq |ac| \\ error & idx > |ac| \end{cases}$$

Figure 2.10 – Auxiliary functions used in assembly level semantics

The original assembly level's semantics are instrumented to produce leakages of the form ℓ_a present in Figure 2.9. The ℓ_a can be $\bullet$, i (where i represents the instruction pointer that is a natural number), $\mathsf{if_a}\ i\ b$ (where i represents the instruction pointer, and b represents the evaluation of the guard of a conditional jump) and $(*p, \ldots, *p)$ represents a sequence of memory addresses. The judgment of the form: $s_a \Downarrow_{\ell_a} s'_a$ defines the instrumented semantics of assembly-level instructions present in the Figure 2.11.

Assembly-level instruction semantics:

$$\frac{s_{\mathsf{ip}} = ip}{\mathsf{align_a} : s \Downarrow_\bullet \mathsf{set_{ip}}(s, ip+1)}\ [\textsc{Align}] \qquad \frac{s_{\mathsf{ip}} = ip}{\mathsf{label_a} : s \Downarrow_\bullet \mathsf{set_{ip}}(s, ip+1)}\ [\textsc{Label}]$$

$$\frac{\mathsf{find_{lbl_a}}(n, s_{\mathsf{acmd}}) = ip' \wedge s_{\mathsf{ip}} = ip}{\mathsf{jump}\ n : s \Downarrow_{(ip'+1)-ip} \mathsf{set_{ip}}(s, ip'+1)}\ [\textsc{Jump}]$$

$$\frac{[cond]^{s_{\mathsf{flag}}} \downarrow tt \wedge \mathsf{find_{lbl_a}}(n, s_{\mathsf{pc}}) = ip' \wedge s_{\mathsf{pc}} = ip}{\mathsf{jcc}(n, cond) : s \Downarrow_{\mathsf{if_a}\ ((ip'+1)-ip)\ tt} \mathsf{set_{ip}}(s, ip'+1)}\ [\textsc{Jcct}]$$

$$\frac{[cond]^{s_{\mathsf{flag}}} \downarrow ff \wedge s_{\mathsf{pc}} = ip}{\mathsf{jcc}(n, cond) : s \Downarrow_{\mathsf{if_a}\ 1\ ff} \mathsf{set_{ip}}(s, ip+1)}\ [\textsc{Jccf}]$$

$$\frac{o(es) \downarrow^{s_{\mathsf{mem}}}_{(*p,\ldots,*p)} m' \wedge s_{\mathsf{pc}} = ip}{\mathsf{asm_{op}}(o, es) : s \Downarrow_{(*p,\ldots,*p)} \langle m', s_{\mathsf{acmd}}, ip+1\rangle}\ [\textsc{Op}]$$

Figure 2.11 – Instrumented semantics of assembly-level language

The semantics of assembly-level $\mathsf{align_a}$, $\mathsf{label_a}\ n$, and $\mathsf{jump}\ n$ are similar to the semantics of intermediate-level instructions. The $\mathsf{jcc}(n, cond)$ instruction jumps to the instruction with instruction pointer n if the condition $cond$ evaluates to tt. It leaks the difference between the instruction pointer and the boolean tt (of the form $\mathsf{if_a}\ ((ip'+1)-ip)\ tt$). The guard represented by cond is evaluated using the judgment $[\mathsf{cond}]^{s_{\mathsf{flag}}}$ that uses s_{flag} (flag map parameter of state s) to evaluate the cond where cond represents various flag conditions like equal, not equal, carry set, overflow, signed less than etc. Similarly, in the case where $cond$ evaluates to ff, the instruction $\mathsf{jcc}(n, cond)$ proceeds to the next instruction and leaks $\mathsf{if_a}\ 1\ ff$. The evaluation of $\mathsf{asm_{op}}(\mathsf{o}, \mathsf{es})$ leaks the memory addresses touched during the application of operation o (x86 operations) on the arguments es (it can be implicit, explicit registers, immediate value or memory addresses).

2.3 Leakage transformers

The job of leakage transformers is to transform the source-level leakage to the target-level leakage. Five kinds of leakage transformers are introduced to instrument all compilation passes of the Jasmin compiler from source to assembly. Three kinds for high-level compiler passes, and two kinds corresponding to linearization and assembly generation passes. In most cases, these functions are independent of the program except for the stack-allocation pass, which needs access to the stack pointer. The syntax of leakage transformers τ_e for expressions and τ_s for accessed addresses are shown in Figure 2.12. The formal semantic of τ_e and τ_s are interpreted using the notations $[\![\tau_e]\!]^{\ell_e}_e$ (for expressions) and $[\![\tau_s]\!]^{v_{\mathsf{sp}}}_s$ (for accessed addresses) where v_{sp} is the value of stack pointer. Their formal definitions are provided Figure 2.14. The syntax of leakage transformers τ for instructions are shown in Figure 2.13, and their formal definitions are provided in Figure 2.21. The formal semantic of τ is interpreted as $[\![\tau]\!]^{\ell}$.

$\tau_e ::=$	$\bullet$	remove	$\tau_s ::=$	$\mathsf{cst}(p)$	constant
	$\mid$ id	identity		$\mid$ sp	stack pointer
	$\mid$ $\tau_e \circ \tau_e$	composition		$\mid$ $\tau_s + \tau_s$	addition
	$\mid$ π_i	projection		$\mid$ $\tau_s \times \tau_s$	multiplication
	$\mid$ $(\tau_e, \ldots, \tau_e)$	map		$\mid$ x	variable
	$\mid$ rev	reverse			
	$\mid$ $(\tau_e; \ldots; \tau_e)$	sequence			
	$\mid$ $C(\tau_s)$	constant addr			
	$\mid$ $I(x \mapsto \tau_s)$	indexed addr			

Figure 2.12 – Leakage transformers for expressions

$\tau :=$	keep	keep
	$\mid$ $\mathsf{op}(\tau_e, \ldots, \tau_e)$	op
	$\mid$ $\mathsf{if}(\tau_e, \tau, \tau)$	cond
	$\mid$ $\mathsf{while}(\tau, \tau_e, \tau)$	while
	$\mid$ $\mathsf{for}(\tau_e, (\tau; \ldots; \tau))$	for
	$\mid$ $\mathsf{call}(fn, (\tau_e; \ldots; \tau_e), (\tau_e; \ldots; \tau_e))$	call
	$\mid$ $(\tau; \ldots; \tau)$	sequence
	$\mid$ remove	remove
	$\mid$ $\mathsf{ceval}_b\ \tau$	cond-eval
	$\mid$ $\mathsf{for}_{\mathsf{unroll}}(n, \tau)$	for-unroll
	$\mid$ $\mathsf{cond}_{\mathsf{low}}((\tau_e; \ldots; \tau_e), \tau_e, (\tau; \ldots; \tau), (\tau; \ldots; \tau))$	cond-low
	$\mid$ $\mathsf{while}_{\mathsf{low}}((\tau_e; \ldots; \tau_e), \tau_e, (\tau; \ldots; \tau), (\tau; \ldots; \tau))$	while-low
	$\mid$ $\mathsf{op}_{\mathsf{low}}\ (\tau_e; \ldots; \tau_e)$	op-low

Figure 2.13 – Leakage transformers for instructions

2.3.1 Leakage transformers for expressions

The leakage transformer $\bullet$ erases the leakage and produces $\bullet$ leakage. id is used where the compiler does not modify the expression; hence, leakage must also be preserved. The leakage transformer $\tau_e \circ \tau_e$ allows the composition of leakage transformers. The leakage transformer π_i returns the leakage at index i from the set of leakages. $(\tau_e, \ldots, \tau_e)$ maps a set of leakage transformers to a set of leakages. rev reverses the set of leakages. $(\tau_e; \ldots; \tau_e)$ generates the target leakage by applying a sequence of leakage transformers to a source leakage. The transformer $C(\tau_s)$ introduces fresh memory access leakages. $C(\tau_s)$ is used in stack-allocation compiler pass where a new leakage is created, and their semantics depend on the value of the stack pointer. Similarly, $I(x \mapsto \tau_s)$ introduces a new leakage based on the free variable x that denotes the actual value of the index. Both $I(x \mapsto \tau_s)$ and $C(\tau_s)$ take a type τ_s as an argument that denotes the accessed addresses. The semantics of leakage transformers τ_e and τ_s are present in Figure 2.14. The semantics of leakage transformers $C(\tau_s)$ and $I(x \mapsto \tau_s)$ only need the value of stack pointer to compute the target leakage because they are involved in transforming source-level read and write to actual memory store and load (in stack-allocation pass, the variables are allocated in the

Leakage transformers for expressions:

$$\frac{}{[\![\bullet]\!]_e^{\ell_e} = \bullet} \qquad \frac{}{[\![id]\!]_e^{\ell_e} = \ell_e} \qquad \frac{}{[\![\mathsf{rev}]\!]_e^{(\ell_1,\ldots,\ell_n)} = (\ell_n,\ldots,\ell_1)}$$

$$\frac{}{[\![\pi_i]\!]_e^{(\ell_1,\ldots,\ell_n)} = \ell_i} \qquad \frac{[\![\tau_i]\!]_e^{\ell} = \ell_i'}{[\![(\tau_1;\ldots;\tau_n)]\!]_e^{\ell} = (\ell_1',\ldots,\ell_n')}$$

$$\frac{[\![\tau_i]\!]_e^{\ell_i} = \ell_i'}{[\![(\tau_1,\ldots,\tau_n)]\!]_e^{(\ell_1,\ldots,\ell_n)} = (\ell_1',\ldots,\ell_n')} \qquad \frac{[\![\tau_1]\!]_e^{\ell_e} = \ell_e'}{[\![\tau_1 \circ \tau_2]\!]_e^{\ell_e} = [\![\tau_2]\!]_e^{\ell_e'}}$$

$$\frac{[\![\tau_s]\!]_s^{v_{\mathsf{sp}}} = p}{[\![C(\tau_s)]\!]_e^{\ell_e} = *p} \qquad \frac{[\![\tau_s[i/x]]\!]_s^{v_{\mathsf{sp}}} = p}{[\![I(x \mapsto \tau_s)]\!]_e^{[i]} = *p}$$

Transformers creating address leakage:

$$\frac{}{[\![\mathsf{cst}(p)]\!]_s^{v_{\mathsf{sp}}} = p} \qquad \frac{}{[\![\tau_{s_1} + \tau_{s_2}]\!]_s^{v_{\mathsf{sp}}} = [\![\tau_{s_1}]\!]_s^{v_{\mathsf{sp}}} + [\![\tau_{s_2}]\!]_s^{v_{\mathsf{sp}}}}$$

$$\frac{}{[\![\mathsf{sp}]\!]_s^{v_{\mathsf{sp}}} = v_{\mathsf{sp}}} \qquad \frac{}{[\![\tau_{s_1} \times \tau_{s_2}]\!]_s^{v_{\mathsf{sp}}} = [\![\tau_{s_1}]\!]_s^{v_{\mathsf{sp}}} \times [\![\tau_{s_2}]\!]_s^{v_{\mathsf{sp}}}}$$

Figure 2.14 – Semantics for leakage transformers

stack). The rest of the leakage transformers only need the source leakage to compute the target leakage.

Illustrative example Here is a table illustrating some examples that show how various leakage transformers produce target leakages from source leakages.

Expression		Leakage	Leakage	
source	target	transformer	source	target
$0 \times e$	0	$\bullet$	$(\bullet, \ell)$	$\bullet$
$0 + e$	e	π_2	$(\bullet, \ell)$	ℓ
$e_1 + e_2$	$e_1' + e_2'$	(τ_e^1, τ_e^2)	(ℓ_1, ℓ_2)	(ℓ_1', ℓ_2')
$e_1 + e_2$	e_2'	$\pi_2 \circ \tau_e^2$	(ℓ_1, ℓ_2)	ℓ_2'

In the first line, the compiler knows statically that multiplication of 0 to an expression e results in 0, hence the source leakage $(\bullet, \ell)$ corresponding to 0 and e transforms to $\bullet$ by the leakage transformer $\bullet$ (because the target expression is just a constant 0, which does not produce any leakage during its evaluation). In the second line, the source expression $0+e$ is transformed to e (which means the leakage associated with e must be preserved at the target level), so the leakage transformer is a projection π_2. In the third line, both the expression e_1 and e_2 present in the source expression are transformed to e_1' and e_2'; hence, the leakage corresponding to e_1 i.e., ℓ_1 and leakage corresponding to e_2 i.e., ℓ_2 needs to be transformed to the leakage corresponding to e_1' i.e. ℓ_1' and the leakage corresponding to e_2' i.e. ℓ_2'. The leakage transformer in the above case is a map consisting of leakage transformers (τ_e^1, τ_e^2) where τ_e^1 transforms the leakage ℓ_1 to ℓ_1' and τ_e^2 transforms the leakage ℓ_2 to ℓ_2'. In the fourth line, the addition is removed (as in the second line), and the second sub-expression is recursively transformed, so we compose a projection with the leakage transformer for the sub-expression.

2.3.2 Transformers creating accessed addresses

In compiler pass like stack-allocation, new leakage is created. In the stack-allocation pass, some variables are allocated into the stack memory, replacing the corresponding accesses (read and write) with memory operations (load and store). This, in turn, creates new leakages as memory operations are involved. Given a variable x allocated at constant offset o_x in the stack, a read from this variable will be compiled into the memory load $[sp + o_x]$ where sp is the stack pointer value register. At the source level, the leakage of x is $\bullet$. It becomes $(\bullet, *(v_{sp} + o_x))$ at the target level where $\bullet$ corresponds to the leakage associated with the evaluation of the offset o_x and v_{sp} is the value of the stack pointer.

The case of an array variable a, allocated at a constant offset o_a works similarly. The source level array access $a[e]$ is compiled into $[sp + o_a + n \times e]$ where n is the size of an array element. At the source level, $a[e]$ leaks (ℓ_e, v_e) where ℓ_e is the leakage obtained while evaluating the expression e and v_e is the value of the index. At the target level, the leakage is $*(v_{\mathsf{sp}} + o_a + n \times v_e)$. o_a and n are statically known values provided to the leakage transformer. So, in this transformation, the target leakage will further depend on the value v_e that can be recovered from the source leakage and on the dynamic value of sp.

These two cases described above are the cases where the new leakage is created by not only using the source leakage but also requiring some extra information, like the value of the stack pointer. Hence, the leakage transformers depend not only on the source leakage but also on the stack pointer, which has some consequence for preserving constant-time that is explained in Section 2.5. As the leakage transformer needs to be parameterized by the stack pointer value, it must be considered as *public* input.

To capture these kinds of transformations, the syntax of leakage transformers includes transformers that can construct new leakages, and their semantics depend on the value of the stack pointer. New memory accesses that are introduced during stack allocation are constructed using the leakage transformer $C(\tau_s)$ and $I(x \mapsto \tau_s)$. They depend on τ_s that represents the accessed address. x is a free variable representing the index's actual value. The τ_s can be computed by performing some operations like $+$ and $\times$ whose semantics are explained in Figure 2.14.

Brief introduction of stack allocation pass

The stack allocation pass consists of an analysis that computes the stack frame's layout and introduces memory operations. As far as leakage transformers are concerned, only the second part is relevant. The leakage transformation only concerns the second part, which involves transforming the read/write operations to load/store operations. The leakage transformation involves transforming leakages due to expression, left values, and instructions. The transformation of expressions and left values is done similarly. Here, a code snippet (Coq encoding) is presented in Figure 2.15 that represents the transformation of expressions in the stack allocation pass. alloc_e is recursively defined that depends on memory m (result of the analysis part that is done to compute the stack frame's layout) and an expression e. The function alloc_e succeeds (and returns the compiled expression and leakage transformer) or returns an error (in the case of stack frames not aligned properly).

```
1  Fixpoint alloc_e (m: map) (e: pexpr) : cexec (pexpr × leak_e_tr) :=
2   match e with
3   | Pconst _ | Pbool _ | Parr_init _ | Pglobal _ ⇒ ret e id
4   | Papp1 o e ⇒
5     Let: (e, r) := alloc_e m e in
6     ret (Papp1 o e) r
7
8   | Pload ws x e ⇒ ...
9     Let: (e, r) := alloc_e m e in
10    ret (Pload ws x e) [ r, id ]
11
12  | Pvar x ⇒
13    if ... (* x has size ws and is allocated at offset ofs *) then
14      ret (Pload ws stk ofs) [ id; C (sp + cst ofs) ]
15    else ... (* x is a register *)
16      ret e id
17
18  | Pget ws x e ⇒
19    Let: (e, r) := alloc_e m e in
20    if ... (* x is allocated at offset ofs *) then
21      let: (ofs', t) := mk_ofs ws e ofs in
22      ret (Pload ws stk ofs') [ r ∘ t, I(x ↦ sp + cst x × cst ws + cst ofs) ]
23    else ... ret (Pget ws x e) [ r, id ]
24
25  | ... ⇒ ...
26  end.
```

Figure 2.15 – Pseudo-code of the stack-allocation of expressions

```
27 Definition mk_ofs ws e ofs : pexpr × leak_e_tr :=
28   let sz := wsize_size ws in
29   if is_const e is Some i then
30     ((cast_const (i × sz + ofs)),
31      I(x ↦ sp + cst x × cst sz + cst ofs))
32   else
33     let: (e, t) := cast_word e in
34     (add (mul (cast_const sz) e) (cast_const ofs), [ [ • ; t] ; • ]).
35 end.
```

Figure 2.16 – Pseudo-code of the mk_ofs

Compiling expressions like constant, boolean, array initialization, or global variable declaration is straightforward (line 3), as the compiled expression is the same as the source. Hence, in these cases, the leakage transformer will be id (that will preserve the source leakage). In the case of a variable (represented as Pvar x in line 12-16), if the variable is of type stack (and is bound to an offset ofs in the memory m) then it is transformed to a memory load operation else its compilation remains the same. In the case of stack variable, the leakage transformer is $id; C(sp + ofs)$ where sp is the value of stack pointer and ofs is the offset at which the variable will be assigned on the stack. In the case of the register variable, the leakage transformer is id as it just preserves the same leakage. The compilation of read (line 18-23) of size ws from an array x at an index e is transformed to a memory load with an offset recursively calculated from e. The ofs is calculated using a function mk_ofs, which simplifies the expression e and returns leakage transformers corresponding to it. In the case of array access, the leakage transformer is a sequence of composition of two leakage transformers related to the offset (the first

part of composition is the leakage transformer obtained by recursively applying alloc_e to e. The second part is the leakage transformer obtained by function mk_ofs), and $I(x \mapsto sp + cst(x) * cst(ws) + cst(ofs))$ where x represents the index, ws represents the size and ofs represents the offset.

Compiling the rest of the expressions works similarly by recursively applying the function alloc_e to its sub-parts. For example, in the case of unary operators o (Papp1 in line 4-6) applied to a sub-expression e produce the same leakage as produced by the sub-expression; therefore, the leakage transformer corresponds to the leakage transformer produced during the compilation of the sub-expression e. Compiling a load expression recursively transforms the address expression but preserves its value (lines 8-10).

The transformation of instructions is pretty straightforward as it applies the function alloc_e and the function transforming the left value on the sub-parts and recursively applies itself to the part concerning the instructions. Hence, the leakage transformers generated by these sub-parts are collectively used to produce the leakage transformer for each kind of instruction.

```
1 fn foo() ⟶ reg u64 {
2   stack u64[2] t;
3   reg u64 p r;
4   t[0] = 0;
5   t[1] = 1;
6   p = 0;
7   r = t[(int)(p + 1)];
8   return r; }
```

```
1 fn foo() ⟶ u64 {
2   stack: 16
3   [RSP + 0] = MOV(0);
4   [RSP + 8] = MOV(1);
5   RAX = MOV(0);
6   RAX = MOV([RSP + (8 * (RAX + 1) + 0)]);
7   return RAX; }
```

Figure 2.17 – Example program: source and after stack-allocation

Illustrative example Figure 2.17 presents a small program that stores two literal values 0 and 1 in an array t and later reads the value from the array an index $p + 1$ where p is assigned to value 0. The leakage associated with the source program present in the left side of Figure 2.17 is: { (•, [0]) := •; (•, [1]) := •; • := •; • := ((•, •), [1]) } (present in Figure 2.18). After the stack allocation, the compiled program is on the right side of Figure 2.17. The type of variable t in the source program is *stack*, which means the programmer wants the array to be stored in the stack. Hence after the stack allocation pass, the write operation $\mathsf{t}[0] = 0$ and $\mathsf{t}[1] = 1$ are transformed to $[\mathsf{RSP} + 0] = \mathsf{MOV}(0)$ and $[\mathsf{RSP} + 8] = \mathsf{MOV}(1)$ where the memory addresses are computed using the stack pointer, held in register RSP. The write operation $\mathsf{r} = \mathsf{t}[(\mathsf{int})(\mathsf{p} + 1)]$ is transformed to $\mathsf{RAX} = \mathsf{MOV}([\mathsf{RSP} + (8 * (\mathsf{RAX} + 1) + 0)])$ where the address is computed using the index, scaling multiplication, constant offsets, and the stack pointer RSP. The leakage transformer produced by the stack allocation pass is presented in Figure 2.20. As explained in Section 2.3.2, the leakage transformer consists of transformers like $C(\tau_s)$ and $I(x \mapsto \tau_s)$ to compute the new memory addresses. The target leakage produced by the application of the leakage transformer to the source leakage is of the form: $\{(\bullet, *v_{\mathsf{sp}}) := \bullet; (\bullet, *(v_{\mathsf{sp}} + 8)) := \bullet; \bullet := \bullet; \bullet := (((\bullet, (\bullet, \bullet)), \bullet), *(v_{\mathsf{sp}} + 8))\}$, where v_{sp} is the value of the stack pointer stored in RSP (present in Figure 2.19). The array index leaked during the evaluation of the source code is replaced by leaking the memory address at the target level.

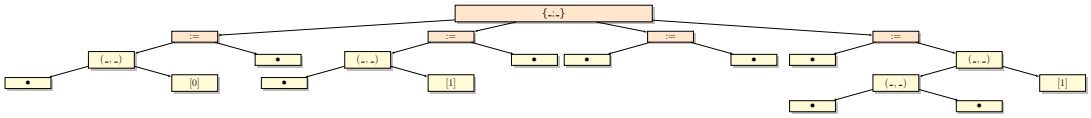


Figure 2.18 – Structured leakage for source code

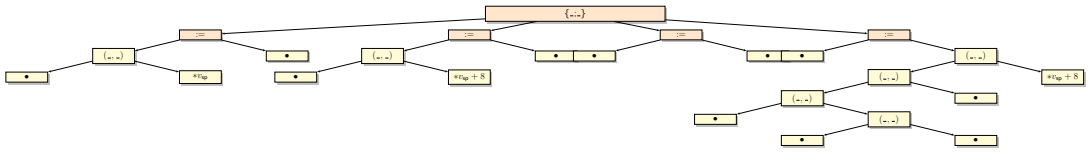


Figure 2.19 – Structured leakage for compiled code

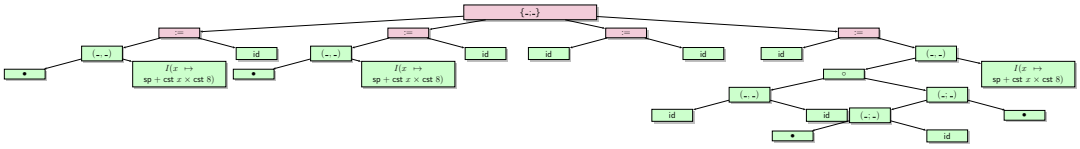


Figure 2.20 – Leakage transformer

2.3.3 Leakage transformers for instructions

There are different kinds of compiler passes. A set of leakage transformers is defined to cover the functionality of different compiler passes. The syntax of leakage transformers is present in Figure 2.13, and their semantics are defined in Figure 2.21. Many compiler passes preserve the program's structure and are defined recursively on the program's structure. For such passes, the compilation of instructions consists of applying a transformation on its sub-expression and sub-instructions. In these cases, the source leakages are transformed similarly. The target leakage after the compilation will have the same structure as the source leakage, and only its sub-components will be modified. To account for these cases, the syntax of leakage transformers includes a constructor per instruction. This constructor recursively transforms leakage without modifying its structure and only applying the transformation to the sub-leakages. The leakage transformers like $\mathsf{op}(\tau_d, \tau_e)$, $\mathsf{if}(\tau_e, \tau_{tt}, \tau_{ff})$, $\mathsf{while}(\tau, \tau_e, \tau')$, $\mathsf{for}(\tau_e, \tau)$ and $\mathsf{call}(fn, \tau_e, \tau'_e)$ fall in the above category. The target leakage produced by these leakages has the same structure as the source leakage. For example, the leakage transformer $\mathsf{op}(\tau_d, \tau_e)$ will expect a leakage of the form $\mathsf{op}_\mathsf{l}(\ell_d; \ell_e)$ and will apply its sup-transformers to the sub-leakages so that the resulting leakage will be of the form $\mathsf{op}_\mathsf{l}([\![\tau_d]\!]_e^{\ell_d}; [\![\tau_e]\!]_e^{\ell_e})$. For conditional instructions, the leakage transformer $\mathsf{if}(\tau_e, \tau_{tt}, \tau_{ff})$ is built from leakage transformers for the condition and each branch. Notice that only τ_{tt} or τ_{ff} will be used to transform the leakage (depending on which branch will be taken, but this cannot be known at compile-time). It is the interpretation of $[\![\mathsf{if}(\tau_e, \tau_{tt}, \tau_{ff})]\!]$ that selects which leakage transformer should be used. The leakage transformer for the loop works similarly. For function calls, the leakage transformer $\mathsf{call}(fn, \tau_e, \tau'_e)$ is built using the function name fn (helps in accessing the leakage transformer, which is used to transform the leakage associated with the body of the function), τ_e (transformer for the arguments) and τ'_e (transformer for leakages obtained while writing the result back).

Leakage transformers for instructions:

$$\frac{}{[\![\mathsf{keep}]\!]^{\ell} = \ell} \qquad \frac{}{[\![\mathsf{remove}]\!]^{\ell} = \{\}} \qquad \frac{[\![\tau_d]\!]_e^{\ell_d} = \ell'_d \qquad [\![\tau_e]\!]_e^{\ell_e} = \ell'_e}{[\![\mathsf{op}(\tau_d, \tau_e)]\!]^{\mathsf{op_l}(\ell_d;\ell_e)} = \mathsf{op_l}(\ell'_d; \ell'_e)}$$

$$\frac{[\![\tau_e]\!]_e^{\ell_e} = \ell'_e \qquad [\![\tau_b]\!]^{\ell_b} = \ell'_b}{[\![\mathsf{if}(\tau_e, \tau_{tt}, \tau_{ff})]\!]^{\mathsf{if}_b(\ell_e,\ell_b)} = \mathsf{if}_b(\ell'_e, \ell'_b)}$$

$$\frac{[\![\tau]\!]^{\ell_c} = \ell''_c \qquad [\![\tau_e]\!]_e^{\ell_e} = \ell'_e \qquad [\![\tau']\!]^{\ell'_c} = \ell'''_c \qquad [\![\mathsf{while}(\tau, \tau_e, \tau)]\!]^{\ell_w} = \ell'_w}{[\![\mathsf{while}(\tau, \tau_e, \tau')]\!]^{\mathsf{while_t}(\ell_c,\ell_e,\ell'_c,\ell_w)} = \mathsf{while_t}(\ell''_c, \ell'_e, \ell'''_c, \ell'_w)}$$

$$\frac{[\![\tau]\!]^{\ell_c} = \ell'_c \qquad [\![\tau_e]\!]_e^{\ell_e} = \ell'_e}{[\![\mathsf{while}(\tau, \tau_e, \tau')]\!]^{\mathsf{while_f}(\ell_c,\ell_e)} = \mathsf{while_f}(\ell'_c, \ell'_e)} \qquad \frac{[\![\tau_e]\!]_e^{\ell_e} = \ell'_e \qquad [\![\tau]\!]^{\ell_c} = \ell'_c}{[\![\mathsf{for}(\tau_e, \tau)]\!]^{\mathsf{for}\ \ell_e\ \ell_c} = \mathsf{for}\ \ell'_e\ \ell'_c}$$

$$\frac{[\![\tau_e]\!]_e^{\ell_e} = \ell''_e \qquad get_{lt}(fn) = \tau \qquad [\![\tau]\!]^{\ell_c} = \ell'_c \qquad [\![\tau'_e]\!]_e^{\ell'_e} = \ell'''_e}{[\![\mathsf{call}(fn, \tau_e, \tau'_e)]\!]^{\mathsf{call}\ \ell_e\ (fn,\ell_c)\ \ell'_e} = \mathsf{call}\ \ell''_e\ (fn, \ell'_c)\ \ell'''_e}$$

$$\frac{[\![\tau_i]\!]^{\ell_i} = \ell'_i}{[\![\tau_1; \ldots; \tau_n]\!]^{\ell_1;\ldots;\ell_n} = \ell'_1; \ldots; \ell'_n}$$

$$\frac{[\![\tau]\!]^{\ell_b} = \ell'_b}{[\![\mathsf{ceval}_b\ \tau]\!]^{\mathsf{if}_b(\ell_e,\ell_b)} = \ell'_b} \qquad \frac{[\![\tau]\!]^{\ell_c} = \ell'_c}{[\![\mathsf{ceval}_b\ \tau]\!]^{\mathsf{while_f}(\ell_c,\ell_e)} = \ell'_c}$$

$$\frac{[\![\tau]\!]^{\ell_c} = \ell'_c \qquad \ell'_c = \ell_{c'_1}; \ell_{c'_2}; \ldots; \ell_{c'_n}}{[\![\mathsf{for_{unroll}}(n, \tau)]\!]^{\mathsf{for}\ \ell_e\ \ell_c} = (\mathsf{op_l}(\bullet;\bullet); \ell_{c'_1}; \ldots; \mathsf{op_l}(\bullet;\bullet); \ell_{c'_n})}$$

Figure 2.21 – Semantics for leakage transformers

There is also a second category of compiler passes that changes the program's structure. Hence, the leakage transformer produced by this kind of compiler pass also transforms the structure of the source leakage. The leakage transformers like remove, $\mathsf{ceval}_b\ \tau$, $\mathsf{for_{unroll}}(n, \tau)$, etc. fall in the category which transforms the structure of the source leakage. remove is used when an instruction is removed, e.g., in dead-code elimination. Assume that we have a program of the form $i; c$, let c' and τ be the code and leakage transformer obtained by compilation of c. If the compiler can detect that the instruction i is redundant statically, then the compiler will remove it, and the compilation of $i; c$ will be c'. In this scenario, the leakage transformation should also remove the leakage associated with instruction i. The source leakage for $i; c$ is $l_i; l_c$, and the target leakage for c' is l'_c. The leakage transformer will be of the form remove; τ. The job of remove is to throw away the leakage l_i, and τ will transform l_c to l'_c.

The leakage transformer $\mathsf{ceval}_b\ \tau$ is used when a conditional instruction is replaced by one of its branches. This is used when an instruction if e then c_{tt} else c_{ff} is replaced by c_{tt} or c_{ff} (when the compiler statically knows the value of conditional guard to be equal to true or false) by the compiler. It can also be utilized where the result of the condition of the while loop is statically known.

Loop unrolling replicates the loop's body as many times as the range of the loop. The instruction of the form for $i\ r\ c$ is compiled to $i := 1; c_i; \ldots; i := n; c_n$ where n is the range of the loop. The source leakage corresponding to the for loop is of the form for $\ell_e\ \ell_c$ where

ℓ_e is the leakage produced during the evaluation of the range and ℓ_c is the leakage produced during the evaluation of the loop's body. After unrolling, the target leakage will be of the form $(\mathsf{op}(\bullet,\bullet);\ell_{c'_1};\ldots;\mathsf{op}(\bullet,\bullet);\ell_{c'_n})$ where $\mathsf{op}_\mathsf{l}(\bullet;\bullet)$ corresponds the leakage associated with the assigning the range (left-hand side is a variable which produces $\bullet$ leakage and the right-hand side is a constant which also produces $\bullet$ leakage). ℓ_i represents the leakage associated with the loop body where i ranges from 1 to n (range of the loop). The leakage transformer $\mathsf{for}_{\mathsf{unroll}}(n,\tau)$ is produced during unrolling where n is the number of instructions produced by the compiler after unrolling the for loop and τ transforms source leakage ℓ_c to $\{\ell_{c'_1};\ldots;\ell_{c'_n}\}$.

Brief introduction of lowering/instruction selection

The compiler produces another kind of leakage transformer during the lowering/instruction selection, where a sequence of instructions replaces one instruction. Lowering/Instruction selection replaces high-level instruction with low-level instructions closer to the assembly. The following table illustrates two examples where a single instruction is lowered to more than one instruction.

Instruction	
source	target
$x := 0; x := x+1$	$x := \mathsf{MOV}(0); (\mathsf{OF},\mathsf{SF},\mathsf{PF},\mathsf{ZF},x) := \mathsf{INC}(x)$
$\mathsf{if}\ x < y\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2$	$(\ldots,CF,\ldots) := CMP(x,y); \mathsf{if}\ CF\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2$

Leakage	
source	target
$\mathsf{op}_\mathsf{l}((\bullet,\bullet);\bullet); \mathsf{op}_\mathsf{l}((\bullet,\bullet);(\bullet,\bullet))$ $\mathsf{if}_b((\bullet;\bullet),\ell_{c_i})$	$\mathsf{op}_\mathsf{l}((\bullet,\bullet);\bullet); \mathsf{op}_\mathsf{l}((\bullet;\bullet;\bullet;\bullet;\bullet);\bullet)$ $\mathsf{op}_\mathsf{l}((\bullet;\bullet;\bullet;\bullet;\bullet);(\bullet,\bullet))\ \mathsf{if}_b((\bullet;\bullet),\ell_{c_i})$

For example, the sequence of instructions $(x := 0; x := x+1)$ in the first row is transformed to $x := \mathsf{MOV}(0); (\mathsf{OF},\mathsf{SF},\mathsf{PF},\mathsf{ZF},x) := \mathsf{INC}(x)$ where assigning 0 is replaced by MOV and $+1$ is replaced by low-level instruction INC. The low-level instruction INC performs addition by 1 and computes extra flags like overflow-flag, sign-flag, parity-flag, and zero-flag. Similarly, the instruction $\mathsf{if}\ x < y\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2$ in the second row is transformed to two instructions $(\ldots,CF,\ldots) := CMP(x,y)$ and $\mathsf{if}\ CF\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2$. The first instruction $(\ldots,CF,\ldots) := CMP(x,y)$ represents the lowering of the guard that uses a low-level CMP instructions (it performs an unsigned comparison of x and y and also assigns extra flags like comparison-flag). The second instruction uses the CF flag to decide whether to take the true or false branch.

The leakage obtained from these instructions before and after lowering is present in the table above. The assignment of the flags will create extra $\bullet$ leakage that has to be justified. Similarly, the leakage generated by the expression $x < y$ must be used to develop the leakage for the CMP instruction. Therefore, this pass relies on leakage transformers that can, on the one hand, split leakages into smaller parts and, on the other hand, construct fresh leakages from these parts.

Leakage transformers for constructing instruction's leakage from expression's leakages:

$$\frac{[\![\tau_i]\!]^{\ell}_{e} = \ell'_i}{[\![(\tau_1;\ldots;\tau_n)]\!]^{\ell}_{ei} = \mathsf{op_l}(\ell'_1,\ldots;\ell'_n)}$$

Leakage transformers for lowering compiler pass:

$$\frac{[\![\tau_i]\!]^{\ell_e}_{ei} = \ell'_i \qquad [\![\tau_e]\!]^{\ell_e}_{e} = \ell'_e \qquad [\![\mathsf{if}\ b\ \mathsf{then}\ \tau_{tt}\ \mathsf{else}\ \tau_{ff}]\!]^{\ell_c} = \ell'_c}{[\![\mathsf{cond_{low}}(\tau_i,\tau_e,\tau_{tt},\tau_{ff})]\!]^{\mathsf{if}_b(\ell_e,\ell_c)} = \ell'_i;\mathsf{if}_b(\ell'_e,\ell'_c)}$$

$$\frac{[\![\tau_i]\!]^{\ell_e}_{ei} = \ell'_i \quad [\![\tau_{ff}]\!]^{\ell_c} = \ell''_c \qquad [\![\tau_e]\!]^{\ell_e}_{e} = \ell'_e \qquad [\![\tau_{tt}]\!]^{\ell'_c} = \ell'''_c \qquad [\![\mathsf{while_{low}}(\tau_i,\tau_e,\tau_{tt},\tau_{ff})]\!]^{\ell_w} = \ell'_w}{[\![\mathsf{while_{low}}(\tau_i,\tau_e,\tau_{tt},\tau_{ff})]\!]^{\mathsf{while_t}(\ell_c,\ell_e,\ell'_c,\ell_w)} = \mathsf{while_t}((\ell''_c;\ell'_i),\ell'_e,\ell'''_c,\ell'_w)}$$

$$\frac{[\![\tau_i]\!]^{\ell_e}_{ei} = \ell'_i \quad [\![\tau_{ff}]\!]^{\ell_c} = \ell''_c \qquad [\![\tau_e]\!]^{\ell_e}_{e} = \ell'_e}{[\![\mathsf{while_{low}}(\tau_i,\tau_e,\tau_{tt},\tau_{ff})]\!]^{\mathsf{while_f}(\ell_c,\ell_e)} = \mathsf{while_f}((\ell''_c;\ell'_i),\ell'_e)}$$

Figure 2.22 – Semantics for leakage transformers used in lowering

The semantics of leakage transformers used in the lowering compiler pass are in the Figure 2.22. It uses an interpretation $[\![\tau]\!]^{\ell_e}_{ei}$ that transforms the leakages of a kind ℓ_e (leakages produced during the evaluation of expressions) to leakages of a kind ℓ (leakages produced during the evaluation of instructions). In lowering, the guard of a branching instruction that is represented using an expression at high-level is transformed into an assignment instruction that assigns various flags and replaces the high-level operation with a low-level operation. The semantics of $\mathsf{cond_{low}}$ and $\mathsf{while_{low}}$ are very similar and are explained below with the help of the example used in the above paragraph. The leakage transformer produced by the lowering compiler pass during the transformation of the instruction $x := 0; x := x+1$ is $\mathsf{op}(\mathsf{id},\mathsf{id});\mathsf{op}((\mathsf{remove};\mathsf{remove};\mathsf{remove};\mathsf{remove};\pi_1),\mathsf{remove})$. The extra leakage $\bullet$ generated due to the flags' computations needs to be justified using the leakage transformer remove, and the rest of the leakages are similar to the source. During the transformation of the instruction $\mathsf{if}\ x < y\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2$, the leakage transformer produced during lowering is $\mathsf{cond_{low}}((\mathsf{op}(\mathsf{remove};\mathsf{remove};\mathsf{remove};\mathsf{remove};\mathsf{remove},\mathsf{id})),\mathsf{remove},\tau_1,\tau_2)$. $\mathsf{cond_{low}}$ takes four parameters (τ_i, τ_e, τ_{tt} and τ_{ff}). τ_i represents the leakage transformer needed to transform the leakage associated with guard expression to leakage associated with assignment instruction at the target level (as the guard is transformed to a CMP instruction and assigns extra flags). In the example, τ_i is $\mathsf{op}(\mathsf{remove};\mathsf{remove};\mathsf{remove};\mathsf{remove};\mathsf{remove},\mathsf{id})$ where remove creates fresh $\bullet$ leakages for flags and id preserves the leakage associated with x and y from the source. τ_e (remove) transforms the leakage associated with the guard from the source to the target level. τ_{tt} or τ_{ff} (τ_1 or τ_2) transform the leakage associated with the true or the false branch. The leakage transformers produced during the loop transformation are similar to the conditional.

Brief introduction of linearization

In linearization compiler pass, the high-level control-flow instructions are replaced by a linear set of instructions with explicit branches and labels explained in Section 2.2.2. Figure 2.23 presents a code snippet showing a source program consisting of branching

```
1 fn bar() ⟶ reg u64 {
2 reg u64 x;
3 reg u64 y;
4 x = 0;
5 y = 1;
6 if (x < y)
7    { y = y + 1; }
8 else
9    { y = 0; }
10 return y;
11 }
```

```
1 fn bar() ⟶ (u64) {
2 stack: 0
3 RCX.62 = MOV_64(((64u) 0));
4 RAX.60 = MOV_64(((64u) 1));
5 OF.94, CF.92, SF.96, PF.95, ZF.97 = CMP_64(RCX.62, RAX.60);
6 If CF.92
7    Goto 1;
8    RAX.60 = MOV_64(((64u) 0));
9    Goto 2;
10   Label 1;
11   OF.94, SF.96, PF.95, ZF.97, RAX.60 =
12   INC_64(RAX.60);
13   Label 2
14   return RAX.60
15 }
```

Figure 2.23 – Example program: source and after linearization

$$
\begin{array}{rlll}
\tau_l := & \mathsf{op_l}\ \tau_e & & \text{opl} \\
\mid & \mathsf{if}_{tt}(\tau_e, \tau) & & \mathsf{cond_t} \\
\mid & \mathsf{if}_{ff}(\tau_e, \tau) & & \mathsf{cond_f} \\
\mid & \mathsf{if}_{*}(\tau_e, \tau, \tau') & & \text{cond} \\
\mid & \mathsf{while}(a, \tau, \tau') & & \text{while} \\
\mid & \mathsf{while}\ \tau & & \mathsf{while_f} \\
\mid & \mathsf{while}(a, \tau) & & \mathsf{while_t}
\end{array}
$$

Figure 2.24 – Leakage transformers produced during linearization

instruction at the source level and a code snippet of the same program after linearization compiler pass. The branching instruction present in line 6 at the source program is transformed to linear-level branching, which uses goto and label intermediate-level instructions defined in Figure 2.6. The leakages (present in Figure 2.6) obtained during the evaluation of the intermediate-level language are also of a different kind as compared to the leakages (present in Figure 2.5) obtained during the evaluation of high-level language (as the two languages also have different structures). Figure 2.24 presents the syntax of leakage transformers that transform the high-level leakage to intermediate-level (linear) leakages. The interpretation $[\![\tau]\!]^{\ell}_{l}$ presented in Figure 2.25 defines the transformation where the leakage transformer τ transforms the high-level leakage ℓ to intermediate-level leakage. The leakage transformer $\mathsf{op_l}\ \tau$ transforms the high-level leakage associated with assignment or operators to intermediate-level leakage of kind $\mathsf{op}\ \ell'_1, \ldots \ell'_n$ where the leakages ℓ_i is obtained by applying the leakage transformer τ to the sub-parts of high-level leakage. In Jasmin, the linearization of conditional instructions like if-else and while is defined using more than one case to increase the overall performance. For example, there is one dedicated case to transform the instruction of the form $\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ [::]$ (the else branch is empty) to $\mathsf{if_i}(e, n1); c_1; \mathsf{label}\ n_1$. These design decisions are made to optimize the transformation process. The generalized case is discussed here. The high-level instruction $\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2$ is transformed to $\mathsf{if_i}(e, n1); c_2; \mathsf{goto}\ n_2; \mathsf{label}\ n_1; c_1; \mathsf{label}\ n_2$ where source-level control flow is transformed into explicit labels and gotos. The leakage transformer generated during this transformation is of the form $\mathsf{if}_{*}(\tau_e, \tau_i, \tau'_i)$ where τ_e transforms the

Auxiliary functions:

$$\mathsf{c_{size}}(\tau) = \begin{cases} 1 & \tau = \mathsf{op_l}\ \tau \\ \mathsf{c_{size}}(\tau_i) + \mathsf{c_{size}}(\tau_i') + 4 & \tau = \mathsf{if}_*(\tau_e, \tau_i, \tau_i') \end{cases}$$

Leakage transformers:

$$\frac{[\![\tau]\!]_e^{\ell_1,\ldots,\ell_n} = \ell_1', \ldots, \ell_n'}{[\![\mathsf{op_l}\ \tau]\!]_l^{\mathsf{op_l}(\ell_1,\ldots;\ell_n)} = \mathsf{op}\ \ell_1', \ldots \ell_n'}$$

$$\frac{[\![\tau_e]\!]_e^{\ell_e} = \ell_e' \qquad [\![\tau_i]\!]_l^{\ell_c} = \ell_c' \qquad [\![\tau_i']\!]_l^{\ell_c} = \ell_c''}{[\![\mathsf{if}_*(\tau_e, \tau_i, \tau_i')]\!]_l^{\mathsf{if}_b(\ell_e,\ell_c)} = \mathsf{if}\ b\ \mathsf{then}\ ((\mathsf{if_l}\ (\mathsf{c_{size}}(\tau_i') + 3)\ \ell_e'\ b), \ell_c', \bullet)\ \mathsf{else}\ ((\mathsf{if_l}\ 1\ \ell_e'\ b), \ell_c'', (\mathsf{c_{size}}(\tau_i) + 3))}$$

Figure 2.25 – Semantics for leakage transformers used in linearization

source leakage obtained during the evaluation of the guard e, τ_i or τ_i' transform the source leakage associated with the true or false branch. The target leakage leaks some extra information compared to the source level leakage. Along with the guard, it also leaks the position where the control flow jumps to. For example, if the guard e evaluates to true, the control must go to the instruction after the label n_1, and hence, it must take the n + 3 steps where n is the number of instructions present in the linear transformation of c_2. The number of instructions present in a set of instructions obtained after linearization is calculated using the auxiliary function $\mathsf{c_{size}}()$, which calculates the steps based on the leakage transformers and does not need the actual program. The leakage transformers play an important role here because they give an intuition about the structure of instruction before and after the transformation. The leakage transformer generated for loops is defined in a similar manner and is not explained here.

2.4 Instrumented correctness

The Jasmin compiler is formally proved to be functionally correct, but there were no guarantees for the correctness of the leakage transformers. Hence, there is a need to verify the instrumented Jasmin compiler's correctness formally. The correctness of the leakage transformer is closely related to the correctness proof for the compiler. A correct compiler is necessary for proving the correctness of the leakage transformer because, in the end, the leakage transformer transforms the leakage associated with the execution of an instruction. The correctness proof of leakage transformers is stated as follows: For each source program p, if the compilation succeeds and produces target program $\overline{p}$ and leakage transformer τ, then for every instrumented execution of the source producing a leakage ℓ, the instrumented execution of the target program is defined and produces a target leakage, which is equal to the leakage obtained by applying the leakage transformer τ to the source leakage ℓ.

Theorem 1 (Instrumented correctness)**.**

$$p : s \Downarrow_\ell s' \implies \overline{p} : s \Downarrow_{[\![\tau]\!]^\ell} s'.$$

Proof. As presented in the Figure 1.10, the Jasmin compiler consists of a set of compiler passes. The correctness proof of the compiler is done independently for each compiler pass. Similarly, the correctness proof for the leakage transformer is also done for each compiler pass. All the correctness theorems are updated to instrumented correctness, and proof for the correctness of leakage transformers is added to them. The proof for the correctness of the leakage transformer boils down to proving the correctness of the functions $[\![.]\!]_e$ and $[\![.]\!]^\cdot$ that they compute the correct target leakage from the source leakage,

There are various kinds of compiler passes in Jasmin. Some of the compiler's passes preserve the program's structure, some modify the structure, and some completely remove instructions. The leakage transformers are designed by keeping the semantics of these compiler passes in mind, and hence, they also transform the source leakages similarly. The correctness proof for these leakage transformers is straightforward if their semantics are correctly defined. □

2.5 Preservation of constant-time

The constant-time property is a software-based countermeasure against side-channel attacks. Even if the programmer succeeds in writing a constant-time program, there still exists a chance of vulnerabilities that the compiler might introduce. Several optimizations introduced by the compiler, as discussed in Section 1.2.3, might introduce vulnerabilities in the program that support constant-time property at the source level. For example, a compiler optimization might introduce branching into the source code that is originally branchless, or a compiler can also introduce time-variable operations like division or modulus, which might break the constant-time property. The solution is to manually inspect the assembly generated at the end to determine whether the constant-time property is preserved. This approach is quite tricky as much information is lost along the way to the assembly, and as the process is manual, it is prone to human error. Hence, there is a need to have a certified way to preserve the constant-time property till the end.

