KU LEUVEN

FACULTY OF
ENGINEERING SCIENCE

Compiler Support for Control-Flow Linearization Leveraging Hardware Defenses

Daan Vanoverloop

Thesis voorgedragen tot het behalen
van de graad van Master of Science
in de ingenieurswetenschappen:
computerwetenschappen, hoofdoptie
Gedistribueerde systemen

Promotoren:
Prof. dr. ir. F. Piessens
Dr. L.-A. Daniel

Evaluatoren:
H. Winderix
Dr. ir. L. Sion

Begeleiders:
H. Winderix
Dr. L.-A. Daniel

Academiejaar 2022 – 2023

Preface

This master's thesis allowed me to delve into the world of LLVM, a significant software project, while also enabling me to apply the theoretical knowledge and skills gained throughout my bachelor's and master's studies. I would like to express my gratitude to my daily advisors Hans and Lesly-Ann for their support and countless hours spent in invaluable meetings. Additionally, I extend my thanks to my supervisor, Prof. Piessens, for granting me this opportunity. My sincere gratitude also goes to my family for supporting me, and my friends for the numerous discussions, experiences and memorable adventures we have had together.

Daan Vanoverloop

Contents

Abstract

By observing microarchitectural state, attackers can observe the control flow of a program and derive information about secret values. Such leakage can be mitigated against by linearizing secret-dependent control flow. State-of-the-art compiler-assisted *control flow linearization* methods commonly make use of conditional execution, relying on selection primitives found within an existing instruction set architecture (ISA) that were not designed with control flow linearization in mind.

Related work of Winderix et al. proposes a novel hardware mechanism and ISA extension for *RISC-V* called *architectural mimicry (AMi)* that provides primitives designed to accelerate side-channel resistant programs by enabling efficient control flow linearization and balancing. This thesis expands upon their proposed programming model to enable linearization of *reducible control flow* without the need for costly structurization and provides systematic methods to linearize secret-dependent branches. We address the challenges involved in ensuring the correctness and security of the linearized program.

We formally show that these methods preserve the semantics of the program, and provide a practical implementation in the *LLVM* compiler. Experimental evaluation of this implementation demonstrates that our automatic linearization adds only 6% overhead to both code size and execution time when compared to the manual linearization efforts of Winderix et al. (AMi), still significantly lower than state-of-the-art control flow linearization that makes use of conditional execution.

Samenvatting

Door het observeren van microarchitecturale toestand kunnen aanvallers de controlestroom van een programma waarnemen en informatie afleiden over geheime waarden. Deze lekkage kan worden verminderd door controlestroom die afhankelijk is van geheime waarden te lineariseren. State-of-the-art compiler-ondersteunde methoden voor *controlestroomlinearisatie* maken vaak gebruik van conditionele uitvoering, waarbij ze vertrouwen op selectieprimitieven die te vinden zijn in bestaande instructiesetarchitecturen (ISA's) die niet zijn ontworpen met controlestroomlinearisatie in gedachten.

Het verwante werk van Winderix et al. stelt een nieuw hardwaremechanisme en ISA-uitbreiding voor *RISC-V* genaamd *architectural mimicry (AMi)* voor die primitieven biedt ontworpen om side-channel resistente programma's te versnellen door efficiënte controlestroomlinearisatie en -balancering mogelijk te maken. Deze masterproef breidt hun voorgestelde programmeermodel uit om linearisatie van *reduceerbare controlestroom* mogelijk te maken zonder de noodzaak van kostbare structurering en biedt systematische methoden om controlestroom te lineariseren die afhankelijk is van geheime waarden. We gaan in op de uitdagingen bij het waarborgen van de correctheid en beveiliging van het gelineariseerde programma.

We tonen formeel aan dat deze methoden de semantiek van het programma behouden en bieden een praktische implementatie in de *LLVM*-compiler. Experimentele evaluatie van deze implementatie toont aan dat onze automatische linearisatie slechts 6% overhead toevoegt aan zowel codegrootte als uitvoeringstijd in vergelijking met de handmatige linearisatie-inspanningen van Winderix et al. (AMi), nog steeds aanzienlijk lager dan state-of-the-art controlestroomlinearisatie die gebruikmaakt van conditionele uitvoering.