Preserving constant-time property helps certify that the Jasmin compiler does not break the constant-time property. Though it transforms the control flow and introduces memory accesses, it never removes the constant-time property. We need a formal notion of constant-time property to carry out the preservation proof.

Formally, constant-time is defined as follows:

Definition 1 (Constant-time)**.** *A program p is* constant-time *w.r.t. the indistinguishability relation $\cdot \sim \cdot$ when the following holds:*

$$\forall s_1\ s_2, s_1 \sim s_2 \implies \exists s_1'\ s_2'\ \ell, p : s_1 \Downarrow_\ell s_1' \wedge p : s_2 \Downarrow_\ell s_2'.$$

The definition of constant-time is parameterized by an indistinguishability relation on the initial states s_1 and s_2. The indistinguishability relation $\sim$ states that the program states only differ in their secret part.

The constant-time definition is used in the preservation of constant-time property. Informally, preservation of constant-time says that if the source program is constant-time with respect to the indistinguishability relation $\sim$ then the compiled program will also be

constant-time with respect to indistinguishability relation $\sim$. But in reality, the source states and target states are different. In the case of the Jasmin compiler, an initial source state is made of a memory m and a list of values $\overline{v}$ (the arguments of the main function), whereas the target state is made of a memory m and a register bank r. From the design of target states, we can see that the source states can be computed from the target states as the memory is kept, and the values of the arguments can be read from the appropriate registers. Hence, we can relate the target states by linking the corresponding source states.

As we have seen in the discussion about the stack-allocation compiler pass, the target leakage presented in Figure 2.19 depends on the value of the initial stack pointer. Also, the leakage transformer shown in Figure 2.20 depends on the stack pointer to compute the target leakage from the source leakage. Hence, we must make the value of the stack pointer public. This requires constant-time preservation for the target language must consider that the stack pointers are equal in both states.

A few issues need to be considered to prove constant-time preservation for a realistic programming language like Jasmin.

- Source and target programs (states) are syntactically and semantically different.
- The semantics or interpretation of the leakage transformers is parameterized by parts of the initial state (mainly the stack pointer)
- The compiler correctness has side conditions, like there should be enough free memory in the initial target state to allocate the local variables.

Hence, the target states must be in indistinguishable relation $\bar{\sim}$ with each other defined as follows:

Definition 2 (Indistinguishability of target states)**.** *Given an equivalence relation $\sim$ between source states, its* lifting to target states $\bar{\sim}$ *is defined as follows. We say that two target states (m_1, r_1) and (m_2, r_2) are indistinguishable, and note $(m_1, r_1)\bar{\sim}(m_2, r_2)$, when all the following conditions hold:*

- *corresponding initial source states are indistinguishable, noted: $(m_1, \vec{v}_1) \sim (m_2, \vec{v}_2)$ (where $\vec{v}_1$, resp. $\vec{v}_2$, denotes the program arguments extracted from register bank r_1, resp. r_2);*
- *stack pointers agree: $r_1[sp] = r_2[sp]$;*
- *there is enough free stack space to allocate the local variables in both memories m_1 and m_2.*

We can formally define constant-time preservation after defining indistinguishable relations for both source and target states. The constant time preservation is formally defined as follows:

Theorem 2 (Constant time preservation)**.** *Given a source program p that is constant time w.r.t. $\sim$, if the Jasmin compiler succeeds and produces a target program $\bar{p}$, then the target program is constant time w.r.t. $\bar{\sim}$.*

Proof. The proof of constant time preservation directly results from the instrumented correctness proof. The leakage transformers are compositional. All the compiler passes produce leakage transformers that can be composed to transform source leakage to the

target leakage. As the instrumented correctness formally verifies that the compiler passes to produce the correct leakage transformer, the target leakage is also correctly calculated using the composition of these leakage transformers.

A particular remark: the leakage transformer must only exist to prove the preservation of constant time. The approach of having an existential quantifier for the leakage transformer, as discussed in Section 2.1.3 can prove preservation without the compilation passes producing it. But to reason about other properties like the cost of the assembly program from the cost of the source program explained in the next chapter, these leakage transformers need to be explicitly produced by the compiler. Hence, leakage transformers as a product of the compilation can be valuable artifacts: looking at the leakage transformer for a particular program, one can get precise information about the compilation of this program. □

2.6 Evaluation

This section evaluates the methodology in terms of proof effort and compile-time overhead.

2.6.1 Proof effort

The Jasmin compiler consists of 16 compilation passes verified for correctness in Coq. To reason about constant-time, all semantics have been instrumented with leakages as described in Section 2.2. All compilation passes are also instrumented to produce the leakage transformers as described in Section 2.3. The validator infers the correct leakage transformer for the passes implemented as an external oracle and validated in Coq. The program analyses and transformations that are implemented in OCaml have remained the same.

The correctness theorem for each pass has been modified, and their proofs are updated accordingly. The constant time preservation theorem stated in Section 2.5 is stated and proved only once for the whole compiler, and proofs are only a few lines long as it is just a corollary of the correctness theorem. The changes made to the Coq files modify 5 thousand lines and add 6 thousand new lines. It is a 20 percent increase in the overall Coq development.

2.6.2 Compiler behavior

The instrumented compiler produces an extra parameter called leakage transformer; hence, it needs to compute more data. To measure the compile-time overhead of the computation of leakage transformers, a set of Jasmin programs is compiled with two versions of the Jasmin compilers (with and without instrumentation). The experiment is done on a machine running Ubuntu Linux on an Intel® Xeon® processor (E5-2687W v3 @ 3.10GHz) using a sample of Jasmin implementations of cryptographic primitives from various sources.

Table 2.1 – Compilation times (s) of selected implementations of cryptographic primitives with (LT) and without (Ref.) computation of leak-transformers

Name	**Ref.** (s)	**LT** (s)
xxhash64	0.06	0.06
poly1305 (ref)	0.06	0.06
gimli (AVX2)	0.09	0.11
chacha20 (ref)	0.16	0.18
poly1305 (AVX2)	0.29	0.32
gimli (ref)	0.8	0.9
bash (AVX2)	2.4	2.6
blake2b	2.8	3.0
chacha20 (AVX2)	3.9	4.0
bash (ref)	6.5	7.4
curve25519	7.6	8.3

The compilation times are reported in Table 2.1. The compile-time overhead is about 10 %. The run-time overhead is zero (not shown in the table): the generated assembly is *identical* with the two versions of the compiler.

2.7 Related work

Compilation and Cryptographic Constant-Time The preservation of constant time is first considered in [Barthe et al., 2018], and proved formally (in Coq) for a toy compiler inspired by Jasmin. The proof is based on CT-simulation. CT-simulation establishes that the equality of leakage in two source executions entails equality of leakage in the corresponding target executions. In general, CT-simulation requires reasoning about four executions and is more difficult to establish than the classic simulations. By introducing the notion of structured leakage and leakage transformers, this work simplifies the overall reasoning of preserving constant time and altogether foregoing the use of CT-simulations. In [Barthe et al., 2020], they use a direct method based on proving that the leakage of target programs is identical (up to erasure) to the leakage of the corresponding source programs. They used their method to establish the preservation of constant time for a patched version of CompCert. Their work does not handle expression leakages: the first verified pass of [Barthe et al., 2020] is C#minorgen. In contrast to [Barthe et al., 2020], our work can reason about expression leakages. In [Barthe et al., 2020], they use CT-simulation in linearization pass, and they can use their direct method for passes that preserve or erase leakages. In contrast, leakage transformers do not impose restrictions on how transformations modify leakage, and the proof for linearization is straightforward. The notion of instrumented correctness introduced in this chapter is new and simplifies the proof. In [Barthe et al., 2020], each compiler pass must be independently proved correct and constantly preserved. Instead, each pass is independently proved correct in our work, and constant-time preserving proof is done only once. The FACT compiler [Cauligi et al., 2019b] transforms an information flow secure program into cryptographic constant-time programs protected against cache-based timing side channels. Motivated by Spectre and other recent micro-architectural attacks, recent works explore compiler-based mitigations under speculative execution. [Patrignani and Guarnieri, 2021] shows

(in)security of several common compiler-based mitigation techniques, including fence insertion and speculative load hardening, against these attacks. Their analysis is based on speculative variants of CCT. [Vassena et al., 2021b] designs and implements a provably sound automated compiler-based method for mitigating the BCB (bound check bypass) variant of Spectre attacks. The correctness of their approach needs to be machine-checked.

Secure compilation [Abate et al., 2018] provides a systematic classification and comparison of the different notions of secure compilation. This work is primarily foundational; it does not target any specific compiler and does not address the problem of deploying secure compilers. To address the latter problem, [Namjoshi and Tabajara, 2020] develop a translation validation framework for hyperproperties, and illustrate its application to several common optimizations.

The interaction between information-flow and compilation has been studied extensively. There exists information flow types preserving compilers, e.g., [Barthe et al., 2006, Chen et al., 2010]. In short, these works define information-flow type systems for source and target programs and show that typable source programs are transformed into typable target programs. [Sison and Murray, 2019] follow a different approach: they define an information flow type system for source programs and develop secure refinement methods to prove that typable source programs are compiled into programs that satisfy (timing-sensitive) non-interference. Their proof is mechanized using the Isabelle proof assistant.

Chapter 3

Enforcing fine-grained constant-time policies

3.1 Introduction

As discussed in Chapter 1 and Chapter 2, cryptographic constant-time is a popular programming discipline used by cryptographers to protect their libraries against timing attacks. It aims to enforce that program execution does not leak secret information. Many tools exist to enforce or check the constant-time property, but most focus on a baseline (BL) leakage model. In the BL leakage model, leakages are defined based on the assumption:

- branching statement leak values of their guards.
- memory operations leak the addresses accessed.
- nothing else leaks

But in reality, there can be more than one kind of leakage model, as there can be different architectures to accommodate different kinds of security vulnerabilities. The leakage models differ subtly based on the considered threat model, the intended target platform, and the tractability of the constant-time verification problem. The formal leakage model described in Section 2.2 of Chapter 2 focuses only on the baseline leakage model. This model is the basis of the constant-time policy, which mandates that leakage does not depend on secrets. But it does not include other attack possibilities which might arise due to time-variable operations or a more granular level of memory accesses.

However, in practice, there could be vulnerabilities due to other leakage models and constant-time cryptography is based on a family of leakage models rather than a single model. There is less development of mitigation against a more advanced leakage model. Figure 3.1 gives an overview of some of the key leakage models. For example, in contrast to the BL model, the CL leakage model leaks the cache line of the address accessed during a memory access operation instead of leaking the exact address. In contrast to the BL model, the TV leakage model leaks some extra information based on the operator instead of assuming operators to be constant time. These models are explained in detail in later sections.

Cryptography libraries like OpenSSL optimize their implementation for each leakage model, providing one implementation per leakage model. Unfortunately, the multiplicity

Program Counter (PC)	Conditionals leak their guards
Baseline (BL)	PC + Memory R/W leak addresses
Cache line (CL)	PC + Mem. accesses leak cache lines
Time-Variable (TV)	BL + TV arithmetic operators leak
TV + CL	TV arithmetic operators leak + CL

Figure 3.1 – Common leakage models

of implementations and leakage models can lead to a false sense of security. For example, here are two potential scenarios:

- A library is verified for constant-time but only for a specific leakage model, so only functions relevant to this model are checked. For instance, OpenSSL provides multiple implementations of the same crypto routines optimized for different leakage models. Therefore, when a library such as OpenSSL is verified for constant-time; it is likely that only functions relevant to the BL leakage model are checked. In contrast, no guarantee is given for functions that target the CL model, which is harder to verify.

- A library is verified for constant-time in a weaker leakage model than intended because of limitations in the verification technology. Consequently, it may still leak in the intended leakage model. For instance, some crypto routines in (the earlier version) of OpenSSL are provable and secure in the BL leakage model but are insecure in the TV model and vulnerable to practical timing attacks—such as Lucky13 [Al Fardan and Paterson, 2013].

These two scenarios reflect the existence of a potentially dangerous gap in computer-aided cryptography. This chapter explains a methodology for generalizing existing works on verification and secure compilation of the constant-time policy to cover different leakage models.

3.1.1 Contributions

This chapter proposes a methodology for formally verifying the preservation of fine-grained constant-time property. All the work is being carried out around the Jasmin compiler, and all the proofs are machine-checked using theorem prover Coq. The development is present here: https://github.com/jasmin-lang/jasmin/tree/constant-time-op. Technical contributions in nut-shell:

- The mechanized proof in the Coq proof assistant showcases that the Jasmin compiler preserves a class of fine-grained constant-time policies.

- A discovered bug in OpenSSL implementation of MEE-CBC and a proposed fix.

- A set of formal proofs that previously unverified cryptographic code is constant-time in a (non-baseline) leakage model. This includes formally verifying Langley's patch to Lucky13 and the proposed fix of OpenSSL. The proofs are carried out via embedding Jasmin's source code into EasyCrypt and using EasyCrypt's implementation of relational Hoare logic.

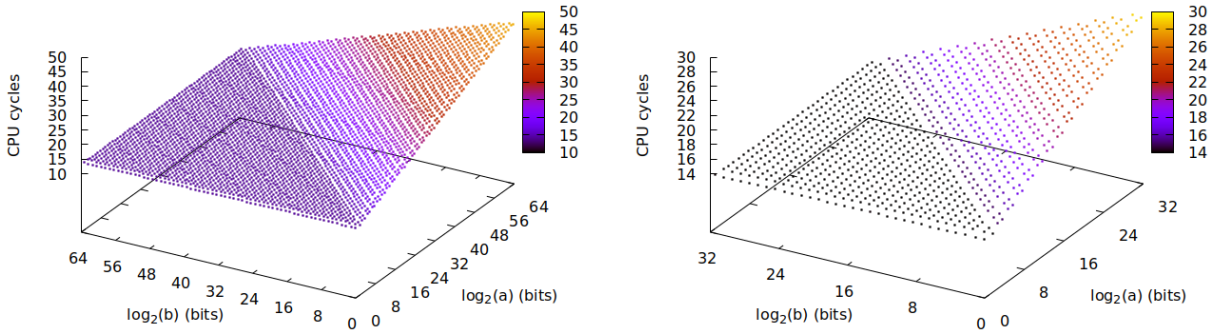


Figure 3.2 – Timing behavior of the div instruction on an x86 microprocessor (AMD EPYC 7F52) with 64-bit (left) and 32-bit (right) operands. It computes at once both quotient and modulo of its arguments a and b. Different microprocessor models exhibit different timing profiles.

3.2 Fine-grained constant-time leakage models

This section gives a brief overview of the leakage models:

- **Baseline (BL) leakage model**: This is the simplest leakage model but not the weakest. This model assumes:

  - branching statements leak values of their guards;
  - memory operations leak the address accessed;
  - nothing else leaks.

  This model is the basis of the baseline constant-time, which mandates that leakage does not depend on secrets. It captures many timing attacks in literature. The model is quite tractable and can be used effectively by cryptographic engineers to guide their implementations. Also, there is a large spectrum of automated tools [Barbosa et al., 2021] for (dis)proving that programs satisfy the baseline constant-time property, i.e., their leakage is independent of secrets. There is a recent line of work [Barthe et al., 2020, Barthe et al., 2021b] that establishes the preservation of the baseline constant-time policy for some realistic compilers.

- **Time-variable (TV) leakage model**: The baseline leakage model does not capture leakage resulting from time-variable instruction. Time-variable instructions leak information about their operands and are widely used in all modern architectures. For example, division and modulo operators are time-variable operators in x86 architecture. As shown in the Figure 3.2, the timing behavior of div instruction on x86 microprocessor varies depending on the size of operands. Consequently, a program that is constant-time in the baseline leakage model may still leak through their time-variable instructions. In specific circumstances, this leakage may be exploited to recover secret data. To address this issue, the TV leakage model strengthens the BL leakage model by making time-variable arithmetic instructions leak a function of their operands. For instance, modulo operation leaks the base-2 integer algorithm of its operands.

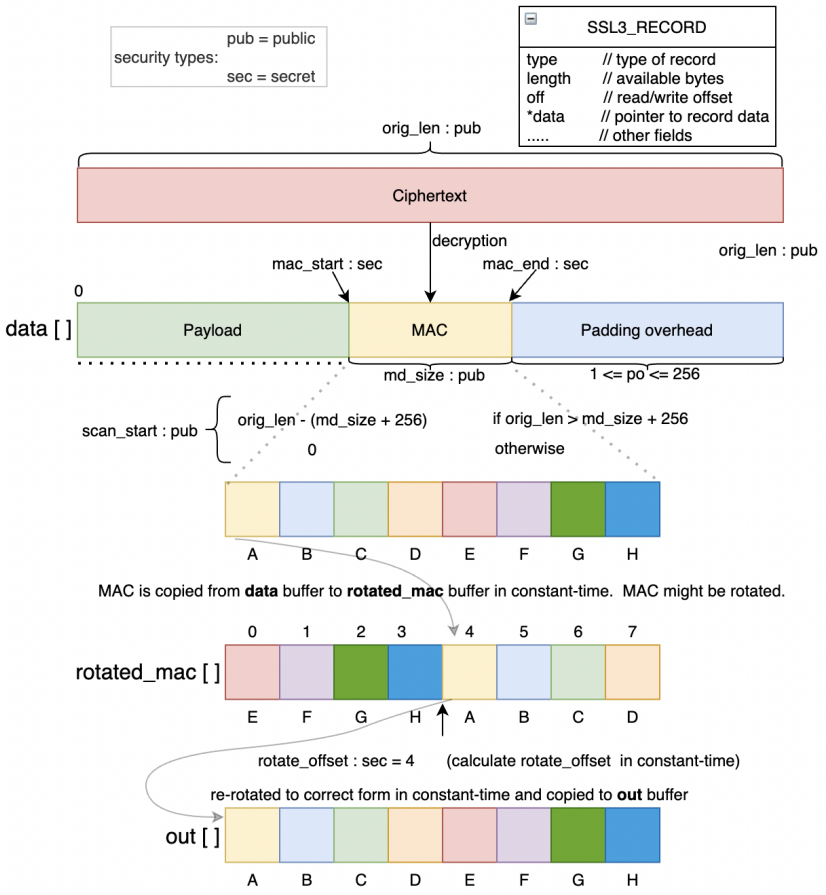


Figure 3.3 – Memory layout of a decrypted SSL3 record

- **Cache line (CL) leakage model**: The baseline leakage model assumes that the memory operations leak the address of the memory accessed. This assumption helps in protecting cache-bank conflicts (when two instructions access data from the same cache bank) but also makes programs harder to write and less efficient. As a consequence, cryptographic libraries often provide the implementation for the CL leakage model. In this leakage model, memory accesses leak the cache line of the addresses being accessed. In reality, a memory is bigger in size as compared to a cache. Data are moved from memory to cache in blocks (cache line) that help in spatial locality. We bring data that the program requires to the cache and data that is close to them. The typical cache block sizes are 32 bytes and 64 bytes. In this work, memory accesses leak the address divided by the length of the cache line (32, 64, ...).

3.3 Motivating Example

MEE-CBC (MAC-then-Encode-then-CBC-Encrypt) is an authenticated encryption scheme used in the TLS 1.2 cipher-suite. Lucky13 [Al Fardan and Paterson, 2013], a sophisticated timing attack against several open-source cryptographic libraries supporting MEE-CBC. In response to the attack, several libraries developed new implementations of MEE-CBC that loosely follow the constant-time programming discipline. However, proving that these

```
1 /* public: md_size, scan_start */
2 /* secret: mac_start */
3 fn rotate_offset_BL(reg u32 md_size, mac_start, scan_start) ⟶ reg u32 {
4   reg u32 rotate_offset;
5   rotate_offset = mac_start;
6   rotate_offset -= scan_start;
7   rotate_offset = rotate_offset % md_size;
8   return rotate_offset;
9 }
```

```
1 /* pre: 16 ≤ md_size ≤ 64  ∧  0 ≤ mac_start - scan_start < 256 */
2 /* public: md_size, scan_start */
3 /* secret: mac_start */
4 fn rotate_offset_TV(reg u32 md_size, mac_start, scan_start) ⟶ reg u32 {
5   reg u32 div_spoiler;
6   reg u32 rotate_offset;
7   div_spoiler = md_size;
8   div_spoiler <<= 23;
9   rotate_offset = mac_start;
10  rotate_offset -= scan_start;
11  rotate_offset += div_spoiler;
12  rotate_offset = rotate_offset % md_size;
13  return rotate_offset;
14 }
```

Figure 3.4 – Two implementations computing the rotation offset

implementations are constant-time according to the baseline leakage model is not always possible, as some implementations use time-variable instructions or optimize code in a way that degrades security to a weaker model.

CBC decoding Figure 3.3 provides a pictorial description of inputs to the function ssl3_cbc_copy_mac. This function aims to extract MAC from the record in constant-time. It copies the bytes representing the MAC of size md_size from the ssl3 record to an out buffer. This function takes four arguments: 1. rec pointer to an ssl3 record structure, which contains a pointer to the data buffer and a secret length field which denotes the length after removing padding overhead; 2. out buffer, where the extracted message authentication code (MAC) will be copied to; 3. md_size denoted the MAC size and 4. orig_len representing the record's original length, including padding. Initially, ciphertext is decrypted into a data buffer containing the payload, the MAC, and finally, the padding overhead. Unfortunately, copying the MAC is not straightforward as mac_start and mac_end are secret. Also, we do not want to scan the whole data buffer for efficiency reasons.

However, we know that orig_len is public and that the padding overhead is at most 256 bytes long. Therefore, it suffices to begin copying from scan_start $= \max(0, \text{orig\_len} - (\text{md\_size} + 256))$. The copied MAC might yield byte-wise rotated values. Therefore, to recover the original MAC, one must compute the offset and perform the rotation relative to this offset in constant-time.

Computing the rotation offset Figure 3.4 shows the functions rotate_offset_BL and rotate_offset_TV for computing the offset. For readability, information about the se-

curity types of the variables is present at the top of the functions. The first function rotat_offset_BL is a simple arithmetic program that computes the offset as (mac_start − scan_start) % md_size. This function is trivially constant-time in the baseline (BL) leakage model, as it does not branch nor perform memory accesses on any secret data. However, the variable mac_start is secret; hence the function is not constant-time in the time-variable (TV) leakage model. The reason behind this is the usage of the modulo operator while calculating the rotate_offset and we know that the modulo operator is not constant-time in the TV leakage model.

The second function rotate_offset_TV is a Jasmin implementation of Langley's original fix to Lucky13 and is constant-time in the TV leakage model. This is achieved by making the first argument large enough, by first setting div_spoiler = md_size << 23 and by setting rotate_offset to div_spoiler + (mac_start − scan_start). Note that the denominator (md_size) is a public value, so only the numerator (rotate_offset) needs to be patched in such a way. Under the assumptions on the parameters set by the precondition, we can prove that the change does not affect the instruction result and makes leakage independent of mac_start. As discussed in the prior section, module operation leaks the base-2 integer algorithm of its operands. Hence, the leakage is equal to the following: $\log_2(\mathsf{md\_size} \times 2^{23})$, $\log_2(\mathsf{md\_size})$ and hence only depends on public values. To justify the above claim, note that by definition of the leakage model, the first component of the leakage is $\log_2(\mathsf{md\_size} \times 2^{23} + (\mathsf{mac\_start} - \mathsf{scan\_start})) = \log_2(\mathsf{md\_size} \times 2^{23})$. The equality above follows from the precondition.

Rotating the MAC Figure 3.5 presents the two Jasmin implementations for rotating the MAC. The first implementation performs a nested loop. Before the loop ro is set to (−rotate_offset) mod mid_size (line 9). For each i, the inner loop writes to out buffer; if j equals rotate_offset, the new value obtained at line 15 is written, else the old value is rewritten. The selection is done at line 16 using a conditional assignment and will be compiled using a constant-time CMOVcc instruction. Line 20 computes (rotate_offset + 1) mod md_size in constant-time.

This implementation is constant-time in the baseline leakage model since the branching statement (the while loop) only depends on public data (i, j and md_size), and similarly for memory and array accesses, which only depend on i, j and out. The drawback is that the implementation performs a nested loop, so the copy is quadratic in md_size.

In contrast, the second implementation rotated_mac_CL performs a single loop. The leakage corresponding to the branching (while loop) instruction depends only on the public data (i and md_size). The memory access in line 13 only depends on public data (out and i) and hence does not leak secrets. However, the instruction at line 12 leaks the secret index rotate_offset in the first iteration. Hence, the function is not constant-time in the baseline leakage model. On the other hand, rotate_mac_CL is constant-time in the CL leakage model assuming that rotated_mac is 64-byte aligned memory pointer, and the MAC data will fit in a cache line–these assumptions correspond to the precondition. This is because in this model, the instruction leaks: ⌊(rotated_mac + ro) / 64⌋. Since rotated_mac mod 64 = 0 and 0 <= ro <= md_size <= 64, it follows:

$$\lfloor(\mathsf{rotated\_mac} + \mathsf{ro}) \;/\; 64\rfloor = \lfloor \mathsf{rotated\_mac}/64\rfloor.$$

Since the value of the pointer rotated_mac is public, the leakage does not depend on secrets.

```
1 /* public: md_size, out */
2 /* secret: rotate_offset */
3 fn rotate_mac_BL (reg u32 md_size, rotate_offset,
4                    reg u64 out, stack u8[128] rotated_mac) {
5   reg u64 i, j;
6   reg u32 old, new, zero, ro;
7   zero = 0;
8   // ro = (-rotate_offset) % md_size
9   ro = opp_mod(rotate_offset, md_size);
10  i = 0;
11  while (i < md_size) {
12    j = 0;
13    while (j < md_size) {
14      old = (32u) (u8)[out + j];
15      new = (32u) rotated_mac[(int) i];
16      new = old if j ≠ ro;
17      (u8)[out + j] = new;
18      j += 1;
19    }
20    ro += 1; ro = zero if md_size <= ro;
21    i += 1;
22  }
23 }
```

```
1 /* rotated_mac % 64 = 0 ∧ 0 ≤ rotate_offset ≤ mdsize ≤ 64*/
2 /* public: md_size, out, rotated_mac */
3 /* secret: rotate_offset */
4 fn rotate_mac_CL (reg u32 md_size, rotate_offset,
5                    reg u64 out, rotated_mac) {
6   reg u8 new;
7   reg u64 i, zero, ro;
8   zero = 0;
9   ro = (64u) rotate_offset;
10  i = 0;
11  while (i < md_size) {
12    new = (u8)[rotated_mac + ro];
13    (u8)[out + i] = new;
14    ro += 1; ro = zero if md_size <= ro;
15    i += 1;
16  }
17 }
```

Figure 3.5 – Two implementations of MAC rotation

OpenSSL bug Figure 3.6 shows the code used by OpenSSL in the CL leakage model to accommodate CPUs with 32-byte cache lines. Here, rotated_mac is a 64-byte aligned buffer (i.e., its address is 64q for some q) and assumed that this data would fit into two 32-byte cache lines. The developer has added a dummy unoptimizable first access to load rotate_mac[rotate_offset^32] and then the actual load to ensure that they touch both lines in every iteration to make it look like constant-time. Since the volatile keyword is added in line 5, the compiler will not remove an unused load operation in the deadcode elimination pass. The comment in the line says it will touch the “second line". However, this is incorrect, and it touches “other line". According to the 32-byte cache line model, the address divided by 32 is leaked. Thus, for the two load operations, when 0 <=

```
#if defined(CBC_MAC_ROTATE_IN_PLACE)
j = 0;
for (i = 0; i < md_size; i++) {
  /* in case cache-line is 32 bytes, touch second line */
  ((volatile unsigned char *)rotated_mac)[rotate_offset ^ 32];
  out[j++] = rotated_mac[rotate_offset++];
  rotate_offset &= constant_time_lt_s(rotate_offset, md_size);
}
#else ...
```

Figure 3.6 – Buggy C implementation of rotate_offset

rotate_offset < 32, values $2q+1$ and $2q$ are leaked (i.e., the second cache line is touched first and then the first line) and when $32 <=$ rotate_offset < 64, values $2q$ and $2q+1$ are leaked (i.e., the first line is touched first and then the second). This has been reported to OpenSSL, and a later section discusses a fix.

3.4 Fine-Grained Policies in Jasmin

This section presents the approach taken to extend the baseline (BL) leakage model (discussed in Chapter 2) to include other leakage models in the context of Jasmin.

3.4.1 Syntax and Semantics

The syntax of Jasmin language is presented in Figure 1.11. To enable a wide range of policies corresponding to various hardware and adversary capabilities, the definition of leakage is layered in two stages. The first layer corresponds to the syntactic shape of the leakage. The baseline leakage model's leakage is presented in Figure 2.5 of Chapter 2. This structure is fixed and corresponds to the "BL" leakage model. The leakage model can be fine-tuned by leaking more or less precise values. The second layer consists of two parameters: $\mathcal{A}^{\diamond}(v_1, \ldots, v_n)$ defines the leakage produced by the evaluation of the operator $\diamond$ applied to arguments $(v_1, \ldots, v_n)$; and $\mathcal{M}(p)$ defines the leakage produced by memory access at address p.

Figure 2.4 in Chapter 2 presents the rules of instrumented semantics of the Jasmin language. The rules represent the baseline leakage (BL) model. We updated the instrumented semantics to make the leakage model more fine-grained. There are three kinds of judgments: 1. $e \downarrow^{s}_{\ell_e} v$ corresponds to the evaluation of expression e in state s producing value v and leakage ℓ_e; 2. $d := v \downarrow^{s}_{\ell_e} s'$ corresponds to the assignment of value v into the left-value d in state s producing the updated state s' and leakage ℓ_e; 3. $i : s \Downarrow_{\ell} s'$ corresponds to the execution of instruction i starting in state s and ending in state s' while producing leakage ℓ. We extend the semantics of operators and memory accesses with the notation $s \Downarrow_{\ell}^{\mathcal{A},\mathcal{M}} s'$ to make explicit the dependency of the semantic on the leakage model. For a given program p, an initial state s is said to be *safe* when there exists an execution from this state, ending in some final state s' and producing some leakage ℓ. The point to note here as compared to instrumented semantics presented in Chapter 2 are: 1. memory accesses leak according to $\mathcal{M}$; 2. the leakage for arithmetic operators correspond to the leakage of the evaluation of their arguments followed by one of the computations of the operator (as defined by $\mathcal{A}$). The parameter $\mathcal{M}$ defines the leakage for memory load and

Parameters:

$$\mathcal{M}(v), \mathcal{A}^{\mathsf{op}}(v_1, \ldots, v_n) \in \ell_e$$

Expression semantics:

$$\frac{e \downarrow^s_{\ell_e} p}{*e \downarrow^s_{(\ell_e, \mathcal{M}(\mathsf{p}))} s[p]}$$

$$\frac{e_i \downarrow^s_{\ell^i_e} v_i \quad \mathsf{op}(v_1, \ldots, v_n) = v \quad \mathcal{A}^{\mathsf{op}}(v_1, \ldots, v_n) = \ell_e}{\mathsf{op}(e_1, \ldots, e_n) \downarrow^s_{((\ell^1_e, \ldots, \ell^n_e), \ell_e)} v}$$

Assignment semantics:

$$\frac{e \downarrow^s_{\ell_e} p}{*e := v \downarrow^s_{(\ell_e, \mathcal{M}(\mathsf{p}))} s\{p \leftarrow v\}}$$

$$\frac{e \downarrow^s_{\ell_e} p \qquad s[p] = v}{x := *e \downarrow^s_{(\ell_e, \mathcal{M}(\mathsf{p}))} s\{x \leftarrow v\}}$$

Figure 3.7 – Instrumented fine-tuned semantics.

store as they involve memory accesses.

The parameters $\mathcal{A}$ and $\mathcal{M}$ play an important role in generating a generic leakage model that can be instantiated by different kinds of leakage models.

Generalization of leakage models in Coq The type class LeakOp implements the leakage class, which includes the parameters $\mathcal{A}$ and $\mathcal{M}$. This class can be instantiated by giving witness to the function $\mathcal{A}$ and $\mathcal{M}$ and can represent different leakage models like BL, TV, or CL.

```
Class LeakOp :=
  { A : signedness → forall (sz:wsize), word sz → word sz → word sz → leak_e;
    M : word Uptr → word Uptr; }.
```

Figure 3.8 – Leakage Class

For example, the leakage obtained during the execution of division and modulo is now based on the function $\mathcal{A}$ in the type class LeakOp. It uses a generalized instance of function $\mathcal{A}$ called div_{leak}. Given a signdness parameter s, size sz and words hi, lo and div of size sz, div_{leak} computes the leakage $\mathcal{A}\ s\ hi\ lo\ div$. The operators like division and modulo use div_{leak} to compute the leakage.

3.4.2 Instances of leakage class

This section presents various instances of the leakage class.

Baseline leakage model In the BL model, arithmetic operations produce an $\bullet$ leakage and memory accesses leak the whole address. This can be expressed by instances $\mathcal{A}_{BL}$ and $\mathcal{M}_{BL}$.

$$\mathcal{A}^{\diamond}_{BL}(a) = \bullet$$
$$\mathcal{M}_{BL}(p) = p.$$

Here a is the set of arguments provided to the operator, and p is the accessed memory address.

Time-variable leakage model The time-variable leakage model captures the leakages generated during the execution of division and modulo operators. This can be expresses by instances $\mathcal{A}_{TV}$ and $\mathcal{M}_{TV}$.

$$\mathcal{A}^{\div}_{TV}(a,b) = (log_2(a), log_2(b)) \quad \mathcal{A}^{\%}_{TV}(a,b) = (log_2(a), log_2(b))$$
$$\mathcal{M}_{TV}(p) = p.$$

The point to note here is the leakages are generated only for division and modulo. For other operators, their execution still produced $\bullet$ leakage and is modeled as constant-time. For example, $\mathcal{A}^{\times}_{TV}(a,b) = \bullet$. This work only focuses on making the division and modulo operator (other operators are assumed to be constant-time) constant-time as Jasmin only supports x86 architecture. However, the current methodology can be extended to support other operators. For example, once Jasmin is extended to support ARM architecture, the current methodology can be extended to remove the assumption that multiplication is constant time and add leakages corresponding to the multiplication operator.

Cache-line leakage model The cache-line leakage model assumes that the truncated address is leaked instead of leaking the whole address.

$$\mathcal{M}_{CL_{64}}(p) = \lfloor p/64 \rfloor$$
$$\mathcal{M}_{CL_{32}}(p) = \lfloor p/32 \rfloor.$$

Here 64 and 32 are the (byte) granularity of cache lines.

Combining models We can combine these leakage instances to yield combinations of different models like the TV + CL leakage model, where operations like division and modulo leak a function of their arguments, and memory accesses leak the truncated address.

3.5 Leakage transformers

As discussed in Section 2.3 of Chapter 2, the Jasmin compiler is instrumented to produce leakage transformers. To model the translation of leakages corresponding to fine-grained leakage models into the target leakage, we need to consider the extra leakages produced during the evaluation of an operation and the different kinds of leakages produced during memory access (depending on the model it is targeting). Hence, the semantics of some of the leakage transformers are slightly changed to incorporate the fine-grained leakage model. Here, the set of leakage transformers that need to be adapted for supporting verification of fine-grained leakage models is only discussed (the rest of the leakage transformers have the same syntax and semantics as presented in Chapter 2).

Leakage transformers for expressions:

$$\frac{[\![\tau_s]\!]_s^{v_{\mathsf{sp}}} = p}{[\![C(\tau_s)]\!]_e^{\ell_e} = *\mathcal{M}(p)} \qquad \frac{[\![\tau_s[i/x]]\!]_s^{v_{\mathsf{sp}}} = p}{[\![I(x \mapsto \tau_s)]\!]_e^{[i]} = *\mathcal{M}(p)}$$

Figure 3.9 – Semantics for adapted leakage transformers

In the case of leakage transformers for expressions, the semantics of $C(\tau_s)$ and $I(x \mapsto \tau_s)$ are updated (presented in Figure 3.9) as these leakage transformers are involved during the stack allocation pass to generate fresh leakage corresponding to memory load and store. After the stack allocation passes, the transformed leakage involves leaking the memory address; hence, it should be computed using the function $\mathcal{M}$.

Next, there are changes in the leakage transformers related to transforming the leakage associated with operators. As discussed in Section 2.3 of Chapter 2, the leakage transformer concerning operations is of the form $(\tau_e; \ldots; \tau'_e)$ where $\tau_e \ldots \tau'_e$ are used to transform the leakage associated with the operands. But we know that the operators can also leak to enforce fine-grained leakage models. In this work, only division and modulo leak something extra than $\bullet$ leakage, and the rest of the operators leak $\bullet$ leakage; we need an extra leakage transformer in the set to transform the leakage associated with the operators.

```
36 Definition snot (e: pexpr) : (pexpr × leak_e_tr) :=
37  match e with
38  | Pbool b ⇒ (Pbool(negb  b), remove)
39  | Papp1 Onot e' ⇒ (e', π1 ∘ π1)
40  | _ ⇒ (Papp1 Onot e, id)
41 end.
```

Figure 3.10 – Pseudo-code of the constant propagation

Figure 3.10 presents a simple example featuring the semantics of not operation. The case of applying not operation twice is discussed here (not(not e) = e) as the operation is simplified further by the constant-propagation pass. In the case of the application of operation *Onot* on $Onot(e')$, the operation *Onot* should also leak according to the fine-grained leakage model. Hence in the constant propagation pass, the leakage transformer produced is $\pi_1 \circ \pi_1$ because we need to compute leakage of e' from the leakage of $Onot(Onot(e'))$. At the source level, the leakage associated with $Onot(Onot(e'))$ is $((\ell'_e, \bullet), \bullet)$ but the target instruction is e'; hence we need to extract the leakage associated with e' (using the leakage transformer $\pi_1 \circ \pi_1$) and ignore the leakage associated with *Onot*. In contrast, the leakage transformer produced in this case in the work described in Chapter 2 is just id as there was no leakage associated with *Onot* at the source level (in the BL model, operators do not leak). This example shows that the leakage transformers produced during various compiler pass needs to be adjusted to consider the fine-grained leakage model.

The point to note here is that the development in this work focuses on division and modulo operators (leak more than empty leakage depending on the target leakage model). Still, it can be extended to incorporate more operators. For example, the support of ARM architecture in Jasmin will require making the multiplication operator constant-time. The execution time of multiplication opcode in ARM Cortex-M3 depends on the size of one or both operands. The development and proof of preservation of constant time (explained

in Section 3.7) in this work are generic. They are carried out based on the leakage class that can be extended to incorporate more operators to leak something meaningful, not just empty leakage.

3.6 Instrumented correctness

The Jasmin compiler is formally proved to be functionally correct and also includes machine-checked proofs for the correctness of the leakage transformers as discussed in Chapter 2. The proofs for the correctness of the leakage transformers need to be updated to incorporate the fine-grained leakage models.

Theorem 3 (Instrumented correctness for fine-grained leakage models)**.**

$$p : s \Downarrow_{\ell}^{\mathcal{A},\mathcal{M}} s' \implies \overline{p} : s \Downarrow_{[\![\tau]\!]^{\ell}}^{\mathcal{A},\mathcal{M}} s'.$$

The implementation of the compiler does not depend on the instance of the leakage model. It just needs to be parameterized over the leakage class. The correctness proof is done independently for each compiler pass, where the proof for the correctness of leakage transformers is added to them. The proof is adapted to include the new changes in the leakage transformers and leakages as discussed in Section 3.5. Overall, the proof of instrumented correctness boils down to proving the functional correctness of the leakage transformers.

3.7 Preservation of fine-grained constant-time policies

Definition 3 (Constant-time)**.** *A program p is* constant-time *w.r.t. the indistinguishability relation $\sim$ when the following holds:*

$$\forall s_1\ s_2\ \mathcal{A}\ \mathcal{M}, s_1 \sim s_2 \implies \exists s_1'\ s_2'\ \ell, p : s_1 \Downarrow_{\ell}^{\mathcal{A},\mathcal{M}} s_1' \wedge p : s_2 \Downarrow_{\ell}^{\mathcal{A},\mathcal{M}} s_2'.$$

The definition of constant-time is similar to as used in Chapter 2, but it needs to be parameterized by $\mathcal{A}\ \mathcal{M}$. The indistinguishability relation for source and target states is the same as defined in Section 2.5 of Chapter 2.

Theorem 4 (Constant time preservation)**.** *Given a source program p that is constant time w.r.t. $\sim$ and $\mathcal{A}\ \mathcal{M}$, if the Jasmin compiler succeeds and produces a target program $\bar{p}$, then the target program is constant time w.r.t. $\bar{\sim}$ and $\mathcal{A}\ \mathcal{M}$.*

This theorem entails that the Jasmin compiler preserves fine-grained constant-time policies. The proof is similar to as explained in Section 2.5 of Chapter 2.

3.8 Deductive Enforcement of fine-grained constant-time policies

We need to instantiate the leakage models to prove the fine-grained constant-time policies at the source level. The Jasmin program is extracted to EasyCrypt. EasyCrypt is used to prove functional correctness and cryptographic security of Jasmin programs. The EasyCrypt development is made parametric in the leakage models to prove the constant-time property at the source level.

3.8.1 Example of instances of the leakage model

This section presents a development of the instances (as discussed in Section 3.4.2) of the leakage class to model BL, TV, and CL models. It describes one way to instantiate the leakage class, but there can be other ways to define the instance depending on the developer's need and purpose.

Focus on BL and TV model instances

$$\div_{leak_{kind}} \ ::= \ \mathsf{dlk_{none}} \mid \mathsf{dlk_{num_{log}}}$$

$\div_{leak_{kind}}$ is a type describing the two constructors to represent the leakages obtained while evaluating the division or modulo. $\mathsf{dlk_{none}}$ represents the BL leakage model where the division and modulo does not leak and $\mathsf{dlk_{num_{log}}}$ represents the TV leakage model.

$$\mathsf{build}_{\div_{\mathsf{leak}}}(dlk, s, sz, h, l, div) = \begin{cases} \bullet & dlk = \mathsf{dlk_{none}} \\ [log_2(|h \times sz + lo| + 1)] & s = Unsigned \wedge dlk = \mathsf{dlk_{num_{log}}} \\ [log_2(|h \times sz + lo| + 1)] & s = Signed \wedge dlk = \mathsf{dlk_{num_{log}}} \end{cases}$$

$\mathsf{build}_{\div_{\mathsf{leak}}}$ is a function to compute the leakage based on the variant kind dlk. In the case where $dlk = \mathsf{dlk_{none}}$, it computes $\bullet$ leakage because $\div$ is not supposed to leak in this model (BL). In the case where $dlk = \mathsf{dlk_{num_{log}}}$, it computes the leakage by taking the logarithm of the operand (the operand is computed using the high and low bits and uses the signdness). The div instruction in the X86 instruction set always divides the 64 bits value across $\mathsf{EDX} : \mathsf{EAX}$ (low 32 bits are in EAX and high 32 bits are in EDX) by value, and the result is stored in EAX and the remainder is stored in EDX. The semantics div in Jasmin represents the same notation where the dividend is represented using high (h), low (lo) bits and size (for example, sz equals 256 for the case of 1 byte). To avoid the case of computing the logarithm of 0, 1 is added to the operand.

Focus on BL and CL model instances

$$\mathsf{mem_{leak_{kind}}} \ ::= \ \mathsf{mlk_{full}} \mid \mathsf{mlk}_{\div 32} \mid \mathsf{mlk}_{\div 64}$$

$\mathsf{mem_{leak_{kind}}}$ is a type describing three constructors to represent the possible leakages obtained while accessing a memory location.

$$\mathsf{build}_{\mathsf{mem_{leak}}}(mlk, p) = \begin{cases} *p & mlk = \mathsf{mlk_{full}} \\ *(p \div 32) & mlk = \mathsf{mlk}_{\div 32} \\ *(p \div 64) & mlk = \mathsf{mlk}_{\div 64} \end{cases}$$

$\mathsf{build}_{\mathsf{mem_{leak}}}$ is a function to compute the leakage for memory access based on the memory address and the target leakage model. For example, if the cache-line is 32-byte aligned and the target leakage model is CL, the leakage obtained while evaluating memory access p is $*(p \div 32)$.

Using these two functions, we can build a generalized model that is an instance of the class LeakOp.

$$build_{model}(dlk, mlk) ::= \{|\mathcal{A} := \mathsf{build}_{\div_{\mathsf{leak}}}(dlk); \mathcal{M} := \mathsf{build}_{\mathsf{mem_{leak}}}(mlk)|\}$$

In EasyCrypt, an abstract theory represents the class in Coq (resembles the leakage class LeakOp). A theory in EasyCrypt is used to group definitions that can be cloned to represent different kinds of specialization. Theory cloning is used to instantiate a theory with a particular implementation (similar to instances of a class in Coq).

As discussed in Section 3.2, in the BL leakage model, the operators do not leak, and the memory accesses reveal the full address. To prove any implementation against the BL leakage model, the theory LeakOp should be cloned to an instance represented as $\{|\mathcal{A} := \mathsf{build}_{\div_{\mathsf{leak}}}(\mathsf{dlk}_{\mathsf{none}}, s, sz, h, l, div); \mathcal{M} := \mathsf{build}_{\mathsf{mem}_{\mathsf{leak}}}(\mathsf{mlk}_{\mathsf{full}}, *p)|\}$. Similarly, to prove an implementation against the TV leakage model, the theory LeakOp should be cloned to an instance $\{|\mathcal{A} := \mathsf{build}_{\div_{\mathsf{leak}}}(\mathsf{dlk}_{\mathsf{num}_{\mathsf{log}}}, s, sz, h, l, div); \mathcal{M} := \mathsf{build}_{\mathsf{mem}_{\mathsf{leak}}}(\mathsf{mlk}_{\mathsf{full}}, *p)|\}$. This cloning mechanism gives freedom to the developer to define their instances depending on their need and purpose. In this work, the instances are defined to target fine-grained leakage models and how they can help mitigate against timing-based attacks based on time-variable operators and cache-line.

3.9 Instrumentation of programs in EasyCrypt

The instrumentation of programs is generic and transforms every source program p into an extracted program in EasyCrypt represented as $[p]$. The semantics of p and $[p]$ are same except $[p]$ accumulates the leakages generated by the evaluation of p in a special fresh program variable named leak.

The instrumentation relies on expressions to capture the leakages in a fresh variable called leak. For verifying fine-grained constant-time policies at the source level, the expressions defining the leakages are extended to add functionalities of $\mathcal{M}$ and $\mathcal{A}$. e represents the expressions. $\mathcal{A}^{\diamond}(e, \ldots, e)$ resembles the leakage associated with division or modulo. For example, depending on the division operators (unsigned or signed) present in x86 architecture, $\mathcal{A}^{\diamond}(e, \ldots, e)$ computes the leakage using the operands. $\uplus$ is used for the concatenation of leakages.

The instrumentation of expressions $\{e\}$ is present in Figure 3.11. The instrumentation computes a new expression that will evaluate the leakage generated by e: $\mathsf{if}\ e : s \downarrow_{\ell_e}^{\mathcal{A},\mathcal{M}} v\ \mathsf{then}\ \{e\} : s \downarrow \ell_e$. The expressions like constant, boolean, variable, and global variables produce no leakage. For array accesses $a[e]$, the leakage contains the index e and the leakage $\{e\}$ generated during the evaluation of e. For memory accesses $*e$, the leakage contains the address computed using $\mathcal{M}$ and also leaks $\{e\}$ generated during the evaluation of e. In the case of operators, their leakage contains the leakage of their arguments and the leakage due to the operation. The conditional expression leaks $\{e\}$ (leakage obtained during evaluation of condition) and also the leakage obtained during the evaluation of true ($\{e_{tt}\}$) or false ($\{e_{ff}\}$) branch. The point to note in the case of conditional expression is it does not leak the value of the condition because the Jasmin compiler will compile it using a conditional move instruction, which is constant-time. The leakages associated with the instrumentation of left-values are similar to expressions.

Extended expressions:

$$\mathsf{Expr} ::= \ldots \mid \mathsf{M}(e) \mid \mathsf{A}^{\diamond}(e, \ldots, e) \mid \emptyset \mid e \uplus e$$

Translation of expressions:

$$\begin{aligned}
\{c\} = \{x\} = \{b\} = \{a[n]\} = \{g\} &= \emptyset \\
\{*e\} &= \mathsf{M}(e) \uplus \{e\} \\
\{a[e]\} &= e \uplus \{e\} \\
\{\diamond(e_1, \ldots, e_n)\} &= \mathsf{A}^{\diamond}(e_1, \ldots, e_n) \uplus \{e_1\} \uplus \ldots \uplus \{e_n\} \\
\{\mathsf{if}\ e\ \mathsf{then}\ e_{tt}\ \mathsf{else}\ e_{ff}\} &= \{e\} \uplus \{e_{tt}\} \uplus \{e_{ff}\}
\end{aligned}$$

Translation of left values:

$$\begin{aligned}
\{x\} &= \emptyset \\
\{*e\} &= \mathsf{M}(e) \uplus \{e\} \\
\{a[e]\} &= e \uplus \{e\}
\end{aligned}$$

Translation of instructions:

$$\begin{aligned}
[d := e] &= \mathsf{leak} := \{d\} \uplus \{e\} \uplus \mathsf{leak}; d := e \\
[\mathsf{if}\ e\ \mathsf{then}\ c_{tt}\ \mathsf{else}\ c_{ff}] &= \mathsf{leak} := e \uplus \{e\} \uplus \mathsf{leak}; \\
&\quad\ \mathsf{if}\ e\ \mathsf{then}\ [c_{tt}]\ \mathsf{else}\ [c_{ff}] \\
[\mathsf{while}\ e\ \mathsf{do}\ c] &= \mathsf{leak} := e \uplus \{e\} \uplus \mathsf{leak}; \\
&\quad\ \mathsf{while}\ e\ \mathsf{do}\ [c]; \mathsf{leak} := e \uplus \{e\} \uplus \mathsf{leak} \\
[i_1; \ldots; i_n] &= [i_1]; \ldots; [i_n]
\end{aligned}$$

Figure 3.11 – Program instrumentation with explicit leakage

Figure 3.11 provides the instrumentation for the instructions $[c]$. The instrumentation of an assignment instruction $[d := e]$ is a sequence of two assignments; the first extends the variable leak with $\{d\}$ and $\{e\}$ and then do the assignment d := e. For conditional instructions, the leak variable leak is extended with the value of condition e, the leakage obtained during the evaluation of condition i.e., $\{e\}$. The instrumentation of the while loop works in a similar manner as if-else. In case of while, the leak is updated once before the loop (to capture the leakage of conditional for the first iteration) and then at the end of each loop iteration.

3.9.1 Correctness of instrumentation

This section explains the correctness of the instrumentation.

Theorem 5 (Correctness of the instrumenation)**.** *For all program c, if its evaluation starting from a state s generates a leakage ℓ and a state s', then the evaluation of its*

instrumentation $[c]$ *starting from the state* s *extended with* ℓ_0 *for the variable* leak *leads to the state* s' *extended with* $\ell \uplus \ell_0$ *for the variable* leak*:*

$$\forall c\ s\ s'\ \ell\ \ell_0\ \mathcal{A}\ \mathcal{M}, c : s \Downarrow_\ell^{\mathcal{A},\mathcal{M}} s' \implies [c] : s + \{\mathsf{leak} \leftarrow \ell_0\} \Downarrow^{\mathcal{A},\mathcal{M}} s' + \{\mathsf{leak} \leftarrow \ell \uplus \ell_0\}$$

where the notation $s + \{\mathsf{leak} \leftarrow \ell_0\}$ *represents the state* s *extended with a fresh variable* leak *and its associated value* ℓ_0*.*

The above theorem shows that the instrumented program correctly accumulates the leakage in the variable leak. As the semantics in EasyCrypt is a one-to-one mapping of the semantics of Jasmin, the proof boils down to computing the leakage correctly using the rules stated in Figure 3.11.

3.9.2 Fine-grained constant-time policies for the extracted programs in EasyCrypt

Fine-grained constant-time policies are 2-safety properties and can be enforced using relational program logics, such as Relational Hoare Logic [Benton, 2004]. Relational Hoare Logic naturally captures the information flow properties, which is enough for representing the properties of interest presented in this chapter. Other kinds of logic, like probabilistic Relational Hoare Logic, derive claims about the probability of events occurring during a program's execution (also supported in EasyCrypt [Barthe et al., 2011b]) are not necessary to prove claims about constant-time policies. Hence, this chapter only focuses on Relational Hoare Logics. The Relational Hoare Logic manipulates judgments of the form:

$$c_1 \sim c_2 : \varphi \implies \psi \tag{3.1}$$

where c_1, c_2 are programs and φ is a relational pre-condition and ψ is a relational post condition. Both the pre-condition and post-condition are interpreted as relations over the states of the two programs. Concretely, the interpretation of such a judgment is:

$$\forall s_1, s_1', s_2, s_2',\ \begin{cases} s_1\ \varphi\ s_2 \\ c_1 : s_1 \Downarrow s_1' \\ c_2 : s_2 \Downarrow s_2' \end{cases} \implies s_1'\ \psi\ s_2'$$

In other words, if we start the evaluation of c_1 and c_2 in two states that are in relation for the pre-condition $(s_1\ \varphi\ s_2)$, the final states will be in relation for the post-condition $(s_1'\ \psi\ s_2')$. Note that the validity of a judgment is implicitly parameterized by an interpretation of operators. $\mathcal{A}, \mathcal{M} \models c_1 \sim c_2 : \varphi \implies \psi$ reflects a judgment is valid w.r.t. an interpretation of operators and memory leakage.

Theorem 6 (Fine-grained constant-time, relationally)**.** *If* $\mathcal{A}, \mathcal{M} \models [c] \sim [c] : \varphi \wedge =_{\{leak\}} \implies =_{\{leak\}}$ *then [c] is constant-time program with respect to* $\mathcal{A}$*,* $\mathcal{M}$ *and* φ*.*

The above theorem shows that relational Hoare logic can verify fine-grained constant-time policies. The proof is a direct consequence of theorem 5 and of the interpretation of relational Hoare logic. It follows that any soundproof system for relational Hoare logic can be used for proving fine-grained constant-time.

3.10 Description of the various proofs done using EasyCrypt

This section describes a high-level idea of proving fine-grained constant-time policies for various examples. As discussed in Section 3.9, the extraction to the EasyCrypt model is generic, and the proof can be instantiated to target a particular leakage model.

3.10.1 Proving ssl3_cbc_copy_mac

This section presents a detailed explanation of the proof of the ssl3_cbc_copy_mac implementation. The function mainly comprises rotated_offset and rotate_mac. Since there are two implementations for each one (rotate_offset_BL, rotate_offset_TV, rotate_mac_BL, rotate_mac_CL), there are a total of four implementations presented in Section 3.3.

Focusing on implementation to be proved for BL leakage model The program at the top of Figure 3.5 presents the functions rotate_mac_BL. The function is constant-time in the BL leakage model (and also in the CL model). Using the relational Hoare logic, explained in Section 3.9.2, the constant-time property is stated as follows:

$$\mathsf{rotate\_mac\_BL} \sim \mathsf{rotate\_mac\_BL} : =_{\{\mathsf{leak,out,md\_size}\}} \implies =_{\{\mathsf{leak}\}} \tag{3.2}$$

The above judgment states that if we start the execution of the function rotate_mac_BL from two states in which values of the variable leak, out and md_size are equal, then the evaluations will end in two states where the value of the variable leak will be equal. From the assumptions of the program, we know that md_size and out buffer (pointer to where data is stored after rotating it back) are public. According to the definition of $\sim$ or state-equivalence explained in Chapter 2, we know that the equivalence relation between two states resembles the fact that they should be equal on public data. The judgment requires no assumptions for the value of the variable rotate_offset as we know it can be secret dependent. For the case of rotate_offset_BL, the pre-conditions are similar, and the modulo operator does not leak anything in the BL model.

The generated program after extraction consists of the EasyCrypt program corresponding to the Jasmin implementation of ssl3_cbc_copy_mac. It consists of an abstract theory called LeakageModel, where the theory LeakageModel is a signature of the various leakage models that can be instantiated in the proof. Programs in EasyCrypt are stated as modules that consist of global variables and procedures that consist of local variables and a sequence of instructions. Hence, it consists of a module M that contains the variable leakages (stores the leakage generated by various instructions) and extracted EasyCrypt procedures for various functions used in ssl3_cbc_copy_mac.

Here is a snippet of the proof script for proving ssl3_cbc_copy_mac to be constant-time in EasyCrypt:

The first two lines represent importing various libraries that are required to carry out the proof. To prove a program to be constant-time, we need to specify which leakage model it should target. In this case, the implementation of ssl3_cbc_copy_mac needs to be secure in the BL leakage model. Hence, using the keyword “clone", we clone the theory LeakageModel with the theory LeakageModelBL that represents the actual implementation of the BL leakage model adhering to the abstract theory LeakageModel. The concept of cloning allows the generic design of the leakage model to be instantiated in various ways, like for the BL, TV, or CL leakage model.

```
from Jasmin require import JModel Leakage_models.
require import AllCore IntDiv CoreMap List Array64 WArray64 Copy_mac_ct.

clone import Copy_mac_ct.T with theory LeakageModel <- LeakageModelBL.

equiv l_final : M.ssl3_cbc_copy_mac_BL_BL ~ M.ssl3_cbc_copy_mac_BL_BL :
  ={M.leakages, md_size, orig_len, out, rec} /\
  (loadW64 Glob.mem (to_uint (rec + (of_int 16)%W64)))){1} = (loadW64 Glob.mem (to_uint (rec + (of_int 16)%W64))){2}
  ==> ={M.leakages}.
proof.
  proc => /=.
  call (: ={M.leakages, out, md_size} ==> ={M.leakages}); 1: by proc; inline *; sim.
  wp; call (: ={M.leakages, md_size, scan_start} ==> ={M.leakages}); 1: by proc; wp; skip.
  wp; call (: ={ M.leakages, data, scan_start, orig_len, md_size } ==> ={ M.leakages }); 1: by proc; sim.
  by inline *; auto.
qed.
```

Figure 3.12 – Proof of ssl3_cbc_copy_mac_BL_BL in BL model

The lemma is stated using the keyword “equiv" and is called l_final. The procedure ssl3_cbc_copy_mac_BL_BL used in the lemma is present in the module M. The proof in EasyCrypt is interactive; hence, various tactics can be used to reduce a goal into zero or more subgoals.

Focusing on implementation to be proved for TV leakage model To prove the implementation of rotate_mac and rotate_offset present in the top and bottom of Figure 3.5 and Figure 3.4 in the TV leakage model, the theory describing the leakage model where the operators leak should be cloned. For the rotate_mac_BL, the proof is similar as it needs to be constant-time for the BL model. To prove the TV implementation of rotate_offset (present at the bottom of Figure 3.4), we need to show equivalence between the leak variable where it holds the leakage associated with the modulo operator used in the program. Line 12 of the function rotate_offset_TV uses a modulo operator where the divisor (md_size) is public but the operand rotate_offset is secret as it depends on mac_start that is a secret data. The rotate_offset is made a bigger number using the computation $\mathsf{mac\_start} - \mathsf{scan\_start} + (\mathsf{md\_size} << 23)$. The constant-time property for the TV implementation is stated as:

$$
\begin{aligned}
&=_{\mathsf{leak},\ \mathsf{md\_size},\ \mathsf{scan\_start}} \\
&\wedge\ (0 \leq \mathsf{mac\_start}\{1\} - \mathsf{scan\_start}\{1\} < 256) \\
&\wedge\ (0 \leq \mathsf{mac\_start}\{2\} - \mathsf{scan\_start}\{2\} < 256) \\
&\wedge\ 16 \leq \mathsf{md\_size}\{1\} \leq 64
\end{aligned}
$$

and the post-condition is simply $=_{\mathsf{leak}}$. The notation $\mathsf{x}\{1\}$ refers to the value of the variable x in the left state while $\mathsf{x}\{2\}$ refers to its value in the right state. Through the pre-condition, we add enough relations so that we will be able to show equivalence on leak after the final execution. The point to note is that the pre-condition relation defined on mac_start does not require its equivalence because it is a secret data, but it requires its distance from scan_start to be bounded on both sides.

The proof requires about 25 lines of EasyCrypt code. It requires basic results on non-linear arithmetic. In the implementation, the div_spoiler is shifted by 23 to match the OpenSSL implementation, but a shift by eight would suffice. In the original implementation of OpenSSL, the shift was of the form $(\mathsf{md\_size} >> 1) << 24$. As md_size is even, it is equivalent to $\mathsf{md\_size} << 23$, but compilers cannot infer it. The idea of writing it in this form by OpenSSL developer was to prevent the compiler from optimizing the code by removing introduced countermeasure (i.e., replace $((\mathsf{md\_size} << 23) + \mathsf{rotate\_offset})\ \%\ \mathsf{md\_size}$

```
for (j = 0, i = 0; i < md_size; i++) {
  aux1 = rotated_mac[rotate_offset & ~32];
  aux2 = rotated_mac[rotate_offset | 32];
  mask = constant_time_eq_8(rotate_offset & ~32, rotate_offset);
  aux3 = constant_time_select_8(mask, aux1, aux2);
  out[j++] = aux3;
  rotate_offset++;
  rotate_offset &= constant_time_lt_s(rotate_offset, md_size);
}
```

Figure 3.13 – Fixed C implementation of OpenSSL rotate_offset

by (rotate_offset) % md_size, this replacement is functionally correct but does not preserve constant-time hyperproperty).

Focusing on implementation to be proved for CL leakage model To prove the implementation of rotate_mac and rotate_offset present at the bottom of Figure 3.5 and Figure 3.4 in the CL leakage model, the theory describing the leakage model where the memory operation does not leak the full address but leak according to the size of the cache line. The function rotate_mac_CL present in the bottom of the Figure 3.5 is proved constant-time in the CL leakage model.

To prove this function to be constant-time in the CL leakage model, it requires stronger pre-conditions:

$$
\begin{aligned}
&=_{\{\mathsf{leak},\ \mathsf{out},\ \mathsf{md\_size},\ \mathsf{rotated\_mac}\}} \\
&\wedge\ \mathsf{rotated\_mac}\{1\}\%64 = 0 \\
&\wedge\ 16 \leq \mathsf{md\_size}\{1\} \leq 64 \\
&\wedge\ 0 \leq \mathsf{rotate\_offset}\{1\} < \mathsf{md\_size}\{1\} \\
&\wedge\ 0 \leq \mathsf{rotate\_offset}\{2\} < \mathsf{md\_size}\{1\}
\end{aligned}
$$

In this implementation rotated_mac is a pointer to a buffer of length md_size, which should be public (= rotated_mac) and 64 byte aligned (it is the role of the caller of rotate_mac_CL to ensure this condition). The proof follows the intuition provided in Section 3.3. Since the specification also requires that $\mathsf{rotate\_offset}\{1\} < \mathsf{md\_size}\{1\}$, the specification of rotate_offset_TV needs to be extended to ensure that the result will satisfy this condition.

3.10.2 Verified countermeasure on rotating MAC with 32-byte cache line

OpenSSL implementation of rotate_mac_CL for 32-byte cache line model has a bug (see Figure 3.6 for original code). Two load operations are present within a loop as the data will fit in two cache lines. When trying to prove it against the CL_{32} model, we realized that this is incorrect and it is dependent on secret rotate_offset. A counterexample using Jasmin evaluator was created. With a 64-byte aligned rotated_mac buffer, rotate_offset with value 31 touches the second cache line first and then the first. However, rotate_offset with value 63 touches the first cache line first and then the second.

Figure 3.13 shows the verified fix where it always accesses the first cache line and then the second cache line. Later, the correct value is selected in constant-time. This incurs

an overhead of 5.9% at ssl3_cbc_copy_mac function granularity. However, the overhead is negligible in OpenSSL macro benchmarks.

3.11 Implementation

The implementation work is carried out in the Jasmin framework. The implementation consists of:

- a Coq formalization of fine-grained leakage and a formal proof that the Jasmin compiler preserves fine-grained constant-time policies;
- an OCaml implementation that extracts an EasyCrypt program from a Jasmin program;
- an OCaml implementation of an evaluator for testing constant-timeness.

The first two components of the implementation preserve the workflow for Jasmin programs:

- Jasmin programs are checked for safety and compiled;
- proofs of constant-timeness are carried on source Jasmin programs via an embedding into EasyCrypt. Program instrumentation is performed during the embedding in a way similar to what is presented in Section 3.9.

The last component adds an additional functionality (testing for constant-timeness), which is extremely helpful when dealing with fine-grained policies, which have considerably more complex proofs.

3.11.1 Coq formalization

The work explained in Chapter 2 is extended to reason about the fine-grained constant-time policies. The instrumented semantics are adapted to take into account the fact that each division-like operator $\diamond$ generates a leakage depending on its arguments $\mathcal{A}^{\diamond}$ whereas, in Chapter 2, this leakage was assumed to be constant. Also, the leakage for memory access has been made generic to support weaker models. All the proofs of the compiler have been adapted and generalized over the leakage model. As discussed in Section 3.5, the leakage transformers described in Chapter 2 are reused with modifications to the ones that are used in the compiler passes like constant folding and instruction selection as they deal with propagation and lowering of operators. The correctness theorem and its proofs are updated accordingly.

The overall adaptation in the formalization made the Coq development grow from about 37×10^3 lines to 38×10^3 lines.

3.11.2 Extraction to EasyCrypt & Jasmin evaluator

Jasmin provides different ways to extract programs to EasyCrypt. EasyCrypt extraction is proven for constant-time property. This part is made parametric in the leakage model. The extraction is done generically, independently of the model, and then the user can specify their model in EasyCrypt (as discussed in Section 3.10). This work presents some

example models, but the users are also free to define their own models. As the proof of preservation of constant-time is generic in the model, any new model will also be supported.

The extraction to EasyCrypt is implemented in OCaml, and the overall changes with respect to the work discussed in Chapter 2 amounts to around 50 lines of code. The evaluator is also implemented as an OCaml wrapper; it uses an automatically generated file (from the extraction of the Jasmin semantic defined in Coq), and the wrapper represents around 200 lines of code.

3.12 Evaluation

This work also evaluates to check that a program that has a fully automated proof for BL constant-time in Chapter 2 also has an automated proof with fine-grained leakage models, and programs that have an interactive proof of BL constant-time in Chapter 2 have a similar interactive proof of constant-time in the fine-grained models. A set of examples from the Jasmin libraries is checked and validated.

Also, the evaluation focuses on validating the cost of providing programs constant-time with respect to other policies. As discussed in prior sections, proving fine-grained constant-time policies involves arithmetic and, hence, is more complex than proofs that focus on data dependencies. A set of examples that target fine-grained constant-time policies are implemented and proved or disproved that these examples satisfy their intended policies. The number of lines of code is reported to indicate the difference with the baseline leakage mode.

The last emphasis is on the fact that formal methods can be used in the early phase of development, which can help achieve constant-time policies. A development of modulo that is proved to be secure in the TV model is provided.

3.12.1 Impact on verifying the baseline policy

For the evaluation, the motivating examples from Section 3.3 (ssl3_cbc_copy_mac_BL_BL and ssl3_cbc_copy_mac_TV_BL) that have been adapted for this work from OpenSSL, all implementations taken from previous works [Almeida et al., 2019, Almeida et al., 2020] are considered. They consist of three versions of the ChaCha20 [Bernstein et al., 2008] stream cipher (a reference implementation and two optimized ones targeting specific vector instruction set extensions, namely AVX and AVX2); three versions of the Poly1305 authenticator; three versions of the Keccak1600 (SHA3 [Bertoni et al., 2013]) hashing algorithm (a reference implementation, an optimized one using AVX2 vector instructions, and an optimized one using scalar instructions only); two versions of MAC extraction based on the functions shown in Figure 3.4.

The results of this case study are reported in the first rows of Table 3.1. Here are a few observations. Eight out of the ten implementations can be proved secure in any of the considered policies in a proof script of seven lines only. The proof is a one-line call to a fully automatic tactic. The other lines are the statement of the theorem. The proof script for the vectorized version of Keccak1600 is much longer: this implementation uses in-memory tables and its proof involves a lot of reasoning about pointers, similar to what is done in the proof of functional correctness. This suggests possible improvements to the Jasmin programming language.

Table 3.1 – Compliance of examples with fine-grained policies. The table reports the size (lines of code) of each version of examples and for each model whether it can be proved constant-time (number of lines of the proof) or if there is a counter-example (marked with a ✘) witnessing a security violation.

Example	**Implementation**	**Leakage model**					
	size (loc)	CL_{32}	CL_{64}	BL	$\mathsf{TV}+\mathsf{CL}_{32}$	$\mathsf{TV}+\mathsf{CL}_{64}$	TV
ChaCha20 (ref)	396	7	7	7	7	7	7
ChaCha20 (AVX)	900	7	7	7	7	7	7
ChaCha20 (AVX2)	1006	7	7	7	7	7	7
Poly1305 (ref)	239	7	7	7	7	7	7
Poly1305 (AVX)	1065	7	7	7	7	7	7
Poly1305 (AVX2)	1037	7	7	7	7	7	7
keccak1600 (ref)	392	7	7	7	7	7	7
keccak1600 (AVX2)	446	361	361	361	361	361	361
keccak1600 (scalar)	469	7	7	7	7	7	7
ssl3_cbc_copy_mac_BL_BL	469	16	16	16	✘	✘	✘
ssl3_cbc_copy_mac_TV_BL	103	16	16	16	59	59	59
ssl3_cbc_copy_mac_BL_CL_{64}	82	✘	56	✘	✘	✘	✘
ssl3_cbc_copy_mac_TV_CL_{32}	89	153	156	✘	159	162	✘
ssl3_cbc_copy_mac_TV_CL_{64}	86	✘	59	✘	✘	90	✘
pmac_verify_hmac	78	118	118	✘	118	118	✘
coding_wolfSSL	34	✘	58	✘	✘	58	✘

The last example also has a very short proof script (sixteen lines): since the precondition of the security statement involves the contents of the initial memory, the proof requires the user interaction in a couple of places; it is, nonetheless, straightforward and mostly automatic.

In all cases, the proof can be carried once in the BL model, and the same script can be reused as-is in any weaker model (CL_{32} and CL_{64}). Also, for implementations that do not use time-variable instructions, proof extends without modification to the stronger TV and $\mathsf{TV}+\mathsf{CL}$ models.

3.12.2 Verification effort of other policies

In this second case study, we consider implementations designed to be secure in non-baseline policies. They either target the TV model and ensure that time-variable operations only leak public information or target one of the CL_{32} and CL_{64} models and make sure that only the least significant bits of pointers may be secret. We studied 10 cryptographic libraries; out of them, we found eight libraries that contain such code, and we selected three examples that are the most representative or challenging. The corpus consists of four implementations of MAC extraction as explained in Section 3.3, one MAC verification, and the char2val routine used in base64 decoding. All of them are new ports to Jasmin of existing code: the first five from OpenSSL and the last one from WolfSSL. Moreover, all of these examples are out of reach of the previous Jasmin pipeline.

The MAC verification (coined pmac_verify_hmac) accesses a 32-byte-aligned buffer of length at most twenty using secret dependent indices. Its security, therefore, relies on the assumption that the size of a cache line is at least 32 bytes. The base64 decoding excerpt (coding_wolfSSL) uses a table lookup at a secret dependent index: said table is

64-byte-aligned, with a size of 80 bytes, and assumed to fit in two cache lines.

The sizes of the proofs corresponding to these examples are reported in the last six lines of Table 3.1. These proofs are generally longer than the ones for the baseline model: some arithmetic invariants about the arguments to time-variable operations and constraints on the base address and offsets for secret-dependent memory accesses must be established.

As illustrated by the coding_wolfSSL case, which does not use time-variable operations, the proof script for the CL_{32} model can be reused without modification in the stronger $\mathsf{TV} + \mathsf{CL}_{32}$ model.

Overall, the complexity of these proofs matches our expectations and is reasonable. For comparison, functional correctness proofs of ChaCha20 and SHA3 (to be found in earlier works) need hundreds and thousands of lines, respectively; the security proof of SHA3 needs tens of thousands of lines.

3.12.3 Efficient Constant-Time Modulo

The methodology discussed in this chapter can be used to guide the design of efficient constant-time code. The code for computing the modulus in constant-time manner (function rotated_offset_TV) works only because the value of the numerator (resp. the denominator) is small, less than 256 (resp. 64). This section shows that it is possible to implement a constant-time modulus without any requirement on the arguments except that the denominator needs to be public.

In certain circumstances, Modulo might be implemented without using the time-variable operations (by using bitwise operators). Here we present a different approach: the inputs to the time-variable operations are tweaked to ensure that they always fall into a restricted domain that is too small to exhibit a noticeable variable time behavior.

Jasmin implementation

Here is an implementation with the following assumptions: a is private, b is public, and the goal is to compute the (unsigned) remainder of the division of a by b. As the divisor is public, the timing behavior of the hardware modulo instruction only depends on the size of the dividend. Therefore, before calling this instruction, the first argument of the division or modulo operator has a particular (public) size: its most significant bit must be set. This means that the modulo instruction is always called with its first argument in the range $[2^{63}; 2^{64} - 1]$.

An integer n is computed to get a meaningful result so that $a' = b \cdot 2^n + a$ falls in the expected range. In this way, the computation of a' mod b will give the expected result.

The complete implementation in Jasmin is given in Figure 3.14. It relies on an auxiliary function lzcnt (lines 1–6) that counts the number of leading zeros of its argument and also returns a boolean flag telling if this number is null. The code in lines 11–16 computes lzb the number m of leading zeros of b and in the variable flag an integer whose value is one if a is already in the range and zero if a is not in the range but b is. A first attempt (lines 18–21) is to compute in variable dividend the value $b \cdot 2^{m-1} + a$. Using the LEA instruction is an optimization that saves a copy. This value may be too small to fall in the target range; therefore, a second attempt (lines 22–23) computes in variable temp2 the value $b \cdot 2^m + a$. This value is used as a dividend only if the addition did not overflow (line 24). In case b is in the range and a is not, then the result is a: in this case, a dummy division of the maximal 64-bit unsigned integer is computed (lines 25–27) and the result accordingly corrected (line 30) using a conditional move.

```
1 inline fn lzcnt(reg u64 x) −→ reg bool, reg u64 {
2   reg u64 result;
3   reg bool zf;
4   _, _, _, _, zf, result = #LZCNT(x);
5   return zf, result;
6 }
7 export fn mod_TV(reg u64 a, reg u64 b) −→ reg u64 {
8   reg u64 flag, one, zero, dividend, modulo, result;
9   reg u64 lzb, lzb_m1, b_lzb, b_lzb_m1, temp2;
10  reg bool lzaz, lzbz, cf;
11  flag = 0x1234;
12  one = 1;
13  zero = 0;
14  lzbz, lzb = lzcnt(b);
15  flag = zero if lzbz;
16  lzaz, _= lzcnt(a);
17  flag = one if lzaz;
18  lzb_m1 = #LEA(lzb - 1);
19  b_lzb_m1 = b;
20  b_lzb_m1 = b_lzb_m1 << lzb_m1;
21  dividend = #LEA(b_lzb_m1 + a);
22  b_lzb = b_lzb_m1 << 1;
23  cf, temp2 = b_lzb + a;
24  dividend = temp2 if ! cf;
25  dividend = a if flag == 1;
26  temp2 = 0xFFFFFFFFFFFFFFFF;
27  dividend = temp2 if flag == 0;
28  modulo = dividend % b;
29  result = modulo;
30  result = a if flag == 0;
31  return result;
32 }
```

Figure 3.14 – Generic constant-time modulus operation

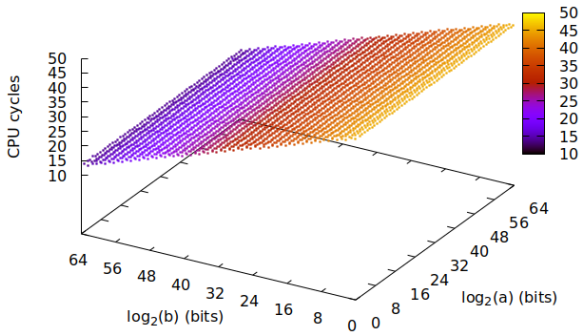


Figure 3.15 – Timing behavior of the mod_TV function on one x86 microprocessor (same experimental setup as in Figure 3.2).

Functional correctness

This implementation is correct (i.e., it always computes the modulo of its arguments). Correctness can be formally stated in EasyCrypt as follows:

$$\{\mathsf{a} = a_0 \wedge \mathsf{b} = b_0 \wedge \mathsf{b} \neq 0\}\ \mathsf{mod\_TV}\ \{\mathsf{result} = a_0 \bmod b_0\}\,.$$

This is a Hoare triple with universally quantified logical variables a_0 and b_0 that allow referring to the initial values of the arguments. The proof shows that no overflow badly interferes with the computation and is about 60 lines long.

As illustrated in Figure 3.15, the experiment shows that there is no longer any timing variation while changing the private input. The execution time of the whole function may still vary depending on the size of the public input. Notice on the right-hand-side plot that the execution times of the secure implementation range between 14 and 45 cycles, which is the same as the range of execution times of the lone hardware modulo instruction reported on the left of Figure 3.2 (for this particular microprocessor). Reproducing the same experiment on a different microprocessor (Intel Xeon E5-2687W) leads to the same conclusion: although the div instruction is time-variable, the execution time of the mod_TV function does not depend on the value of its first argument.

Constant-time security

This implementation is secure in the TV model, under the precondition that argument b is public and non-zero. Formally, the security is stated as follows, where the instrumentation of the mod_TV function is interpreted in the TV model:

$$\mathsf{mod\_TV} \sim \mathsf{mod\_TV}_{=\{\mathsf{leak,b}\}} \wedge b\{1\} \neq 0_{=\{\mathsf{leak}\}}$$

The proof methodology is similar to the other examples discussed earlier. The central argument is that the leakage produced by the execution of this function is a known function of b (hence independent of the value of the first argument a). Thanks to the simplicity of the control-flow structure of this program, the EasyCrypt machinery for computing the weakest preconditions can transform the program-verification task into a pure arithmetic formula. Discharging this proof is a bit tedious, as usual with machine arithmetic, due to the possible overflows. The proof script is about 130 lines long.

3.13 Related work

There is a large spectrum of tools for analyzing side-channel attacks. Several of those tools are also discussed in Chapter 1 and Chapter 2. Tools like ct-verif [Almeida et al., 2016b], flowtracker [Rodrigues et al., 2016a], virtualcert [Barthe et al., 2014a], and binsec/rel [Daniel et al., 2020] explicitly target the baseline constant-time policy. These tools are supported by soundness claims. Only CacheD [Wang et al., 2017] considers a weaker leakage model where the cache line is leaked. CacheD favors automation and precision over soundness and is therefore not supported by a soundness claim. As noted by Bernstein, no tool supports time-variable operations. There are many other tools, such as CacheAudit [Doychev et al., 2013] or CacheFix [Chattopadhyay and Roychoudhury, 2018], which use automated techniques to reason about cache behavior. However, these tools do not target a constant-time policy. In particular, they do not consider control-flow leakage and do not allow values to carry a security level. We refer to [Barbosa et al., 2021, Jancar, 2021]

for a description of other tools. In addition, there are general-purpose tools that can also be applied to side-channel analysis. This is the case of Themis [Chen et al., 2017], which introduces QCHL, a quantitative variant of Cartesian Hoare Logic [Sousa and Dillig, 2016] and uses QCHL to reason about side-channels of Java bytecode. Another (earlier) instance is Blazer [Antonopoulos et al., 2017], which introduces a proof technique to reason about hypersafety and applies the proof technique to reason about side-channels of Java bytecode. Another line of work develops constraint-based methods for verifying relational properties of hardware and applies these methods to reason about constant-time [von Gleissenthall et al., 2019, von Gleissenthall et al., 2021]. There is also a large body of work that develops automated transformation methods for making programs constant-time, see e.g. [Cauligi et al., 2019a, Borrello et al., 2021], or that develops frameworks that are secure by design: [Zhang et al., 2012] features a language with mechanisms to control timing channels and a type system that quantitatively bound the information leakage of well-typed programs; [Zagieboylo et al., 2019] introduces a timing-channel aware ISA that serves as the contract between software and hardware.

Chapter 4

Cost Analysis

4.1 Introduction

This chapter presents an application of the leakages and leakage transformers presented in Chapter 2. It explains how they can be used to justify a program's execution cost. The instruction counting model that tracks how often each instruction is executed in a program run is one of the simplest cost models. It is also the basis of many approaches for computing the upper bounds on the program's cost. These approaches are mainly developed for source programs, and transferring the source-level analysis result to target programs is not straightforward.

The computation of the run-time cost of a program - computational complexity, worst-case execution time, peak memory usage, etc. - crucially depends on low-level details; hence, it involves decisions made at compile-time: control-flow transformations and code layout, register spilling, and memory layout of local variables, instruction selection, and scheduling. Therefore, a precise static cost analysis must be carried out near the end of the compilation pipeline (ideally on assembly code or even at the binary level). However, the estimation of the run-time cost relies on loop bounds or other flow information, either inferred by static analysis or provided by the programmer as annotations. In both cases, these flow facts are provided at the source level: programmers are more inclined towards annotating source code than target code, and static analyses are much more precise and efficient when they can rely on high-level abstractions from the source language.

This chapter explains how the findings described in Chapter 2 offer a means to transfer the source-level analysis results to the target level. The instruction counting model is designed based on the leakage model described in Chapter 2. The leakages give an understanding of the program's structure as it is closely related to the syntax and semantics of the program. It is possible to compute the cost of execution as a function of its leakage, i.e., $\kappa = tocost(\ell)$, where $p : s \Downarrow_\ell s'$. The leakage corresponding to an instruction helps to compute its execution cost. Also, the Jasmin compiler emits a function F called leakage transformers described in Section 2.3 of Chapter 2. The explicit representation of leakage transformers makes it possible to compute the cost of a Jasmin target program by analyzing the source program. Therefore, any function F that correctly transforms the leakage of p satisfies $\overline{\kappa} = tocost(F(\ell))$. This approach is precise because, for most optimization passes, one can compute the target program's cost from the source program's cost. The cost of the target program is exact if the cost of the source program is exact. Moreover, the cost of the target program is a sound overapproximation if the cost of the source program is a sound approximation. The overall beauty of the methodology explained in

this chapter is that the complete cost analysis (either at the source or target level) only depends on leakages and leakage transformers; it does not require the program.

4.2 Contribution

- The definition of cost and cost transformers;
- A certified algorithm for computing the cost of assembly programs from the cost of Jasmin programs.

4.3 Cost analysis

This section describes how leakage transformers presented in Chapter 2 can be used to transport the source-level cost information down to the assembly level. A cost model is introduced as an abstraction of leakages and a methodology to deduce the "cost transformers" from leakage transformers.

4.3.1 Cost models

The cost of program execution at the source or target level is modeled as the number of times each instruction is executed. In other words, cost is a finite map between the program points and natural numbers. The program point is a position in the program text for unstructured intermediate language (like linear level or assembly). For structured language, a language of paths is described to define a program point in the abstract syntax tree. The cost is defined by means of a function that evaluates a leakage trace into a cost map. The structure of leakage plays an important role here; as it reveals a lot about the semantics of the program, it helps compute the cost without looking at the program.

Definition 4 (Cost)**.** *Each intermediate language is equipped with a tocost(.) function that, given a leakage, computes a cost, i.e., a count for each program point.* $p : s \Downarrow_\ell s'$*, its cost is tocost(*ℓ*)*

4.3.2 Cost model for source level

This section describes the cost model for the source level language of Jasmin defined in Figure 1.11. There are various notations defined that are used to describe a path in the abstract syntax tree of the source program. The cost of a source-level instruction is a mapping from the path to a natural number. The path is computed using labels *lbl* defined as follows:

$$lbl := lblF\ fn \mid lblB\ b \mid lblL \qquad bpath := \overline{(lbl, n)} \qquad path := (bpath, n)$$

The labels *lbl* define the tag for labeling function calls, conditional, and non-branching instructions. For example, the label to a function is of the form $lblF\ fn$ where fn is the function name.

The source-level language is equipped with a function called $tocost_s(p, \ell)$, given a source-level leakage ℓ (where ℓ is of the form defined in Figure 2.5) and path p, computes a cost, i.e., a count for each source-level instruction. Given an execution, $p : s \Downarrow_\ell s'$,

its cost is $tocost_s(p, \ell)$. The semantics of $tocost_s(p, \ell)$ are defined as rules present in Figure 4.1.

Auxiliary functions:

$$next_p(p) = \lambda\ p.(p.1, p.2 + 1) \qquad path_b(b, p) = \lambda\ b\ p.((lblB\ b, p.2) :: p.1)$$

$$path_{fun}(fn, p) = \lambda\ b\ p.((lblF\ fn, p.2) :: p.1)$$

$$path_{for}\ p = \lambda\ p.((lblL, p.2) :: p.1)$$

Cost of source level instructions:

$$\frac{}{tocost_{\overline{s}}(p, \bullet) = \{\_ \to 0\}}\ [\mathsf{Empty_{cost}}]$$

$$\frac{tocost_s(p, \ell_i) = c_i \wedge next_p(p) = np \wedge tocost_{\overline{s}}(np, \ell_c) = c_c}{tocost_{\overline{s}}(p, \{\ell_i; \ell_c\}) = c_i(p) + c_c(p)}\ [\mathsf{Seq_{cost}}]$$

$$\frac{}{tocost_s(p, \mathsf{op_l}(\ell_d; \ell_e)) = \{\_ \to 0\}}\ [\mathsf{Assign_{cost}}]$$

$$\frac{}{tocost_s(p, \mathsf{if}_b(\ell_e, \ell_c)) = \{path_b(b, p) \to 1\} + tocost_{\overline{s}}((path_b(b, p), 0), \ell_c)}\ [\mathsf{Cond_{cost}}]$$

$$\frac{}{tocost_s(p, \mathsf{while_f}(\ell_c, \ell_e)) = \{path_b(\mathit{ff}, p) \to 1\} + tocost_{\overline{s}}((path_b(\mathit{ff}, p), 0), \ell_c)}\ [\mathsf{WhileF_{cost}}]$$

$$\frac{\begin{array}{c}\{path_b(\mathit{ff}, p) \to 1\} + tocost_{\overline{s}}((path_b(\mathit{ff}, p), 0), \ell_c) = c_1 \wedge \\ \{path_b(\mathit{tt}, p) \to 1\} + tocost_{\overline{s}}((path_b(\mathit{tt}, p), 0), \ell'_c) = c_2 \wedge tocost_s(p, \ell_w) = c_3\end{array}}{tocost_s(p, \mathsf{while_t}(\ell_c, \ell_e, \ell'_c, \ell_w)) = c_1(p) + c_2(p) + c_3(p)}\ [\mathsf{WhileT_{cost}}]$$

$$\frac{}{\begin{array}{c}tocost_s(p, \mathsf{call}\ (\ell_e; \ldots; \ell_e)\ (fn, \ell_f)\ (\ell_w; \ldots; \ell_w)) = \\ \{path_{fun}(fn, p) \to 1\} + tocost_{\overline{s}}((path_{fun}(fn, p), 0), \ell_f)\end{array}}\ [\mathsf{Call_{cost}}]$$

Figure 4.1 – Function computing cost for source-level instructions.

The cost of an assignment instruction is null as per the cost model defined in this work. $\mathsf{Cond_{cost}}$ explains how to compute the cost of a conditional instruction. It is computed using the leakage of the form $\mathsf{if}_b(\ell_e, \ell_c)$. It merges the cost associated while entering the branch (entering the branch updates the cost related to the label $lblB\ b$ to 1) and the cost of the basic block corresponding to the true or false branch (it is computed using the leakage associated with true or false branch). The cost of a while loop is defined (similar to the conditional) using two rules: $\mathsf{WhileF_{cost}}$ and $\mathsf{WhileT_{cost}}$.

$\mathsf{Call_{cost}}$ explains how to compute the cost of a function call. It is computed using the leakage of the form $\mathsf{call}\ (\ell_e; \ldots; \ell_e)\ (fn, \ell_f)\ (\ell_w; \ldots; \ell_w)$. It merges the cost associated while entering the function call (entering the function call updates the cost related to the

label $lblF\ fn$ to 1) and the cost of executing the body of the function (it is computed using the leakage associated with the body).

The rules to calculate the cost for various instructions are computed using multiple auxiliary functions. The rule $\mathsf{Seq}_{\mathsf{cost}}$ defines how to compute the cost of a sequence of instructions. It uses the sequences of leakages $\{\ell_i; \ell_c\}$ corresponding to the instructions, computes cost related to the first instruction using $tocost_s(,)$ and then recursively computes the cost of the rest of the instructions. $path_b(b, p)$ computes the position corresponding to the conditional instruction present in a program with a path p, and it assigns the label $lblB\ b$ to the position. Similarly, $path_{fun}(fn, p)$ computes the position corresponding to the function call in a program.

4.3.3 Cost model for linear level

This section describes the cost analysis carried out for the linear-level language. The linear level language is described in Figure 2.6 present in Chapter 2. The cost of a linear instruction is a mapping from a program counter to a natural number. An update function is defined to update the cost of a program point. Given a program counter n and a linear cost map m, the update function increases the cost associated with n by 1 ($m\{n \to (m(n) + 1)\}$). A merge function merges the cost associated with a program counter. Given a program counter n and two maps m_1 and m_2, the merge function returns $m_1(n) + m_2(n)$.

The intermediate language is equipped with a $tocost_l(.,.)$ function that, given a leakage l_i (where l_i is of the form defined in Figure 2.6 of Chapter 2) and a program counter n, computes a cost, i.e., a count for each program counter. Given a execution, $p : s \Downarrow_{l_i} s'$, its cost is $tocost_l(n, l_i)$. As the program counter plays a role in computing the cost, the $tocost_l(.,.)$ function also returns the next program counter that can be used to compute the cost of a sequence of linear instructions.

$next_{pc}(n, l_i)$ is a function that computes the next program counter for a given program counter n and a leakage l_i.

$$next_{pc}(n, l_i) = \begin{cases} n+1 & l_i = \bullet \\ |n+i| & l_i = i \\ n+1 & l_i = \mathsf{op}\ \ell_e \\ |n+i| & l_i = \mathsf{if_l}\ i\ \ell_e\ b \end{cases}$$

The function $tocost_l(n, l_i)$ computes the cost corresponding to a single linear instruction and is defined as follows:

$$tocost_l(n, l_i) = \lambda\ n\ l_i.(\{n \to 1\}, next_{pc}(n, l_i))$$

The function $tocost_{\overline{l}}(n, \overline{l_i})$ computes the cost corresponding to a sequence of linear instructions $\overline{l_i}$ and is defined as follows:

$tocost_{\bar{l}}(.,.)$ rules:

$$\frac{}{tocost_{\bar{l}}(n, \bullet) = (\{\_ \to 0\}, n)} \qquad \frac{tocost_l(n, l_i) = (m, n') \wedge tocost_{\bar{l}}(n', l_i') = (m', n'')}{tocost_{\bar{l}}(n, l_i :: l_i') = (m(n) + m'(n), n'')}$$

Figure 4.2 – Semantics of $tocost_{\bar{l}}(.,.)$ function

4.3.4 Cost model for assembly level

This section describes the cost analysis for the assembly level. The assembly level language is described in Figure 2.9 present in Chapter 2. The cost of an assembly instruction is a mapping from the instruction pointer to a natural number. The assembly language is also equipped with a $tocost_a(.,.)$ function that, given a leakage l_a (where l_a is of the form defined in Figure 2.9 of Chapter 2) and a program counter n, computes a cost, i.e., a count for each instruction pointer. Given a execution, $p : s \Downarrow_{l_a} s'$, its cost is $tocost_a(n, l_a)$.

$next_{pc}(n, l_i)$ is a function that computes the next program counter for a given instruction pointer n and a leakage l_a.

$$next_{pc}(n, l_a) = \begin{cases} n+1 & l_a = \bullet \\ |n+i| & l_a = i \\ n+1 & l_i = (*p, \ldots, *p) \\ |n+i| & l_a = \mathsf{if_a}\ i\ b \end{cases}$$

The function $tocost_a(n, l_a)$ computes the cost corresponding to a single assembly instruction and is defined similarly to $tocost_l(.,.)$. The function $tocost_{\bar{a}}(n, \overline{l_a})$ computes the cost corresponding to a sequence of assembly instructions $\overline{l_a}$ and is defined similarly as the rules presented in Figure 4.2.

4.4 Cost Transformers

Program transformations found in compilers introduce, remove, or reorder instructions according to the program being compiled; they do not makeup instructions out of the blue. Even though predicting how many times each instruction emitted by a compilation pass will be executed at run-time is usually not possible, the execution counts for target instructions can be related to execution counts for the corresponding source executions. More precisely, for most transformations found in the Jasmin compiler, the target costs can be precisely described by relating each target program point to one basic block of the source program. This link is to be interpreted as follows: the instruction at this program point is executed in the target execution as often as the basic block is executed in the source execution. Unfortunately, the predicted target count is an upper bound for some compilation passes.

Definition 5 (Cost Transformers)**.** *A cost transformer maps the target program point to source basic blocks. It, therefore, enables the translation of source-level cost into target-level cost. A leakage transformer can be seen as a cost transformer by an interpretation*

function $[\![.]\!]_\kappa$*; such an interpretation is sound when for all (source) leakage* ℓ*, and the matching leakage transformer* τ *the following holds (where* $\sqsubseteq$ *is a partial order of costs):*

$$tocost([\![\tau]\!]^\ell) \sqsubseteq [\![\tau]\!]_\kappa^{tocost(\ell)}. \tag{4.1}$$

The interpretation of a leakage transformer as a cost transformer is defined from the leakage transformer only: it neither depends on the program nor the compilation pass. It is a sound way to transport source-level cost information down to the assembly level.

Cost transformers are monotone; therefore, they can be soundly composed. Indeed, given two leakage transformers τ_1 and τ_2 corresponding to two successive compilation passes, the following inequality holds:

$$tocost([\![\tau_2]\!]^{[\![\tau_1]\!]^\ell}) \sqsubseteq [\![\tau_2]\!]_\kappa^{[\![\tau_1]\!]_\kappa^{tocost(\ell)}}.$$

It means that a sequence of two leakage transformers can be interpreted as a sound cost transformer by composing their interpretation.

The actual definition of the interpretation functions (for each of the three languages of leakage transformers described in this work) is tedious but unsurprising.

As already mentioned, the cost transformers for all but one pass are exact: the soundness relation (equation (4.1) above) holds even when the partial order on costs $\sqsubseteq$ is equality. For loop unrolling, however, it only holds for the slightly less precise pointwise ordering of counters (with natural numbers ordered as usual).

4.5 Evaluation

With accurate leakage transformers at hand, a range of source-level reasoning becomes possible. A source-level cost analysis is combined with the leakage transformers to statically compute the upper bounds of the run-time cost of target programs. The result is compared with actual run-time measurements. This experiment aims to assess the precision of the leakage transformers and not to design a cycle-accurate cost analysis for x86 assembly programs. In particular, the cost model explained in this work is fairly simple: we count the (total) number of executed instructions.

Methodology

A sample of representative Jasmin programs (permutations, hash functions, etc.) is selected to experiment. For each program, the source-level cost analysis computes a set of linear constraints between execution counters (at the granularity of basic blocks) and initial values of the (main) function arguments. The leakage transformers produced at compile-time yield *cost transformers* that map each target instruction to a source basic block. From this cost-transformer, a symbolic upper bound of the total run-time cost is computed: an affine combination of source execution counters.

By fixing some run-time parameters (typically, the size of the inputs) and solving the resulting integer linear program: The maximal cost satisfying the constraints is obtained. This gives a static numerical estimate of the cost for the given input size.