Chapter 1

Introduction

Throughout the last decade, timing and microarchitectural attacks have exploited the physical side effects of processors. This includes differences in execution time, but also microarchitectural state such as caches. This is especially devastating in shared environments in which an attacker can closely monitor microarchitectural state [18]. One example of such shared systems are Infrastructure-as-a-Service (IaaS) cloud computing platforms where multiple tenants are supported using virtual machines (VM).

Although virtualization provides isolation of the architectural state, secret data can still leak on side channels, such as cache state, for which hardware is shared between an attacker and his victim. Practical attacks on micro-architectural state have been demonstrated by numerous research [31, 38, 49], which can lead to recovery of full cryptographic keys [10, 26]. A common countermeasure against such attacks is constant-time programming, in which the programmer ensures that the leakage trace does not depend on secrets.

1.1 Context

Constant-Time Programming Under constant-time programming model, code is written such that control flow and memory accesses do not depend on secrets. Memory accesses should not depend on secrets, since this would leak secrets through, for example, cache-timing based side channels. This programming model is the de facto standard to protect against microarchitectural attacks and commonly applied in cryptographic libraries [6, 9].

Control flow linearization techniques are applied to eliminate secret-dependent branches, replacing them with equivalent constant-time code. For example, by executing all paths of a secret-dependent branch in sequence, but only retaining the results of one path. Currently, constant-time programming in cryptographic code is generally a manual effort, and mainstream compilers provide no support or guarantees for this programming model.

Compiler-Assisted Control Flow Linearization Numerous research explores automatic control-flow linearization, in which programs are transformed into their constant-time equivalent by eliminating secret-dependent branches [33, 12, 48, 15, 43]. In linearized code, all instructions are executed regardless of secret branch conditions, and those conditions are instead used to select results in constant-time. However, this results into significant binary size and runtime overhead, as it relies on conditional execution [46].

Architectural Mimicry To accelerate programs hardened against side-channel leakage, Winderix et al. [46] introduce a novel hardware mechanism called mimic execution and instruction set architecture (ISA) support for RISC-V called architectural mimicry (AMi), which aims to replace these conditional execution primitives while still providing the means the balance or linearize code. Mimic execution is an execution mode in which instructions are executed only for their microarchitectural effects, while suppressing most architectural effects. When combined with the ability to conditionally enable mimicry mode for the duration of a branch, control flow can be linearized with very low overhead. Unfortunately, there is no full compiler support to automatically balance or linearize programs making use of AMi. Hence, hardening programs with AMi is a manual effort.

Secure Compilation Compilation of sensitive code, such as code found in cryptographic libraries, poses several challenges. When writing such code, cryptographers rely on implicit invariants or assumptions about compilers, such as timing or memory access patterns. However, such invariants cannot be formally expressed today in compilers, making it hard control side effects in modern C compilers [42]. Compiler optimizations can break such implicit invariants by introducing secret-dependent control flow or memory access. For example, a conditional selection may be transformed into a branch to improve performance, Simon et al. [42] have proposed extensions to existing mainstream compilers to support primitives such as constant-time selection to make such implicit invariants explicit in the LLVM compiler.

A promising solution to this problem is secure compilation, which is the compilations of programs while preserving security properties, such as constant-timeness. Methods for secure compilation or side channel countermeasures have been proposed by Barthe et al. [9], and practical implementations exist, such as CompCert [28] and Jasmin [5]. Although initial implementations such as [28] only provided formal verification of correctness and safety properties, more recent research such as [5] also proves constant-time security. However, this does not apply to mainstream compilers such as GCC and LLVM.

1.2 Problem and Challenges

We adopt the same scope as AMi, and assume an adversary may be able to run software on the same physical core as the victim. Furthermore, we assume the adversary has access to the source code of the vulnerable program. The adversary

aims to gain information about a secret that is leaked on software-based timing side channels. Other types of side channels, such as power based channels [29], or side channels based on transient execution [14] are outside the scope of this thesis.

AMi [46] can be applied to linearize secret-dependent branches, but their work does not include full compiler support, making it difficult to apply in practice, as assembly code has to be hardened manually. An assembler and disassembler is implemented in LLVM by the authors of [46], but there is no support for automatic hardening. Furthermore, the programming model they propose for linearizing control flow can only be applied to a limited set of programs that adhere to a certain structure. Although they provide definitions for correctness and security as a way to guide developers when to use AMi instructions, they do not state any concrete rules.