Each compiled program is evaluated on inputs of the corresponding sizes and measures[1] the number of executed instructions and elapsed CPU cycles. Elapsed time is

[1] by reading Linux performance counters on a laptop running Linux 5.4 on a Intel® Core™ i7-8665U CPU @ 1.90GHz.

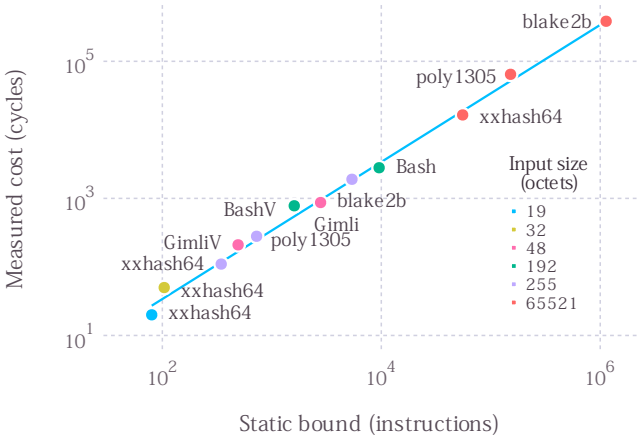


Figure 4.3 – Run-time cost analysis (blue line is 2.9 instr/cycle)

estimated by a Rust program that calls the Jasmin functions; it is built on top of Criterion.rs [Criterion Developers, 2019], a "statistics-driven micro-benchmarking tool".

Results

Experimental results are shown in Figure 4.3. Each point of the graph corresponds to one program and one choice of input size. The programs Gimli and Bash are permutations: their inputs have fixed sizes. Both come in two versions: a reference and one optimized for platforms with AVX2 vectorized instructions (the capital V at the end of the names mark the vectorized versions). The xxhash64 and poly1305 programs are a (non-cryptographic) hash algorithm and a MAC function (respectively). In both cases the control-flow structure is slightly complex as there are two different paths for short and long messages, and there are many loops to handle the input message in chunks of decreasing sizes. Finally, the program blake2b is a cryptographic hash algorithm that can produce digests of any size between 1 and 64 bytes. It is also made of several loops to first consume the message and then produce the digests of the appropriate size. In all cases, the measured number of executed instructions is *exactly* predicted by the static analysis (not shown on the graph). The measurement shows that the processor executes between 2 and 4 instructions per cycle. The plain line on the graph, obtained through linear regression of the measurements, has a slope of 2.9 instructions per cycle (with a correlation coefficient of 0.999).

Remark on precision loss

As mentioned in Section 4.4, the cost-transformer for loop unrolling may lose some precision, as illustrated in the (artificial) example depicted in Figure 4.4. When the loop is unrolled, its body is replicated, and each copy is executed as many times as the original loop. However, at most, a single copy of the nested basic block (labeled A) is executed, but the compiler cannot predict which one, hence the loss of precision, assuming that each copy *may* be executed.

Such pathological cases do not occur in practice as conditions that are nested in unrolled loops and involve the loop counters are usually resolved at compile-time.

.

```
1 param int N = 10;
2
fn inc(reg u64 x) ⟶ reg u64 {
3   inline int i;
4   reg u64 r;
5   r = x;
6   for i = 0 to N {
7     if x == i {
8       r += 1; // A
9     }
10  }
11  return r;
12 }
13
```

Exact cost: $21 + A$ instructions; computed bound: $21 + 10 \cdot A$.

Figure 4.4 – Precision loss in cost transformation

4.6 Related work

There is a vast body of work on automatically analyzing program efficiency. In particular, the fields of WCET (Worst-Case Execution Time) and cost analyses aim to provide estimates (upper and lower bounds) of program execution. These estimates use a broad range of cost models. One of the simplest models is the instruction counting model considered in this paper. However, many works in the WCET community also consider very precise cost models that account for underlying micro-architectural features. Analyses are carried out on source programs (prevailing for cost analysis) and low-level programs (prevailing for WCET), but only a few works connect the costs of source and target programs. One of the first works in this direction is [Crary and Weirich, 2000], which develops a time bounds-certifying compiler from a safe dialect of C to assembly. However, their work focuses on upper rather than exact bounds for assembly programs and follows the principles of certifying compilation. In contrast, our work is more focused on transferring the results of source-level cost analysis. In this sense, our work is more closely related to the CerCo compiler [Armadio et al., 2011], which connects a cost analysis for source programs with the cost of target programs. Their work goes beyond the goals of our present study, as their compiler provides, via annotations in the source program, realistic estimates of the time and space cost of basic blocks of the target programs. A similar approach is taken by Carbonneaux et al. [Carbonneaux et al., 2014] to provide upper bounds on stack usage of assembly programs generated by the CompCert verified compiler. In a functional setting, Paraskevopoulou and Appel [Paraskevopoulou and Appel, 2019] prove preservation of stack space by closure conversion, whereas Gómez-Londoño et al. [Gómez-Londoño et al., 2020] prove a similar result for the CakeML compiler. To the exception of [Crary and Weirich, 2000], all these works support mechanized correctness proofs using proof assistants.

Chapter 5

High-Assurance Cryptography in the Spectre Era

5.1 Introduction

As discussed in Chapter 2 and Chapter 3, constant-time is a programming discipline to avoid timing-based attacks in programs featuring sequential execution semantics. However, this semantics is not aligned with the behavior of modern processors that make use of speculative execution to improve performance. The prior chapters' results do not guarantee against Spectre-style attacks discussed in Section 1.1.2 of Chapter 1. This chapter presents an end-to-end methodology for proving constant-time property for cryptographic software under speculative execution.

High-assurance cryptography leverages methods from program verification and cryptographic engineering to deliver cryptographic software with machine-checked proofs of memory safety, functional correctness, provable security, and the absence of timing attacks. Cryptography implementations must achieve the *Big Four* guarantees: (i) It should be *memory safe* to prevent leaking secrets due to illegal memory accesses, (ii) It should be *functionally correct* with respect to a standard specification, (iii) *provable secure* to avoid important class of attacks, and (iv) *protected against timing-based side-channel attacks* that can be carried out remotely without physical access to the device under attack. Cryptographers use high-assurance cryptography techniques to implement efficient cryptography libraries to achieve these goals. Unfortunately, the guarantees provided by *Big Four* can still be vulnerable to micro-architectural side-channel attacks, such as Spectre attacks [Kocher et al., 2019b], which exploit speculative execution in modern CPU because these properties established under a sequential execution cannot be extended blindly for the speculative semantics.

Contributions

- A formalization of an adversarial semantics of speculative execution and a notion of speculative constant-time for a core language with support for software-level countermeasures against speculative execution attacks.

- A weaker "forward" semantics definition is introduced, which means an execution is forced into early termination when misspeculation is detected instead of backtracking.

- A proof of a key property called “secure forward consistency", that shows that a program is speculative constant-time iff forward executions do not leak secrets via timing side-channels.

- A verification method for speculative constant-time. The method is decomposed into two steps: (i) check that the program does not perform illegal memory accesses under speculative execution (speculative safety), and (ii) check that leakage does not depend on secrets.

- An implementation of the approach in the Jasmin verification framework.

5.1.1 Methodology

This section gives an overview of the methodology described in this work.

Threat model The sequential semantics described in the previous chapters model the scenario where an attacker observes all branch decisions and the addresses of all memory accesses throughout the course of program execution. An extension to this threat model assumes that an attacker can make the same observations about the speculatively executed code. The semantics described in Chapter 2 do not take into account how to capture attackers that deliberately influence predictors. Hence, it is necessary to model how code is speculatively executed and what kind of values can be speculatively retrieved by load instructions.

In this work, an approach is taken where an active attacker is taken into account that controls branch and load decisions. Whereas in sequential semantics described in previous chapters, a passive attacker is taken into account. The well-known approach to limit the attacker is by using fence instruction. The active attacker model helps in capturing attackers that not only mount traditional timing attacks [Kocher, 1996a] discussed in Section 1.1.1 of Chapter 1, but also helps in capturing different kinds of Spectre attacks described in Section 1.1.2 of Chapter 1. The threat model described in this work explicitly assumes that the execution platform enforces control flow and memory isolation, and fences act effectively as a speculative barrier. Attackers cannot read the values of arbitrary memory addresses, cannot force execution to jump to arbitrary program points, and cannot bypass or influence the execution of fence instructions.

Memory fences as a Spectre mitigation Memory fence instructions act as speculative barriers, preventing further speculative execution until prior instructions have been completed. For example, placing a fence after the conditional branch (between line 1 and line 2) in Figure 1.4a and Figure 1.4b, prevents the processor from speculatively reading from p until the branch condition has resolved, at which point any misspeculation will have been caught. Similarly, placing a fence in Figure 1.5 before loading a[i] on line 6 forces the processor to commit all prior stores to memory before continuing, leaving nothing for the attacker to mispredict.

Unfortunately, inserting fences after every conditional and before each load instruction severely hurts the performance of programs. An experiment inserting LFENCE instructions around the conditional jumps in the main loop of SHA-256 implementation showed a nearly 60% decrease in performance. Also, minimizing the use of fences without the

support of any formal reasoning or verification mechanism showing their sufficient insertions might lead to shaky security guarantees (e.g., Microsoft's C/C++ compiler-level countermeasures against conditional-branch variants of Spectre v1) [Spe, 2018].

Speculative constant-time The notion of constant-time defined in Section 2.5 of Chapter 2 aims to protect cryptographic code against the standard timing side-channel threat model [Jean-Philippe Aumasson, 2019]. To carry out the formal reasoning, the semantics presented in Section 2.2 of Chapter 2 is instrumented with explicit leakages/observations that represent what values are leaked to an attacker during the execution of an instruction. Unfortunately, this notion falls short in the presence of speculative execution.

The formalization of speculative constant-time is based on the same idea of leakages as for constant-time, but is defined under an adversarial semantics of speculation. To reflect active adversarial choices, each step of execution is parameterized with an adversarially issued *directive* indicating the next course of action. For example, to model the attacker's control over the branch predictor upon reaching a conditional, the attacker issues either a step directive to follow the due course of execution or a force b directive to speculatively execute a target branch b. The set of directives is described in the later section.

Under the adversarial semantics, a program is speculative constant-time if, for every set of directives, the observations accumulated over the course of the program's execution do not depend on the values of secret inputs. Importantly, this notion is independent of cache and predictor models and delivers stronger, more general guarantees that are also easier to verify.

This work proves that programs are speculative constant-time using a relatively standard dependency analysis. The soundness proof of the analysis is non-trivial and relies on a key property of the semantics, which is called *secure forward consistency*. This shows that a program is speculative constant-time iff forward executions (rather than arbitrary speculative executions) do not leak secrets via timing side-channels. The forward execution forces the processor to halt in case of misspeculation; hence, no semantics backtracks to restart the execution from the state that produced the wrong result. This result greatly simplifies the verification of speculative constant-time, drastically reducing the number of execution paths to be considered. Moreover, with secure forward consistency, code that is proven functionally correct and provable secure in sequential semantics also enjoys these properties in speculative semantics.

Speculative safety The semantics in this work assume that unsafe memory accesses, whether speculatively or not, leak the entire memory μ via an observation unsafe μ. Therefore, programs that perform unsafe memory accesses cannot be speculative constant-time. Programs are proved to be speculatively safe, i.e., do not perform illegal memory accesses for any choice of directives, using a value analysis. The analysis relies on standard abstract interpretation techniques [Cousot and Cousot, 1977], but with some modifications to reflect speculative semantics.

Jasmin integration The methodology described in this work is integrated into Jasmin's framework. Jasmin is extended with a fence instruction. The Jasmin language is extended to include speculative semantics (backtracking and no backtracking). All the proofs showing the equivalence between various semantics and the soundness of the analysis are done on paper and are explained in later sections. The Jasmin language is extended with a fence instruction. All the checkers, like for memory safety, functional

correctness, etc., are extended to include speculative semantics and are written in OCaml (their correctness is proved on paper and explained in the later sections).

5.2 Adversarial semantics

This section presents the language and its adversarial semantics and defines speculative safety and constant-time.

5.2.1 Language

The language is present in Figure 1.11 and is extended with a fence instruction.

5.2.2 Memory

For sequential semantics, memory consists of the main memory m: $\mathcal{A} \times \mathcal{V} \to \mathcal{V}$ (maps addresses (pairs of array names and indices) to values) and a register map ρ: $\mathcal{X} \to \mathcal{V}$ (map registers to values). A buffered memory is used instead of the main memory to represent out-of-bound memory operations. A buffered memory adds a *write buffer*, or a sequence of *delayed writes* to the main memory. Every delayed write is of the form $[(a, w) := v]$, representing a pending write of value v to an array a at an index w. The buffered memory is of the form $[(a_1, w_1) := v_1] \ldots [(a_n, w_n) := v_n]m$, where the sequence of updates represent the pending writes not yet committed to main memory.

Buffered Memory

Main memory $m : \mathcal{A} \times \mathcal{V} \to \mathcal{V}$

Buffered memory $\mu ::= m \mid [(a, w) := v]\mu$

Location access

$$\begin{cases} m(\!|(a,w)|\!)^i = m[(a,w)], \bot & w \in [0, |a|] \\ [(a,w) := v]\mu(\!|(a,w)|\!)^0 = v, \bot & w \in [0, |a|] \\ [(a,w) := v]\mu(\!|(a,w)|\!)^{i+1} = v', \top & \mu(\!|(a,w)|\!)^i = v', \_ \\ [(a',w') := v]\mu(\!|(a,w)|\!)^i = \mu(\!|(a,w)|\!)^i & (a',w') \neq (a,w) \end{cases}$$

Flushing Memory

$$\overline{m} = m$$

$$\overline{[(a,w) := v]\mu} = \overline{\mu}\{(a,w) := v\}$$

Figure 5.1 – Formal definitions of buffered memory, location access, and flushing

The memory writes are applied as delayed writes to the write buffer, and memory reads may look up values in the write buffer instead of the main memory. As the semantics models the steps taken by the attacker, the memory read or write should also allow the attacker to read any value from the write buffer or from the main memory of his choice. Hence, memory reads may not always use the value from the most recent write to the same address. The adversary can force load instructions to read any compatible values from the write buffer or even skip the buffer entirely and load from the main memory. Buffered memory access is denoted as $\mu(\!|(a, w)|\!)^i$ where array a is being read at offset w, and i is an integer specifying which entry in the buffered memory to use (0 represents the most

recent write to that address in the buffer). The access returns the corresponding value and a flag representing whether the fetched value is correct concerning non-speculative semantics. If i is 0 the flag is $\bot$, signifying that the value is correct; otherwise, the flag is $\top$. When the program encounters a fence instruction, the write buffer is flushed to the main memory. Each delayed write is committed to the main memory in order, and the write buffer is cleared. This operation is represented as $\overline{\mu}$.

The formal definitions of buffered memories, accessing a location, and flushing the write buffer are present in Figure 5.1. The notation $m[(a, w)]$ represents memory lookup and $m\{(a, w) := v\}$ represents updating the memory with value v.

[(a, 9) := pub; (a, 9) := sec]m

```
1 fun foo(int sec, int pub) {
2     if (x < 8) {
3         a[x] = sec;    Speculatively load secret
4         a[x] = pub;    Speculatively load public
5         r = a[x];      Speculatively leaks secret
6     }
7     return r;
8 }
```

Figure 5.2 – Spectre-PHT attack

The example in the Figure 5.2 presents the function foo, which speculatively leaks the secret at line 5. The condition x < 8 is bypassed speculatively, and two stores are performed at lines 3 and 4. These store operations add two delayed writes to the write buffer, and the attacker can read any value from the write buffer. In line 5, the read can be performed at secret index (leaking the secret) because the attacker can decide to read the not recent write but the old one, which stores sec.

5.2.3 States

States are a set of configurations. A configuration is a tuple of the form $\langle c, \rho, \mu, ms \rangle$, where c is a command, ρ is a register map, μ is a buffered memory, and ms is a boolean. The boolean ms represents the misspeculation flag and is $\top$ in case of misspeculation and is $\bot$ in case of no misspeculation.

5.2.4 Semantics

The one-step execution of programs is instrumented to add observations (leakages) and directives: $S \xrightarrow[d]{o} S'$. o represents the visible observations to the outside world. The set of observations is defined as follows:

$$o \in Obs \ ::= \ \bullet \mid \mathsf{read}\ a, v, b \mid \mathsf{write}\ a, v \mid \mathsf{bool}\ b \mid \mathsf{bt}\ b \mid \mathsf{unsafe}\ \mu$$

The $\bullet$ observation means there is no visible observation to the outside world during an execution. The observation read a, v, b and write a, v represents the memory accesses leakages. The control-flow leaks the boolean that is represented by bool b. bt b is leaked during the backtrack semantics that backtracks depending on the boolean b where b is the value of the misspeculation flag. The unsafe access leaks an observation represented by unsafe μ where μ is the whole buffered memory.

The program execution depends on directives d issued by an adversary. The set of directives is defined as follows:

$$d \in Dir \;::=\; \mathsf{step} \mid \mathsf{force}\ b \mid \mathsf{load}\ i \mid \mathsf{backtrack} \mid \mathsf{ustep}$$

The step directive allows execution to proceed normally while the force b directive forces execution to follow the branch b. The directive load i determines the step taken by the attacker to read previously stored value from the buffered memory. The backtrack checks if misspeculation has occurred and backtracks in case of misspeculation. The ustep directive captures the unsafe memory action.

Instruction semantics:

$$\frac{}{S \to^{\epsilon}_{\epsilon} S}\ \textbf{0-STEP} \qquad \frac{S \xrightarrow[d]{o} S' \quad S' \overset{O}{\underset{D}{\twoheadrightarrow}}_{n} S''}{S \overset{o::O}{\underset{d::D}{\twoheadrightarrow}} S''}\ \textbf{S-STEP}$$

$$\frac{C = \langle x := e; c, \mu, \rho, ms\rangle}{C :: S \xrightarrow[\mathsf{step}]{\bullet} \langle c, \rho\{x \leftarrow [\![e]\!]_\rho\}, \mu, ms\rangle :: S}\ \textbf{ASSGN}$$

$$\frac{C = \langle x := e; c, \mu, \rho, ms\rangle \quad \mu(\!|(a, [\![e]\!]_\rho)|\!)^i = (v, ms_v)}{C :: S \xrightarrow[\mathsf{load}\ i]{\mathsf{read}\ a, [\![e]\!]_\rho, ms_v} \langle c, \rho\{x \leftarrow [\![e]\!]_\rho\}, \mu, ms \vee ms_v\rangle :: C :: S}\ \textbf{LOAD}$$

$$\frac{C = \langle a[e] := e'; c, \mu, \rho, ms\rangle \quad [\![e]\!]_\rho \in [0, |a|)}{C :: S \xrightarrow[\mathsf{step}]{\mathsf{write}\ a, [\![e]\!]_\rho} \langle c, \rho, [(a, [\![e]\!]_\rho) := [\![e']\!]_\rho]\mu, ms\rangle :: S}\ \textbf{STORE}$$

$$\frac{C = \langle i; c, \mu, \rho, ms\rangle \quad [\![e]\!]_\rho \notin [0, |a|) \quad i = a[e] := e' \vee x := a[e]}{C :: S \xrightarrow[\mathsf{ustep}]{\mathsf{unsafe}\ \mu} \langle c, \rho', \mu', ms\rangle :: S}\ \textbf{UNSAFE}$$

$$\frac{C = \langle \mathsf{if}\ t\ \mathsf{then}\ c_\top\ \mathsf{else}\ c_\bot; c, \mu, \rho, ms\rangle \quad b' = \mathsf{if}\ d = \mathsf{force}\ b\ \mathsf{then}\ b\ \mathsf{else}\ [\![t]\!]_\rho}{C :: S \xrightarrow[d]{\mathsf{bool}\ [\![t]\!]_\rho} \langle c'_b; c, \rho, \mu, b \vee b' \neq [\![t]\!]_\rho\rangle :: C :: S}\ \textbf{COND}$$

$$\frac{\begin{array}{c} C = \langle \mathsf{while}\ t\ \mathsf{do}\ c_0; c, \mu, \rho, ms\rangle \quad c_\top = c_0; \mathsf{while}\ t\ \mathsf{do}\ c_0; c \quad c_\bot = c \\ b' = \mathsf{if}\ d = \mathsf{force} b\ \mathsf{then}\ b\ \mathsf{else}\ [\![t]\!]_\rho \end{array}}{C :: S \xrightarrow[d]{\mathsf{bool}\ [\![t]\!]_\rho} \langle c'_b; c, \rho, \mu, b \vee b' \neq [\![t]\!]_\rho\rangle :: C :: S}\ \textbf{WHILE}$$

$$\frac{}{\langle c, \rho, \mu, \top\rangle :: C :: S \xrightarrow[\mathsf{backtrack}]{\mathsf{bt}\ \top} C :: S}\ \textbf{BTT} \qquad \frac{}{\langle c, \rho, \mu, \bot\rangle :: C :: S \xrightarrow[\mathsf{backtrack}]{\mathsf{bt}\ \bot} \langle c, \rho, \mu, \bot\rangle :: \epsilon}\ \textbf{BTF}$$

$$\frac{}{\langle \mathsf{fence}; c, \rho, \mu, \bot\rangle :: S \xrightarrow[\mathsf{step}]{\bullet} \langle c, \rho, \overline{\mu}, \bot\rangle :: S}\ \textbf{FENCE}$$

Figure 5.3 – Instrumented semantics of language with speculation

One-step execution The relation $S \xrightarrow[d]{o} S'$ means under the directive d; the state S executes in one step to state S' and yields leakage o. The rules are present in Figure 5.3. All rules, except those executing the fence instruction or a backtrack directive, either modify the top configuration on the stack (assignment and stores) or push a new configuration onto the stack (instructions that can lead to misspeculation like conditionals, loops, and load).

The rule **ASSGN** evaluates the expression e and assigns it to register x. It only updates the top configuration and does not produce any leakage.

The rule **LOAD** creates a new configuration in which the buffered memory remains unchanged, and the register map is updated with a value read from memory. The directive is load i, indicating that the memory load happens at index i. The leakage is of the form read $a, [\![e]\!]_\rho, ms_v$. This rule assumes that the memory access is in bounds.

The rule **STORE** stores the value at a memory location. It leaks the memory address (write $a, [\![e]\!]_\rho$) and assumes that the memory access is inbound.

The rule **UNSAFE** resembles an unsafe memory read or write. The accessed address can be out of bounds; hence, the whole buffered memory is leaked unsafe μ. This rule is non-deterministic due to unsafe memory access; therefore, μ' and ρ' are non-deterministically computed.

The rule **COND** creates a new configuration with the same register map and memory map as the top configuration of the current state. It updates both the command and misspeculation flag according to the directive. If the adversary uses the directive force b with $b \in \{\bot, \top\}$, then the execution proceeds into the corresponding branch (c_b). If the adversary uses the step, the condition is evaluated, and execution enters the correct branch. The misspeculation flag is updated accordingly depending on the directive chosen by the attacker. The rule **WHILE** follows the same pattern.

The rule **BTT** and **BTF** defines the semantics of backtrack directives. These directives can occur at any point during execution. If the execution encounters the backtrack directive and misspeculation flag is $\top$, then the rule **BTT** pops the top configuration and restarts execution from the next configuration. It also leaks bt $\top$. If the adversary wants to backtrack further, they may issue multiple backtrack directives. In the situation where the execution encounters a backtrack directive, and the misspeculation flag is $\bot$, then the rule **BTF** clears the stack so that only the top configuration remains. The observation bt $\bot$ is leaked in this case.

The rule **FENCE** resembles the execution of the fence instruction. The fence instruction is executed only with step directive if the misspeculation flag is $\bot$ (no prior misspeculation). After executing a fence instruction, all pending writes in μ are flushed to memory, resulting in the updated memory $\overline{\mu}$.

Multi-step execution The multi-step execution is represented using the relation $S \xrightarrow[D]{O}\!\!\!\!\!\rightarrow S'$. The rules **0-STEP** and **S-STEP** represent the multi-step execution and present in Figure 5.3.

5.3 Speculative safety and Speculative constant-time

This section presents the formal definition of speculative safety and speculative constant time.

Speculative safety Speculative safety states that executing a command, even speculatively, should not lead to illegal memory access.

Definition 6 (Speculative safety)**.**

- *An execution* $S \overset{O}{\underset{D}{\twoheadrightarrow}} S'$ *is safe if* S' *is not of the form* $\langle i; c, \rho, \mu, ms\rangle :: S_0$*, with* $i = x := a[e]$ *or* $i = a[e] := e'$*, and* $[\![e]\!]_\rho \notin [0, |a|)$*.*
- *A state* S *is safe iff every execution* $S \overset{O}{\underset{D}{\twoheadrightarrow}} S'$ *is safe.*
- *A command* c *is safe, written* $c \in safe$ *iff every initial state* $\langle c, \rho, \mu, ms\rangle :: \epsilon$ *is safe.*

Speculative constant-time Speculative constant-time states that if we execute a command from two initial states that do not differ on their public data, we should not be able to distinguish between the sequence of visible observations (provided that the executions can be speculative).

Definition 7 (Speculative constant-time)**.** *Let* ϕ *be an indistinguishable relation on register maps and memories stating that all variables with public data in register maps and memories are equal. A command* c *is speculatively constant-time w.r.t.* ϕ*, written* $c \in \phi - SCT$*, iff for every two executions* $\langle c, \rho_1, m_1, \bot\rangle :: \epsilon \overset{O_1}{\underset{D}{\twoheadrightarrow}} S_1$ *and* $\langle c, \rho_2, m_2, \bot\rangle :: \epsilon \overset{O_2}{\underset{D}{\twoheadrightarrow}} S_2$ *such that* $(\rho_1, m_1)\ \phi\ (\rho_2, m_2)$ *we have* $O_1 = O_2$*.*

5.4 Consistency

The adversarial semantics is called sequentially consistent if it coincides with the standard semantics of programs. From the set of directives $\{\mathsf{load}\ i, \mathsf{step}, \mathsf{force}\ b, \mathsf{backtrack}, \mathsf{ustep}\}$, subset of directives can be derived. They are defined as follows:

- $\mathcal{S} = \{\mathsf{load}\ 0, \mathsf{step}\}$: sequential directives
- $\mathcal{F} = \{\mathsf{load}\ i, \mathsf{step}, \mathsf{force}\ b\}$: forward directives
- $\mathcal{L} = \{\mathsf{load}\ i, \mathsf{step}, \mathsf{force}\ b, \mathsf{backtrack}\}$: legal directives

Three different fragments of the semantics can be defined using these directives and are represented using the relation $S \overset{O}{\underset{D}{\twoheadrightarrow_{\mathcal{X}}}} S'$, where $\mathcal{X}$ is the subset of directives. $\mathcal{X}$ can be $\mathcal{S}$, $\mathcal{F}$, or $\mathcal{L}$. The definitions of safe and $\phi - SCT$ can also be adapted to the subset of directives and are represented as $\mathsf{safe}_{\mathcal{X}}$ and $\phi - SCT_{\mathcal{X}}$

5.4.1 Equivalence relation between configurations

A configuration equivalence relation is defined using the relation $\equiv$

Definition 8 (Configuration equivalence)**.** *Two configurations* C_1 *(*$\langle c_1, \rho_1, m_1, ms_1\rangle$*) and* C_2 *(*$\langle c_2, \rho_2, m_2, ms_2\rangle$*) are equivalent if* $c_1 = c_2$ *and* $ms_1 = ms_2$*.*

Theorem 7 (Configuration equivalence after one-step in forward semantics)**.** *If* $C_1 \overset{O}{\underset{D}{\rightarrow_{\mathcal{F}}}} C_1'$ *and* $C_2 \overset{O}{\underset{D}{\rightarrow_{\mathcal{F}}}} C_2'$ *and* $C_1 \equiv C_2$ *then* $C_1' \equiv C_2'$*.*

Proof. Assume $C_1 = \langle c, \rho_1, m_1, ms\rangle$ and $C_2 = \langle c, \rho_2, m_2, ms\rangle$. The proof follows by doing a case analysis on the first instruction in c. The reasoning involves only forward directives ($\{\mathsf{load}\ i, \mathsf{step}, \mathsf{force}\ b\}$).

- $x := e$, $a[e] := e'$ or fence: The assignment, store, and fence instructions do not update the misspeculation flag and semantics is deterministic under the same set of directives. Hence, it is trivial that $C_1' \equiv C_2'$.
- $x := a[e]$: The load instruction uses the same directive in both the execution and also generates the same leakage o. ms is updated according to memory access done at index $[\![e]\!]_\rho$. Since the directive is the same in both executions, the misspeculation flag obtained during loading data from the buffered memory is also the same. This makes the misspeculation flag in C_1' and C_2' equal as they are computed using the same value. Hence, it is concluded that $C_1' \equiv C_2'$.
- $\mathsf{if}\ t\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2$ or $\mathsf{while}\ t\ \mathsf{do}\ c$: The conditional and loop instructions use the same directive in both the execution and produce the same leakages o that suffices to show that $[\![t]\!]_{\rho_1} = [\![t]\!]_{\rho_2}$. The same directive in both executions suffices to show that the misspeculation flag is updated in a similar manner in both executions as it depends on the directive taken (force or step). An equal directive in both executions also shows that the new state will have the same command. Hence, it is concluded that $C_1' \equiv C_2'$.

□

Theorem 8 (Configuration equivalence after multi-step in forward semantics)**.** *If* $C_1 \overset{O}{\underset{D}{\twoheadrightarrow_{\mathcal{F}}}} C_1'$ *and* $C_1 \overset{O}{\underset{D}{\twoheadrightarrow_{\mathcal{F}}}} C_2'$ *and* $C_1 \equiv C_2$ *then* $C_1' \equiv C_2'$.

Proof. The proof follows by doing induction on D.

- $D = []$: This case is trivial.
- $D \neq []$: This case is proved using the theorem 7 for one step and induction hypothesis for the rest of the steps.

□

Theorem 9 (Legal to forward execution)**.** *If* $\langle c, \rho_1, m_1, \bot\rangle :: \epsilon \overset{O}{\underset{D}{\twoheadrightarrow_{\mathcal{L}}}} C_1 :: S_1$ *and* $\langle c, \rho_2, m_2, \bot\rangle :: \epsilon \overset{O}{\underset{D}{\twoheadrightarrow_{\mathcal{L}}}} C_2 :: S_2$ *and* $\langle c, \rho_1, m_1, \bot\rangle \equiv \langle c, \rho_2, m_2, \bot\rangle$ *then there exists* D' *and* O' *such that* $\langle c, \rho_1, m_1, \bot\rangle :: \epsilon \overset{O'}{\underset{D'}{\twoheadrightarrow_{\mathcal{F}}}} C_1 :: S_1$ *and* $\langle c, \rho_2, m_2, \bot\rangle :: \epsilon \overset{O'}{\underset{D'}{\twoheadrightarrow_{\mathcal{F}}}} C_2 :: S_2$.

Proof. The proof follows by doing induction on D.

- $D = []$: This case is trivial.
- $D = D_1 :: d$: For this case there exists O_1 and o such that

$$\langle c, \rho_1, m_1, \bot\rangle :: \epsilon \overset{O_1}{\underset{D_1}{\twoheadrightarrow_{\mathcal{L}}}} C_1' :: S_1' \overset{o}{\underset{d}{\rightarrow_{\mathcal{L}}}} C_1 :: S_1$$
$$\langle c, \rho_2, m_2, \bot\rangle :: \epsilon \overset{O_1}{\underset{D_1}{\twoheadrightarrow_{\mathcal{L}}}} C_2' :: S_2' \overset{o}{\underset{d}{\rightarrow_{\mathcal{L}}}} C_2 :: S_2$$

The induction hypothesis consists of the first $n-1$ steps. From the induction hypothesis, we can conclude that $\langle c, \rho_1, m_1, \bot\rangle :: \epsilon \overset{O_1'}{\underset{D_1'}{\twoheadrightarrow_{\mathcal{F}}}} C_1' :: S_1'$ and $\langle c, \rho_2, m_2, \bot\rangle :: \epsilon \overset{O_1'}{\underset{D_1'}{\twoheadrightarrow_{\mathcal{F}}}} C_2' :: S_2'$. Now focusing on the last step corresponding to o and d. The proof follows by doing a case analysis on d:

- d $\neq$ backtrack: The case where d is not a *backtrack* directive, the proof is trivial as the directive d will also exist in the directive set $\mathcal{F}$.
- d = backtrack: From theorem 8 applied to $\langle c, \rho_1, m_1, \bot\rangle :: \epsilon \overset{O_1}{\underset{D_1}{\twoheadrightarrow_{\mathcal{F}}}} C_1' :: S_1'$, $\langle c, \rho_2, m_2, \bot\rangle :: \epsilon \overset{O_1}{\underset{D_1}{\twoheadrightarrow_{\mathcal{F}}}} C_2' :: S_2'$ and the hypothesis $\langle c, \rho_1, m_1, \bot\rangle \equiv \langle c, \rho_2, m_2, \bot\rangle$, we know that $C_1' :: S_1' \equiv C_2' :: S_2'$. From the definition of $\equiv$ present in 8, we know that commands and misspeculation flags of C_1' and C_2' are equal. Since the misspeculation flags are same in both the configuration C_1' and C_2', the semantics rule $C_1' :: S_1' \overset{o}{\underset{d}{\rightarrow_{\mathcal{L}}}} C_1 :: S_1$ and $C_2' :: S_2' \overset{o}{\underset{d}{\rightarrow_{\mathcal{L}}}} C_2 :: S_2$ can use **BTT** or **BTF** in both the executions.
  - **BTF** rule: In this case we know that $C_1 = C_1'$ and $C_2 = C_2'$, hence concluded.
  - **BTT** rule: In this case, the semantics backtrack to C_1' and C_2'. The proof follows in this case by doing induction on D_1'.
    - $D_1' = []$: This case is not possible as we know that there exists another step next to it that uses **BTT** rule.
    - $D_1' = D_1'' :: d_1''$: There are various cases to analyze based on d_1''.
      1. case 1: When d_1'' corresponds to assignment, store, or fence semantics, the misspeculation flag is set to true before this step because these instructions do not update the misspeculation flag. Hence, it is proved by the induction hypothesis that there already exists such execution in the forward semantics.
      2. case 2: When d_1'' corresponds to fence instruction, this case is not possible as fence instruction requires the misspeculation flag to be false, but it is true.
      3. case 3: When d_1'' corresponds to load, cond, or while semantics, the misspeculation flag can be updated to true depending on the speculative load or forced branching. According to the semantics of these instructions, the updated configuration is pushed on the top of the stack, and in case of encountering backtrack, the top configuration is popped that is already present in the forward semantics. Hence concluded.

□

5.4.2 Sequential consistency

This section shows that the adversarial semantics is equivalent to the sequential semantics of commands. Sequential semantics does not allow misspeculation. The directive set $\mathcal{S}$

does not use force b or backtrack or ustep directives. It only uses the top configuration, always loads the correct value from the memory, and does not modify the misspeculation flag. The sequential semantics is represented as $\langle c, \rho, \mu, \bot\rangle :: S \overset{O}{\underset{D}{\twoheadrightarrow_{\mathcal{S}}}} \langle c', \rho', \mu', \bot\rangle :: S'$. For readability, it is represented as $\langle c, \rho, \mu\rangle \overset{O}{\underset{D}{\twoheadrightarrow_{\mathcal{S}}}} \langle c', \rho', \mu'\rangle$ because sequential semantics does not update the misspeculation flag and the configurations in S. The sequential consistency can be achieved because any command that is functionally correct under the sequential semantics will be also functionally correct under the adversarial semantics.

Theorem 10 (Sequential consistency)**.** *If* $\langle c, \rho, m, \bot\rangle :: \epsilon \overset{O_1}{\underset{D}{\twoheadrightarrow}} \langle [], \rho', \mu, \bot\rangle :: S$ *then there exists* O_2 *such that* $\langle c, \rho, m\rangle \overset{O_2}{\twoheadrightarrow_{\mathcal{S}}} \langle [], \rho', \overline{\mu}\rangle$.

Proof. According to the definition of $\twoheadrightarrow_{\mathcal{S}}$, we need to consider only the executions, which do not involve backtrack, force b and ustep directives but only uses the directive in the set $\mathcal{S}$. The proof follows by doing case analysis on c and is very trivial because the execution does not consider any misspeculation. □

The above theorem shows that any functionally correct under the sequential semantics command is also functionally correct under the adversarial semantics.

5.4.3 Secure forward consistency

Proving properties about semantics with speculation is more complex than establishing them for sequential semantics because there are more execution paths to be considered. Also, the execution may backtrack at any point, making verifying speculative safety and constant-time more complex. This section shows a way to reason about speculative semantics by only considering the semantics with directives present in set $\mathcal{F}$. The directive set $\mathcal{F}$ is enough to reason about speculative safety and speculative constant-time for executions with backtracking.

Theorem 11 (Safe legal to forward consistency)**.** *A command* c *is* $safe_{\mathcal{L}}$ *iff* c *is* $safe_{\mathcal{F}}$.

Proof. The formal definition of $safe$ is present in 6, and $safe_{\mathcal{F}}$ is the same definition applied to the directive set $\mathcal{F}$. There are two goals to prove:

- $c \in safe_{\mathcal{L}} \implies c \in safe_{\mathcal{F}}$: In this case, the goal is to prove that for all ρ, m and C if $\langle c, \rho, m, \bot\rangle \overset{O}{\underset{D}{\twoheadrightarrow_{\mathcal{F}}}} C$ then C is a safe configuration. We know from the assumption that c is $safe_{\mathcal{L}}$, which shows $\langle c, \rho, m, \bot\rangle$ is safe, $\langle c, \rho, m, \bot\rangle \overset{O}{\underset{D}{\twoheadrightarrow_{\mathcal{L}}}} C$ is a safe execution and resultant command C is also safe. This suffices to show that c will never use the UNSAFE rule. Since c cannot execute using the UNSAFE rule, the execution of c will always produce a safe command.

- $c \in safe_{\mathcal{F}} \implies c \in safe_{\mathcal{L}}$: In this case, the goal is to prove that for all ρ, m and C, if $\langle c, \rho, m, \bot\rangle \overset{O}{\underset{D}{\twoheadrightarrow_{\mathcal{L}}}} C$ then C is a safe configuration. We know that $\langle c, \rho, m, \bot\rangle \equiv \langle c, \rho, m, \bot\rangle$ by reflexivity. Applying the theorem 9 twice on $\langle c, \rho, m, \bot\rangle \overset{O}{\underset{D}{\twoheadrightarrow_{\mathcal{L}}}} C$ and $\langle c, \rho, m, \bot\rangle \equiv \langle c, \rho, m, \bot\rangle$, we know that there exists D' and O' such that

$\langle c, \rho, m, \bot\rangle \overset{O'}{\underset{D'}{\twoheadrightarrow_{\mathcal{F}}}} C$. Also by assumption we know that C is a safe configuration and $c \in safe_{\mathcal{F}}$. Since C is a safe configuration, we can conclude $c \in safe_{\mathcal{L}}$ because C is obtained by evaluating the command c, and an unsafe command cannot lead to a safe configuration.

□

Theorem 12 (Safe legal consistency)**.** *A command c is safe iff it is $safe_{\mathcal{L}}$.*

Proof. There are two goals to prove: (i) $c \in safe \implies c \in safe_{\mathcal{L}}$. (ii) $c \in safe_{\mathcal{L}} \implies c \in safe$.

- $c \in safe \implies c \in safe_{\mathcal{L}}$: From the assumption, we know that c is safe; hence it will lead to a configuration C which will be a safe configuration. Since c is safe, it will never use the UNSAFE rule. Hence concluded that $c \in safe_{\mathcal{L}}$.

- $c \in safe_{\mathcal{L}} \implies c \in safe$: We need to show that forall ρ, m and C, if $\langle c, \rho, m, \bot\rangle :: \epsilon \overset{O}{\underset{D}{\twoheadrightarrow}} C :: S$ then C is a safe configuration. From the assumption, we know that $c \in safe_{\mathcal{L}}$ which means it never uses the UNSAFE rule. Since it never uses UNSAFE rule, it can never lead to unsafe configuration; hence, C is a safe configuration that suffices to show that $c \in safe$.

□

Theorem 13 (Safe forward consistency)**.** *A command c is safe iff it is $safe_{\mathcal{F}}$.*

Proof. There are two goals to prove: (i) $c \in safe \implies c \in safe_{\mathcal{F}}$. (ii) $c \in safe_{\mathcal{F}} \implies c \in safe$.

- $c \in safe \implies c \in safe_{\mathcal{F}}$: Applying the theorem 12 to the hypothesis $c \in safe$, we know that $c \in safe_{\mathcal{L}}$. Applying the theorem 11 to the result of theorem 12 i.e., $c \in safe_{\mathcal{L}}$, we conclude that $c \in safe_{\mathcal{F}}$.

- $c \in safe_{\mathcal{F}} \implies c \in safe$: Applying the theorem 11 to the hypothesis $c \in safe_{\mathcal{F}}$, we know that $c \in safe_{\mathcal{L}}$. Applying the theorem 12 to the result of the theorem 11 i.e., $c \in safe_{\mathcal{L}}$, we conclude that $c \in safe$.

□

Theorem 14 (Secure forward consistency)**.** *For any speculative safe command c, c is $\phi - SCT$ iff c is $\phi - SCT_{\mathcal{F}}$.*

Proof. By contraposition, we will prove if c is not $\phi - SCT$ then c is not $\phi - SCT_{\mathcal{F}}$. From the assumption, we know that c is a speculative safe command; hence, it suffices to replace the statement "c is not $\phi - SCT$" with "c is not $\phi - SCT_{\mathcal{L}}$" because speculative safe command c will never execute using *ustep* directive. $c \notin \phi - SCT_{\mathcal{L}}$ implies there exists two derivations such that:

- $(\rho_1, m_1)\phi(\rho_2, m_2)$

- $\langle c, \rho_1, m_1, \bot\rangle :: \epsilon \overset{O}{\underset{D}{\twoheadrightarrow_{\mathcal{L}}}} C_1' :: S_1' \overset{o_1}{\underset{d}{\rightarrow_{\mathcal{L}}}} C_1 :: S_1$

- $\langle c, \rho_2, m_2, \bot\rangle :: \epsilon \xrightarrow[D]{O}_{\mathcal{L}} C_2' :: S_2' \xrightarrow[d]{o_2}_{\mathcal{L}} C_2 :: S_2$
- $o_1 \neq o_2$

By theorem 9, there exists D' and O' such that: $\langle c, \rho_1, m_1, \bot\rangle :: \epsilon \xrightarrow[D']{O'}_{\mathcal{F}} C_1'$ and $\langle c, \rho_2, m_2, \bot\rangle :: \epsilon \xrightarrow[D']{O'}_{\mathcal{F}} C_2'$. Now, the proof follows by doing a case analysis on d.

- d $\neq$ backtrack: From assumption we know that $o_1 \neq o_2$. Hence for the whole execution $\langle c, \rho_1, m_1, \bot\rangle :: \epsilon \xrightarrow[D']{O'::o_1}_{\mathcal{F}} C_1 :: S_1$ and $\langle c, \rho_2, m_2, \bot\rangle :: \epsilon \xrightarrow[D']{O'::o_2}_{\mathcal{F}} C_2 :: S_2$, we can conclude $O' :: o_1 \neq O' :: o_2$.
- d = backtrack: From the theorem 8 applied to $\langle c, \rho_1, m_1, \bot\rangle :: \epsilon \xrightarrow[D]{O}_{\mathcal{F}} C_1'$ and $\langle c, \rho_2, m_2, \bot\rangle :: \epsilon \xrightarrow[D']{O'}_{\mathcal{F}} C_2'$, we know that $C_1' \equiv C_2'$ that means the misspeculation flag of C_1' and C_2' are equal. It also suffices to show that the leakages o_1 and o_2 during the two execution will be the same (either bt $\bot$ or bt $\top$) as the directive is a backtrack directive. Hence, this contradicts our assumption $o_1 \neq o_2$.

□

$$\frac{\{I\}\ c\ \{O\} \qquad I \subseteq I' \qquad O' \subseteq O}{\{I'\}\ c\ \{O'\}}\ [\text{SCT-Conseq}]$$

$$\frac{}{\{O\}\ \mathsf{fence}\ \{O\}}\ [\text{SCT-Fence}]$$

$$\frac{O \setminus \{x\} \subseteq I \qquad x \in O \implies \mathsf{fv}(e) \subseteq I}{\{I\}\ x := e\ \{O\}}\ [\text{SCT-Assign}]$$

$$\frac{(O \setminus \{x\}) \cup \mathsf{fv}(i) \subseteq I \qquad x \in O \implies a \in I}{\{I\}\ x := a[i]\ \{O\}}\ [\text{SCT-Load}]$$

$$\frac{O \cup \mathsf{fv}(i) \subseteq I \qquad a \in O \implies \mathsf{fv}(e) \subseteq I}{\{I\}\ a[i] := e\ \{O\}}\ [\text{SCT-Store}]$$

$$\frac{\{I\}\ c_1\ \{O\} \qquad \{I\}\ c_2\ \{O\} \qquad \mathsf{fv}(e) \subseteq I}{\{I\}\ \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \{O\}}\ [\text{SCT-Cond}]$$

$$\frac{\{O\}\ c\ \{O\} \qquad \mathsf{fv}(e) \subseteq O}{\{O\}\ \mathsf{while}\ e\ \mathsf{do}\ c\ \{O\}}\ [\text{SCT-While}] \qquad \frac{}{\{O\}\ []\ \{O\}}\ [\text{SCT-Empty}]$$

$$\frac{\{X\}\ c\ \{O\} \qquad \{I\}\ i\ \{X\}}{\{I\}\ i; c\ \{O\}}\ [\text{SCT-Seq}]$$

Figure 5.4 – Proof system for speculative constant-time.

5.5 Verification of speculative safety and speculative constant-time

This section overviews the verification methods for speculative safety and speculative constant-time. The speculative constant-time analysis is a fully automated analysis and is presented using a declarative style by means of a proof system.

5.5.1 Speculative safety

The speculative safety checker developed in this work is based on abstract interpretation techniques [Cousot and Cousot, 1977]. The checker executes programs by soundly over-approximating the semantics of every instruction. Sound transformations of the abstract state must be designed for every instruction of the language. The program is then abstractly executed using these sound abstract transformations.

The checker differs from the Jasmin safety analyzer. It modifies the abstract semantics of conditional and memory operations to include speculative semantics. For example, when entering the then branch of an if statement, the condition of the if is not assumed to hold always. This modification supports the adversary's behavior (soundly accounting for misspeculation). Only weak updates are performed on values stored in memory to resemble the delay in memory writes. For example, a memory store $a[i] := e$ will update the possible value of $a[i]$ to any possible value (the abstract evaluation) of e or any possible old value of $a[i]$. This soundly reflects the adversary's ability to pick stale values from the write buffer.

To model fences, the analyzer computes simultaneously a pair of abstract values $(\mathcal{A}_{std}{}^{\#}, \mathcal{A}_{spec}{}^{\#})$, where $\mathcal{A}_{std}{}^{\#}$ follows the non-speculative semantics, while $\mathcal{A}_{spec}{}^{\#}$ follows the speculative semantics. On the execution of fence, the speculative abstract value is replaced by the standard abstract value.

The goal of the analysis is to check that there are no memory safety violations throughout the execution of the program (carried out on abstract values). The soundness of the abstraction mechanism suffices to show that a safe program under abstract semantics entails safety under concrete (speculative) semantics.

5.5.2 Speculative constant-time

The speculative constant-time analysis manipulates a judgment of the form $\{I\}\ c\ \{O\}$, where I and O are sets of variables (registers and arrays) and c is a command. Informally, it ensures that if two executions of c start on equivalent states w.r.t I, then the resulting states are equivalent w.r.t. O, and the generated leakages are equal. The methodology is based on dependency analysis that produces constraints based on the executions of commands. The dependency analysis is a bit different from the analysis done for constant-time property as it needs to define equivalence relation, considering the speculation.

The proof rules are present in Figure 5.4. The rule [SCT-CONSEQ] is the rule for the consequence that says that we can use the subset relation of input and output variables to reason about the bigger set of variables.

The rule [SCT-FENCE] states that equivalence w.r.t O is preserved by executing a [FENCE] instruction. The semantics of the [FENCE] instruction ensures that the buffer memory is flushed to the main memory; hence, the declarative rule is the direct consequence of it.

The rule [SCT-ASSGN] requires that $O \setminus \{x\} \subseteq I$. This guarantees that equivalence on all arrays and registers in O except x already holds prior to this execution. It also requires that if $x \in O$, then $\mathsf{fv}(e) \subseteq I$ where $\mathsf{fv}(e)$ are the free variables of e. This inclusion ensures that evaluations of e will be the same in both executions, giving equal values for x as well.

The rule [SCT-LOAD] also requires that $O\setminus\{x\} \subseteq I$. Additionally, it requires $\mathsf{fv}(i) \subseteq I$, which resembles the memory access does not leak. Finally, it requires that if $x \in O$, then $a \in I$. The latter enforces that the buffered memories coincide on a, and the same values are stored in x.

The rule [SCT-STORE] requires that $O \subseteq I$ and $\mathsf{fv}(i) \subseteq I$. The first inclusion guarantees that equivalence on all arrays in O and on all registers in O already holds before executing the store. The second inclusion guarantees that both execution of the index i will be equal, i.e., that the access does not leak. Moreover, it requires that if $a \in O$, then $\mathsf{fv}(e) \subseteq I$. This ensures that both evaluations of e give equal values so that (together with $\mathsf{fv}(i) \subseteq I$) equivalence of buffered memories is preserved.

The rule [SCT-COND] requires that $\mathsf{fv}(e) \subseteq I$ (so that the conditions in the two executions are equal) and that the judgments $\{I\}\ c_i\ \{O\}$ hold for $i = 1, 2$. The rule [SCT-WHILE] requires that $\mathsf{fv}(e) \subseteq O$ and O is an invariant, i.e. the loop body preserves O-equivalence.

5.5.3 Soundness of the declarative judgment for Speculative constant-time

Definition 9 (Equivalence $\approx$)**.** *Equivalence between the states must satisfy the following:*

- *Two register maps ρ_1 and ρ_2 are equivalent w.r.t O, written as $\rho_1 \approx_o \rho_2$, iff $\rho_1[x] = \rho_2[x]$ for all $x \in \mathcal{X} \cap O$.*
- *Two buffered memories μ_1 and μ_2 are equivalent w.r.t O, written as $\mu_1 \approx_o \mu_2$ is derivable from the rules*

$$\frac{\forall a \in \mathcal{A} \cap O,\ \forall v \in [0, |a|), m_1[a, v] = m_2[a, v]}{m_1 \approx_O m_2}$$

$$\frac{\mu_1 \approx_O \mu_2 \quad a \in O \Rightarrow v_1 = v_2}{[(a, w) := v_1]\mu_1 \approx_O [(a, w) := v_2]\mu_2}\ .$$

- *The relation $\approx_O$ is defined by the clause $\rho_1, \mu_1 \approx_O \rho_2, \mu_2$ iff $\rho_1 \approx_O \rho_2$ and $\mu_1 \approx_O \mu_2$.*

Theorem 15. *If $\mu_1 \approx_X \mu_2$ then $\overline{\mu_1} \approx_X \overline{\mu_2}$.*

Theorem 16. *If $\rho_1 \approx_X \rho_2$ and $\mathsf{fv}(e) \subseteq X$ then $[\![e]\!]_{\rho_1} = [\![e]\!]_{\rho_2}$.*

Theorem 17 (Subject Reduction)**.** *Let $C_1 = \langle i; c, \rho_1, \mu_1, b\rangle$ and $C_2 = \langle i; c, \rho_2, \mu_2, b\rangle$. If the following holds*

- *$\{I\}\ \ i\ \ \{X\}$ and $\{X\}\ \ c\ \ \{O\}$*
- *$C_1 \approx_I C_2$*
- *$C_1 \underset{d}{\overset{o_1}{\rightarrow}}_F C_1'$ and $C_2 \underset{d}{\overset{o_2}{\rightarrow}}_F C_2'$*

where $C_1' = \langle c', \rho_1', \mu_1', b' \rangle$*, then there exists* I' *such that* $\{I'\}\ c'\ \{O\}$ *and* $C_1' \approx_{I'} C_2'$ *and* $o_1 = o_2$.

Proof. The proof proceeds by doing induction on $\{I\}\ i\ \{X\}$. There are various sub-goals to prove depending on the set of rules defined in Figure 5.4.

- [SCT-Conseq]: there exists I_1 and X_1 such that $\{I_1\}\ i\ \{X_1\}$, $I_1 \subseteq I'$ and $X \subseteq X_1$. We can trivially conclude using the induction hypothesis on $\{I_1\}\ i\ \{X_1\}$, notice that we have $\{X_1\}\ c\ \{O\}$.

- [SCT-Fence]: we have $I = X$ and $C_1 \xrightarrow[d]{o_1}_F C_1'$ and $C_2 \xrightarrow[d]{o_2}_F C_2'$ necessarily correspond to an application of the rule [Fence] (so $o_1 = o_2 = \bullet$). We conclude using $I' = X$ and by applying Proposition 15.

- [SCT-Assign]: we have $i = x := e$, $X \setminus \{x\} \subseteq I$ and $x \in X \implies \mathsf{fv}(e) \subseteq I$. $C_1 \xrightarrow[d]{o_1}_F C_1'$ and $C_2 \xrightarrow[d]{o_2}_F C_2'$ correspond to the rule [Assign], so

$$C_j' = \langle c, \rho_j\{x := [\![e]\!]_{\rho_j}\}, \mu_j, b_j \rangle$$

  and $o_1 = o_2 = \bullet$. $\rho_1 \approx_I \rho_2$ and conditions $X \setminus \{x\} \subseteq I$ and $x \in X \implies \mathsf{fv}(e) \subseteq I$ implies $\rho_1\{x := [\![e]\!]_{\rho_1}\} \approx_X \rho_2\{x := [\![e]\!]_{\rho_2}\}$. This allows to conclude using $I' = X$.

- [SCT-Load]: we have $i = x := a[e]$ and $(X \setminus \{x\}) \cup \mathsf{fv}(e) \subseteq I$ and $x \in X \implies a \in I$. $C_1 \xrightarrow[d]{o_1}_F C_1'$ and $C_2 \xrightarrow[d]{o_2}_F C_2'$ correspond to the rule [Load], so

$$\begin{gathered} d = \mathsf{load}(i,) \\ \mu_j(\!|((a,,)[\![e]\!]_{\rho_j})|\!)^i = (v_j, b_v^j) \\ C_j' = \langle c, \rho_j\{x := v_j\}, \mu_j, b_j \vee b_v^j \rangle \\ o_j = \mathsf{read}\ a, [\![e]\!]\rho_j, b_v^j \end{gathered}$$

  $\rho_1 \approx_I \rho_2$ and $\mathsf{fv}(e) \subseteq I$ implies $[\![e]\!]_{\rho_1} = [\![e]\!]_{\rho_2}$. $\mu_1 \approx_I \mu_2$ implies $b_v^1 = b_v^2$ and so $o_1 = o_2$. Furthermore if $x \in X$ we have $a \in I$ and so $v_1 = v_2$ and $\rho_1\{x := v_1\} \approx_X \rho_2\{x := v_2\}$. We conclude using $I' = X$.

- [SCT-Store]: we have $i = a[e] := e'$ and $X \cup \mathsf{fv}(e) \subseteq I$ and $a \in X \implies \mathsf{fv}(e') \subseteq I$. Both evaluations use the [Store] rule, so

$$\begin{aligned} C_j' &= \langle c, \rho_j, [(a, [\![e]\!]_\rho \rho_j) := [\![e']\!]_{\rho_j}]\mu_j, b_j \rangle \\ o_j &= \mathsf{write}\ a, [\![e]\!]_{\rho_j} \end{aligned}$$

  $\mathsf{fv}(e) \subseteq I$ implies $[\![e]\!]_{\rho_1} = [\![e]\!]_{\rho_2}$, so $o_1 = o_2$. Furthermore, we prove that

$$[(a, [\![e]\!]_{\rho_1}) := [\![e']\!]_{\rho_1}]\mu_1 \approx_X [(a, [\![e]\!]_{\rho_2}) := [\![e']\!]_{\rho_2}]\mu_2$$

  and we can conclude using $I' = X$.

- [SCT-Cond]: we have $i = \mathsf{if}\ e\ \mathsf{then}\ c_\top\ \mathsf{else}\ c_\bot$ and $\{I\}\ c_\top\ \{X\}$, $\{I\}\ c_\bot\ \{X\}$, and $\mathsf{fv}(e) \subseteq I$. Both evaluations use the [Cond] rule with d, so we have

$$\begin{aligned} C_j' &= \langle c_{b'}; c, \rho_j, \mu_j, b_j \vee b_j' \neq [\![e]\!]_{\rho_j} \rangle \\ o_j &= \mathsf{branch}\ [\![e]\!]_{\rho_j} \end{aligned}$$

  $\mathsf{fv}(e) \subseteq I$ implies $[\![e]\!]_{\rho_1} = [\![e]\!]_{\rho_2}$ which concludes $b_1' = b_2'$ and $o_1 = o_2$. Remark that $\{I\}\ c_\top; c\ \{O\}$ and $\{I\}\ c_\bot; c\ \{O\}$ are derivable. So, we conclude using $I' = I$.

- [SCT-While]: We have $I = X$. This case is similar to the previous one; the key point is to show that $\{I\}\ c_0; \mathsf{while}\ e\ \mathsf{do}\ c_0; c\ \{O\}$.

□

Theorem 18 (Forward Soundess). *Let* $C_1 = \langle c, \rho_1, \mu_1, b\rangle$ *and* $C_2 = \langle c, \rho_2, \mu_2, b\rangle$. *If the following holds*

- $\{I\}\ \ c\ \ \{O\}$
- $C_1 \approx_I C_2$
- $C_1 \overset{O_1}{\underset{D}{\twoheadrightarrow_F}} C_1'$ *and* $C_2 \overset{O_2}{\underset{D}{\twoheadrightarrow_F}} C_2'$

where $C_1' = \langle c', \rho_1', \mu_1', b'\rangle$, *then there exists* I' *such that* $\{I'\}\ c'\ \{O\}$ *and* $C_1' \approx_{I'} C_2'$ *and* $O_1 = O_2$.

Proof. By induction on D, using theorem 17. The base case $D = \phi$ is trivially proved by taking $I = I'$. □

Theorem 19 (Soundness). *If* c *is speculative safe and* $\{I\}\ c\ \{\emptyset\}$ *is derivable then* $c \in \approx_I$*-SCT.*

Proof. The proof is a direct consequence of theorem 18. □

5.6 Integration into the Jasmin compiler

As discussed in Chapter 1, the Jasmin compiler consists of several compiler passes. The Chapter 2 shows that the Jasmin compiler preserves the constant-time property, but we cannot guarantee this for the Jasmin programs that can speculate. The problem lies with the *stack sharing* compiler pass. The *stack sharing* compiler passes try to reduce the stack size by merging different stack variables. This transformation can create Spectre-STL vulnerabilities. Here are two examples showing the possibilities of vulnerabilities.

```
1 a[0] = s; store secret value
2 ...
3 b[0] = p; store public value at a diff location
4 x = b[0]; can only load public value p
5 y = c[x]; secret independent memory access
```

```
1 a[0] = s; store secret value
2 ...
3 a[0] = p; store public value at same location
4 x = a[0]; can speculatevily load secret s
5 y = c[x]; secret dependent memory access
```

Figure 5.5 – Example program: before and after stack sharing

Figure 5.5 presents two programs; on the left is the program before the stack sharing, and on the right is the program after the stack sharing. The variable s is of type secret, and the variable p is of type public. In the program on the left, the memory access $c[x]$ leaks no information related to secret data because x is assigned the value p, which is public.

If the array a is dead after line 2, then the stack-sharing transformation preserves the semantics of programs, leading to the transformed program on the right. Due to stack sharing transformation, the arrays a and b from the program on the left now share the array a. Now, the program can speculatively reach to the secret data in the last line $c[x]$.

One solution is to modify this pass to restrict the merging of stack variables, i.e., by requiring that only stack variables isolated by a fence instruction are merged. Unfortunately, this solution incurs a significant performance cost and is not aligned with Jasmin's philosophy of keeping the compiler predictable. Instead, in this work, Jasmin is modified to check speculative safety and speculative constant-time after stack sharing. This will prevent any insecure variable merging.

After stack sharing passes, each stack variable corresponds to exactly one stack position. As a result, the remaining compiler passes in Jasmin all preserve speculative constant-time and safety. Lowering replaces high-level Jasmin instructions with low-level assembly equivalent instructions. The only new variables that may be introduced are register variables, mainly boolean flags, so there is no issue. The register allocation renames register variables to actual register names. This pass leaves stack variables and the leakages untouched. The deadcode compiler pass does not exploit the branch condition, leaving the program's speculative semantics untouched. The stack allocation pass maps stack variables to stack positions. Since each stack variable corresponds to exactly one stack position after stack sharing, there is no further issue. In stack allocation, new leakages are introduced due to transforming the stack variables into memory loads, but we have already proved the leakages are preserved by this pass in chapter 2. Then, linearization removes structured control-flow instructions and replaces them with jumps that preserve leakages in a direct way. The final pass is assembly generation, which also preserves leakage.

5.6.1 Integration into the Jasmin workflow

The typical workflow for Jasmin verification is to establish functional correctness, safety, provable security, and timing side-channel protection of Jasmin implementations and then derive the same guarantees for the generated assembly programs. By theorem 10, there exists a sequential semantics for every semantics with speculation. We know that the properties need to be established only for the sequential semantics of the source as it will also be carried to the speculative semantics of the source. Arguing that the guarantees extend to the speculative semantics of assembly programs requires a bit more work. First, we need to define the adversarial semantics of assembly programs and prove that the assembly-level counterpart of theorem 10. Since the Jasmin compiler is proved to be functionally correct for the sequential semantics and together with theorem 10, it entails that the Jasmin compiler is also correct for the speculative semantics. This, in turn, suffices to obtain the guarantees for the speculative semantics of assembly programs.

The proofs of functional correctness and provable security can simply use the existing poof infrastructure, based on the interpretation of Jasmin programs to EasyCrypt [Barthe et al., 2011a] [Barthe et al., 2014b]. Proving functional correctness and provable security of new (speculative secure) implementations can be significantly simplified when verified implementations already exist with proofs of functional correctness and provable security for the sequential semantics. Specifically, it suffices to show functional equivalence between the two implementations.

5.7 Evaluation

The evaluation of the approach described in this chapter answers two questions:

- How much development and verification effort is required to harden implementations to be speculatively constant-time?

- What is the runtime performance overhead of code that is speculatively constant-time?

The answers to these questions are based on benchmarking the Jasmin implementations of ChaCha20 and Poly1305.

5.7.1 Methodology

Benchmarks The baselines for benchmarks are Jasmin-generated/verified assembly implementations of ChaCha20 and Poly1305 developed by Almeida et al. [Almeida et al., 2020]. Each primitive has a scalar implementation and an AVX2-vectorized implementation. The scalar implementations are platform-agnostic but slower. Conversely, the AVX2 implementations are platform-specific but faster, taking advantage of Intel's AVX2 vector instructions that operate on multiple values simultaneously. All of these implementations have mechanized proofs of functional correctness, memory safety, and constant-time, and have performance competitive with the fast, widely deployed (but unverified) implementations from OpenSSL [Openssl, 2013]—it includes the scalar and AVX2-vectorized implementations of ChaCha20 and Poly1305 from OpenSSL in benchmarks to serve as reference points.

Experimental setup Experiments are conducted on one core of an Intel Core i7-8565U CPU clocked at 1.8 GHz with hyperthreading and TurboBoost disabled. The CPU is running microcode version `0x9a`, i.e., without the transient-execution-attack mitigations introduced with update `0xd6`. The machine has 16 GB of RAM and runs Arch Linux with kernel version 5.7.12. We collect measurements using the benchmarking infrastructure offered by SUPERCOP [Bernstein and Lange, 2009].

Benchmarks are collected on an otherwise idle system. As the cost for LFENCE instructions typically increases on busy systems with large cache-miss rate, the relative cost for the countermeasures reported should be considered a lower bound.

5.7.2 Developer and verification effort

For ensuring that the Jasmin source code is speculative constant-time, two different methods are used in practice. First, use of a fence-only approach, where the fence is added after every conditional in the program. In particular, this requires a fence at the beginning of the body of every `while` loop. This approach has the advantage of being simple and trivially leaves the non-speculative semantics of the program unchanged, leading to simpler functional correctness proofs. However, using the fence method sometimes leads to a large performance penalty. We also examined another, more subtle approach using conditional moves (movcc) instructions: In certain cases, it is possible to replace a fence with a few conditional move instructions, which can reset the program's state to safe values whenever misspeculation occurs. This recovers the lost performance but requires marginally more functional, correctness-proof effort.

```
while(inlen >= 16){
    h = load_add(h, in);
    h = mulmod(h, r);
    in += 16;
    inlen -= 16;
}
```

```
while(inlen >= 16){
    #LFENCE;
    h = load_add(h, in);
    h = mulmod(h, r);
    in += 16;
    inlen -= 16;
}
```

```
stack u64 s_in;
s_in = in;
if (inlen >= 16) {
    #LFENCE;
    while{
        in = s_in
                if inlen < 16;
        inlen = 16
                if inlen < 16;
        h = load_add(h, in);
        h = mulmod(h, r);
        in += 16;
        inlen -= 16;
        } (inlen >= 16)
}
```

Figure 5.6 – Speculative safety violation in Poly1305 (top-left) and countermeasures (bottom-left and right). By convention, inlen is a 64-bit register variable.

Speculative safety Most of the development effort for protecting implementations is in fixing speculative safety issues. This section presents some examples in Figure 5.6 to illustrate the changes required to prove speculative safety. Top-left in Figure 5.6 presents the main loop of the Poly1305 scalar implementation. Initially, the pointer in points to the beginning of the input (to be authenticated), and inlen is the message length. At each iteration of the loop, a block of 16 bytes of the input is read using load_add(h, in), the message authentication code h is updated by mulmod(h, r), and finally the input pointer in is increased so that it points to the next block of 16 bytes, and inlen is decreased by 16.

While this code is safe under sequential semantics, it is unsafe under adversarial semantics. If the condition of while loop is misspeculated, then the loop will execute the body even when the guard is unsatisfied. Misspeculation might cause a buffer overflow on the input. More precisely, if we misspeculate k times, then we overflow by $16.(k-1)+1$ to $16.k$ bytes.

The program on the bottom-left and right-side of Figure 5.6 presents two different countermeasures to give protection against speculative overflow. The fence-based countermeasure (bottom-left) adds a fence instruction at the beginning of each loop iteration to ensure that the loop condition has been correctly evaluated. The program in the right of Figure 5.6 first stores the initial value of the input pointer in the stack variable s_in. The fence at the beginning of the if statement ensures that the store to the s_in is correctly performed when entering the loop. The costly fence is replaced by conditional moves, which resets the pointer and length to safe values if misspeculated. The if instruction ensures safety by checking that the inlen is at least 16, even for misspeculating execution.

Speculative constant-time We found that, after addressing speculative safety, there was relatively little additional work needed to achieve speculative constant-time, aside from occasional fixes necessary to address stack sharing issues (see Section 5.6). This is perhaps not surprising, since the speculative constant-time checker differs little from the classic constant-time checker. Stack-sharing issues showed up just once throughout our case studies in the scalar implementation of ChaCha20, and only required a simple code fix to prevent the offending stack share.

Functional correctness and provable security The functional correctness of the implementations is proved by equivalence checking with the implementations present in the paper [Almeida et al., 2020], for which functional correctness is already established. The equivalence proofs are mostly automatic, except for the proof of the movcc version of Poly1305, which requires providing a simple invariant.

5.7.3 Performance overhead

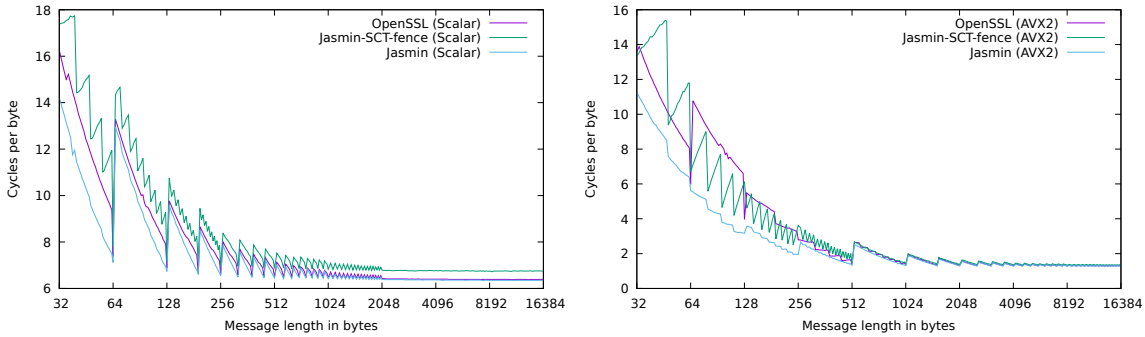


Figure 5.7 – ChaCha20 benchmarks, scalar and AVX2. Lower numbers are better.

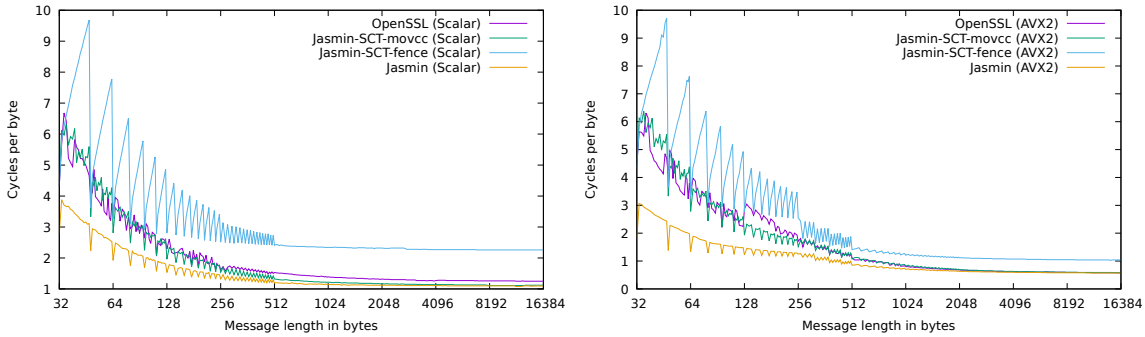


Figure 5.8 – Poly1305 benchmarks, scalar and AVX2. Lower numbers are better.

Figures 5.7 and 5.8 show the benchmarking results for ChaCha20 and Poly1305, respectively. They report the median cycles per byte for processing messages ranging in length from 32 to 16384 bytes.

For both the scalar and AVX2 implementations of ChaCha20, the movcc method resulted in nearly identical performance as the fence method, so we only report on the latter. For the ChaCha20 scalar implementations, the baseline Jasmin implementation enjoys performance competitive with OpenSSL, even slightly beating it. As expected, the SCT implementation is slightly slower across all message lengths, with the gaps being more prominent at the smaller message lengths. For the ChaCha20 AVX2 implementations, all implementations, whether SCT or not, enjoy a similar performance at the mid to larger message lengths. For small messages, however, the baseline Jasmin implementation is the fastest, while the other implementations trade positions in the range of small message lengths.

For the Poly1305 scalar implementations, the baseline Jasmin implementation outperforms OpenSSL across all message lengths, with the gaps being more prominent at the smaller message lengths. The Jasmin-SCT-movcc implementation enjoys performance

competitive with OpenSSL. The Jasmin-SCT-fence implementation, however, is considerably slower than the rest. For Poly1305 AVX2 implementations, the baseline Jasmin implementation outperforms OpenSSL and Jasmin-SCT-movcc, which are comparable at the smaller message lengths, but enjoy a similar performance at the mid to larger message lengths. Again, the Jasmin-SCT-fence implementation is considerably slower, but the gap is less apparent than in the scalar case.

Overall, the performance overhead of making code SCT is relatively modest. Interestingly, platform-specific, vectorized implementations are easier to protect due to the availability of additional general-purpose registers, leading to fewer (potentially dangerous) memory accesses. Consequently, SCT vectorized implementations incur less overhead than their platform-agnostic, scalar counterparts. Moreover, the best method for protecting code while preserving efficiency varies by implementation. For ChaCha20, the movcc and fence methods fared similarly. For Poly1305, the movcc method performed significantly better.

5.8 Discussion

This section discusses limitations, generalization, complementary problems, and comparison with other approaches.

5.8.1 Machine-checked guarantees

The adversarial semantics is not mechanized and hence is not formally verified for correctness. In contrast to this, the sequential semantics is mechanized and proved for its correctness using the Coq proof assistant. Formalization of adversarial semantics is required to provide machine-checked proofs for all the theorems defined in the above sections.

The constant-time property involves sequential semantics, and its preservation is also machine-checked, as explained in Chapter 2. However, mechanizing a proof of preservation of speculative constant-time seems simpler and less work than preservation of constant-time because the analysis is carried out at a lower level. It requires developing methods for proving the preservation of speculative constant-time; however, it will not be challenging to adapt the techniques from Chapter 2 on constant-time preserving compilation. A well-adapted methodology with full machine-checked proof is presented in Chapter 6 for providing guarantees against timing-based side-channel attacks where the semantics are speculative and revolve around a simple language. The same approach is extended for the Jasmin framework in Chapter 7, but the proof for soundness is done on paper.

5.8.2 Requirement of memory safety

The approach explained in this chapter requires the programs to be memory-safe in order to achieve speculative constant-time property. The reasons are:

- The dependency analysis explained in Figure 5.4 is not expressive enough to reason about speculative constant-time in case of out-of-bound memory accesses.

- The operational semantics of unsafe load and store leak the whole memory. Proving equivalence on leakages in case of unsafe load and store is not desirable because it is impossible to prove equivalence over the entire memory.

Proving memory safety for each implementation requires some extra work, and it can be resolved by using a more expressive information-flow-based type system and semantics that do not leak whole memory in case of out-of-bound access (Chapter 6 and Chapter 7 explain methodology to enforce speculative constant-time without a need of memory safety).

5.8.3 Other speculative execution attacks

The adversarial semantics primarily covers Spectre-PHT and Spectre-STL attacks (falls in the category of Spectre v1 and Spectre v4), but it does not cover Spectre-BTB (falls in the category of Spectre v2) [Kocher et al., 2019a]. It is a variant of Spectre in which the attacker mistrains the Branch Target Buffer (BTB), which predicts the destinations of indirect jumps. Spectre-BTB attacks can speculatively redirect control flow. The development is carried out for Jasmin, and in Jasmin, there was no support for indirect jumps (during the time of development of this work) because the design decision is made to write efficient and correct cryptographic code that tends to have simple structures control flow. For the cases where the software must include indirect jumps, hardware manufacturers have developed CPU-level mitigations to prevent an attacker from Spectre-BTB [Intel, b, Intel, a].

Spectre-RSB [Koruyeh et al., 2018] [Maisuradze and Rossow, 2018] attacks abuse the Return Stack Buffer (RSB) to speculatively redirect control flow similar to a Spectre-BTB attack. The RSB may misdirect the destinations of return addresses when the call and return instructions are unbalanced or when there are too many nested calls and the RSB overflows or underflows. These kinds of attacks are not considered in this development as the Jasmin compiler inlines all code into a single function; the generated assembly consists of a single flat function with no call instructions. So, in the case of Jasmin and its compiler, no RSB attacks are possible. There also exist efficient hardware-based mitigations such as Intel's shadow stack [Shanbhogue et al., 2019] for protecting code that may be susceptible to Spectre-RSB.

5.8.4 Beyond high-assurance cryptography

Speculative constant-time is a necessary step to protect cryptographic keys and other sensitive material. However, it does not suffice because non-cryptographic (and unprotected) code living in the same memory space may leak. Carruth [Carruth, 2020] proposes to address this issue by putting high-value (long-term) cryptographic keys into a separate crypto-provider process and using inter-process communication to request cryptographic operations rather than just linking against cryptographic libraries. This modification should preserve functional correctness and speculative constant-time, assuming that inter-process communication can be implemented in a way that respects speculative constant-time. This integration into Jasmin is left for future work.

Chapter 6

Formally verified type checker for constant-time and speculative constant-time

6.1 Introduction

This chapter explores a different methodology to enforce mitigation against timing-based side-channel attacks using an information-flow-based type system. Information-flow-based type systems help us include rules for deriving facts about types of expressions and instructions in a program. The type system helps capture the security levels of any variable present in the program that is useful in reasoning about security properties like constant-time and speculative constant-time. As discussed in Section 1.1.3 of Chapter 1, there are various programming choices made by the programmer to provide mitigation against timing-based side-channel attacks. An information-flow-based type system can be used to track that these choices made by the programmer are correctly used. For example, in the case of providing mitigation like Speculative load hardening [Chandler Carruth, 2021], extra primitives are used by the programmer to mask the value obtained during a memory load in case of misspeculation. The type system approach can help in ensuring whether these primitives are used at the right place by the programmer or not.

Type system methodology differs from methodology discussed in Chapter 2 because it gives no guarantees about the compiler. It guarantees the program is secure against timing attacks instead of the compiler preserving the security properties. The low-level program also needs a type system to ensure the compiler does not break the constant-time property.

The work described in this chapter also differs from the work described in Chapter 5 in several ways. In Chapter 5, there is a declarative-style judgment of the form $\{I\}\ c\ \{O\}$ that uses dependency analysis to produce constraints based on the execution of commands in a speculative setting. This type system helps in ensuring that a typable program is speculative constant-time but requires the program to be *Speculative memory safe* (described in Section 5.3 of Chapter 5). *Speculative memory safety* is required due to the way operational semantics of load and store and their typing derivatives are designed. In case of unsafe load/store (where the access does not respect the bound), the whole memory is leaked instead of leaking a particular address as presented in Section 5.2.4. Hence, reasoning about the equality of the entire memory is not desirable. Also, the possible mitigation against Spectre attacks in Chapter 5 is inserting fence instruction at

the potential threat program points, which is not an efficient solution.

The methodology described in this chapter does not need memory safety even in the speculative setting because the information-flow-based type system is more powerful and expressive. It is explained in detail by referring to the two code snippets in Figure 6.1. Figure 6.1 presents two straightforward examples where the load and store operations are based on the guard (i <= |a|). In the code snippet present in Figure 6.1a, the types of x and a are public, and the program is assumed to be constant-time (i is also public). In case of misspeculation, the memory access will go out-of-bound and might load a secret data into the variable x. The type system present in this chapter ensures that the type of a[i] should be transient. The compiler ensures that the speculatively loaded value will be hardened before flowing into x. Hence, even if secret data is speculatively read from memory, it will never flow into the public variable x; instead, some default value will be assigned to x. The type derivative of load operation ensures no direct data flow under speculative execution by assigning type transient to the data loaded from memory. Similarly, the code snippet present in Figure 6.1b can bypass the guard (i <= |a|) and write in some other part of memory. The type system ensures that the type of all different arrays is at least the type of the loaded value (e′).

```
1 if (i <= |a|) {
2     x := a[i]; x and a are public
3 }
```

(a) Load operation

```
1 if (i <= |a|) {
2     a[i] = e; e is secret
3 }
```

(b) Store operation

Figure 6.1 – Illustrative example showcasing how to avoid the requirement of memory safety

The type system present in this chapter is more expressive than the one present in Chapter 5. It helps in avoiding the necessity of having memory safety. This chapter includes more efficient mitigation (as compared to inserting fence) against Spectre attacks, and together with the type system, their correct usage is established.

6.1.1 Necessity to prove the soundness of the type system

The type systems developed in this work help to ensure statically that the program written in the language L present in Figure 6.2 is secure against timing-based side-channel attacks. There are three sets of typing rules defined in the following sections. These typing rules help to provide the guarantee that if the program is typable, then the program is secure against timing-based side-channel attacks. The type soundness theorem for programming language L will prove that if a program P written in programming language L successfully type-checks, then P should be guaranteed to be secure against timing-based side-channel attacks. There is a need to prove the soundness of these type systems because a human can make mistakes while designing or implementing these type systems. The soundness proof in Coq will give formal guarantees that a typable program is indeed constant-time.

6.1.2 Contribution

- A formal language and core primitives to support protection against timing-based side-channel attacks.

$v \in Val := nat$	$e \in \mathsf{Expr} ::=$	x	variable
		$\mid op(e, e)$	operator
	$op \in Bop :=$	$+$	plus
		$\mid -$	minus
		$\mid <$	lessthan
		$\mid >$	greaterthan
		$\mid =$	equal
	$i \in \mathsf{Instr} ::=$	$x := e$	assignment
		$\mid \mathsf{if}\ be\ \mathsf{then}\ i\ \mathsf{else}\ i'$	conditional
		$\mid x := a[e]$	load
		$\mid a[e] := e'$	store
$c \in Cmd := \overline{i}$	$s \in State :=$	$\langle c, \rho, \mu \rangle$	state

Figure 6.2 – Syntax of simple language

- An extension to the formal language to support “declassify" that can be used to construct intensional leakages.
- A set of type systems to enforce constant-time and speculative constant-time policies.
- A mechanized proof in Coq showing the soundness of type systems.

6.2 Language

Figure 6.2 presents a language consisting of expressions and instructions. An expression can be a variable or a binary operator ($<$, $>$ or $=$ that operates on boolean values and $+$ and $-$ that operates on naturals). Instructions can be an assignment $x := e$, a conditional $\mathsf{if}\ be\ \mathsf{then}\ i\ \mathsf{else}\ i$ where be represents boolean operators applied to expressions, a load $x := a[e]$ or a store $a[e] := e'$ instruction. A program operates on a state made up of register map ρ (maps the registers to the values stored in it) and memory μ (maps the memory addresses to the values). The language featuring declassify is presented in Section 6.5. The language is extended with new primitives to support protection against speculative constant-time in Section 6.4.2.

6.2.1 Operational Semantics

This section presents the operational semantics of the language described in Figure 6.2. To reason about constant-time property, the semantics is instrumented to produce visible observations called leakages. Leakages can be empty $\bullet$, an index $\mathsf{index}\ i$ or a boolean $\mathsf{bool}\ b$.

$$o \in Obs \ ::= \ \bullet \mid \mathsf{index}\ i \mid \mathsf{bool}\ b$$

The operational semantics of the program are represented using the judgment: $\langle c, s \rangle \xrightarrow{\ell} \langle c', s' \rangle$ that represents the execution of c starting from state s producing the observation ℓ, command c' and updated state s'. Expressions do not produce any leakage. Evaluating a

variable x produces a value v, obtained from the register map ρ. Evaluation of an operator *op* involves evaluating its operands and applying the operator to obtain the result.

Expression semantics:

$$\frac{}{[\![x]\!]_\rho = v}\ \textsc{Variable} \qquad \frac{[\![e_1]\!]_\rho = v_1 \quad [\![e_2]\!]_\rho = v_2}{[\![op(e_1, e_2)]\!]_\rho = op(v_1, v_2)}\ \textsc{Operator}$$

Instruction semantics:

$$\frac{}{\langle \{\}, s\rangle \to^{\epsilon}_{\epsilon} s}\ \textsc{0-step} \qquad \frac{\langle i, s\rangle \xrightarrow{o} s_1 \quad \langle c, s_1\rangle \overset{O}{\underset{n}{\twoheadrightarrow}} s_2}{\langle \{i; c\}, s\rangle \overset{o::O}{\underset{n+1}{\twoheadrightarrow}} s_2}\ \textsc{s-step}$$

$$\frac{}{\langle x := e; c, \rho, \mu\rangle \xrightarrow{\bullet} \langle c, \rho\{x \leftarrow [\![e]\!]_\rho\}, \mu\rangle}\ \textsc{assgn}$$

$$\frac{[\![e_1]\!]_\rho = v_1 \quad [\![e_2]\!]_\rho = v_2 \quad [\![bop(v_1, v_2)]\!]_\rho = b'}{\langle \mathsf{if}\ bop(e_1, e_2)\ \mathsf{then}\ c_{tt}\ \mathsf{else}\ c_{ff}; c, \rho, \mu\rangle \xrightarrow{\mathsf{bool}\ b'} \langle \mathsf{if}\ b'\ \mathsf{then}\ c_{tt}\ \mathsf{else}\ c_{ff}; c, \rho, \mu\rangle}\ \textsc{cond}$$

$$\frac{[\![e]\!]_\rho = i \quad \mu(a, i) = v}{\langle x := a[e]; c, \rho, \mu\rangle \xrightarrow{\mathsf{index}\ i} \langle c, \rho\{x \leftarrow v\}, \mu\rangle}\ \textsc{load}$$

$$\frac{[\![e]\!]_\rho = i \quad [\![e']\!]_\rho = v}{\langle a[e] := e'; c, \rho, \mu\rangle \xrightarrow{\mathsf{index}\ i} \langle c, \rho, \mu(a, i) \leftarrow v\rangle}\ \textsc{store}$$

Figure 6.3 – Instrumented semantics.

The semantics for instructions are defined as small-step semantics and are present in Figure 6.3. The assignment instruction $x := e$ produces $\bullet$ leakage and assigns the evaluation of e to the variable x. Evaluation of a conditional instruction $\mathsf{if}\ bop(e_1, e_2)\ \mathsf{then}\ c_{tt}\ \mathsf{else}\ c_{ff}$ leaks the guard that is obtained by applying the operator *bop* on the evaluation of e_1 and e_2. Depending on the guard, it takes step to either the true or false branch. Loading a value from memory $x := a[e]$ (a is an array that represents the memory that is accessed at an index represented by the expression e) to a variable x leaks the $\mathsf{index}\ i$ that is obtained by evaluation of expression e and also assigns the value v (present at index i in the memory) to the variable x in the register map ρ. Store $a[e] := e'$ works similarly as load and leaks the index. More than one step is defined using the judgment of the form $c : s_1 \overset{o}{\underset{n}{\twoheadrightarrow}} s_2$ where n is the number of steps taken to reach state s_2 from state s_1.

6.3 Enforcing constant time policy

This section presents a type system that enforces constant-time. The security types in this work are inspired by the work Volpano and Smith [Volpano and Smith, 1997]. The motive behind the type system is to ensure that if the program written in the language described in Figure 6.2 type checks using the type system defined in Figure 6.4 then the program is constant-time.

Expressions:

$$\frac{\Gamma_v(x) = \mathit{Some}\ t}{\Gamma_v \vdash x : t}\ \textbf{VAR} \qquad \frac{\Gamma_v \vdash e_1 : t_1 \qquad \Gamma_v \vdash e_2 : t_2}{\Gamma_v \vdash \mathit{op}(e_1, e_2) : t_1\ U\ t_2}\ \textbf{OP}$$

Array:

$$\frac{\Gamma_v(a) = \mathit{Some}\ t \qquad \Gamma_v \vdash e : \mathit{Public}}{\Gamma_v \vdash a[e] : t}\ \textbf{ARRAY}$$

Instructions:

$$\frac{}{\Gamma_v, \Gamma_a \vdash \{\}}\ \textbf{0-STEP} \qquad \frac{\Gamma_v, \Gamma_a \vdash i \quad \Gamma_v, \Gamma_a \vdash c}{\Gamma_v, \Gamma_a \vdash \{i; c\}}\ \textbf{S-STEP}$$

$$\frac{\Gamma_v(x) = t \qquad \Gamma_v \vdash e : t_e \qquad t_e\ <=\ t}{\Gamma_v, \Gamma_a \vdash x := e}\ \textbf{ASSGN}$$

$$\frac{\Gamma_v \vdash b : \mathit{Public} \qquad \Gamma_v, \Gamma_a \vdash c_{tt} \qquad \Gamma_v, \Gamma_a \vdash c_{ff}}{\Gamma_v, \Gamma_a \vdash \mathsf{if}\ b\ \mathsf{then}\ c_{tt}\ \mathsf{else}\ c_{ff}}\ \textbf{COND}$$

$$\frac{\Gamma_v(x) = \mathit{Some}\ t_x \qquad \Gamma_v, \Gamma_a \vdash a[e] : t_a \qquad t_a\ <=\ t_x}{\Gamma_v, \Gamma_a \vdash x := a[e]}\ \textbf{LOAD}$$

$$\frac{\Gamma_v, \Gamma_a \vdash a[e] : t_a \qquad \Gamma_v \vdash e' : t_e \qquad t_e\ <=\ t_a}{\Gamma_v, \Gamma_a \vdash a[e] := e'}\ \textbf{STORE}$$

Figure 6.4 – Typing rules

6.3.1 Security types

There are two security types {Secret, Public} with order Public <= Secret. The following grammar defines the security types:

$$t\ ::=\ \mathit{Secret} \mid \mathit{Public}$$

The union of two types t_1 and t_2 is represented as $t_1\ U\ t_2$ is defined as follows:

$$t_1\ U\ t_2 = \begin{cases} \mathit{Secret} & t_1 = \mathit{Secret} \\ t_2 & t_1 = \mathit{Public} \end{cases}$$

The typing environment is a partial map between string (representing variables in a program) and option type.

6.3.2 Typing rules

The type system manipulates a judgment of the form:

$$\Gamma_v \vdash e : t \qquad \Gamma_v, \Gamma_a \vdash c$$

where Γ_v and Γ_a are security environments for variables and arrays.

Figure 6.4 provides the typing rules for the language introduced in Figure 6.2. The convention $\Gamma(x) = Some\ t$ is used to indicate that the type of variable x is obtained from the environment Γ.

The typing rule **VAR** and **OP** are used for typing expressions. They use the judgment of the form: $\Gamma_v \vdash e : t$. The rule **VAR** inspects the type from the context Γ_v and generates no constraint. The rule **OP** for binary operators collects type for both operands e_1 and e_2 and generates the new type that is the maximum of the security type of e_1 and e_2.

The rule **ASSIGN** requires that the security type of value assigned (e) is a sub-type of the type of variable x. The rule **COND** checks that the guard b has security type *Public* and also ensures that both the branches c_{tt} and c_{ff} are typable in environment Γ_v and Γ_a. The rule **LOAD** ensures that the array is accessed with index e of type *Public* and the type of x is greater than or equal to the type of array access. The rule **STORE** ensures that the accessed index e is of type *Public*. The relation $t_e <= t_a$ ensures that the array type t_a is at least the type of the stored expression. Rules **0-STEP** and **S-STEP** are pretty straightforward and ensure that a sequence of instructions is typable if all the instructions present in the sequence type-checks.

6.3.3 Soundness

The type system is sound, i.e., it only accepts constant-time programs. This section defines the statement of soundness and other definitions used in it. Informally, soundness states that if a program p type checks using the judgment $\Gamma_v, \Gamma_a \vdash p$, then p is constant-time. The soundness of the type system depends on the definition of state equivalence and constant time.

Definition 10 (State equivalence)**.** *State equivalence relation* $=_{(\Gamma_v,\Gamma_a)}$ *between two states* $\langle c_1, \rho_1, \mu_1\rangle$ *and* $\langle c_2, \rho_2, \mu_2\rangle$ *is defined as follows:*

$$\langle c_1, \rho_1, \mu_1\rangle =_{(\Gamma_v,\Gamma_a)} \langle c_2, \rho_2, \mu_2\rangle \equiv \begin{cases} \forall x, \Gamma_v(x) = Public \implies \rho_1(x) = \rho_2(x) \\ \forall a, \Gamma_a(a) = Public \implies \mu_1(a) = \mu_2(a) \\ c_1 = c_2 \end{cases}$$

Two states are said to be equivalent if they do not differ in their public data. Equivalence on public data is stated using the type system that ensures whether data is public or secret. In terms of reasoning about expression, for readability $=_{(\Gamma_v,\Gamma_a)}$ is written as $=_{\Gamma_v}$.

Definition 11 (Safe)**.** *A pair of state and command* s_{safe} *is called safe if it satisfies one of these properties:* $c = \phi \vee \exists\ l\ s', s \xrightarrow{l} s'$ *where* c *is the command in the state* s*.*

Definition 12 (Constant time)**.** *A program* c *is constant time with respect to state* s_1 *and* s_2

$$=^{CT}_{(\Gamma_v,\Gamma_a,s_1,s_2)} \equiv \left\{ \langle c, s_1\rangle =_{(\Gamma_v,\Gamma_a)} \langle c, s_2\rangle \wedge \langle c, s_1\rangle \overset{l_1}{\underset{n}{\twoheadrightarrow}} s_1' \wedge \langle c, s_2\rangle \overset{l_2}{\underset{n}{\twoheadrightarrow}} s_2' \implies l_1 = l_2 \wedge s_{1safe}' \leftrightarrow s_{2safe}' \right.$$

The constant time property is stated formally as above. It states that two executions of a program c starting from state s_1 and s_2 that are equivalent produce the same visible observations, and also, the execution leads to a safe state.

Soundness for expressions The soundness of the type system for expressions is necessary to prove soundness for instructions. It states that the evaluation of public expression e in two equivalent states is equal. This will play a major role in proving the equality of leakages, as leakages are mostly constructed using expressions.

Theorem 20 (Soundness for expressions)**.**

$$\forall \rho_1\ \rho_2\ e\ t, \rho_1 =_{\Gamma_v} \rho_2 \wedge \Gamma_v \vdash e : Public \implies [\![e]\!]_{\rho_1} = [\![e]\!]_{\rho_2}$$

Proof. The proof follows by doing induction on type derivation of e. There are two cases:

- Variable case: This case is trivial as the state equivalence justifies that the public variables are equal in the register map ρ_1 and ρ_2.
- Operator case: The induction hypothesis says that the type of expression $op(e_1, e_2)$ is the maximum of types of e_1 and e_2 represented as $t_1\ U\ t_2$. From the hypothesis we know that $t_1\ U\ t_2 = Public$, which means $t_1 = t_2 = Public$. Using the equivalence relation, we know that the evaluation of e_1 in ρ_1 will be equal to the evaluation of e_2 in ρ_2 as they are of type $Public$. Since the evaluation of operands is equal, applying operator op on them will result in the same value.

□

Preservation Preservation justifies that after any number of steps, a well-typed program will remain well-typed.

Theorem 21 (Preservation)**.**

$$\forall\ \Gamma_v\ \Gamma_a\ s_1\ s_1'\ l_1\ c\ c',$$
$$\Gamma_v, \Gamma_a \vdash c \wedge \langle c, s_1\rangle \xrightarrow{l_1} \langle c', s_1'\rangle \implies \Gamma_v, \Gamma_a \vdash c'$$

Proof. The proof follows by doing induction on the step-reduction for commands c, which produces different cases for the instruction at the head of c. In all the cases, the proof is pretty straightforward. The only interesting case is conditional instruction if $bop(e_1, e_2)$ then c_{tt} else $c_{ff}; c$ where we need to justify two cases: $\Gamma_v, \Gamma_a \vdash c_{tt}; c$ and $\Gamma_v, \Gamma_a \vdash c_{ff}; c$. The typing derivation for conditional instruction generates the hypothesis $\Gamma_v, \Gamma_a \vdash c_{tt}$ and $\Gamma_v, \Gamma_a \vdash c_{ff}$ and then using the typing rule **S-STEP** present in Figure 6.4 we prove $\Gamma_v, \Gamma_a \vdash c_{tt}; c$ and $\Gamma_v, \Gamma_a \vdash c_{ff}; c$. □

One-step soundness We first establish the soundness of the type system for one-step execution and, in the later section, use the following lemma to prove soundness for multi-step execution.