The goal of this thesis is to extend LLVM, a widely used compiler toolchain, to support automatic control-flow linearization using AMi. This presents several challenges. Firstly, a practical algorithm is needed to linearize control flow without changing the semantics of the program. Secondly, we need a concrete implementation in the LLVM compiler. This poses several challenges, since secret-dependent branches need to be identified in the compiler backend and control flow linearization algorithms must be implemented in a compilation pipeline where other compiler transformation may not preserve the desired properties. Secrets are annotated in the source program, to avoid unnecessary linearization of all branches. Lastly, the resulting program must be both correct and secure. However, this results in additional challenges as some AMi instructions may break the correctness of a program by overwriting registers that should not be overwritten in mimicry mode.

1.3 Proposal

This thesis proposes both fundamental and practical compiler support for AMi. We propose an implementation to propagate annotations from clang (a C compiler frontend of LLVM) to the RISC-V backend, and implement static taint analysis in this backend to identify secret-dependent branches. This thesis aims to relax the hypotheses, as well as the well-behavedness proposition introduced by [46], with the goal of enabling a more general programming models that support reducible control flow.

Several practical algorithms for linearization are proposed. We first propose to structurize the control-flow graph (CFG) to a triangle-structured CFG, such that the linearization pattern of [46] can be applied. Additionally, we provide a method that can be applied to structured control flow by making use of activating jump instructions. Lastly, we generalize these methods to reducible control flow by adapting the partial control-flow linearization (PCFL) algorithm proposed by Moll and Hack [32] and applied for security by Soares et al. [43]

To ensure that the resulting program is secure, we ensure that branch conditions and accessed memory addresses are computed persistently. Furthermore, we nullify

side effects of store operations using ghost loads as proposed by [46], and we constrain register allocation to maintain the correctness of the program.

1.4 Contributions

The contributions of this thesis are summarized as follows.

- We generalize the programming model for linearization proposed by Winderix et al. [46], and broaden its applicability, such that AMi can be used to linearize reducible control flow without an additional cost from structurization.
- We address challenges when ensuring that the resulting program is correct and secure, by using persistent computation of memory addresses, nullification of persistent side effects and register allocation constraints to maintain correctness.
- We propagate annotations to the RISC-V backend of LLVM where we implement static taint analysis to identify secret-dependent branches.
- We formalize and implement three linearization methods based on AMi and other research [12, 48, 32, 43] in the RISC-V backend of LLVM.
- We formally show that these methods preserve the high-level semantics of the program under mild conditions, such as the assumption that loop bounds cannot be secret-dependent.
- We evaluate and compare the execution time and code size overhead of different linearization methods. Results show that all three linearization methods yield similar results for most control flow graphs. When comparing our results with the manual efforts of Winderix et al. [46], we observe a small overhead of 6% for both code size and execution time, which still greatly outperforms state-of-the-art automatic control flow linearization methods.
- We make our modifications to the LLVM compiler available under a permissive license to foster new research[1].

[1] `https://github.com/Danacus/llvm-project-ami-cfl`

1.5 Outline

This section provides an outline of the structure of this thesis.

- **Chapter 2 Background** provides background information about microarchitectural attacks, gives an overview of compiler-assisted hardening against side channel leakage, and introduces relevant AMi and compilation concepts that will be used throughout this thesis.
- **Chapter 3 Related Work** discusses related work about structurization and linearization.
- **Chapter 4 System and Leakage Model** introduces the language AMiL, a formal abstraction of an ISA, as the system model, along with the leakage model.
- **Chapter 5 Linearization with Architectural Mimicry** explains different methods to linearize control flow using AMi, along with the conditions under which such linearization is correct and secure.
- **Chapter 6 Implementation in LLVM** elaborates on the implementation of these linearization methods in the RISC-V compiler backend of LLVM, in addition to an explanation of how information about the secrecy of data can be provided to this backend.
- **Chapter 7 Evaluation** discusses the results of benchmarks by comparing the execution time and code size overhead of different linearization methods.
- **Chapter 8 Conclusion** concludes this thesis by summarizing the contributions and elaborating on its limitations. Furthermore, we briefly discuss related work and explore future directions.