Theorem 22 (One-step soundness)**.**

$$\forall\ \Gamma_v\ \Gamma_a\ s_1\ s_2\ s_1'\ s_2'\ l_1\ l_2\ c_1\ c_2\ c_1'\ c_2',$$
$$\Gamma_v, \Gamma_a \vdash c_1 \wedge \langle c_1, s_1\rangle =_{(\Gamma_v,\Gamma_a)} \langle c_2, s_2\rangle\ \wedge \langle c_1, s_1\rangle \xrightarrow{l_1} \langle c_1', s_1'\rangle \wedge \langle c_2, s_2\rangle \xrightarrow{l_2} \langle c_2', s_2'\rangle \implies$$
$$l1 = l2\ \wedge\ \langle c_1', s_1'\rangle =_{(\Gamma_v,\Gamma_a)} \langle c_2', s_2'\rangle.$$

Proof. The proof follows by doing a case analysis on c. Also for readability, states are unfolded as: $s_1 = \langle \rho_1, \mu_1 \rangle$, $s_2 = \langle \rho_2, \mu_2 \rangle$, $s_1' = \langle \rho_1', \mu_1' \rangle$ and $s_2' = \langle \rho_2', \mu_2' \rangle$. The equality over the commands c_1' and c_2' is trivial as the language is deterministic in nature.

- $x := e$. The equality on leakages is trivial as the assignment produces $\bullet$ leakage. The semantics of update function $\{x \leftarrow [\![e]\!]_\rho\}$ helps in justifying the proof by reasoning about the type of x. The update is done based on the fact that the register x is present in the register map or not. If it is not present, then we return the same register map; else, we update it with the value $[\![e]\!]_\rho$.
  - $x' = x$. Since we know that the type of x' is *Public* and $x = x'$, this gives us the information $\Gamma_v(x) = \mathit{Public}$. The typing rule of assignment instruction also suffices that the type of expression e should be less than the type of x. Hence, the type of e is *Public*. According to the semantics of assignment instructions we also know that $s_1' = \langle \rho_1\{x \leftarrow [\![e]\!]_{\rho_1}\}, \mu_1 \rangle$ and $s_2' = \langle \rho_2\{x \leftarrow [\![e]\!]_{\rho_2}\}, \mu_2 \rangle$. Now using theorem 20, we know that $[\![e]\!]_{\rho_1} = [\![e]\!]_{\rho_2}$ that suffices $\rho_1'(x) = \rho_2'(x)$. The memory map equivalence is trivial as the assignment does not update the memory.
  - For $x' \neq x$. We know that the type of x' is *Public* and $\Gamma_v(x) = t$. Since x is not equal to x', we know from the semantic of assignment instruction that $\rho_1'(x') = \rho_1\{x \leftarrow [\![e]\!]_{\rho_1}\}(x')$ and $\rho_2'(x') = \rho_2\{x \leftarrow [\![e]\!]_{\rho_2}\}(x')$. And from the assumption $\langle c_1, s_1 \rangle =_{(\Gamma_v, \Gamma_a)} \langle c_2, s_2 \rangle$, we know that $\rho_1(x') = \rho_2(x')$. Hence from transitivity we know that $\rho_1'(x') = \rho_2'(x')$ which suffices to prove that $\langle c_1', s_1' \rangle =_{(\Gamma_v, \Gamma_a)} \langle c_2', s_2' \rangle$.
- $\mathsf{if}\ \mathit{bop}(e_1, e_2)\ \mathsf{then}\ c_{\mathit{tt}}\ \mathsf{else}\ c_{\mathit{ff}}$: Here *bop* is a boolean operator that operates on two operands. From the typing rule of conditional, we know that the type of $\mathit{bop}(e_1, e_2)$ is *Public*, which shows that the type of e_1 and e_2 are also *Public*. The two executions leaks the guard which is $[\![\mathit{bop}(e_1, e_2)]\!]_{\rho_1}$ and $[\![\mathit{bop}(e_1, e_2)]\!]_{\rho_2}$. Since theorem 20 justifies $[\![e_1]\!]_{\rho_1} = [\![e_1]\!]_{\rho_2}$ and $[\![e_2]\!]_{\rho_1} = [\![e_2]\!]_{\rho_2}$, we can say that $[\![\mathit{bop}(e_1, e_2)]\!]_{\rho_1} = [\![\mathit{bop}(e_1, e_2)]\!]_{\rho_2}$. Hence, the leakages are equal, and as the semantics of conditional does not update the register and memory maps, the new state s_1' and s_2' are trivially equivalent.
- x := a[e]: The typing rule of load instruction gives the information that the type of index e is *Public*. The two executions leak the index $[\![e]\!]_{\rho_1}$ and $[\![e]\!]_{\rho_2}$. Using theorem 20, we know that $[\![e]\!]_{\rho_1} = [\![e]\!]_{\rho_2}$ that suffices to prove that leakages are equal. The semantics of load instruction updates the register with the value present at index $[\![e]\!]_{\rho_i}$ in the memory map μ at cell a. Precisely, $s_1' = \langle \rho_1\{x \leftarrow \mu_1(a, [\![e]\!]_{\rho_1})\}, \mu_1 \rangle$ and $s_2' = \langle \rho_2\{x \leftarrow \mu_2(a, [\![e]\!]_{\rho_2})\}, \mu_2 \rangle$. The update function, as discussed above, gives us two cases:
  - $x' = x$ and $[\![e]\!]_{\rho_1} = [\![e]\!]_{\rho_2}$. Since we know that the type of x' is *Public* and $x = x'$, this gives us the information $\Gamma_v(x) = \mathit{Public}$. Also, the sub-type relation ensures $\Gamma_a(a) = \mathit{Public}$. From the assumption $\langle c_1, s_1 \rangle =_{(\Gamma_v, \Gamma_a)} \langle c_2, s_2 \rangle$, we know that $\mu_1(a) = \mu_2(a)$. Hence $\langle c_1', s_1' \rangle =_{(\Gamma_v, \Gamma_a)} \langle c_2', s_2' \rangle$ as $\mu_1(a) = \mu_2(a)$ and $[\![e]\!]_{\rho_1} = [\![e]\!]_{\rho_2}$.
  - $x' \neq x$ and $[\![e]\!]_{\rho_1} = [\![e]\!]_{\rho_2}$. We know that the type of x' is *Public* and $\Gamma_v(x) = t$. Since x is not equal to x', we know from the semantic of load instruction that $\rho_1'(x') = \rho_1\{x \leftarrow \mu_1(a, [\![e]\!]_{\rho_1})\}(x')$ and $\rho_2'(x') = \rho_2\{x \leftarrow \mu_2(a, [\![e]\!]_{\rho_2})\}(x')$.

And from the assumption $\langle c_1, s_1\rangle =_{(\Gamma_v,\Gamma_a)} \langle c_2, s_2\rangle$, we know that $\rho_1(x') = \rho_2(x')$. Hence from transitivity we know that $\rho_1'(x') = \rho_2'(x')$ which suffices to prove that $\langle c_1', s_1'\rangle =_{(\Gamma_v,\Gamma_a)} \langle c_2', s_2'\rangle$.

- $a[e] = e'$: The typing rule of store instruction gives the information that the type of index e is *Public*. The two executions leak the index $[\![e]\!]_{\rho_1}$ and $[\![e]\!]_{\rho_2}$. Using theorem 20, we know that $[\![e]\!]_{\rho_1} = [\![e]\!]_{\rho_2}$ that suffices to prove that leakages are equal. The semantics of store instruction updates the memory μ (cell a) at index $[\![e]\!]_{\rho_i}$ with the value $[\![e']\!]_{\rho_i}$. Precisely, $s_1' = \langle \rho_1, \mu(a, [\![e]\!]_{\rho_1}) \leftarrow [\![e']\!]_{\rho_1}\rangle$ and $s_2' = \langle \rho_2, \mu(a, [\![e]\!]_{\rho_2}) \leftarrow [\![e']\!]_{\rho_2}\rangle$. The update function of the memory map works in a similar manner. It checks for the string equality to update the array cell.

  - $a' = a$ and $[\![e]\!]_{\rho_1} = [\![e]\!]_{\rho_2}$. Since we know that the type of a' is *Public* and $a = a'$, this gives us the information $\Gamma_v(a) = \mathit{Public}$. Also, the sub-type relation ensures $\Gamma_v(e') = \mathit{Public}$. From the assumption $\langle c_1, s_1\rangle =_{(\Gamma_v,\Gamma_a)} \langle c_2, s_2\rangle$, we know that $\mu_1(a) = \mu_2(a)$. Theorem 20 ensures that $[\![e']\!]_{\rho_1} = [\![e']\!]_{\rho_2}$. Hence $\langle c_1', s_1'\rangle =_{(\Gamma_v,\Gamma_a)} \langle c_2', s_2'\rangle$ because $\mu_1(a) = \mu_2(a)$, $[\![e]\!]_{\rho_1} = [\![e]\!]_{\rho_2}$ and $[\![e']\!]_{\rho_1} = [\![e']\!]_{\rho_2}$ and register map is not updated.
  - $a' \neq a$ and $[\![e]\!]_{\rho_1} = [\![e]\!]_{\rho_2}$. We know that the type of a' is *Public* and $\Gamma_a(a) = t$. Since a is not equal to a', we know from the semantic of store instruction that $\mu_1'(a') = \mu_1((a, [\![e]\!]_{\rho_1}) \leftarrow [\![e']\!]_{\rho_1})(a')$ and $\mu_2'(a') = \mu_2((a, [\![e]\!]_{\rho_2}) \leftarrow [\![e']\!]_{\rho_2})(a')$ And from the assumption $\langle c_1, s_1\rangle =_{(\Gamma_v,\Gamma_a)} \langle c_2, s_2\rangle$, we know that $\mu_1(a') = \mu_2(a')$. Hence, from transitivity, we know that $\mu_1'(a') = \mu_2'(a')$ and it does not update the register map that suffices to prove that $\langle c_1', s_1'\rangle =_{(\Gamma_v,\Gamma_a)} \langle c_2', s_2'\rangle$.

□

Multi-step soundness The final soundness theorem is stated as follows:

Theorem 23 (Multi-step soundness)**.**

$$\forall\ \Gamma_v\ \Gamma_a\ s_1\ s_2,\ c$$
$$\Gamma_v, \Gamma_a \vdash c \implies c =^{CT}_{(\Gamma_v,\Gamma_a,s_1,s_2)}$$

where c start executing from equivalence state s_1 and s_2

Proof. The proof follows by unfolding the definition of $=^{CT}_{(\Gamma_v,\Gamma_a,s_1,s_2)}$ that generates two goals and set of hypothesis $\langle c, s_1\rangle =_{(\Gamma_v,\Gamma_a)} \langle c, s_2\rangle$, $\langle c, s_1\rangle \overset{l}{\underset{n}{\twoheadrightarrow}} s_1'$ and $\langle c, s_2\rangle \overset{l'}{\underset{n}{\twoheadrightarrow}} s_2'$. The first goal concerns the equivalence of leakages, and the second is proving that states s_1' and s_2' are safe. The second goal is trivial, as the language never gets stuck. The proof for proving the equivalence on leakages follows by doing induction on n, where n is the number of steps taken during the multi-step semantics.

- $n = 0$: This case is trivial as there will be no leakage as the number of steps taken by the command c from s_1 and s_2 are 0.
- $n = n' + 1$: This case produces the goal $l = l'$ where $l = l_1 :: l_2$ and $l' = l_1' :: l_2'$. More precisely, l_1 and l_2 are leakages produced during one-step execution, and l_2 and l_2' are the leakages produced during the rest of the steps. Using the theorem 22,

we conclude $l_1 = l_1'$. Using theorem 21, we know that the command obtained after one-step execution is also well-typed. Generalizing the induction hypothesis on the result of 21, we conclude $l_2 = l_2'$.

□

Hence, soundness helped establish that the type system for checking constant time property presented in Figure 6.4 is correct.

6.4 Enforcing speculative constant time policy

To enforce speculative constant time, we need to consider the processor's speculative behavior.

6.4.1 State extension

The state $\langle c, \rho, \mu \rangle$ is extended to include an extra variable called ms variable. The extended state is of the form $\langle c, \rho, \mu, ms \rangle$. The ms variable keeps track of whether execution is misspeculating or not. The ms variable is of type boolean and represents misspeculation when set to true.

6.4.2 Language extension

The language present in Figure 6.2 is extended to include an extra primitive called protect(x, ms). For illustration, this notation of protect is used. But in the Coq development, the ms value is included in the state, and protect instruction has notation as protect x y where y represents the correct value which is assigned to x in case of no speculation. The use of protect is illustrated using following example:

```
1 foo (int length) {
2 if (length < bound) {
3     x := a[length]; a is public, length is public
4     y := b[x]; leaks secret index
5 }
```

```
1 fooProtect (int length) {
2 if (length < bound) {
3     x := a[length];
4     x := protect(x, ms);
5     y := b[x];
6 }
```

Figure 6.5 – Example program: Illustration of protect primitive

In the program on the left side of Figure 6.5, the attacker can pass any value of length and speculatively reach the secret part of memory by bypassing the guard. Program foo is constant time as there is no secret dependent branching or memory access, but speculatively, it can still leak the secret data through public loads. The attacker can load an arbitrary value from memory into x, which is later leaked via the memory access $b[x]$. Inserting fence instruction after line 2 is one of the solutions to avoid leaking secrets due to speculation. Fence instruction is inefficient as it stops speculations of all instructions, not only for the instructions that leak. The program fooProtect adds protect at line 4. protect(x, ms) protects the public load. In case of misspeculation (when ms is true), x gets a dummy value instead of loading the secret data. Hence, in line 5, no secret data is loaded in y, and the leaked index x does not depend on the secret.

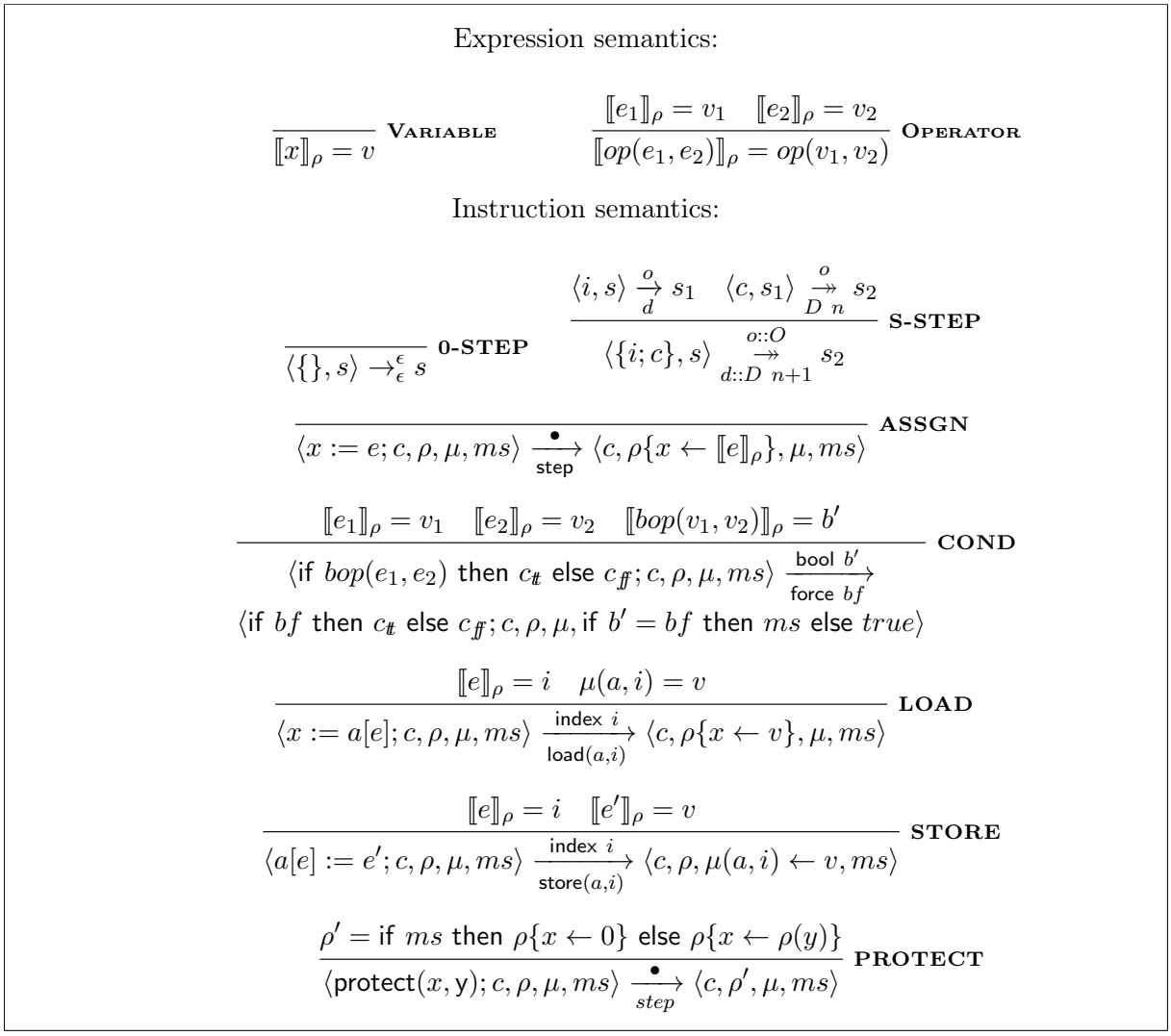


Figure 6.6 – Instrumented semantics with speculation

6.4.3 Operational semantics

To reason about speculations, the operational semantics presented in Figure 6.3 is enriched with directives to model the steps taken by the attackers. The directive d represents the attacker's decision before the instruction is executed.

$$d \in Dir \ ::= \ \mathsf{step} \mid \mathsf{force}\ b \mid \mathsf{load}(a,i) \mid \mathsf{store}(a,i)$$

The attacker uses the directive step when the execution proceeds normally. During a step directive, a conditional branch does not speculate, and the program enters the correct branch. The attacker issues the directive force to force execution to enter into a misspeculated branch. The force directive is the only directive that updates the misspeculation flag. The directives $\mathsf{load}(a,i)$ and $\mathsf{store}(a,i)$ allows the attacker to read or write to addresses of its choice speculatively. The leakages are similar to the one defined in Section 6.2.

The judgment defines the operational semantics:

$$\langle c, s\rangle \xrightarrow[d]{\ell} \langle c', s'\rangle$$

where ℓ is the observation visible to the outside world obtained while executing the command c, d is the directive, and s and s' are the initial and final states.

The operational semantics is present in Figure 6.6. The semantics is very similar to the one presented in Figure 6.3, except it is instrumented with directives. The instructions like assignment, load, and store do not update the misspeculation flag; hence, their semantics are similar to as described in Section 6.2.1. In conditional instruction, the ms flag is updated based on the directive $\mathsf{force}\ bf$ where bf indicates whether the attacker speculatively takes the true or false branch. In the case of $b' \neq bf$ (b' shows the correct guard value), the ms flag is set to $true$, indicating the program is misspeculating. The semantics of $\mathsf{protect}\ x\ y$ ensures that x gets the dummy value 0 in case of misspeculation and gets the value y where the ms flag is false.

Expressions:

$$\frac{\Gamma_v(x) = Some\ t}{\Gamma_v \vdash x : t}\ \textbf{VAR} \qquad \frac{\Gamma_v \vdash e_1 : t_1 \qquad \Gamma_v \vdash e_2 : t_2}{\Gamma_v \vdash op(e_1, e_2) : t_1\ U\ t_2}\ \textbf{OP}$$

Array:

$$\frac{\Gamma_v(a) = Some\ t \qquad \Gamma_v \vdash e : Public}{\Gamma_v \vdash a[e] : t}\ \textbf{ARRAY}$$

Instructions:

$$\frac{}{\Gamma_v, \Gamma_a \vdash \{\}}\ \textbf{0-STEP} \qquad \frac{\Gamma_v, \Gamma_a \vdash i \quad \Gamma_v, \Gamma_a \vdash c}{\Gamma_v, \Gamma_a \vdash \{i; c\}}\ \textbf{S-STEP}$$

$$\frac{\Gamma_v(x) = t \qquad \Gamma_v \vdash e : t_e \qquad t_e\ <=\ t}{\Gamma_v, \Gamma_a \vdash x := e}\ \textbf{ASSGN}$$

$$\frac{\Gamma_v \vdash b : Public \qquad \Gamma_v, \Gamma_a \vdash c_{tt} \qquad \Gamma_v, \Gamma_a \vdash c_{ff}}{\Gamma_v, \Gamma_a \vdash \mathsf{if}\ b\ \mathsf{then}\ c_{tt}\ \mathsf{else}\ c_{ff}}\ \textbf{COND}$$

$$\frac{\Gamma_v(x) = Some\ t_x \qquad \Gamma_v, \Gamma_a \vdash a[e] : t_a \qquad to_{t_v}(t_a)\ <=\ t_x}{\Gamma_v, \Gamma_a \vdash x := a[e]}\ \textbf{LOAD}$$

$$\frac{\Gamma_v, \Gamma_a \vdash a[e] : t_a \qquad \Gamma_v \vdash e' : t_e \qquad t_e\ <=\ to_{t_v}(t_a)}{\Gamma_v, \Gamma_a \vdash a[e] := e'}\ \textbf{STORE}$$

$$\frac{\Gamma_v(x) = Some\ Public \qquad \Gamma_v(y) = Some\ t \qquad t\ <=\ Transient}{\Gamma_v, \Gamma_a \vdash \mathsf{protect}\ x\ y}\ \textsc{Protect}$$

Figure 6.7 – Typing rules for speculative instructions

6.4.4 Security types

The type system is extended to enforce speculative constant time with a new type called $Transient$. Data with $Transient$ type indicate that it is public under sequential semantics but may depend on secrets under the speculative semantics. There are two types of arrays:

Public and *Secret* (arrays are used to represent memory).

$$\begin{aligned} t_v &::= Secret \mid Public \mid Transient \\ t_a &::= Secret \mid Public \end{aligned}$$

The U relation is extended to add the fact that $Transient\ U\ Public = Transient$, and similarly, the <= is extended to add the fact that $Public <= Transient$ and leads to the order $Public <= Transient <= Secret$. A function $to_{t_v}(t_a)$ is defined to transform the array type to the value type. It returns the *Transient* type when the array type is *Public* and the *Secret* type when the array type is *secret*. This function helps in reasoning about the instructions like load and store, where secret data can be leaked speculatively through public loads.

6.4.5 Typing rules

The typing rules for speculative language are present in Figure 6.7. It is defined using the same typing judgment described in Section 6.2. It is similar to the typing rules explained in Figure 6.4 and adds a new typing rule for protect instruction. The typing rule **LOAD** and **STORE** uses the function $to_{t_v}(t_a)$ that ensures that in terms of public-loads or public-stores, the type assigned to the value obtained by public memory access is *Transient*. A protect instruction is used to avoid leakages through public loads and stores. At the semantic level of protect instruction, all the variables with *Transient* types are assigned a default value in case of misspeculation and the actual value through the load in case of no misspeculation.

The typing rule **PROTECT** ensures that the type of y is either *Transient* or *Public*. We are only interested in protecting public loads and stores because the secret loads are already covered using the constant-time property.

6.4.6 Soundness

The type system is sound, i.e., it only accepts speculative constant-time programs. Informally, soundness states that if a program p type checks using the judgment $\Gamma_v, \Gamma_a \vdash p$, then p is speculative constant-time. The soundness of the type system depends on the definition of state equivalence and speculative constant-time.

Definition 13 (State equivalence)**.** *State equivalence relation* $=^s_{(\Gamma_v,\Gamma_a)}$ *between two states* $\langle c_1, \rho_1, \mu_1, ms_1\rangle$ *and* $\langle c_2, \rho_2, \mu_2, ms_2\rangle$ *is defined as follows:*

$$\langle c_1, \rho_1, \mu_1, ms_1\rangle =^s_{(\Gamma_v,\Gamma_a)} \langle c_2, \rho_2, \mu_2, ms_2\rangle \equiv \begin{cases} ms_1 = ms_2 \\ c_1 = c_2 \\ \forall x, \Gamma_v(x) = Public \implies \rho_1(x) = \rho_2(x) \\ (!ms_1 \implies \\ \quad \forall x, \Gamma_v(x) = Transient \implies \rho_1(x) = \rho_2(x) \\ \quad \forall a, \Gamma_a(a) = Public \implies \mu_1(a) = \mu_2(a)) \end{cases}$$

State equivalence for speculative programs differs slightly from sequential programs as we must consider speculative semantics. Two states are equivalent if they do not differ in their public data stored in registers in the case of speculation. They do not differ on transient and public data in case of no speculation. State equivalence for speculative

semantics also adds an extra condition for misspeculation flags ms_1 and ms_2 that should be equal. In terms of reasoning about expression, for readability $=^{s}_{(\Gamma_v,\Gamma_a)}$ is written as $=^{s}_{\Gamma_v}$.

Definition 14 (Speculative constant time)**.** *A program c is speculative constant time with respect to state s_1 and s_2*

$$=^{SCT}_{(\Gamma_v,\Gamma_a)} \equiv \left\{ \langle c_1, s_1\rangle =_{(\Gamma_v,\Gamma_a)} \langle c_2, s_2\rangle \wedge \langle c_1, s_1\rangle \overset{l_1}{\underset{d\ n}{\twoheadrightarrow}} s_1' \wedge \langle c_2, s_2\rangle \overset{l_2}{\underset{d\ n}{\twoheadrightarrow}} s_2' \implies l_1 = l_2 \wedge s_{1safe}' \leftrightarrow s_{2safe}' \right.$$

Speculative constant-time ensures that in the presence of the same set of directives, the leakages should be equal if the executions start from equivalent states.

Soundness for expressions The soundness of the speculative type system for expressions is necessary to prove the soundness of the type system for instructions. It states that the evaluation of expression e in two equivalent states is equal if the type of expression is $Public$ in case of misspeculation and $Transient$ in case of no speculation. This will play a significant role in proving the equality of leakages, as leakages are mostly constructed using expressions.

Theorem 24 (Soundness of expressions)**.**

$$\forall\ \rho_1\ \rho_2\ \mu_1\ \mu_2\ ms_1\ ms_2\ e\ t, \langle \rho_1, \mu_1, ms_1\rangle =^{s}_{\Gamma_v} \langle \rho_2, \mu_2, ms_2\rangle \wedge$$
$$\Gamma_v \vdash e : \mathsf{if}\ ms_1\ \mathsf{then}\ Public\ \mathsf{else}\ Transient \implies [\![e]\!]_{\rho_1} = [\![e]\!]_{\rho_2}$$

Proof. The proof follows by doing induction on type derivation of e. There are two cases:

- Variable case: There are two cases:
  - $ms_1 = ms_2 =$ true: This is the case where the semantics is speculative. We know that $\Gamma_v(x) = Some\ Public$. $\rho_1(x) = \rho_2(x)$ is direct consequence of state equivalence.
  - $ms_1 = ms_2 =$ false: This is where the semantics is not speculative. We know that $\Gamma_v(x) = Some\ Transient$. This is also a direct consequence of state equivalence as it ensures that transient data are equal in case of no speculation.
- Operator case: The induction hypothesis states that the type of expression $op(e_1, e_2)$ is the maximum of types of e_1 and e_2, and it also says that the resultant type is $Public$ or $Transient$ depending on the value of ms_1. Let t_1 and t_2 be the type of e_1 and e_2.
  - $ms_1 = ms_2 =$ true: According to the union relation, $t_1\ U\ t_2 = Public$ means $t_1 = Public$ and $t_2 = Public$. Using the equivalence relation, we know that the evaluation of e_1 will be equal to the evaluation of e_2 as they are of type $Public$ and semantics is speculative. Since the evaluation of operands is equal, the evaluation of operator op on them will also be equal in ρ_1 and ρ_2.

– $ms_1 = ms_2 =$ false: According to the union relation, $t_1\ U\ t_2 = Transient$ means $t_1 <= Transient$ and $t_2 <= Transient$ (the subtyping relation ensures that t_1 and t_2 can be either $Public$ or $Transient$). Using the equivalence relation, we know that the evaluation of e_1 will be equal to the evaluation of e_2 as they are of type $Public$ or $Transient$. Since the evaluation of operands is equal, the evaluation of operator op on them will also be equal in ρ_1 and ρ_2.

□

Preservation Preservation justifies that after any number of steps, a well-typed program will remain well-typed. This property is used to prove soundness for multi-step.

Theorem 25 (Preservation)**.**

$$\forall\ \Gamma_v\ \Gamma_a\ s_1\ s_1'\ l_1\ c\ c',$$
$$\Gamma_v, \Gamma_a \vdash c \wedge c : s_1 \xrightarrow[d_1]{l_1} c' : s_1' \implies \Gamma_v, \Gamma_a \vdash c'$$

Proof. The proof is similar to the proof of preservation in Section 6.2. It only adds the preservation proof for protect instruction, which is very trivial. □

One-step soundness We first establish the soundness of the type system for one-step execution and, in the later section, use the following lemma to prove soundness for multi-step execution.

Theorem 26 (One-step soundness)**.**

$$\forall\ \Gamma_v\ \Gamma_a\ s_1\ s_2\ s_1'\ s_2'\ l_1\ l_2\ c_1\ c_2\ c_1'\ c_2',$$
$$\Gamma_v, \Gamma_a \vdash c_1 \wedge \langle c_1, s_1\rangle =^s_{(\Gamma_v,\Gamma_a)} \langle c_2, s_2\rangle\ \wedge \langle c_1, s_1\rangle \xrightarrow[d]{l_1} \langle c_1', s_1'\rangle \wedge \langle c_2, s_2\rangle \xrightarrow[d]{l_2} \langle c_2', s_2'\rangle \implies$$
$$l1 = l2\ \wedge\ \langle c_1', s_1'\rangle =^s_{(\Gamma_v,\Gamma_a)} \langle c_2', s_2'\rangle.$$

Proof. The proof follows by doing a case analysis on c_1 and unfolding the definition of $=^s_{(\Gamma_v,\Gamma_a)}$. The $c_1' = c_2'$ is trivial to prove as the language is deterministic, and the set of directives in both traces are equal. Also for readability, states are unfolded as: $s_1 = \langle \rho_1, \mu_1, ms_1\rangle$, $s_2 = \langle \rho_2, \mu_2, ms_2\rangle$, $s_1' = \langle \rho_1', \mu_1', ms_1'\rangle$ and $s_2' = \langle \rho_2', \mu_2', ms_2'\rangle$. State equivalence proof for all the instructions is similar to the proof explained in 6.3.3. The proof is tweaked to consider both cases: speculation and sequential. The proof for equality on the ms value is added for each case. For all the instructions except the conditional instruction, ms variable is never updated, so the proof is trivial. In the case of conditional, the ms variable is updated similarly in both the semantics because directives are the same; hence, they are equal.

Proof for protect x y: The equality on leakages is trivial as protect produces $\bullet$ leakage. The semantics of protect depends on the ms flag to decide whether x gets value 0 or y. The semantics used the update function to update the value of x ($\{x \leftarrow 0\}$ *or* $\{x \leftarrow \rho(y)\}$). There are two cases to consider:

- $ms_1 = true$: This ensures that the semantic is speculative and x gets the value 0. There are two cases to consider based on the semantics of update:

  - $x = x'$: This case is trivial as $0 = 0$.
  - $x \neq x'$: The type of x' is $Public$ and x is t. From the semantics of protect, we know that $\rho_1'(x') = \rho_1\{x \leftarrow 0\}(x')$ and $\rho_2'(x') = \rho_2\{x \leftarrow 0\}(x')$. From hypothesis we know that $\langle c_1, s_1\rangle =^{s}_{(\Gamma_v,\Gamma_a)} \langle c_2, s_2\rangle$ which ensures $\rho_1(x') = \rho_2(x')$. Hence from transitivity it follows that $\rho_1'(x') = \rho_2'(x')$. Also, memory map equivalence is trivial as it is not updated during protect. Hence $\langle c_1', s_1'\rangle =^{s}_{(\Gamma_v,\Gamma_a)} \langle c_2', s_2'\rangle$.

- $ms_1 = false$: This ensures that the semantics is not speculative and x gets the value of y. There are two cases to consider based on the semantics of update:
  - $x = x'$: We know that type of x' is $Transient$. Also, from the typing rule of protect, we know that the type of y is less than the $Transient$ type. This ensures that the type of y can be $Public$ or $Transient$. From the hypothesis $\langle c_1, s_1\rangle =^{s}_{(\Gamma_v,\Gamma_a)} \langle c_2, s_2\rangle$, we know that in case of no speculation variables of type $Transient$ and $Public$ are equal in register maps ρ_1 and ρ_2. This ensures that $\rho_1(y) = \rho_2(y)$. The semantics of protect updates the register maps as $\rho_1' = \rho_1\{x \leftarrow \rho_1(y)\}$ and $\rho_2' = \rho_2\{x \leftarrow \rho_2(y)\}$. As $\rho_1(y) = \rho_2(y)$, we conclude $\rho_1'(x)$ will be equal to $\rho_2'(x)$. The memory map equivalence is trivial as it is not updated during protect
  - $x \neq x'$: The type of x' is $Transient$ and x is t. From the semantics of protect, we know that $\rho_1'(x') = \rho_1\{x \leftarrow \rho_1(y)\}(x')$ and $\rho_2'(x') = \rho_2\{x \leftarrow \rho_2(y)\}(x')$. From hypothesis we know that $\langle c_1, s_1\rangle =^{s}_{(\Gamma_v,\Gamma_a)} \langle c_2, s_2\rangle$ which ensures $\rho_1(x') = \rho_2(x')$. Hence from transitivity it follows that $\rho_1'(x') = \rho_2'(x')$. Also, memory map equivalence is trivial as it is not updated during protect. Hence $\langle c_1', s_1'\rangle =^{s}_{(\Gamma_v,\Gamma_a)} \langle c_2', s_2'\rangle$.

□

Multi-step soundness The final soundness theorem is stated as follows:

Theorem 27 (Multi-step soundness)**.**

$$\forall\ \Gamma_v\ \Gamma_a\ s_1\ s_2\ c,$$
$$\Gamma_v, \Gamma_a \vdash c \implies c =^{SCT}_{(\Gamma_v,\Gamma_a,s_1,s_2)}$$

Proof. The proof is similar to the proof explained in 6.3.3. □

6.5 Language with Declassify

This section adds a declassify feature to the language. The cryptographic ciphertexts are computed from secret data and are viewed as secret by information flow analysis like the type system discussed above. The declassify construct can make these kinds of data public so that it is no longer sensitive.

$$
\begin{array}{rlll}
i \in \mathsf{Instr} ::= & x_d := e & & \text{assignment} \\
| & \mathsf{if}\ be\ \mathsf{then}\ i\ \mathsf{else}\ i' & & \text{conditional} \\
| & x_d := a[e] & & \text{load} \\
| & a[e]_d := e' & & \text{store} \\
| & \mathsf{protect}\ x\ y & & \text{protect}
\end{array}
$$

Figure 6.8 – Syntax of language with declassify

The language with declassification is present in the Figure 6.8. The set of instructions like assignment, load, and store is updated with a boolean called d that represents the declassify feature. For example, in case of load operation $x_d := a[e]$ if the boolean d is set to true, then the value $a[e]$ is declassified (after declassification, it is treated as public value).

6.5.1 Illustrative example showing potential leakage due to declassification

This section presents an example of vulnerability caused by declassification in the case where speculations are allowed. In cryptography, data is encrypted into ciphertext and sent over the insecure network. The type system classifies data like ciphertext as secret because they are computed from the secret data. The "declassify" primitive can be used to make these kinds of data public as it is no longer secret in the theoretical concept of encryption and decryption. The program present in Figure 6.9 illustrates how declassification in programs that can speculate could leak to potential leakages. The program encryptFun encrypts a secret message m to produce a cipher-text called cipher.

```
1 encryptFun (secret int m, secret int key) ⟶int {
2 secret int cipher;
3 public int i;
4 public int d;
5 cipher = m;
6 for (i = 0; i <= 8; i++) {
7     cipher[i] = m[i] ⊕ key[i] ;
8 }
9 d = declassify(cipher);
10 return d;
11 }
```

Figure 6.9 – Example program

The secret message m is encrypted by performing a bit-wise operation on secret message m and key in line 7. Since cipher is computed using the secrets m and key, the type of cipher will also be secret. Using the declassify mechanism, the cipher is converted to type public and returned as a result. In the sequential execution, there is no chance of leaking the information related to the secret message m as there is no secret-dependent branching or memory access. But consider the case where speculation is allowed. In case of speculation, the condition $i <= 8$ in for-loop is ignored, and declassify will leak the cipher, which is set to the secret message m in line 5. We have seen in Section 1.1.3 of Chapter 1 that there are various mitigation mechanisms possible against Spectre attacks, like Speculative load

hardening, that are more efficient than the insertion of fence. But unfortunately, applying Speculative load hardening to the program in Figure 6.9 does not offer any protection, as the attack does not happen due to speculative load operation.

This work analyzes these undesirable leakages caused due to declassification. It presents a type system to enforce mitigation against timing attacks for the program written in a language that features a "declassify" operation.

6.5.2 Operational Semantics

This section presents the operational semantics of the language with declassification. The semantics are presented in the Figure 6.10. The leakage language present in Section 6.2.1 is extended to include a new kind of leakage called $\mathsf{db}(\mathsf{option}\ v)$. The leakage $\mathsf{db}(\mathsf{option}\ v)$ represents the declassified value that will be visible to the outside world as it is no longer secret data after declassification.

Instruction semantics:

$$\frac{}{\langle \{\}, s\rangle \to^{\epsilon}_{\epsilon} s}\ \textbf{0-STEP} \qquad \frac{\langle i, s\rangle \xrightarrow[d]{o} s_1 \quad \langle c, s_1\rangle \overset{o}{\underset{D\ n}{\twoheadrightarrow}} s_2}{\langle \{i; c\}, s\rangle \overset{o::O}{\underset{d::D\ n+1}{\twoheadrightarrow}} s_2}\ \textbf{S-STEP}$$

$$\frac{}{\langle x_d := e; c, \rho, \mu, ms\rangle \xrightarrow[\text{step}]{\text{if } d \text{ then db}(some\ [\![e]\!]_\rho) \text{ else } \bullet} \langle c, \rho\{x \leftarrow [\![e]\!]_\rho\}, \mu, ms\rangle}\ \textbf{ASSGN}$$

$$\frac{[\![e_1]\!]_\rho = v_1 \quad [\![e_2]\!]_\rho = v_2 \quad [\![bop(v_1, v_2)]\!]_\rho = b'}{\langle \text{if } bop(e_1, e_2) \text{ then } c_{tt} \text{ else } c_{ff}; c, \rho, \mu, ms\rangle \xrightarrow[\text{force } bf]{\text{bool } b'} \langle \text{if } bf \text{ then } c_{tt} \text{ else } c_{ff}; c, \rho, \mu, \text{if } b' = bf \text{ then } ms \text{ else } true\rangle}\ \textbf{COND}$$

$$\frac{[\![e]\!]_\rho = i \quad \mu(a, i) = v}{\langle x_d := a[e]; c, \rho, \mu, ms\rangle \xrightarrow[\text{load}(a,i)]{\text{if } d \text{ then index } i\text{;db}(some\ v) \text{ else index } i} \langle c, \rho\{x \leftarrow v\}, \mu, ms\rangle}\ \textbf{LOAD}$$

$$\frac{[\![e]\!]_\rho = i \quad [\![e']\!]_\rho = v}{\langle a[e]_d := e'; c, \rho, \mu, ms\rangle \xrightarrow[\text{store}(a,i)]{\text{if } d \text{ then index } i\text{;db}(some\ v) \text{ else index } i} \langle c, \rho, \mu(a, i) \leftarrow v, ms\rangle}\ \textbf{STORE}$$

$$\frac{\rho' = \text{if } ms \text{ then } \rho\{x \leftarrow 0\} \text{ else } \rho\{x \leftarrow \rho(y)\}}{\langle \mathsf{protect}(x, \mathsf{y}); c, \rho, \mu, ms\rangle \xrightarrow[step]{\bullet} \langle c, \rho', \mu, ms\rangle}\ \textbf{PROTECT}$$

Figure 6.10 – Instrumented semantics of language with declassification and speculation

The operational semantics only presents the semantics of instructions. The declassification feature is only added to the instructions. There are different ways to add a declassification feature. For example, it can be added as a separate instruction: $x := \mathsf{declassify}(e)$. But all different designs resemble the same semantics. The semantics is very similar to the original semantics presented in Section 6.4.3 except the execution of **ASSGN**, **LOAD** and **STORE** also leaks the declassified value along with the original leakage in case of declassification (if declassify d boolean is set to true). For example, execution of $a[e]_d := e'$

leaks both the index ($[\![e]\!]_\rho$) and also the declassified value that is $[\![e']\!]_\rho$ when d is set to true. In case of no declassification, the execution of $a[e]_d := e'$ only leaks the index ($[\![e]\!]_\rho$).

6.5.3 Typing rules

This section presents the typing rules for the instructions with declassification. The typing rules are similar to those presented in Section 6.4.5 except that in the case of declassification, the declassified value gets the *Transient* type. This helps in making a decision to protect them so that there are no transient attacks due to them. For example, in the example present in Figure 6.9, the value declassify(cipher) gets the type *Transient* that helps in determining the compiler-level mitigation like putting a protect instruction after line 9; hence d will be protected and will not leak the secret value m.

Instructions:

$$\frac{}{\Gamma_v, \Gamma_a \vdash \{\}}\ \textbf{0-STEP} \qquad \frac{\Gamma_v, \Gamma_a \vdash i \quad \Gamma_v, \Gamma_a \vdash c}{\Gamma_v, \Gamma_a \vdash \{i; c\}}\ \textbf{S-STEP}$$

$$\frac{\Gamma_v(x) = t \qquad \Gamma_v \vdash e : t_e \qquad \text{if } d \text{ then } \mathit{Transient} \text{ else } t_e \ <= \ t}{\Gamma_v, \Gamma_a \vdash x_d := e}\ \textbf{ASSGN}$$

$$\frac{\Gamma_v \vdash b : \mathit{Public} \qquad \Gamma_v, \Gamma_a \vdash c_{tt} \qquad \Gamma_v, \Gamma_a \vdash c_{ff}}{\Gamma_v, \Gamma_a \vdash \text{if } b \text{ then } c_{tt} \text{ else } c_{ff}}\ \textbf{COND}$$

$$\frac{\Gamma_v(x) = \mathit{Some}\ t_x \qquad \Gamma_v, \Gamma_a \vdash a[e] : t_a \qquad \text{if } d \text{ then } \mathit{Transient} \text{ else } to_{t_v}(t_a) \ <= \ t_x}{\Gamma_v, \Gamma_a \vdash x_d := a[e]}\ \textbf{LOAD}$$

$$\frac{\Gamma_v, \Gamma_a \vdash a[e] : t_a \qquad \Gamma_v \vdash e' : t_e \qquad \text{if } d \text{ then } \mathit{Transient} \text{ else } t_e \ <= \ to_{t_v}(t_a)}{\Gamma_v, \Gamma_a \vdash a[e]_d := e'}\ \textbf{STORE}$$

$$\frac{\Gamma_v(x) = \mathit{Some}\ \mathit{Public} \qquad \Gamma_v(y) = \mathit{Some}\ t \qquad t \ <= \ \mathit{Transient}}{\Gamma_v, \Gamma_a \vdash \text{protect } x\ y}\ \textbf{PROTECT}$$

Figure 6.11 – Typing rules for instructions with declassify and speculation

The rule **ASSGN** ensures that in the case where d is true in $x_d := e$, the declassified value e gets the type *Transient* instead of type t_e and also its type should be less than the type of x. The rule **LOAD** ensures that in the case where d is true in $x_d := a[e]$, the type of $a[e]$ should be *Transient* instead of $to_{t_v}(t_a)$ and also its type should be less than the type of x. The typing rule for store instruction is similar to the load instruction. There is no declassification involved in conditional instruction; hence, the typing rule remains the same as described in Section 6.4.5.

6.5.4 Soundness

The type system is sound, i.e., it only accepts speculative constant-time programs (with declassification). Informally, soundness states that if a program p with declassification type checks using the judgment $\Gamma_v, \Gamma_a \vdash p$, then p is speculative constant-time. The

soundness of the type system depends on the definition of state equivalence and speculative constant-time. State equivalence for the speculative programs is the same as described in the definition 13. Speculative constant-time definition is also the same as described in the definition 14. The declassification process makes the data public that is also leaked during the execution of instructions like assignment, load, or store. Hence, to prove that leakages are equal, we need the assumption that the declassified values are the same in both executions.

Declassify specification $\langle c_1, s_1\rangle =_{DSPEC} \langle c_2, s_2\rangle$ is defined as follows:

Definition 15 (Declassify Specification)**.**

$$\langle c_1, s_1\rangle =_{DSPEC} \langle c_2, s_2\rangle \equiv \begin{cases} \langle c_1, s_1\rangle =_{(\Gamma_v,\Gamma_a)} \langle c_2, s_2\rangle \wedge \langle c_1, s_1\rangle \overset{l}{\underset{d\ n}{\twoheadrightarrow}} s_1' \wedge \langle c_2, s_2\rangle \overset{l'}{\underset{d\ n}{\twoheadrightarrow}} s_2' \implies \\ l_{dec} = {l'}_{dec} \wedge {s'}_{1safe} \leftrightarrow {s'}_{2safe} \end{cases}$$

Declassification specification states that if the program c starts its execution from two equivalent states s_1 and s_2 produces leakage l and l' then the declassified value present in l and l' will be equal. The declassified value is extracted from the leakage l and l' using the function l_{dec}.

$$l_{dec} = \begin{cases} v & l = \mathsf{db}(v) \\ None & l \neq \mathsf{db}(v) \end{cases}$$

The definition ls_{dec} is recursively defined using l_{dec} to extract the declassified value from the sequence of leakages ls.

Soundness for expressions The Soundness of the type system of expression is required to prove the soundness of the type system for instructions. It states that the evaluation of expression e in two equivalent states is equal if the type of expression is $Public$ in case of misspeculation and $Transient$ in case of no speculation.

Theorem 28 (Soundness of expressions)**.**

$$\forall\ s_1\ s_2\ e\ t, s_1 =^{s}_{\Gamma_v} s_2 \wedge \Gamma_v \vdash e : \mathsf{if}\ ms_1\ \mathsf{then}\ Public\ \mathsf{else}\ Transient \implies [\![e]\!]_{s_1} = [\![e]\!]_{s_2}$$

Proof. The proof of soundness for expressions does not change, and it is the same as described in 24 because the declassification feature is only added to the instructions. □

Auxiliary functions and lemmas To prove the soundness of the type system, this section defines some auxiliary functions and lemmas that are used in the final proof.

Definition 16 (l_{build})**.** *The function l_{build} builds the sequence of leakages for a given instruction i, starting state s ($s = \langle \rho, \mu, ms\rangle$) and a set of declassified values vs:*

$$\mathsf{l_{build}}(\langle i, s\rangle, vs) = \begin{cases} \mathsf{if}\ d\ \mathsf{then}\ \{\mathsf{db}(vs_0)\}\ \mathsf{else}\ \{\bullet\} & i = x_d := e \\ \mathsf{if}\ d\ \mathsf{then}\ \{\mathsf{index}\ [\![e]\!]_\rho; \mathsf{db}(vs_1)\}\ \mathsf{else}\ \{\mathsf{index}\ [\![e]\!]_\rho\} & i = x_d := a[e] \\ \mathsf{if}\ d\ \mathsf{then}\ \{\mathsf{index}\ [\![e]\!]_\rho; \mathsf{db}(vs_1)\}\ \mathsf{else}\ \{\mathsf{index}\ [\![e]\!]_\rho\} & i = a[e]_d := e' \\ \{\mathsf{bool}\ (bop([\![e_1]\!]_\rho, [\![e_2]\!]_\rho)\} & i = \mathsf{if}\ bop(e_1, e_2)\ \mathsf{then}\ i_1\ \mathsf{else}\ i_2 \\ \{\bullet\} & i = \mathsf{protect}\ x\ y \end{cases}$$

The definition l_{build} helps in constructing the leakage corresponding to an instruction for a given set of declassified values.

Lemma 1 (Leakages build from the same set of declassified values are equal)**.**

$$\forall\ \Gamma_v\ \Gamma_a\ c_1\ c_2, \Gamma_v, \Gamma_a \vdash c \implies (\forall\ s_1\ s_2\ vs, \langle c_1, s_1\rangle =^{s}_{\Gamma_v,\Gamma_a} \langle c_2, s_2\rangle \implies \mathsf{l_{build}}(\langle c_1, s_1\rangle, vs) = \mathsf{l_{build}}(\langle c_2, s_2\rangle, vs))$$

Proof. The proof follows doing induction on the typing derivation and unfolding the definition of l_{build}. State s_1 and s_2 are unfolded as $\langle \rho_1, \mu_1, ms_1\rangle$ and $\langle \rho_2, \mu_2, ms_2\rangle$.

- $x_d := e$: In the case of assignment, the leakage constructed using the l_{build} function depends on the declassified value vs or is $\bullet$ depending on the value of d. Since the same declassified value vs is provided to l_{build} function on both sides, the proof is trivial.

- $x_d := a[e]$: In the case of load, the leakage constructed by the function l_{build} on the left-hand side is $\mathsf{if}\ d\ \mathsf{then}\ \{\mathsf{index}\ [\![e]\!]_{\rho_1}; \mathsf{db}(vs_1)\}\ \mathsf{else}\ \{\mathsf{index}\ [\![e]\!]_{\rho_1}\}$ and on the right-hand side is $\mathsf{if}\ d\ \mathsf{then}\ \{\mathsf{index}\ [\![e]\!]_{\rho_2}; \mathsf{db}(vs_1)\}\ \mathsf{else}\ \{\mathsf{index}\ [\![e]\!]_{\rho_2}\}$. The second element $\mathsf{db}(vs_1)$ of the sequence of leakages is trivially equal. The typing derivation of load instruction ensures that the type of e is $Public$. Using the theorem 28 on the hypothesis $\Gamma_v \vdash e : Public$ and $\langle c_1, s_1\rangle =^{s}_{\Gamma_v,\Gamma_a} \langle c_2, s_2\rangle$, we know that $[\![e]\!]_{\rho_1} = [\![e]\!]_{\rho_2}$ which suffices to prove that $\mathsf{index}\ [\![e]\!]_{\rho_1} = \mathsf{index}\ [\![e]\!]_{\rho_2}$.

- $a[e]_d := e'$: The proof for store is similar to the above case.

- $\mathsf{if}\ bop(e_1, e_2)\ \mathsf{then}\ i_1\ \mathsf{else}\ i_2$: In the case of conditional, the leakage constructed using the l_{build} on the left-hand side is $\{\mathsf{bool}\ (bop([\![e_1]\!]_{\rho_1}, [\![e_2]\!]_{\rho_1})\}$ and on the right-hand side is $\{\mathsf{bool}\ (bop([\![e_1]\!]_{\rho_2}, [\![e_2]\!]_{\rho_2})\}$. From the typing derivation, we know that the type of e_1 and e_2 is $Public$. Using the theorem 28 on the hypothesis $\Gamma_v \vdash e_1 : Public$ and $\langle c_1, s_1\rangle =^{s}_{\Gamma_v,\Gamma_a} \langle c_2, s_2\rangle$, we know that $[\![e_1]\!]_{\rho_1} = [\![e_1]\!]_{\rho_2}$. Similarly, using the theorem 28 on the hypothesis $\Gamma_v \vdash e_2 : Public$ and $\langle c_1, s_1\rangle =^{s}_{\Gamma_v,\Gamma_a} \langle c_2, s_2\rangle$, we know that $[\![e_2]\!]_{\rho_1} = [\![e_2]\!]_{\rho_2}$. This suffices to prove $\{\mathsf{bool}\ (bop([\![e_1]\!]_{\rho_1}, [\![e_2]\!]_{\rho_1})\} = \{\mathsf{bool}\ (bop([\![e_1]\!]_{\rho_2}, [\![e_2]\!]_{\rho_2})\}$.

- $\mathsf{protect}\ x\ y$: This case is trivial because $\mathsf{protect}$ produces $\bullet$ leakage.

□

Lemma 2 (Leakages build from the extracted declassified value from a leakage produce the same leakage)**.**

$$\forall\ s\ s'\ c\ c'\ l\ d, \langle c, s\rangle \xrightarrow[d]{l} \langle c', s'\rangle \implies \mathsf{l_{build}}(\langle c, s\rangle, l_{dec}) = l$$

Proof. The proof follows by doing induction on the typing derivation and unfolding the definition of l_{build}. The proof is trivial and boils down to proving the correctness of the function l_{dec} (that it correctly extracts the declassified value from the leakage l) and l_{build}. □

To prove the above lemmas 1 and 2 for multi-step executions, we need to establish the definition l_{build} for constructing the leakages for a set of instructions from a set of set of declassified values. The function to construct leakages corresponding to a set of

instructions $i; c$ for a given set of declassified value $v :: vs$ is called l_{buildc}. The semantic of the function l_{buildc} is present in Figure 6.12.

$$\frac{}{\mathsf{l_{buildc}}(\langle\{\}, s\rangle, \{\}) = \{\}}\ \textbf{B-EMPTY}$$

$$\frac{\mathsf{l_{build}}(\langle i, s\rangle, v) = l \qquad \mathsf{s_{build}}(\langle i, s\rangle, v, d) = s' \qquad \mathsf{l_{buildc}}(\langle c, s'\rangle, vs) = ls}{\mathsf{l_{buildc}}(\langle\{i; c\}, s\rangle, \{v; vs\}) = \{l; ls\}}\ \textbf{B-S-STEP}$$

$$\frac{}{\mathsf{l_{buildc}}(\langle\{\}, s\rangle, \{v; vs\}) = \{\}}\ \textbf{B-EC-STEP}$$

$$\frac{}{\mathsf{l_{buildc}}(\langle\{i; c\}, \{\}\rangle, s) = \{\}}\ \textbf{B-ED-STEP}$$

Figure 6.12 – Function to construct leakages corresponding to a set of instructions for a given set of declassified values

The rule **B-S-STEP** represents the semantics of l_{buildc} to build the leakage for the set of instruction $\{i; c\}$ from a set of set of declassified values $\{v; vs\}$. It uses l_{build} function defined in 16 to calculate the leakage l for instruction i for a state s and declassified value v. To calculate the leakages for the rest of the instructions in the sequence i.e. c, we need to compute the updated state. It uses s_{build} function to compute the next state. With the updated state s', l_{buildc} recursively computes the leakage for c using the declassified values vs. The semantics of s_{build} is pretty straightforward and based on the instructions' operational semantics. The rules of s_{build} is present in Figure 6.13.

$$\frac{\rho' = \rho\{x \leftarrow \mathsf{if}\ d\ \mathsf{then}\ vs\ \mathsf{else}\ [\![e]\!]_\rho\}}{\mathsf{s_{build}}(\langle x_d := e; c, \rho, \mu, ms\rangle, vs, \mathsf{step}) = \langle c, \rho', \mu, ms\rangle}\ \textbf{S-ASSGN}$$

$$\frac{\rho' = \rho\{x \leftarrow \mathsf{if}\ d\ \mathsf{then}\ vs\ \mathsf{else}\ \mu(a, [\![e]\!]_\rho)\}}{\mathsf{s_{build}}(\langle x_d := a[e]; c, \rho, \mu, ms\rangle, vs, \mathsf{load}(a, [\![e]\!]_\rho)) = \langle c, \rho', \mu, ms\rangle}\ \textbf{S-LOAD}$$

$$\frac{\mu' = \mu(a, [\![e]\!]_\rho) \leftarrow \mathsf{if}\ d\ \mathsf{then}\ vs\ \mathsf{else}\ [\![e']\!]_\rho)}{\mathsf{s_{build}}(\langle x_d := a[e]; c, \rho, \mu, ms\rangle, vs, \mathsf{store}(a, [\![e]\!]_\rho)) = \langle c, \rho', \mu, ms\rangle}\ \textbf{S-STORE}$$

$$\frac{ms' = \mathsf{if}\ bop([\![e_1]\!]_\rho, [\![e_2]\!]_\rho) = bf\ \mathsf{then}\ ms\ \mathsf{else}\ true}{\mathsf{s_{build}}(\langle \mathsf{if}\ bop(e_1, e_2)\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2; c, \rho, \mu, ms\rangle, vs, \mathsf{force}\ bf) = \langle \mathsf{if}\ bf\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \rho, \mu, ms'\rangle}\ \textbf{S-IF}$$

$$\frac{\rho' = \mathsf{if}\ ms\ \mathsf{then}\ \rho\{x \leftarrow 0\}\ \mathsf{else}\ \rho\{x \leftarrow \rho(y)\}}{\mathsf{s_{build}}(\langle protect\ x\ y; c, \rho, \mu, ms\rangle, vs, \mathsf{step}) = \langle c, \rho', \mu, ms\rangle}\ \textbf{S-PROTECT}$$

Figure 6.13 – Function to construct updated state from a given set of declassified values

Lemma 3 (Build next state from declassified values)**.**

$$\forall\ s\ s'\ c'\ l\ d, \langle c, s\rangle \xrightarrow[d]{l} \langle c', s'\rangle \implies \mathsf{s_{build}}(\langle c, s\rangle, l_{dec}, d) = \langle c', s'\rangle$$

Proof. The proof follows by doing a case analysis on c and unfolding the definition of s_{build}. The proof is trivial and boils down to proving the correctness of the function s_{build} (i.e., it correctly computes the next state). □

Lemma 4 (After a single step execution, the resultant states are also in equivalence)**.**

$$\begin{aligned}\forall\ s_1\ s_2\ c_1\ c_2\ s_1'\ s_2'\ c_1'\ c_2'\ l\ d,\\ \Gamma_v, \Gamma_a \vdash c_1 \wedge \langle c_1, s_1\rangle =^s_{(\Gamma_v,\Gamma_a)} \langle c_2, s_2\rangle \wedge\\ \mathsf{s_{build}}(\langle c_1, s_1\rangle, vs, d) = \langle c_1', s_1'\rangle \ \wedge\ \mathsf{s_{build}}(\langle c_2, s_2\rangle, vs, d) = \langle c_2', s_2'\rangle \implies\\ \langle c_1', s_1'\rangle =^s_{(\Gamma_v,\Gamma_a)} \langle c_2', s_2'\rangle.\end{aligned}$$

Proof. The proof follows by doing induction on the type derivation. The states are unfolded as $s_1 = \langle \rho_1, \mu_1, ms_1\rangle$, $s_2 = \langle \rho_2, \mu_2, ms_2\rangle$, $s_1' = \langle \rho_1', \mu_1', ms_1'\rangle$ and $s_2' = \langle \rho_2', \mu_2', ms_2'\rangle$. The two resultant commands c_1' and c_2' are equal because s_{build} function builds the next state using the same set of directive d.

- $x_d := e$: The state equivalence definition generates two cases:
  - With speculation: In this case, $ms_1 = ms_2$ is trivially true as the assignment does not update the misspeculation flag. According to the semantics of s_{build} function represented by rule **S-ASSGN** in Figure 6.13, ρ_1' and ρ_2' are calculated as, $\rho_1' = \rho_1\{x \leftarrow \mathsf{if}\ d\ \mathsf{then}\ vs\ \mathsf{else}\ [\![e]\!]_{\rho_1}\}$ and $\rho_2' = \rho_2\{x \leftarrow \mathsf{if}\ d\ \mathsf{then}\ vs\ \mathsf{else}\ [\![e]\!]_{\rho_2}\}$. The rest of the proof follows by doing a case analysis on d.
    - d = true: For the case where declassification is enabled: $\rho_1' = \rho_1\{x \leftarrow vs\}$ and $\rho_2' = \rho_2\{x \leftarrow vs\}$. The typing derivation of the assignment gives the relation $Transient <= t$, where t is the type of x. The type t can be $Secret$ or $Transient$.
      - $t = Secret$. There are two cases according to the update function:
        1. $x = x'$ and $\Gamma_v(x') = Public$. This is trivially proved by contradiction.
        2. $x \neq x'$ and $\Gamma_v(x') = Public$. The goal in this case is $\rho_1'(x') = \rho_2'(x')$. The proof follows by using the induction hypothesis $s_1 =^s_{(\Gamma_v,\Gamma_a)} s_2$.
      - $t = Transient$. The proof is similar to the above case.
      - $t = Public$. This case is trivial as we get $Transient <= Public$ in the hypothesis, which is a false assumption.
    - d = false: For the case where declassification is disabled: $\rho_1' = \rho_1\{x \leftarrow [\![e]\!]_{\rho_1}\}$ and $\rho_2' = \rho_2\{x \leftarrow [\![e]\!]_{\rho_2}\}$. The type derivation gives the relation $t' <= t$, where t' is the type of e and t is the type of x. Based on the semantics of the update, there are two cases:
      1. $x = x'$ and $\Gamma_v(x') = Public$. Using these two hypothesis, we know $\Gamma_v(x) = Public$ and $t' <= Public$ (this also ensures that $t' = Public$ and $\Gamma_v \vdash e : Public$). By applying the theorem 28 on the hypothesis $\Gamma_v \vdash e : Public$, we get $[\![e]\!]_{\rho_1} = [\![e]\!]_{\rho_2}$ that suffices to prove $\rho_1' = \rho_2'$. Since there is no change in the memory map, we conclude $s_1' =^s_{(\Gamma_v,\Gamma_a)} s_2'$
      2. $x \neq x'$ and $\Gamma_v(x') = Public$. The goal in this case is $\rho_1'(x') = \rho_2'(x')$. The proof follows by using the induction hypothesis $\langle c_1, s_1\rangle =^s_{(\Gamma_v,\Gamma_a)} \langle c_2, s_2\rangle$ and transitivity.

  - With no speculation: The proof is similar to the above case except that the type of x' will be $Transient$. Based on the type of x', the type of x will be $Trasient$ in the case of $x = x'$. This will generate an extra case for the type of e as it should be less than equal to $Transient$ (the type of e can be $Public$ or $Transient$). Overall, the proof structure is very similar to the above case.

- $x_d := a[e]$: According to the semantics of s_{build} function represented by rule **S-LOAD** in Figure 6.13, $\rho_1' = \rho_1\{x \leftarrow \mathsf{if}\ d\ \mathsf{then}\ vs\ \mathsf{else}\ \mu(a, [\![e]\!]_{\rho_1})\}$ and $\rho_2' = \rho_2\{x \leftarrow \mathsf{if}\ d\ \mathsf{then}\ vs\ \mathsf{else}\ \mu(a, [\![e]\!]_{\rho_2})\}$. The proof structure is similar to the above case.

- $a[e]_d := e'$: According to the semantics of s_{build} function represented by rule **S-STORE** in Figure 6.13, $\mu_1' = \mu_1(a, [\![e]\!]_{\rho_1}) \leftarrow (\mathsf{if}\ d\ \mathsf{then}\ vs\ \mathsf{else}\ [\![e']\!]_{\rho_1})$ and $\mu_2' = \mu_2(a, [\![e]\!]_{\rho_2}) \leftarrow (\mathsf{if}\ d\ \mathsf{then}\ vs\ \mathsf{else}\ [\![e']\!]_{\rho_2})$. The proof follows a similar style as described above.

- $\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2$ and $\mathsf{protect}\ x\ y$: The proof is similar to as explained in theorem 6.4.6 as there is no declassification in these two instructions.

□

Preservation Preservation justifies that after any number of steps, a well-typed program will remain well-typed. This property is used to prove soundness for multi-step.

Theorem 29 (Preservation)**.**

$$\forall\ \Gamma_v\ \Gamma_a\ s_1\ s_1'\ l_1\ d_1\ c\ c',$$
$$\Gamma_v, \Gamma_a \vdash c \wedge \langle c, s_1\rangle \xrightarrow[d_1]{l_1} \langle c', s_1'\rangle \implies \Gamma_v, \Gamma_a \vdash c'$$

Proof. The proof is similar to the proof of preservation is Section 2.5. □

Multi step soundness The final soundness theorem is stated as follows:

Theorem 30 (Multi-step soundness)**.**

$$\forall\ \Gamma_v\ \Gamma_a\ s_1\ s_2\ c_1\ c_2, \Gamma_v, \Gamma_a \vdash c_1 \wedge \langle c_1, s_1\rangle =_{DSPEC} \langle c_2, s_2\rangle \implies c_1 =^{SCT}_{(\Gamma_v, \Gamma_a, s_1, s_2)} .$$

Proof. The proof follows by unfolding the definition of $=_{DSPEC}$ and $=_{SCT}$ and doing induction on n. It generates two goals and a set of hypothesis $\langle c_1, s_1\rangle =_{(\Gamma_v, \Gamma_a)} \langle c_2, s_2\rangle$, $\langle c_1, s_1\rangle \overset{l}{\underset{n}{\twoheadrightarrow}} s_1'$ and $\langle c_2, s_2\rangle \overset{l'}{\underset{n}{\twoheadrightarrow}} s_2'$. The first goal concerns the equivalence of leakages, and the second is proving that states s_1' and s_2' are safe. The second goal is trivial, as the language never gets stuck. The proof for proving the equivalence on leakages follows by doing induction on n, where n is the number of steps taken during the multi-step semantics.

- n = 0: This case is trivial.

- n ≠ 0: The goal is to prove $l_1; l_2 = l_1'; l_2'$ where l_1 and l_2 are leakages from single-step execution corresponding to the head of sequence c_1 and l_2 and l_2' are the rest of the leakages obtained during multi-step execution. From lemma 2, we know that $\mathsf{l_{build}}(\langle c_1, s_1\rangle, l_{1_{dec}}) = l_1$ and $\mathsf{l_{build}}(\langle c_2, s_2\rangle, l_{2_{dec}}) = l_1'$. Also, from the hypothesis $\langle c_1, s_1\rangle =_{DSPEC} \langle c_2, s_2\rangle$, we know that the declassified values $l_{1_{dec}}$ and $l_{2_{dec}}$ are

equal. Substituting $l_{1_{dec}}$ with $l_{2_{dec}}$, the final goal transforms to $\mathsf{l_{build}}(\langle c_1, s_1\rangle, l_{1_{dec}}); l_2 = \mathsf{l_{build}}(\langle c_2, s_2\rangle, l_{1_{dec}}); l_2'$. Also, from the lemma 1, we know that if the two starting states for c_1 are in equivalence, then the leakages build from same set of declassified values are equal i.e. $\mathsf{l_{build}}(\langle c_1, s_1\rangle, l_{1_{dec}}) = \mathsf{l_{build}}(\langle c_2, s_2\rangle, l_{1_{dec}})$. Hence substituting $\mathsf{l_{build}}(\langle c_1, s_1\rangle, l_{1_{dec}})$ with $\mathsf{l_{build}}(\langle c_2, s_2\rangle, l_{1_{dec}})$ generates equality on the head of the sequence of leakages.

Now rest of the proof focuses on proving $l_2 = l_2'$. The lemma 3 applied to the single-step execution (the execution of the head of c_1 starting from s_1 and s_2) gives the relation (using the s_{build} function) to build the next initial state for the execution of the rest of the commands in c_1. Applying lemma 4 on typing derivation, state equivalence relation (corresponding to s_1 and s_2), and results of lemma 3 (i,e. $\mathsf{s_{build}}(\langle c_1, s_1\rangle, l_{1_{dec}}, d) = s_1''$ and $\mathsf{s_{build}}(\langle c_2, s_2\rangle, l_{2_{dec}}, d) = s_2''$) gives the result $s_1'' =^s_{(\Gamma_v,\Gamma_a)} s_2''$. From the preservation theorem 29, we know that commands present in state s_1'' and s_2'' are also well-typed. Generalizing the induction hypothesis on the above results concludes $l_2 = l_2'$.

□

Chapter 7

Typing High-Speed Cryptography against Spectre v1

7.1 Introduction

This chapter extends the work presented in Chapter 6 to provide mitigation against Spectre v1 attacks for the Jasmin language. There are other existing solutions to protect cryptographic implementations from Spectre v1 attacks, like inserting fences after the guard expression, but it incurs a significant performance overhead. An efficient method of protection is based on (value or address) hardening, including all forms of Speculative Load Hardening [Chandler Carruth, 2021] discussed in Section 1.1.3 of Chapter 1.

Chapter 6 explains the approach of using a new primitive called protect to harden the values in case of speculative loads in Section 6.4.2. The approach in Chapter 6 is applied for a simple language and has only one level of transformation compared to a more realistic language where we need to take care of many transformations and other programming details related to registers and stacks. Hence, in reality, applying these methods takes work. To write highly efficient cryptography algorithms, we intend to use as many available resources as possible to increase the overall performance. For example, we are aware of the fact that accessing data stored in a register is faster as compared to accessing from main memory. Hence, cryptographers tend to use registers as much as possible to avoid loads and storage from memory. However, the number of registers provided in a particular architecture is limited; hence, applying efficient mitigation based on hardening is difficult. The implementation needs to be rewritten so that one register is always reserved for a misspeculation flag (not necessarily one fixed register, as a misspeculation flag can be moved back and forth between the available registers). This register will help to decide whether a program is misspeculating; based on it, masking is performed to avoid speculative read/write.

This chapter presents a realistic and extended version of the approach presented in Chapter 6. The approach presented in this manuscript protects the cryptographic implementation written in Jasmin against Spectre v1 attacks with minimal performance overhead. For example, the cost of protecting a highly-optimized AVX2 implementation of Kyber768 against Spectre v1 is less than 1%.

7.1.1 Contributions

- A set of primitives that support fine-grained protection against Spectre v1 attacks by encoding both compiler-level mitigation, including Speculative Load Hardening [Chandler Carruth, 2021] and [Shivakumar et al., pear] Selective Speculative Load Hardening , and algorithm-specific mitigation. An interface to update and use misspeculation flags. Programmers can use these flags to track whether program execution is normal or misspeculating.

- A type system to track that primitives are correctly used. Type-system uses a value-dependent information flow analysis, where security levels depend on the misspeculation flag.

- An idea about how to prove the soundness of the type system: typable programs are speculative constant-time.

- An implementation of the approach in the Jasmin framework and modified existing implementations of several cryptographic algorithms so that they can be type-checked using the type system (protected against Spectre v1).

7.1.2 Discussion

Section 1.1.2 of Chapter 1 presents an introduction and various examples showing the possibilities of Spectre v1 attacks. There are many ways to protect constant-time programs against Spectre v1 attacks. As seen in Section 5.1.1 of Chapter 5, one simple countermeasure is to insert fence instruction after each conditional guard. As discussed in Section 6.1 of Chapter 1, this approach suffers from some limitations like the requirement of *speculative memory safety* and performance overhead.

Another more efficient alternative is to insert enough fence instructions so that there is always a speculative barrier between when a register becomes transient and when it leaks via control-flow or memory accesses [Vassena et al., 2021a]. This ensures that a program is speculative constant-time and performs better, but it is still not a great solution due to its noticeable performance overhead. It is better to minimize the use of fences.

The LLVM compiler implements an alternative approach called Speculative Load Hardening [Chandler Carruth, 2021] as discussed in Section 1.1.3 of Chapter 1. It is more efficient than the methods discussed in prior paragraphs, but it focuses on hardening all loads.

Another more efficient approach is to perform selective speculative load hardening by protecting only transient values assigned to public registers. Selective speculative load hardening minimizes the performance overhead by only masking transient values. This work implements selective speculative load hardening.

7.1.3 Illustrative Example

This section presents examples that show how new primitives can be used to implement countermeasures and algorithm-specific protections which guarantee speculative constant-time for the programs present in Figure 1.4.

p[10]	s[5]

```
1 ms = init_msf();
2 b = i < 10;
3 if b {
4     ms = set_msf(b, ms);
5     x = p[i];
6     x = protect(x, ms);
7     y = p[x];
8 } else {
9     ms = set_msf(! b, ms);
10 }
```

(a) Protected v1-read

s[5]	p[10]

```
1 ms = init_msf();
2 b = i < 5;
3 if b {
4     ms = set_msf(b, ms);
5     s[i] = sec;
6     x = p[0];
7     x = protect(x, ms);
8     p[x] = 0;
9 } else {
10    ms = set_msf(! b, ms);
11 }
12
```

(b) Protected v1-write

Figure 7.1 – Protected v1

Speculative Load Hardening Figure 7.1a shows a speculative load hardening process that protects the program present in Figure 1.4a against Spectre V1 attacks. Line 1 uses an instruction called init_msf() that ensures that the program's execution starts with a non-speculating state (sequential execution and misspeculation flag is set to $\perp$). Line 4 and Line 9 set the misspeculation flag based on the condition $i < 10$ using the primitive $\mathsf{set\_msf}(b, ms)$ and $\mathsf{set\_msf}(!b, ms)$. This ensures that the ms register is updated correctly after entering the branch. Line 6 uses primitive $\mathsf{protect}(x, ms)$ to mask the speculatively loaded value (p[i]) in the variable x. This ensures that in case of misspeculation, the value from the secret part (s) of the array does not reach the attacker (instead, the attacker gets the default value when $i = 13$). Hence, the program is protected against Spectre v1. A similar procedure is applied to example Figure 1.4b and makes its speculative constant-time.

```
1 i = 0;
2 while (i < 8) {
3     t1 = msg[i];
4     t2 = key[i];
5     t1 ^= t2;
6     msg[i] = t1;
7     i += 1;
8 }
```

(a) Unprotected

```
1 ms = init_ms();
2 i = 0;
3 while (i < 8) {
4     ms = set_msf(i < 8, ms);
5     t1 = msg[i];
6     t1 = protect(t1, ms);
7     t2 = key[i];
8     t2 = protect(t2, ms);
9     t1 ^= t2;
10    msg[i] = t1;
11    i += 1;
12 }
13 ms = set_msf(i >= 8, ms);
```

(b) SLH

```
1 ms = init_ms();
2 i = 0;
3 while (i < 8) {
4     ms = set_msf(i < 8, ms);
5     t1 = msg[i];
6     t2 = key[i];
7     t1 ^= t2;
8     msg[i] = t1;
9     i += 1;
10 }
11 ms = set_msf(i >= 8, ms);
```

(c) selSLH

Figure 7.2 – Protected one-time pad

Selective Speculative Load Hardening Figure 7.2a presents an example that can perform secret loads from two secret arrays msg and key. The program is assumed to be constant-time which means the index i is public. Figure 7.2b presents the protected version

of the program Figure 7.2a by applying speculative load hardening. It adds init_msf() at the beginning of the program in line 1 to ensure that the program's execution starts with ms set to ⊥. In lines 4 and 13, the ms register is set correctly according to the guard (i < 8). In lines 6 and 8, protect primitive is used to mask the value speculatively assigned to t1 and t2. Figure 7.2b presents another protected version of the program Figure 7.2a where no protect primitive is added to the program because as the type of t1 and t2 are secret; there is no need to protect the secret loads (as they will be protected by constant-time programming discipline).

7.2 Programming hardened implementation

This section presents the methodology in detail.

7.2.1 Threat model and security notion

To provide mitigation against micro-architectural attacks, we need to model the threat model. The threat model is designed to implement the attacker's actions. It assumes that the attacker can observe branching decisions and addresses of memory accesses and has full control of branching. An attacker can also misuse all unsafe reads and values of all unsafe writes, which was restricted in the work presented in Chapter 5.

7.2.2 Extenison to the Jasmin language and its semantics

Figure 1.11 presents the Jasmin language. The Jasmin language is extended to include four more primitives:

- ms := init_msf() sets the misspeculation flag ms to 0. This primitive is typically used at the beginning of a program to ensure that the program starts executing from a state that is not misspeculating.
- x := mov_msf(ms) moves the misspeculation flag value to variable x
- ms := set_msf(e, ms) updates the misspeculation flag. This primitive may be used immediately after a branching instruction conditioned on e. However, it is sometimes possible to postpone updating the flag or even not update it without compromising security.
- x := protect(x, ms) conditionally masks the register x depending on the value of ms. Specifically, the value of x remains unchanged in a correct execution and is set to −1 in a misspeculated execution.

Semantics

To reason about the speculative semantics, the semantic is instrumented to include observations o, directives d.

$$p : s \xrightarrow[d]{o} p' : s'$$

The misspeculation flag tracks whether the semantic is misspeculating or not. In case of no speculation, the value of misspeculation flag is set to $false$.

Instruction semantics:

$$\frac{}{\{\} : s \to^{\epsilon}_{\epsilon} s}\ \textbf{0-STEP} \qquad \frac{i : s \xrightarrow[d]{o} s_1 \quad c : s_1 \xrightarrow[d]{o} s_2}{\{i; c\} : s \xrightarrow[d::D]{o::O} s_2}\ \textbf{S-STEP}$$

$$\frac{}{x := e; c : \langle \rho, \mu, b\rangle \xrightarrow[step]{\bullet} c : \langle \rho\{x \leftarrow [\![e]\!]_\rho\}, \mu, b\rangle}\ \textbf{ASSGN}$$

$$\frac{[\![e]\!]_\rho = i \quad i \in |a| \quad \mu(a, i) = v}{\mathsf{ssafe} x := a[e]; c : \langle \rho, \mu, b\rangle \xrightarrow[step]{\mathsf{read}(a,i)} c : \langle \rho\{x \leftarrow v\}, \mu, b\rangle}\ \textbf{SAFE LOAD}$$

$$\frac{[\![e]\!]_\rho = i \quad i \notin |a| \quad i' \in |\alpha| \quad \mu(\alpha, i') = v}{x := a[e]; c : \langle \rho, \mu, \top\rangle \xrightarrow[\mathsf{load}(\alpha,i')]{\mathsf{read}(a,i)} c : \langle \rho\{x \leftarrow v\}, \mu, \top\rangle}\ \textbf{LOAD}$$

$$\frac{[\![e]\!]_\rho = i \quad i \in |a|}{\mathsf{ssafe}\ a[e] := e'; c : \langle \rho, \mu, b\rangle \xrightarrow[step]{\mathsf{write}(a,i)} c : \langle \rho, \mu\{(a, i) \leftarrow [\![e']\!]_\rho\}, b\rangle}\ \textbf{SAFE STORE}$$

$$\frac{[\![e]\!]_\rho = i \quad i \notin |a| \quad \alpha \in \mathcal{A}, i' \in |\alpha|}{\mathsf{ssafe}\ a[e] := e'; c : \langle \rho, \mu, \top\rangle \xrightarrow[\mathsf{store}(\alpha,i')]{\mathsf{write}(a,i)} c : \langle \rho, \mu\{(\alpha, i') \leftarrow [\![e']\!]_\rho\}, \top\rangle}\ \textbf{STORE}$$

$$\frac{e = \mathsf{if}\ (d = \mathsf{force})\ \mathsf{then}\ \neg[\![e]\!]_\rho\ \mathsf{else}\ [\![e]\!]_\rho}{\mathsf{if}\ e\ \mathsf{then}\ c_{tt}\ \mathsf{else}\ c_{ff}; c : \langle \rho, \mu, b\rangle \xrightarrow[d]{\mathsf{branch}\ e} c_e; c : \langle \rho, \mu, b \wedge d = \mathsf{force}\rangle}\ \textbf{COND}$$

$$\frac{e = \mathsf{if}\ (d = \mathsf{force})\ \mathsf{then}\ \neg[\![e]\!]_\rho\ \mathsf{else}\ [\![e]\!]_\rho \quad c_{tt} = c_{ff}; \mathsf{while}\ c_{ff}\ e\ c_{tt}; c \quad c_{ff} = c_{ff}; c}{\mathsf{while}\ c_{ff}\ e\ c_{tt}; c : \langle \rho, \mu, b\rangle \xrightarrow[d]{\mathsf{branch}\ e} c_e; c : \langle \rho, \mu, b \wedge d = \mathsf{force}\rangle}\ \textbf{WHILE}$$

$$\frac{\rho' = \rho\{ms \leftarrow 0\}}{ms = \mathsf{init\_msf}(); c : \langle \rho, \mu, \bot\rangle \xrightarrow[step]{\bullet} c : \langle \rho', \mu, \bot\rangle}\ \textbf{INIT}$$

$$\frac{m = \mathsf{if}\ ([\![e]\!]_\rho = \top)\ \mathsf{then}\ [\![ms]\!]_\rho\ \mathsf{else}\ -1}{ms = \mathsf{set\_msf}(e, \mathsf{ms}); c : \langle \rho, \mu, b\rangle \xrightarrow[step]{\bullet} c : \langle \rho\{ms \leftarrow m\}, \mu, b\rangle}\ \textbf{SET}$$

$$\frac{v = \mathsf{if}\ ([\![ms]\!]_\rho = -1)\ \mathsf{then}\ -1\ \mathsf{else}\ [\![x]\!]_\rho}{y = \mathsf{protect}(x, \mathsf{ms}); c : \langle \rho, \mu, b\rangle \xrightarrow[step]{\bullet} c : \langle \rho\{y \leftarrow v\}, \mu, b\rangle}\ \textbf{PROTECT}$$

$$\frac{}{x = \mathsf{mov\_msf}(\mathsf{ms}); c : \langle \rho, \mu, b\rangle \xrightarrow[step]{\bullet} c : \langle \rho\{x \leftarrow ms\}, \mu, b\rangle}\ \textbf{MOVE}$$

Figure 7.3 – Instrumented semantics.

The directive d represents the attacker's move before executing the instruction, and the observation o represents the information that will be observed/visible when the instruction is executed. Formally, the sets of directives and observations are defined as follows:

$$\begin{aligned} d \in Dir &::= \mathsf{step} \mid \mathsf{force} \mid \mathsf{load}(a, i) \mid \mathsf{store}(a, i) \\ o \in Obs &::= \bullet \mid \mathsf{read}(a, v) \mid \mathsf{write}(a, v) \mid \mathsf{branch}\ b \end{aligned}$$

The attacker uses the directive step in case of sequential execution. In the case of step directive issued by the attacker, the guard for the conditional instruction is evaluated first, then proceeds to either the $true$ or $false$ branch, depending on the guard's value. The directive force is used by the attacker to force execution to enter into a misspeculated branch; hence, in turn, it also modifies the misspeculation flag. The directives $\mathsf{load}(a, i)$ and $\mathsf{store}(a, i)$ are used by the attacker to read from and write to addresses of their choice in case of a (speculatively) unsafe memory read or write.

The semantics is defined as a small-step relation represented as $p : s \xrightarrow[d]{o} p' : s'$. It states that a single step of execution of the program p starting from state s produces the state s' and observation o with the remaining program p' under the directive d. The state s is defined as a triplet $\langle \rho, \mu, b \rangle$ where ρ is the register map (assigns values to registers), μ is the memory map (gives value to the valid addresses) and b is the misspeculation flag.

Also, as discussed in Chapter 5, we know that adversarial semantics with backtracks are equivalent to forward semantics (semantics without backtracks). Following a number of lemmas in Section 5.4.3 of Chapter 5 has successfully proved that the directive set without *backtrack* is enough to reason about speculative constant-time. Hence, this work considers the semantics without backtracking. The rules are present in Figure 7.3.

The ASSGN rule defines the semantics of assignment instruction. It updates the register map by assigning the value obtained by evaluating e to the register x under the directive step. It does not produce any observation (as it involves only computation on registers). The SAFE LOAD rule defines the semantics of safe load operation. It ensures that the evaluation of the expression e (the index of the array being accessed) is always less than the array length $|a|$. The value present at the index i in the memory map μ is assigned to the register x and produces the observation $\mathsf{read}(a, i)$ (leaking the address). The load is carried out under ssafe annotation, so the execution is carried out under step directive. The LOAD rule defines the semantics of unsafe load. It updates the register with the value obtained by any arbitrary address (index i' present in array α) present in the memory as governed by the attacker using the directive $\mathsf{load}(\alpha, i')$. It also leaks the address ($\mathsf{read}(a, i)$), which is obtained by evaluating the index e. Because the value can be loaded speculatively from any arbitrary address, the speculation flag is set to $\top$ ($true$), and it should also start from the state where the program is misspeculating. We assume that all programs are safe; hence, unsafe execution can only happen under speculative execution ($b = \top$). The semantics of safe and unsafe store is very similar to the load operation and is presented in the rule SAFE STORE and STORE.

The COND rule defines the semantics of conditional instruction. The conditional instruction gives the freedom to the attacker to force the execution to any of the two branches irrespective of the evaluation of the guard. The misspeculation flag is updated by evaluating the condition ($b \wedge d = \mathsf{force}$). If the execution is carried out under the force directive, then the misspeculation flag is set to $\top$. Both in case of speculation or sequential, evaluation of guard is leaked. The WHILE rule defines the semantics of the while loop, and it is similar to the conditional instruction.

The INIT rule defines the semantics of initialization of the ms variable to 0, and it

operates like a lfence instruction present in x86. It ensures the execution is non-speculating and ms state is set to $\bot$. The **SET** rule defines the semantics of $\mathsf{set\_msf}(e, \mathsf{ms})$, which updates the value of ms based on the evaluation of the boolean expression e.

The **PROTECT** rule defines the semantics of $\mathsf{protect}(x, \mathsf{ms})$. It protects the value of x in case of misspeculation. It assigns the default value to x when the misspeculation flag ms contains the value -1 and assigns the correct value to x in case of no speculation. The **MOVE** rule defines the semantics of $x := \mathsf{mov\_msf}(\mathsf{ms})$, which moves the value of ms to a variable x.

7.3 Type system

This section presents a constraint-based type system that enforces speculative constant-time.

7.3.1 Security types

The security lattice is defined as {H, L}, with order L $\leq$ H. The high-security level H is used to classify secret data and the low-security level L is used to classify public data. The formal notation of security level is defined as

$$T := t \mid H \mid L$$

where t ranges over level (or type) variables. A security type is a pair $T = (T_n, T_s)$ of security levels. T_n represents the security level under normal (sequential) execution, and T_s represents the security level under all executions. The relation $T_n \leq T_s$ can always be justified in both sequential and speculative execution. The formal definition is as follows:

- (L, L) denotes the public data in both sequential and speculative executions.
- (H, H) denotes the secret data.
- (L, H) denotes the transient data. Transient data is public under sequential execution but may depend on secrets under speculative execution.

7.3.2 Constraint sets

A constraint set is a set of inequalities of the form $t_1 \leq t_2$ where t_1 and t_2 are security levels.

Definition 17 (Closure)**.** *The closure $\overline{C}$ of a constraint C is the smallest set of constraints that contains C and is closed under transitivity.*

Definition 18 (Consistent)**.** *The constraint set C is consistent if (H, L) $\notin \overline{C}$ and iff there exists a valuation θ which maps level variables to {L, H} such that for every constraint $(t_1 \leq t_2) \in C$, $\theta(t_1) = \theta(t_2)$ or $\theta(t_1) = L$ and $\theta(t_2) = H$.*

7.3.3 Misspeculation type (MSF-type)

The type system assigns different types depending on whether execution is misspeculating or not. The following grammar defines the MSF-type:

$$\Sigma := \mathsf{unknown} \mid \mathsf{ms} \mid \mathsf{ms}_{|e}$$

Informally speaking, these misspeculation types are defined to represent these scenarios:

- During misspeculation, a situation might arise where the states are not known to be misspeculating. No register variables are expected to contain this information. The unknown state represents such a scenario.
- The type ms resembles the state when the execution is misspeculating. The other known fact about the misspeculating state is that the register containing the misspecualtion flag ms contains -1. The type ms is used to ensure that the values are correctly masked.
- The type $\mathsf{ms}_{|e}$ resembles the state when the execution is misspeculating and the boolean expression e is assumed to be true, the register containing the misspeculation flag ms is set to -1. The type $\mathsf{ms}_{|e}$ is used by the type system to postpone the update of misspeculation flag ms after a branching instruction.

There are various operations defined on the misspeculation type:

$$\begin{array}{lcl}
\Sigma_1 \subseteq \Sigma_2 & := & \Sigma_1 = \mathsf{unknown} \vee \Sigma_1 = \Sigma_2 \\
\Sigma_1 \cap \Sigma_2 & := & \text{if } \Sigma_1 = \Sigma_2 \text{ then } \Sigma_1 \text{ else } \mathsf{unknown} \\
\Sigma^{|x} & := & \text{if } x \in \mathsf{fv}(\Sigma) \text{ then } \mathsf{unknown} \text{ else } \Sigma \\
\Sigma^{=x} & := & \text{if } \Sigma = \mathsf{ms} \text{ then } x \text{ else } \mathsf{unknown} \\
\Sigma_{|b} & := & \text{if } \Sigma = \mathsf{ms} \text{ then } \mathsf{ms}_{|b} \text{ else } \mathsf{unknown}
\end{array}$$

These operations are used in maintaining the MSF type during the program's execution.

7.3.4 Typing rules

The typing judgments are of the form:

$$\Sigma, \Gamma \vdash c : \Sigma', \Gamma' \mid C \qquad \Gamma \vdash e : \Gamma' \mid C$$

where Γ and Γ' are security environments, Σ and Σ' are MSF-types, and C is the set of constraints. The typing rules for expressions and instructions of Jasmin (language presented in Figure 1.11) is presented in Figure 7.4.

There are two sets of typing rules: one for expressions and one for instructions.

Typing rules for expressions The typing rules for expressions use the judgment of the form (does not mention MSF-type):

$$\Gamma \vdash e : \Gamma' \mid C$$

The typing rules CONST, BOOL and ARRAY INIT are used for typing constants, booleans and array initialization. They are all public and do not generate any constraints. The typing rule VAR obtains the type from the context and does not generate any constraints. The rule OP presents the rule for operators; it collects the constraints from each of its arguments and generates a new constraint, which forces the security level of $op(e_1, \ldots, e_2)$ to be maximum of the security levels of arguments.

Typing rule for instructions The typing rules for instructions use the judgment of the form (does not mention MSF-type):

$$\Sigma, \Gamma \vdash c : \Sigma', \Gamma' \mid C$$

The typing rule **INIT** type checks the $\mathsf{init\_msf}()$ primitive. It ensures that the MSF-type is set to ms after initialization. The security level of ms is set to (L, L) as the value assigned to ms variable during initialization is 0, and the security level of all remaining variables is set to $(\Gamma_n(x), \Gamma_n(x))$ as transient values are committed during initialization. The typing rule **SET** type checks the $\mathsf{set\_msf}(e, \mathsf{ms})$ primitive. It updates the ms depending on the value of e. The MSF-type before the update is $\mathsf{ms}_{|e}$, and ensures that the MSF-type after the update is ms.

The typing rule **PROTECT** type checks the $\mathsf{protect}(x, \mathsf{ms})$ primitive. It requires that the MSF-type is ms and ensures that the output type of y is $(\Gamma_n(x), \Gamma_n(x))$. The $\mathsf{protect}$ primitive copies the value of x to y in case of sequential execution, and in the case of misspeculation, the value of y is set to default value. The speculative type of y is not set to L to ensure $\tau_n \leq \tau_s$. In the case $y = \mathsf{ms}$, the output type of y is set to unknown.

The typing rule **MOVE** type checks the $\mathsf{mov\_msf}(\mathsf{ms})$ primitive. It requires that the MSF-type is ms. The security level of x is set to (L,L) as the value assigned to the ms variable is always public, and the security level of all remaining variables does not change. The output MSF-state $\mathsf{ms}^{=x}$ ensures that x is added to the set of ms variables.

The typing rule **ASSGN** describes the typing of an assignment instruction. It enforces the absence of direct flow by not restricting the subtyping relation between e and x. If $x := \mathsf{ms}$ or x occurs in b, the rule sets the result MSF-type to unknown. This ensures that only $x := \mathsf{mov\_msf}(\mathsf{ms})$ can move the ms value from one variable to another.

The typing rule **LOAD** ensures that the array is accessed with an index e, which is assigned the type (L, L). The value obtained after array access is assigned to the variable x, which gets the fresh type t_{n_x}, t_{s_x}. $t_{n_a} \leq t_{n_x}$ constraint prevents the direct flow in case of sequential execution, and $H \leq t_{s_x}$ constraint prevents direct flow in speculative execution (as in case of speculation, the array access may lead to load from an unsafe location). This shows that there might be the case that a public load in sequential semantics turns into loading a secret in the subject of speculation, and there is a need to protect such loads. It helps avoid the necessity of memory safety because even if out-of-bound access happens during a load operation, the types (L, H) ensure that the variable x needs to be protected.

The typing rule **SAFE LOAD** considers the case where the load is speculatively safe ($ssafe$ annotation distinguishes a safe load from an unsafe load). The constraint for the index is similar to the unsafe load, but the constraint for the variable x ($H \leq t_{s_x}$) can be replaced by $\Gamma_s(a) \leq t_s$.

The typing rule **STORE** ensures that the array is accessed at an index i which is assigned the type (L, L) and the stored expression is assigned type (t_{n_e}, t_{s_e}) which should be less than equal to the type of a. As this can lead to an unsafe store because the store can be performed at an arbitrary address, the type system creates a fresh environment Γ' and adds some constraints. The constraint $\Gamma \leq \Gamma'$ ensures that the type in Γ' is at least the one in Γ. The constraint $t_{n_e} \leq \Gamma'(a)$ and $t_{s_e} \leq \Gamma'(a)$ ensures that the type assigned to a is at least at the level of the stored expression. The constraint $t_{s_e} \leq \Gamma'_s(a')$ captures the fact that the speculative type of all other arrays should be at least the speculative type t_{s_e} of the stored value due to a speculatively unsafe store.