Chapter 2

Background

2.1 Microarchitectural Attacks

The last decade has seen an uprise of *infrastructure-as-a-service (IaaS) cloud computing*, which supports running multiple *virtual machines (VM)* on the same physical machines. Significant progress has been made to achieve strong software isolation of VMs, which includes confidentiality, integrity and availability of VMs that share the same hardware [18]. One such example is the se4L microkernel, which has formal proofs of such isolation properties [25, 24]. However, such methods only achieve complete *architectural* isolation, and they do not mitigate against *side channels*. Microarchitectural components are shared between different stakeholders that execute code on the same hardware. This has led to new exploits that allow attackers to leak confidential information through timing-based side and covert channels [18, 44].

One example of such side-channel is leakage through control flow. The outcome of a conditional branch can be observed by an adversary by measuring for instance the execution time of the branch, the instruction cache or the predictor state. Another example is leakage through cache state, which is shared between the victim and the adversary. Consider a memory access, such as a load operation where the address is secret-dependent. If an attacker can measure which cache lines were touched, information about the secret that was used to determine the memory location will leak to the adversary [18]. These are traditional, *software-based* side channel attacks, in contrast to physical side channels, such as power-based side channels. Leakage through control flow can be mitigated against using software hardening, such as *code balancing* [47] and *linearization* [12, 48, 15, 33, 43].

More recently, different kinds of microarchitectural attacks have been discovered, such as Spectre [26] and Meltdown [30], which exploit *speculative execution*, where the processor executes instructions speculatively based on predictions about the most likely path of execution. These attacks rely on the fact that while architectural effects of speculative execution are discarded, microarchitectural effects, such as the cache state, remain. Even when a correct and secure non-speculative program does not leak secret information through architectural or microarchitectural state, speculative execution may leak this information through microarchitectural state.

For example, a misprediction of a branch that guards access to an array to prevent out-of-bounds access may still cause access to out-of-bounds data during speculative execution. While the architectural state remains unaffected, secret information may leak through attacker-observable cache state [14]. Such attacks are outside the scope of this thesis since they cannot be mitigated using software hardening.

2.1.1 Program Hardening

The *constant-time programming* model is a common method to mitigate software-based side channel attacks. A program is constant-time if two executions whose inputs only differ in secret information leak the same observations to an attacker [6]. The program is assumed to not contain secret-dependent memory accesses or variable-time instructions with secret operands.

Constant-time programming is often applied to cryptographic libraries, such as OpenSSL, BearSSL, libsodium [10] and HACL* [50], to mitigate timing attacks like Lucky 13 [4]. To achieve this security goal, all secret-dependent branches must be eliminated by *linearizing* the program. A linearized program always executes the same sequence of instructions in the same order, such that the same control flow trace is leaked regardless of the inputs.

Existing research has proposed several methods to linearize programs [12, 48, 15, 33, 43], which commonly rely on conditional execution primitives, such as a conditional move, to ensure that the linearized program outputs the same results as the original program, without using secret-dependent branches. This is known as *control-flow linearization*, which linearizes control flow to meet the *PC-Security* requirement as defined by Molnar et al. [33]. PC-Security implies that for each set of secret inputs, the sequence of program counter values observed by an attacker must be identical.

Borrello et al. [12] also implement a different kind of linearization called *data flow linearization*, in which memory accesses are linearized by accessing all locations the original program could reference. Other research implements similar methods to mitigate such side channels [48, 36], but data flow linearization is outside the scope of this thesis. A related method to harden against side-channel leakage is *balancing* [2]. However, code balancing does not adhere to the constant-time policy [47].

Software hardening can be effectively automated through various techniques such as automatic code balancing and linearization, which have been proposed in recent research papers such as [47, 12, 48, 43]. These techniques are commonly implemented through compilation passes and will be further explained in the next section.

2.2 Secure Compilation and Automatic Hardening

2.2.1 Compilation

A *compiler* is a program that translates code written in one language, called the *source* language, to another language, called the *target* language. Commonly, the source language is a *high-level* programming language used by a programmer, which

is independent of the target machine that executes code. Examples of source languages include C, C++ and Rust. The target language is a *low-level* programming language that can be used to create an executable program for a specific target machine. Many modern compilers, such as GCC and LLVM make use of an *Intermediate representation (IR)*, on which several analysis and transformation *passes* run to optimize code and to lower target-independent abstractions to target-specific code.