Typing rules for expressions:

$$\frac{}{\Gamma \vdash c : (L, L) \mid \phi}\ \textbf{CONST} \qquad \frac{}{\Gamma \vdash b : (L, L) \mid \phi}\ \textbf{BOOL}$$

$$\frac{}{\Gamma \vdash x : \Gamma(x) \mid \phi}\ \textbf{VAR} \qquad \frac{}{\Gamma \vdash a[n] : (L, L) \mid \phi}\ \textbf{ARRAY INIT}$$

$$\frac{\Gamma \vdash e_i : (t_{n_i}, t_{s_i}) \mid C_i \qquad t(fresh)}{\Gamma \vdash op(e_1, \ldots, e_2) : t \mid C_i \cup (t_{n_i}, t_{s_i}) \leq t}\ \textbf{OP}$$

Typing rules for instructions:

$$\frac{}{\Sigma, \Gamma \vdash \epsilon : \Sigma, \Gamma \mid \phi}\ \textbf{EMPTY} \qquad \frac{\Sigma, \Gamma \vdash i : \Sigma_i, \Gamma_i \mid C_i \quad \Sigma_i, \Gamma_i \vdash c : \Sigma_c, \Gamma_c \mid C_c}{\Sigma, \Gamma \vdash i; c : \Sigma_c, \Gamma_c \mid C_i \cup C_c}\ \textbf{SEQ}$$

$$\frac{\Sigma, \Gamma \vdash e : \Sigma, (t_{n_e}, t_{s_e}) \mid C}{\Sigma, \Gamma \vdash x := e : \Sigma^{|x}, \Gamma\{x \leftarrow (t_{n_x}, t_{s_x})\} \mid C}\ \textbf{ASSGN}$$

$$\frac{\Gamma'(\mathsf{ms}) = L \quad \forall x \neq \mathsf{ms}, \Gamma'(x) = \Gamma_n(x)}{\Sigma, \Gamma \vdash \mathsf{ms} := \mathsf{init\_msf}() : \mathsf{ms}, \Gamma' \mid \phi}\ \textbf{INIT}$$

$$\frac{}{\mathsf{ms}_{|e}, \Gamma \vdash \mathsf{ms} := \mathsf{set\_msf}(e, \mathsf{ms}) : \mathsf{ms}, \Gamma \mid \phi}\ \textbf{SET}$$

$$\frac{\Gamma' = \Gamma\{y \leftarrow (\Gamma_n(x), \Gamma_n(x))\}}{\mathsf{ms}, \Gamma \vdash y := \mathsf{protect}(x, \mathsf{ms}) : \mathsf{ms}^{|y}, \Gamma' \mid \phi}\ \textbf{PROTECT}$$

$$\frac{\Gamma'(x) = L}{\mathsf{ms}, \Gamma \vdash x := \mathsf{mov\_msf}(\mathsf{ms}) : \mathsf{ms}^{=x}, \Gamma' \mid \phi}\ \textbf{MOVE}$$

$$\frac{\Gamma \vdash e : (t_{n_e}, t_{s_e}) \mid C_i \quad (t_{n_x}, t_{n_s})(fresh)}{\begin{array}{c}\Sigma, \Gamma \vdash x := a[e] : \Sigma^{|x}, \Gamma\{x \leftarrow (t_{n_x}, t_{s_x})\} \mid \\ C_i \cup \{t_{n_e} \leq L, t_{s_e} \leq L, \Gamma_n(a) \leq t_{n_x}, H \leq t_{s_x}\}\end{array}}\ \textbf{LOAD}$$

$$\frac{\Gamma \vdash e : (t_{n_e}, t_{s_e}) \mid C_i \quad t(fresh)}{\begin{array}{c}\Sigma, \Gamma \vdash \mathsf{ssafe}\ x := a[e] : \Sigma^{|x}, \Gamma\{x \leftarrow t\} \mid \\ C_i \cup \{t_{n_e} \leq L, t_{s_e} \leq L, \Gamma(a) \leq t\}\end{array}}\ \textbf{SAFE LOAD}$$

$$\frac{\Gamma \vdash i : (t_{n_i}, t_{s_i}) \mid C_i \quad \Gamma \vdash e : (t_{n_e}, t_{s_e}) \mid C_e \quad \Gamma'(fresh)}{\begin{array}{c}\Sigma, \Gamma \vdash a[i] := e : \Sigma, \Gamma' \mid C_i \cup C_e \cup \{\Gamma \leq \Gamma'\} \cup \\ \{t_{n_i} \leq L, t_{s_i} \leq L, t_{n_e} \leq \Gamma'(a), t_{s_e} \leq \Gamma'(a)\} \cup \{t_{s_e} \leq \Gamma'_s(a') | a' \in A, a' \neq a\}\end{array}}\ \textbf{STORE}$$

$$\frac{\Gamma \vdash i : (t_{n_i}, t_{s_i}) \mid C_i \quad \Gamma \vdash e : (t_{n_e}, t_{s_e}) \mid C_e \quad \Gamma'(fresh)}{\begin{array}{c}\Sigma, \Gamma \vdash \mathsf{ssafe}\ a[i] := e : \Sigma, \Gamma' \mid \\ C_i \cup C_e \cup \{\Gamma \leq \Gamma'\} \cup \{t_{n_i} \leq L, t_{s_i} \leq L, t_{n_e} \leq \Gamma'(a), t_{s_e} \leq \Gamma'(a)\}\end{array}}\ \textbf{SAFE STORE}$$

$$\frac{\begin{array}{c}\Gamma \vdash b : (t_{n_b}, t_{s_b}) \mid C_b \quad \Sigma_{|b}, \Gamma \vdash c_1 : \Sigma_1, \Gamma_1 \mid C_1 \\ \Sigma_{|\neg b}, \Gamma \vdash c_2 : \Sigma_2, \Gamma_2 \mid C_2 \quad \Gamma'(fresh)\end{array}}{\begin{array}{c}\Sigma, \Gamma \vdash \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 : \Sigma_1 \cap \Sigma_2, \Gamma' \mid \\ C_b \cup C_1 \cup C_2 \cup \{t_{n_b} \leq L\} \cup \{t_{s_b} \leq L\} \cup \{\Gamma_1 \leq \Gamma'\} \cup \{\Gamma_2 \leq \Gamma'\}\end{array}}\ \textbf{COND}$$

$$\frac{\Gamma' \vdash b : (t_{n_b}, t_{s_b}) \mid C_b \quad \Sigma'_{|b}, \Gamma' \vdash c : \Sigma_0, \Gamma_0 \mid C \quad \Gamma'(fresh) \quad \Sigma' \subseteq \Sigma \quad \Sigma' \subseteq \Sigma_0}{\begin{array}{c}\Sigma, \Gamma \vdash \mathsf{while}\ b\ \mathsf{do}\ c : \Sigma'_{|\neg b}, \Gamma' \mid \\ C_b \cup C \cup \{t_{n_b} \leq L\} \cup \{t_{s_b} \leq L\} \cup \{\Gamma \leq \Gamma'\}\{\Gamma_0 \leq \Gamma'\}\end{array}}\ \textbf{WHILE}$$

Figure 7.4 – Type system

The typing rule **SAFE STORE** ensures that the store is speculatively safe and relaxes a lot of constraints, as discussed in the unsafe store. In this case, we do not have to deal with all other arrays and assign them types based on unsafe speculation.

The typing rule **COND** ensures that the boolean guard is public; hence, it assigns the type (L, L) to it. Both the branches are typed with MSF-type $\Sigma_{|b}$ and $\Sigma_{|\neg b}$ respectively. The output MSF-type is $(\Sigma_1 \cap \Sigma_2)$, which ensures that the result MSF-type is unknown if both branches disagree on their result MSF-type. The output security type is Γ', a maximum of Γ_1 and Γ_2. The typing rule **WHILE** follows the same idea.

Figure 7.5 presents a function branch that illustrates how Jasmin programs are annotated with security types.

```
1 export fn branch(#transient reg u64 a, #secret reg u64 b c) −→#secret reg u64 {
2   reg u64 r;
3   _= #init_msf();
4   if a == 0 {
5     r = b;
6   } else {
7   r = c;
8   }
9   return r;
10 }
```

Figure 7.5 – Illustrative example with type annotation

The function branch takes three arguments a, b and c. The variables taken as arguments to the export function should be atleast transient to protect the Jasmin code from the non-trusted code that might be misspeculating. The type of local variables are inferred using the type associated with the function's arguments. The #init_msf() primitive at line 3 ensures that the misspeculating path is backtracked to proceed with correct value at the beginning of the function.

7.4 Speculative constant-time

The load and store operations with $ssafe$ annotations are assumed to be speculatively safe, which means the attacker cannot speculatively load or store at any arbitrary address. This property is stated formally as "Speculative Safety" and is presented as follows:

Definition 19 (Speculative Safety)**.** *A load or a store instruction of the form ss(a, e) = x := a[e] or ss(a,e) = a[e] := e' is speculatively safe in a configuration consisting of program p and state s if* $\forall p\ s, p : s \xrightarrow[d]{l\!\!\!\!\rightarrow} ssafe\ ss(a,e); p' : s'$ *then* $[\![e]\!]_\rho \in |a|$.

All the loads and stores with "ssafe" notation in the input program are assumed to verify this property.

Definition 20 (Speculative constant-time)**.** *Let ϕ be an equivalence relation on states. A program c is speculative constant-time with respect to ϕ (or $\phi - SCT$),*
iff $\forall\ D\ \rho_1\ \mu_2\ \rho_2\ \mu_2,\ \langle \rho_1, \mu_1, \bot\rangle\ \phi\ \langle \rho_2, \mu_2, \bot\rangle \wedge c : \langle \rho_1, \mu_1, \bot\rangle \xrightarrow[D]{O_1\!\!\!\!\rightarrow} c_1 : s_1 \wedge c : \langle \rho_2, \mu_2, \bot\rangle \xrightarrow[D]{O_2\!\!\!\!\rightarrow} c_2 : s_2$, *implies* $O_1 = O_2$.

A program suffices to be speculative constant-time if the observations produced during its executions starting from state $\langle \rho_1, \mu_1, \bot \rangle$ and $\langle \rho_2, \mu_2, \bot \rangle$ (which are in equivalence relation ϕ, presented as $=^{\Sigma}_{\theta\Gamma}$ and is defined below) are equal. Equality of the observations ensures that there are no leakages that the attacker can predict from two executions. To carry out the proof, we also require that the resulting states will be in equivalence. The result that a program is speculative constant-time also implies that it is constant-time, but the vice-versa is not true. The constant-time property corresponds to cases where directives can be any directive except the force directive.

Definition 21 (Equivalence)**.** *Equivalence relation* $=^{\Sigma}_{\theta\Gamma}$ *between two states* $\langle \rho_1, \mu_1, b_1 \rangle$ *and* $\langle \rho_1, \mu_1, b_1 \rangle$ *is defined as follows:*

$$\langle \rho_1, \mu_1, b_1 \rangle =^{\Sigma}_{\theta\Gamma} \langle \rho_2, \mu_2, b_2 \rangle \equiv \begin{cases} b_1 = b_2 \\ \forall x, \Gamma_s(x) = L \implies \rho_1(x) = \rho_2(x) \\ \neg b_1 \implies \forall x, \Gamma_n(x) = L \implies \rho_1(x) = \rho_2(x) \\ \forall a, \Gamma_s(a) = L \implies \mu_1(x) = \mu_2(x) \\ \neg b_1 \implies \forall a, \Gamma_n(a) = L \implies \mu_1(x) = \mu_2(x) \\ \rho_1, b_1 \vDash \Sigma \\ \rho_2, b_2 \vDash \Sigma \end{cases}$$

the MSF state correspondence is defined as follows:

- $\rho, b \vDash \bot$ is always valid.
- $\rho, b \vDash ms \equiv b \iff [\![ms]\!]_\rho = -1$
- $\rho, \top \vDash \mathsf{ms}_{|e} \equiv [\![e]\!]_\rho = \bot \Rightarrow [\![ms]\!]_\rho = -1$
- $\rho, \bot \vDash \mathsf{ms}_{|e} \equiv [\![e]\!]_\rho = \top \wedge [\![ms]\!]_\rho \neq -1$

7.5 Soundness

The type system is sound, i.e., it only accepts speculative constant-time programs. The soundness theorem proves the correctness of the type system stated in Figure 7.4. For soundness, we show that if p is safe and

$$\Sigma, \Gamma \vdash p : \Sigma', \Gamma' \mid C$$

then p is speculative constant-time. Informally, the partial equivalence relation $=^{\Sigma}_{\Gamma,C}$ is defined as setting in relation states that coincide on their public parts, as defined by necessity from C and Γ, and are furthermore compatible with the MSF type Σ.

The first step of the proof is to show that for every two executions

$$p : s_1 \xrightarrow[d]{o_1} p_1 : s_1'$$
$$p : s_2 \xrightarrow[d]{o_2} p_2 : s_2'$$

such that $s_1 =^{\Sigma}_{\Gamma,C} s_2$, we have $o_1 = o_2$, $p_1 = p_2$ and $s_1' =^{\Sigma_0}_{\Gamma_0,C_0} s_2'$. There exist such Σ_0, Γ_0, C_0 such that $\Sigma_0, \Gamma_0 \vdash p_1 : \Sigma', \Gamma' \mid C_0$ and $s_1' =^{\Sigma_0}_{\Gamma_0,C_0} s_2'$. Using the latter, we can

prove by induction on the set of directives that executions started in related states yield equal leakage, as required.

The proof will be similar to as explained for the simple language in Chapter 6. The overall proof script and idea are very similar; it only needs to be extended for a more realistic compiler than the toy-level compiler.

7.6 Integration in Jasmin

This methodology is implemented on top of the latest version of the Jasmin framework.

7.6.1 New extensions in Jasmin framework

Section 1.3 of Chapter 1 describes the Jasmin framework. A recent extension to Jasmin allows array variables to have another optional attribute: **ptr**, along with **reg** and **stack**. The type **ptr** indicates that the variable contains a pointer to an array. It can also be combined with **reg** to form type **reg ptr**, which means the variable is a pointer stored in a register. Similarly, it can also be a **stack ptr**, meaning the variable is a pointer stored in the stack.

Another recent extension is first-class functions. In the initial development of Jasmin, all functions were automatically inlined. However, such inlining is inappropriate for implementations with a non-trivial call graph. It is worth noting that function calls use **reg ptr** commonly as arguments.

Finally, the Jasmin language was extended with a system call randombytes. Previously, randomness was passed as a parameter. However, this is not viable for implementations that may need an arbitrary amount of randomness or to implement widely used cryptographic APIs. A call to randombytes takes a reg ptr as an argument (a pointer to an array of fixed length) and fills the array with random bytes. The random bytes are assumed to be safe for this work.

7.6.2 Implementation details

The Jasmin framework is enriched with an intra-procedural analysis that implements the type system presented in Figure 7.4. Some key aspects are discussed below:

Functions To enable intra-procedural analysis, the type-system implementation infers the security levels for the function's inputs and outputs together with a security effect. This level is an upper bound for the speculative stored potentially performed by the function outside its expected memory scope. The security effect is used in the rule for function calls to update the speculative type of the caller's local variables, similar to what is done by the STORE rule. This is similar to the handling of effectful functions in information-flow-type systems.

Pointer variables The implementation associates two types, i.e., two pairs of security levels, to ptr variables. The first type is for the pointer, while the second type is for the pointed data. The use of two types can be exploited by programmers when spilling pointers into stack memory. Again, having two security types for pointers is common in information-flow type systems for pointer languages.

System calls The type system requires that the pointer and the length argument of randombytes be public L and considers that the output is secret H. The type system assumes that the stack effect of the system call is high, which sets to secret H the speculative types of all arrays—similar to a speculatively unsafe store.

Declassification The type system implements a *declassify* construct that the programmer can use to declare intended leakage. However, the guarantees made by this work do not include declassification, and it only type-checks the program without declassification. But in general, the techniques explained in the section Section 6.5 of Chapter 6 and [Shivakumar et al., pear] will apply to this setting as well and will help to yield a proof of relative non-interference for typable programs.

Constraint generation In the type system explained in Figure 7.4, there are constraints to represent inequality between security levels. A set of constraints can be represented with a directed graph where there is an edge between between ℓ_1 and ℓ_2 iff $\ell_1 <= \ell_2$. Constraint generation can be implemented efficiently using a union-find data structure. Each time a constraint is added, the constraint generation algorithm checks if it creates a cycle. If a cycle is found, either it contains L and H, and an error is immediately reported (providing a relatively good location error), or all the variables of the cycle are merged, allowing us to reduce the size of the graph.

7.6.3 Integration in Jasmin

The Jasmin language is extended with new primitives and security annotations to tag variables and arrays with security levels. Once written, programs are checked for safety using the Jasmin safety checker to guarantee that programs are safe. All array accesses of the form $a[n]$ where n is a constant within the array's bounds are considered annotated as speculatively safe. The implementation of the type system is typically used for checking speculative constant-time before compiling programs. Typing source programs generally simplifies analysis but may occasionally cause a loss of precision due to the inability to verify side conditions such as speculative safety or variable conditions on MSF-type. This is easily compensated by using compilation passes such as *inlining*, *loop unrolling*, and *constant propogation* to reveal which array accesses have constant indices and can be marked as speculatively safe, or to make sure that information on MSF-type is not lost.

The lowering pass of the compiler is also extended to lower down the new primitives.

- $\mathsf{ms} = \mathsf{init\_msf}()$ is compiled into $\mathsf{lfence}; \mathsf{ms} = 0;$
- $\mathsf{ms} = \mathsf{set\_msf}(\mathsf{e}, \mathsf{ms})$ into branchless conditional move $\mathsf{ms} = -1 \text{ if } !\mathsf{e}$, where $!\mathsf{e}$ is the negation of e, and thus is not subject to speculation on x86 architectures;
- $\mathsf{x} = \mathsf{protect}(\mathsf{x}, \mathsf{ms})$ is compiled into $\mathsf{x} \mid = \mathsf{ms}$, where x is updated with bitwise OR operation between x and ms.

7.6.4 Application to crypto software

The type system described in Figure 7.4 is used to protect high-performance implementations of cryptographic primitives. The various implementations protected against speculative constant-time in this work are ChaCha20, Poly1305, secretbox, X25519, and Kyber.

7.6.5 Benchmarking and results

The cost of protecting cryptographic software against Spectre v1 is monitored for evaluation. The cost in terms of developer effort is evaluated, and then the cost of type-checking Jasmin software with protections is discussed in this work. Finally, the computational overhead is also measured to evaluate the execution of new primitives. The evaluation is carried out on the code written in Jasmin that is already protected against traditional timing attacks.

The implementation done in this work targets libjade, a cryptographic library that is currently in development and is written entirely in Jasmin. It implements multiple symmetric primitives (hash functions, stream ciphers, and authenticators); elliptical-curve scalar multiplication using X25519 [Bernstein, 2006]; and lattice-based key encapsulation using Kyber [Bos et al., 2018].

The original libjadelibrary[1] consists of about 16 k lines of Jasmin code and provides 72 entry points. For most primitives, it has multiple implementations, including reference implementations and optimized implementations using different instruction-set extensions (e.g., the AVX2 vector-instruction extension). At the moment, Jasmin only supports AMD64 as the target architecture, so all implementations are targeting this architecture.

7.6.6 Development effort

The complete library is adapted to protect each entry point against Spectre v1. In total this required inserting 79 init_msf, 74 set_msf, and 73 protect primitives. Insertion of init_msf is straightforward: one such primitive is needed for each entry point before performing any leaking operation (e.g., a load, store, or branch) on any data. Additionally one init_msf is needed after each call to randombytes; libjade has a total of 4 such calls, all in the implementations of Kyber. The remaining 3 init_msf primitives are in 3 different implementations of secretbox; the misspeculation flag is not chosen to be tracked up to the declassification, but place an init_msf right after declassify instead. Insertion of set_msf and protect is not quite as straight-forward, but guided by the type system and corresponding compiler errors. Overall, the developer effort is remarkably low. This efficiency is made possible by the enhanced type system in Jasmin. Without the help of this type system, insertion of protections would be much more cumbersome and, more importantly, error-prone.

7.6.7 Performance of the type-checker

Type-checking SCT all implementations from the whole libjade library takes a total of a few seconds on a developer's laptop. For reference, the complete compilation of libjade from Jasmin to assembly takes a couple of minutes on the same laptop.

7.6.8 Compilation overhead

To assess the run-time impact of our Spectre v1 protections added to libjade, the execution time of each primitive is measured; for primitives with variable-length input a sample of representative message sizes is considered.

[1] Publicly available on the web: https://github.com/formosa-crypto/libjade/tree/ece99a3bbd8ebd831f285da0c909daba1ce2972d.

Table 7.1 – Benchmark results of the fastest implementations of select primitives in libjade without Spectre v1 protections ("constant-time", CT) and with Spectre v1 protections ("speculative constant-time", SCT) on an Intel Core i7-10700K (Comet Lake) CPU

Primitive	Impl.	Op.	CT	SCT	overhead [%]
ChaCha20	avx2	32 B	314	352	12.10
	avx2	32 B xor	314	352	12.10
	avx2	128 B	330	370	12.12
	avx2	128 B xor	338	374	10.65
	avx2	1 KiB	1190	1234	3.70
	avx2	1 KiB xor	1198	1242	3.67
	avx2	1 KiB	18872	18912	0.21
	avx2	16 KiB xor	18970	18994	0.13
Poly1305	avx2	32 B	46	78	69.57
	avx2	32 B verif	48	84	75.00
	avx2	128 B	136	172	26.47
	avx2	128 B verif	140	170	21.43
	avx2	1 KiB	656	686	4.57
	avx2	1 KiB verif	654	686	4.89
	avx2	16 KiB	8420	8450	0.36
	avx2	16 KiB verif	8416	8466	0.59
secretbox	avx2	32 B	1104	1138	3.08
	avx2	32 B open	1862	1950	4.73
	avx2	128 B	1198	1234	3.01
	avx2	128 B open	1960	2044	4.29
	avx2	1 KiB	3066	3110	1.44
	avx2	1 KiB open	3886	3950	1.65
	avx2	16 KiB	31298	31376	0.25
	avx2	16 KiB open	32146	32208	0.19
X25519	mulx	smult	98352	98256	-0.098
	mulx	base	98354	98262	-0.094
Kyber512	avx2	keypair	25694	25912	0.848
	avx2	enc	35186	35464	0.790
	avx2	dec	27684	27976	1.055
Kyber768	avx2	keypair	42768	42888	0.281
	avx2	enc	54518	54818	0.550
	avx2	dec	43824	44152	0.748

All measurements were performed on a single core of a machine featuring an Intel Core i7-10700K (Comet Lake) CPU with hyperthreading and TurboBoost turned off. Each reported cycle count is the median of 8192 runs for primitives with fixed input length and 1024 runs for each input length for primitives with variable input length. The standard practice of carrying out the benchmarks on an otherwise idle machine is followed in this work. While this helps to reduce variance in cycle counts, it also means that the cost of fence instructions is measured on the optimistic side and may be larger on systems under full load.

Table 7.1 reports the measurements for the fastest implementation of each of the cryptographic primitives. The fastest implementation of each primitive is focused here because this is the most relevant number for performance-critical applications.

The central result is that for sufficiently long cryptographic computations in Jasmin (i.e., when the constant overhead of the init_msf()becomes negligible), the performance impact is extremely low, typically less than 1%. The benchmark numbers that require explanation are the somewhat larger overhead for Poly1305 and the slightly negative overhead for X25519. We started investigating the reasons for these numbers, and preliminary results suggest that they are due to different code alignments. We will continue to look into this.

7.7 Related work

There is a large body of work on enforcing and mitigating Spectre attacks. Following [Cauligi et al., 2021], this work can be classified according to its target policy. Typically, the target policy is some variant of [Cauligi et al., 2020] speculative constant-time (SCT), or relative constant-time (RCT) [Guarnieri et al., 2018], a weaker property which ensures that speculative execution does not leak more than sequential execution—note that RCT is also called speculative non-interference in the literature.

Spectector [Guarnieri et al., 2018] and Pitchfork [Cauligi et al., 2020] use symbolic execution to enforce RCT and SCT for programs with fences. Their symbolic semantics over-approximates the behavior of programs and cannot be used to verify programs that use Speculative Load Hardening. BinsecRel [Daniel et al., 2021] uses relational symbolic execution to enforce SCT for programs with fences and index masking. In principle, the relational symbolic execution of BinsecRel is sufficiently precise to verify programs that Speculative Load Hardening protects. However, BinsecRel does not support all the language features required for high-speed cryptography.

Blade [Vassena et al., 2021a] is an automated tool that enforces SCT using fence and index masking—the latter can be more efficient than fencing but requires that the size of arrays is known statically, which excludes algorithms that take arbitrary length inputs as parameters. Blade is sound, i.e., it transforms every program that is typable with a constant-time system is transformed into a speculative constant-time program. The salient feature of Blade is that it carefully minimizes the number of protections. Precisely, Blade constructs a data-flow graph where nodes can be annotated as sources, i.e., they create a transient value, or sinks, i.e., they use a transient value in a leaking instruction. Blade then uses a classic min-cut algorithm to protect every path from sources to sinks. However, the protected programs obtained by Blade are less efficient than programs based on Selective Speculative Load Hardening.

[Barthe et al., 2021a] defines an information-flow type system to protect Jasmin programs against Spectre attacks. Their type system enforces a stronger form of SCT that

covers against v1 and a very limited form of v4 attacks. However, their approach is based on fences, which impose a high-performance overhead and require programs to be speculatively safe, which also incurs some performance overhead. In contrast, our type system is significantly more elaborate and accepts protected programs using (Selective) Speculative Load Hardening. Finally, our type system is implemented for the latest version of Jasmin, which includes many new challenging features.

[Shivakumar et al., pear] formalizes Selective Speculative Load Hardening. They also define a constant-time type system for a core language with fences and a declassify construct, and show that selSLH transforms typable programs into programs that satisfy RCT—it also follows from their results that typable programs without declassify are transformed into programs that satisfy SCT. In contrast, we offer mechanisms that can be implemented but are not limited to selSLH, and we prove that typable programs satisfy SCT. In addition, [Shivakumar et al., pear] uses Pitchfork to estimate the performance benefits of Selective Speculative Load Hardening on three examples: ChaCha, Donna, and Ed25519. Our implementations are better than their predicted overhead.

Chapter 8

Conclusion

Timing-based side-channel attacks are an essential class of vulnerability that needs to be addressed separately as functional correctness and memory safety do not guarantee protection against them. This thesis aims to protect programs against a broad range of timing-based attacks.

The work in this thesis starts with a formally-verified Jasmin compiler, which is extended to produce constant-time assembly programs in sequential settings. It explains a methodology on how to turn a formally-verified Jasmin compiler into a formally-verified secure Jasmin compiler that guarantees the preservation of constant-time property. The methodology is unique as it does not break the functional-correctness proofs and can be easily extended for new compiler passes. The notion of leakages and leakage transformers used in this work captures the syntax and semantics of programs in an abstract manner that can also be useful to reason about other properties like the cost of programs. This thesis also extends the work on constant-timeness to include more fine-grained policies like the model where operators like division or modulo are not considered constant-time but leaks based on the size of their operands.

Furthermore, formal models to reason about speculative constant-time are developed. The formal definition of constant-time reasoning about timing-based attacks is extended from sequential semantics to the speculative domain by incorporating speculation into the formal semantics. This thesis also includes a formal type-system (dependency analysis style and information-flow-based style) to reason about speculative constant-time at the source level. Also, it checks the correct usage of compiler-level mitigation against Spectre attacks like insertion of fence and selective speculative load-hardening. The dependency analysis helps ensure protection against Spectre v1. Ensuring protection against Spectre v4 has some limitations, like the requirement of memory safety and significant performance overhead. We later developed another more expressive type-system based on information flow that does not require memory safety and has significantly less overhead due to selective speculative load hardening adaption. The soundness of these type-systems is proved on paper. We also developed a toy language that supports declassification and an information-flow-based type system to reason about Spectre attacks. The soundness of the type-system is formally verified using the Coq theorem prover and will help in the future to carry out the soundness of the type-system for the Jasmin language.

Overall, the thesis aims to help cryptographic developers to write efficient and secure algorithms without worrying about the compiler breaking the security property like constant-time on the way to assembly. This thesis presents a formally verified method to preserve constant-time and a sound type system to ensure Speculative constant-time

at the source level with a future goal of preservation of the Speculative constant-time property from the source to assembly.

The work presented in this thesis can be further extended to provide more efficient mitigation against transient execution attacks; for example, selective speculative load-hardening only helps in providing protection against Spectre v1, and protection against Spectre v4 is provided by insertion of `fence` which could be more efficient. A more efficient mitigation against Spectre v4 will be desirable. The methodology presented in this thesis for the preservation of constant-time property can be further extended to support it for different architectures like ARM, RISC-V, etc. This thesis mainly focuses on timing-based side-channel attacks, which can be further extended to support other side-channel leaks, such as power consumption. The goal would be to make the leakage model parametric so that different side-channel models can be instantiated and used to reason about various side-channel leakages like timing, power consumption, etc.

Bibliography

[Spe, 2018] (2018). Spectre mitigations in microsoft's c/c++ compiler. https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html.

[Abate et al., 2018] Abate, C., Blanco, R., Garg, D., Hritcu, C., Patrignani, M., and Thibault, J. (2018). Exploring robust property preservation for secure compilation. In *Computer Security Foundations 2019*.

[Al Fardan and Paterson, 2013] Al Fardan, N. J. and Paterson, K. G. (2013). Lucky thirteen: Breaking the tls and dtls record protocols. In *2013 IEEE Symposium on Security and Privacy*, pages 526–540. IEEE.

[Almeida et al., 2017] Almeida, J. B., Barbosa, M., Barthe, G., Blot, A., Grégoire, B., Laporte, V., Oliveira, T., Pacheco, H., Schmidt, B., and Strub, P.-Y. (2017). Jasmin: High-assurance and high-speed cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM.

[Almeida et al., 2016a] Almeida, J. B., Barbosa, M., Barthe, G., Dupressoir, F., and Emmi, M. (2016a). Verifying Constant-Time implementations. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 53–70, Austin, TX. USENIX Association.

[Almeida et al., 2016b] Almeida, J. B., Barbosa, M., Barthe, G., Dupressoir, F., and Emmi, M. (2016b). Verifying constant-time implementations. In Holz, T. and Savage, S., editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 53–70. USENIX Association.

[Almeida et al., 2020] Almeida, J. B., Barbosa, M., Barthe, G., Grégoire, B., Koutsos, A., Laporte, V., Oliveira, T., and Strub, P.-Y. (2020). The last mile: High-assurance and high-speed cryptographic implementations. In *2020 IEEE Symposium on Security and Privacy (S&P)*, pages 965–982.

[Almeida et al., 2019] Almeida, J. B., Baritel-Ruet, C., Barbosa, M., Barthe, G., Dupressoir, F., Grégoire, B., Laporte, V., Oliveira, T., Stoughton, A., and Strub, P.-Y. (2019). Machine-checked proofs for cryptographic standards: Indifferentiability of sponge and secure high-assurance implementations of sha-3. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1607–1622, New York, NY, USA. Association for Computing Machinery.

[AMD, 2018] AMD (2018). Software Techniques for Managing Speculation on AMD Processors. https://www.amd.com/en/server-docs/software-techniques-for-managing-speculation-amd-processors.

[Antonopoulos et al., 2017] Antonopoulos, T., Gazzillo, P., Hicks, M., Koskinen, E., Terauchi, T., and Wei, S. (2017). Decomposition instead of self-composition for proving the absence of timing channels. In Cohen, A. and Vechev, M. T., editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 362–375. ACM.

[Armadio et al., 2011] Armadio, R., Asperti, A., Ayache, N., Campbell, B., Mulligan, D., Pollack, R., Regis-Gianas, Y., Coen, C. S., and Stark, I. (2011). Certified complexity. *Procedia Computer Science*, 7:175–177. Proceedings of the 2nd European Future Technologies Conference and Exhibition 2011 (FET 11).

[Barbosa et al., 2021] Barbosa, M., Barthe, G., Bhargavan, K., Blanchet, B., Cremers, C., Liao, K., and Parno, B. (2021). Sok: Computer-aided cryptography. In *42nd IEEE Symposium on Security and Privacy*.

[Barthe et al., 2014a] Barthe, G., Betarte, G., Campo, J. D., Luna, C. D., and Pichardie, D. (2014a). System-level non-interference for constant-time cryptography. In Ahn, G., Yung, M., and Li, N., editors, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 1267–1279. ACM.

[Barthe et al., 2020] Barthe, G., Blazy, S., Grégoire, B., Hutin, R., Laporte, V., Pichardie, D., and Trieu, A. (2020). Formal verification of a constant-time preserving C compiler. *Proc. ACM Program. Lang.*, 4(POPL):7:1–7:30.

[Barthe et al., 2021a] Barthe, G., Cauligi, S., Grégoire, B., Koutsos, A., Liao, K., Oliveira, T., Priya, S., Rezk, T., and Schwabe, P. (2021a). High-assurance cryptography in the spectre era. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1884–1901.

[Barthe et al., 2013] Barthe, G., Dupressoir, F., Grégoire, B., Kunz, C., Schmidt, B., and Strub, P.-Y. (2013). Easycrypt: A tutorial. In *Foundations of Security Analysis and Design*.

[Barthe et al., 2014b] Barthe, G., Dupressoir, F., Grégoire, B., Kunz, C., Schmidt, B., and Strub, P.-Y. (2014b). *EasyCrypt: A Tutorial*, pages 146–166. Springer International Publishing, Cham.

[Barthe et al., 2011a] Barthe, G., Grégoire, B., Heraud, S., and Béguelin, S. Z. (2011a). Computer-aided security proofs for the working cryptographer. In Rogaway, P., editor, *Advances in Cryptology – CRYPTO 2011*, pages 71–90, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Barthe et al., 2021b] Barthe, G., Grégoire, B., Laporte, V., and Priya, S. (2021b). Structured leakage and applications to cryptographic constant-time and cost. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 462–476, New York, NY, USA. Association for Computing Machinery.

[Barthe et al., 2011b] Barthe, G., Grégoire, B., Zanella-Béguelin, S., and Heraud, S. (2011b). Computer-Aided Security Proofs for the Working Cryptographer. In *Advances in Cryptology - {CRYPTO} 2011 - 31st Annual Cryptology Conference*, Santa Barbara, United States.

[Barthe et al., 2018] Barthe, G., Grégoire, B., and Laporte, V. (2018). Secure compilation of side-channel countermeasures: The case of cryptographic "constant-time". In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 328–343.

[Barthe et al., 2006] Barthe, G., Rezk, T., and Naumann, D. A. (2006). Deriving an information flow checker and certifying compiler for java. In *2006 IEEE Symposium on Security and Privacy (S&P 2006), 21-24 May 2006, Berkeley, California, USA*, pages 230–242. IEEE Computer Society.

[Benton, 2004] Benton, N. (2004). Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, page 14–25, New York, NY, USA. Association for Computing Machinery.

[Bernstein, 2005] Bernstein, D. J. (2005). Cache-timing attacks on aes.

[Bernstein, 2006] Bernstein, D. J. (2006). Curve25519: new Diffie-Hellman speed records. In *PKC*, pages 207–228.

[Bernstein et al., 2008] Bernstein, D. J. et al. (2008). Chacha, a variant of salsa20. In *Workshop record of SASC*, volume 8, pages 3–5.

[Bernstein and Lange, 2009] Bernstein, D. J. and Lange, T. (2009). ebacs: Ecrypt benchmarking of cryptographic systems.

[Bertoni et al., 2013] Bertoni, G., Daemen, J., Peeters, M., and Assche, G. V. (2013). Keccak. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 313–314. Springer.

[Borrello et al., 2021] Borrello, P., D'Elia, D. C., Querzoni, L., and Giuffrida, C. (2021). Constantine: Automatic side-channel resistance using efficient control and data flow linearization. In Kim, Y., Kim, J., Vigna, G., and Shi, E., editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 715–733. ACM.

[Bos et al., 2018] Bos, J. W., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J. M., Schwabe, P., Seiler, G., and Stehlé, D. (2018). CRYSTALS – Kyber: A CCA-secure module-lattice-based KEM. In *IEEE EuroS&P*, pages 353–367.

[Bulck et al., 2018] Bulck, J. V., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T. F., Yarom, Y., and Strackx, R. (2018). Foreshadow: Extracting the keys to the intel SGX kingdom with transient Out-of-Order execution. In *27th USENIX Security Symposium (USENIX Security 18)*, page 991–1008, Baltimore, MD. USENIX Association.

[Canella et al., 2019a] Canella, C., Bulck, J. V., Schwarz, M., Lipp, M., von Berg, B., Ortner, P., Piessens, F., Evtyushkin, D., and Gruss, D. (2019a). A systematic evaluation of transient execution attacks and defenses. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 249–266, Santa Clara, CA. USENIX Association.

[Canella et al., 2019b] Canella, C., Genkin, D., Giner, L., Gruss, D., Lipp, M., Minkin, M., Moghimi, D., Piessens, F., Schwarz, M., Sunar, B., Van Bulck, J., and Yarom, Y.

(2019b). Fallout: Leaking data on meltdown-resistant cpus. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 769–784, New York, NY, USA. Association for Computing Machinery.

[Carbonneaux et al., 2014] Carbonneaux, Q., Hoffmann, J., Ramananandro, T., and Shao, Z. (2014). End-to-end verification of stack-space bounds for c programs. *SIGPLAN Not.*, 49(6):270–281.

[Carruth, 2020] Carruth, C. (2020). Cryptographic softare in a post-Spectre world. Talk at the Real World Crypto Symposium. https://chandlerc.blog/talks/2020_post_spectre_crypto/post_spectre_crypto.html#1.

[Cauligi et al., 2020] Cauligi, S., Disselkoen, C., Gleissenthall, K. v., Tullsen, D., Stefan, D., Rezk, T., and Barthe, G. (2020). Constant-time foundations for the new spectre era. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 913–926, New York, NY, USA. Association for Computing Machinery.

[Cauligi et al., 2021] Cauligi, S., Disselkoen, C., Moghimi, D., Barthe, G., and Stefan, D. (2021). Sok: Practical foundations for spectre defenses. *ArXiv*, abs/2105.05801.

[Cauligi et al., 2017] Cauligi, S., Soeller, G., Brown, F., Johannesmeyer, B., Huang, Y., Jhala, R., and Stefan, D. (2017). Fact: A flexible, constant-time programming language. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 69–76.

[Cauligi et al., 2019a] Cauligi, S., Soeller, G., Johannesmeyer, B., Brown, F., Wahby, R. S., Renner, J., Grégoire, B., Barthe, G., Jhala, R., and Stefan, D. (2019a). Fact: a DSL for timing-sensitive computation. In McKinley, K. S. and Fisher, K., editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 174–189. ACM.

[Cauligi et al., 2019b] Cauligi, S., Soeller, G., Johannesmeyer, B., Brown, F., Wahby, R. S., Renner, J., Grégoire, B., Barthe, G., Jhala, R., and Stefan, D. (2019b). Fact: a DSL for timing-sensitive computation. In McKinley, K. S. and Fisher, K., editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019.*, pages 174–189. ACM.

[Chandler Carruth, 2021] Chandler Carruth (2021). Speculative Load Hardening – A Spectre Variant #1 Mitigation Technique. https://llvm.org/docs/SpeculativeLoadHardening.html.

[Chattopadhyay and Roychoudhury, 2018] Chattopadhyay, S. and Roychoudhury, A. (2018). Symbolic verification of cache side-channel freedom. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 37(11):2812–2823.

[Chen et al., 2010] Chen, J., Chugh, R., and Swamy, N. (2010). Type-preserving compilation of end-to-end verification of security enforcement. In Zorn, B. G. and Aiken, A., editors, *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 412–423. ACM.

[Chen et al., 2017] Chen, J., Feng, Y., and Dillig, I. (2017). Precise detection of side-channel vulnerabilities using quantitative cartesian hoare logic. In Thuraisingham, B. M., Evans, D., Malkin, T., and Xu, D., editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 875–890. ACM.

[Clarkson and Schneider, 2008] Clarkson, M. R. and Schneider, F. B. (2008). Hyperproperties. In *2008 21st IEEE Computer Security Foundations Symposium*, pages 51–65.

[Cousot and Cousot, 1977] Cousot, P. and Cousot, R. (1977). Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, page 238–252, New York, NY, USA. Association for Computing Machinery.

[Crary and Weirich, 2000] Crary, K. and Weirich, S. (2000). Resource bound certification. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '00, page 184–198, New York, NY, USA. Association for Computing Machinery.

[Criterion Developers, 2019] Criterion Developers (2019). Statistics-driven benchmarking library for Rust. https://github.com/bheisler/criterion.rs.

[Daniel et al., 2020] Daniel, L., Bardin, S., and Rezk, T. (2020). Binsec/rel: Efficient relational symbolic execution for constant-time at binary-level. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1021–1038. IEEE.

[Daniel et al., 2021] Daniel, L.-A., Bardin, S., and Rezk, T. (2021). Hunting the haunter - efficient relational symbolic execution for spectre with haunted relse. In *Network and Distributed System Security Symposium*.

[Doychev et al., 2013] Doychev, G., Feld, D., Köpf, B., Mauborgne, L., and Reineke, J. (2013). Cacheaudit: A tool for the static analysis of cache side channels. In King, S. T., editor, *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 431–446. USENIX Association.

[Ge et al., 2018] Ge, Q., Yarom, Y., Cock, D. A., and Heiser, G. (2018). A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J. Cryptogr. Eng.*, 8(1):1–27.

[Genkin et al., 2016] Genkin, D., Shamir, A., and Tromer, E. (2016). Acoustic cryptanalysis. *Journal of Cryptology*, 30.

[Goldwasser et al., 1985] Goldwasser, S., Micali, S., and Rackoff, C. (1985). The knowledge complexity of interactive proof-systems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, page 291–304, New York, NY, USA. Association for Computing Machinery.

[Gómez-Londoño et al., 2020] Gómez-Londoño, A., Åman Pohjola, J., Syeda, H. T., Myreen, M. O., and Tan, Y. K. (2020). Do you have space for dessert? a verified space cost semantics for cakeml programs. *Proc. ACM Program. Lang.*, 4(OOPSLA).

[Gruss et al., 2016] Gruss, D., Maurice, C., Wagner, K., and Mangard, S. (2016). Flush+flush: A fast and stealthy cache attack. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*, DIMVA 2016, page 279–299, Berlin, Heidelberg. Springer-Verlag.

[Guarnieri et al., 2018] Guarnieri, M., Köpf, B., Morales, J., Reineke, J., and Sánchez, A. (2018). Spectector: Principled detection of speculative information flows.

[Intel, a] Intel. Deep dive: Indirect branch predictor barrier.

[Intel, b] Intel. Deep dive: Indirect branch restricted speculation.

[Intel, 2018] Intel (2018). Intel Analysis of Speculative Execution Side Channels. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/analysis-speculative-execution-side-channels.html.

[Jancar, 2021] Jancar, J. (2021). The state of tooling for verifying constant-timeness of cryptographic implementations.

[Jean-Philippe Aumasson, 2019] Jean-Philippe Aumasson (2019). Guidelines for Low-Level Cryptography Software. https://github.com/veorq/cryptocoding.

[Kaufmann et al., 2016] Kaufmann, T., Pelletier, H., Vaudenay, S., and Villegas, K. (2016). When constant-time source yields variable-time binary: Exploiting curve25519-donna built with msvc 2015. pages 573–582.

[Kocher et al., 2019a] Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. (2019a). Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19.

[Kocher et al., 2019b] Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. (2019b). Spectre attacks: Exploiting speculative execution. In *IEEE S&P*, pages 1–19.

[Kocher, 1996a] Kocher, P. C. (1996a). Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Koblitz, N., editor, *Advances in Cryptology — CRYPTO '96*, pages 104–113, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Kocher, 1996b] Kocher, P. C. (1996b). Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer.

[Koruyeh et al., 2018] Koruyeh, E. M., Khasawneh, K. N., Song, C., and Abu-Ghazaleh, N. (2018). Spectre returns! speculation attacks using the return stack buffer. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, Baltimore, MD. USENIX Association.

[Leroy, 2009] Leroy, X. (2009). Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115.

[Lipp et al., 2018] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. (2018). Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, Baltimore, MD. USENIX Association.

[Maisuradze and Rossow, 2018] Maisuradze, G. and Rossow, C. (2018). Ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 2109–2122, New York, NY, USA. Association for Computing Machinery.

[Namjoshi and Tabajara, 2020] Namjoshi, K. S. and Tabajara, L. M. (2020). Witnessing secure compilation. In Beyer, D. and Zufferey, D., editors, *Verification, Model Checking, and Abstract Interpretation - 21st International Conference, VMCAI 2020, New Orleans, LA, USA, January 16-21, 2020, Proceedings*, volume 11990 of *Lecture Notes in Computer Science*, pages 1–22. Springer.

[Openssl, 2013] Openssl (2013). OpenSSL: Cryptography and SSL/TLS Toolkit. https://www.openssl.org/.

[Osvik et al., 2006] Osvik, D. A., Shamir, A., and Tromer, E. (2006). Cache attacks and countermeasures: The case of aes. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, CT-RSA'06, page 1–20, Berlin, Heidelberg. Springer-Verlag.

[Paraskevopoulou and Appel, 2019] Paraskevopoulou, Z. and Appel, A. W. (2019). Closure conversion is safe for space. *Proc. ACM Program. Lang.*, 3(ICFP).

[Patrignani and Guarnieri, 2021] Patrignani, M. and Guarnieri, M. (2021). Exorcising spectres with secure compilers. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 445–461.

[Picek et al., 2021] Picek, S., Heuser, A., Perin, G., and Guilley, S. (2021). Profiled side-channel analysis in the efficient attacker framework. In *Smart Card Research and Advanced Applications: 20th International Conference, CARDIS 2021, Lübeck, Germany, November 11–12, 2021, Revised Selected Papers*, page 44–63, Berlin, Heidelberg. Springer-Verlag.

[Poletto and Sarkar, 1999] Poletto, M. and Sarkar, V. (1999). Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21(5):895–913.

[Rivest et al., 1978] Rivest, R. L., Shamir, A., and Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126.

[Rodrigues et al., 2016a] Rodrigues, B., Pereira, F. M. Q., and Aranha, D. F. (2016a). Sparse representation of implicit flows with applications to side-channel detection. In Zaks, A. and Hermenegildo, M. V., editors, *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pages 110–120. ACM.

[Rodrigues et al., 2016b] Rodrigues, B., Quintão Pereira, F. M., and Aranha, D. F. (2016b). Sparse representation of implicit flows with applications to side-channel detection. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, page 110–120, New York, NY, USA. Association for Computing Machinery.

[Sayakkara et al., 2019] Sayakkara, A., Le-Khac, N.-A., and Scanlon, M. (2019). A survey of electromagnetic side-channel attacks and discussion on their case-progressing potential for digital forensics. *Digital Investigation*, 29:43–54.

[Schwarz et al., 2019] Schwarz, M., Lipp, M., Moghimi, D., Van Bulck, J., Stecklina, J., Prescher, T., and Gruss, D. (2019). Zombieload: Cross-privilege-boundary data sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 753–768, New York, NY, USA. Association for Computing Machinery.

[Shanbhogue et al., 2019] Shanbhogue, V., Gupta, D., and Sahita, R. (2019). Security analysis of processor instruction set architecture for enforcing control-flow integrity. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '19, New York, NY, USA. Association for Computing Machinery.

[Shivakumar et al., pear] Shivakumar, B. A., Barnes, J., Barthe, G., Cauligi, S., Chuengsatiansup, C., Genkin, D., O'Connell, S., Schwabe, P., Sim, R. Q., and Yarom, Y. (2023 (to appear)). Spectre declassified: Reading from the right place at the wrong time. In *IEEE S&P*.

[Sison and Murray, 2019] Sison, R. and Murray, T. (2019). Verifying that a compiler preserves concurrent value-dependent information-flow security. In *International Conference on Interactive Theorem Proving*, Lecture Notes in Computer Science. Springer-Verlag.

[Sousa and Dillig, 2016] Sousa, M. and Dillig, I. (2016). Cartesian hoare logic for verifying k-safety properties. In Krintz, C. and Berger, E. D., editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 57–69. ACM.

[Tramèr et al., 2020] Tramèr, F., Boneh, D., and Paterson, K. (2020). Remote Side-Channel attacks on anonymous transactions. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2739–2756. USENIX Association.

[Tromer et al., 2010] Tromer, E., Osvik, D. A., and Shamir, A. (2010). Efficient cache attacks on aes, and countermeasures. *J. Cryptology*, 23:37–71.

[van Schaik et al., 2019] van Schaik, S., Milburn, A., Österlund, S., Frigo, P., Maisuradze, G., Razavi, K., Bos, H., and Giuffrida, C. (2019). Ridl: Rogue in-flight data load. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 88–105.

[Vassena et al., 2021a] Vassena, M., Disselkoen, C., Gleissenthall, K. v., Cauligi, S., Kıcı, R. G., Jhala, R., Tullsen, D., and Stefan, D. (2021a). Automatically eliminating speculative leaks from cryptographic code with blade. *Proc. ACM Program. Lang.*, 5(POPL).

[Vassena et al., 2021b] Vassena, M., Disselkoen, C., von Gleissenthall, K., Cauligi, S., Kici, R. G., Jhala, R., Tullsen, D. M., and Stefan, D. (2021b). Automatically eliminating speculative leaks from cryptographic code with blade. *Proc. ACM Program. Lang.*, 5(POPL):1–30.

[Volpano and Smith, 1997] Volpano, D. and Smith, G. (1997). A type-based approach to program security. In *TAPSOFT'97: Theory and Practice of Software Development: 7th International Joint Conference CAAP/FASE Lille, France, April 14–18, 1997 Proceedings 22*, pages 607–621. Springer.

[von Gleissenthall et al., 2019] von Gleissenthall, K., Kici, R. G., Stefan, D., and Jhala, R. (2019). IODINE: verifying constant-time execution of hardware. In Heninger, N. and Traynor, P., editors, *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 1411–1428. USENIX Association.

[von Gleissenthall et al., 2021] von Gleissenthall, K., Kici, R. G., Stefan, D., and Jhala, R. (2021). Solver-aided constant-time hardware verification. In Kim, Y., Kim, J., Vigna, G., and Shi, E., editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 429–444. ACM.

[Wang et al., 2017] Wang, S., Wang, P., Liu, X., Zhang, D., and Wu, D. (2017). Cached: Identifying cache-based timing channels in production software. In Kirda, E. and Ristenpart, T., editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 235–252. USENIX Association.

[Yarom and Falkner, 2014] Yarom, Y. and Falkner, K. (2014). FLUSH+RELOAD: A high resolution, low noise, l3 cache Side-Channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA. USENIX Association.

[Zagieboylo et al., 2019] Zagieboylo, D., Suh, G. E., and Myers, A. C. (2019). Using information flow to design an isa that controls timing channels. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, pages 272–27215.

[Zhang et al., 2012] Zhang, D., Askarov, A., and Myers, A. C. (2012). Language-based control and mitigation of timing channels. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, page 99–110, New York, NY, USA. Association for Computing Machinery.