A frequently used pattern in compiler design is to assign different passes to one of three stages: the frontend, the middle end, and the backend. On the one hand, the frontend verifies programs written in the source language, parses the program into a syntax tree, and emits appropriate code in the intermediate representation. On the other hand, the middle end applies target-independent optimization passes to the intermediate representation. Finally, the backend lowers the intermediate representation into target-specific machine code that can be executed on the target architecture.

This thesis focuses on the compiler backend, in which target-specific hardening passes are implemented for the RISC-V *instruction set architecture (ISA)*. The goal of these passes is to mitigate control-flow leakage through automatic program linearization.

2.2.2 Taint Analysis

Program hardening to mitigate side channels usually comes with an additional cost on the performance of the resulting code. For example, linearized code takes significantly longer to execute when compared to its branched counterpart, since code from all branches will always be executed. In practice, only leakage of confidential information must be avoided. Information that is already public to the adversary may be leaked on side-channels without compromising security.

As a result, it is desirable to minimize the impact on performance, by only mitigating leakage of secret information. However, it is not always obvious which information can be used by an attacker to derive information about a secret. To mitigate leakage through control flow, secret-dependent branches must be linearized. Even if a branch condition has not been marked as secret explicitly, its value may be influenced by a secret. Hence, leakage of the branch condition may also leak information about a secret.

Taint analysis can be used to track the flow of information throughout a program to determine all variables that depend on secret information. Some research uses *dynamic taint analysis*, where instrumented machine code or intermediate representation is profiled to determine secret-dependent variables [12]. Other research makes use of *static analysis* to determine the flow of secret information, by propagating taint by following data and control dependencies [48]. In both cases, taint flows from an initial set of manually annotated secrets.

2.2.3 Contract-Aware Secure Compilation

Controlling side-channel leakage in the constant-time programming remains challenging in practice, due to compilers failing to understand the intentions of the programmer and optimizing away linearized code, opening new side channels as a result [42]. For example, a smart compiler may replace uses of conditional selection with branches to improve performance, such as the `X86CmovConversion` pass in the x86 backend of LLVM. Simon et al. [42] propose an implementation of a constant-time selection primitive in LLVM to improve constant-timeness guarantees. Furthermore, they propose that compilers should explicitly support controls for implicit properties such as execution time.

Guarnieri and Patrignani [19] say that programmers should be able to rely on compilers to produce secure code, which would enable decoupling between program-level security (leakage under ISA semantics), and microarchitectural security. The former is the job of the programmer, while the latter should be enforced by the compiler. They propose the idea of *Contract-Aware Secure COmpilation (CASCO)* to achieve this. Such a compiler would respect a *hardware/software security contract*. A security contract is an abstraction of the processor's security guarantees that can be leveraged by compilers to preserve security properties.

2.3 Control Flow Graphs

One important concept used during compilation is *control flow*, which describes the order in which instructions can be executed. Depending on the semantics of the instruction at a certain location, execution may continue at different locations. Control flow describes how the program counter can change during the evaluation of an instruction in a certain configuration. This section introduces the concepts necessary to describe control flow.

2.3.1 Successor Relation and Basic Blocks

We assume a total order on the locations in the set Loc, such that Loc consists of locations $l_0, l_1, \ldots, l_n$ for which $l_0 \leq l_1 \leq \cdots \leq l_n$. This order is inherited from the set of natural numbers, since $Loc \subseteq \mathbb{N}$. The *successor* relation $\rightarrow$ is a relation over the set Loc, where $l_x \rightarrow l_y$ if and only if execution may continue from l_x to l_y. Furthermore, l_x is called a *predecessor* of l_y.

For instructions **add**, **mul**, **load**, **store**, and activating branches at location l, l is succeeded only by $l+1$ ($l \rightarrow l+1$, and there is no other l' such that $l \rightarrow l'$). Branch, jump, and call instructions have different successors. A **jmp** or **call** instruction at location l can only be succeeded by its target location l'. Similarly, branch instructions (**beqz** and **bnez**) at l can be succeeded by their target location l'. Additionally, they can be succeeded by $l+1$, in case the branch condition evaluates to false. There is a *path* from location l_x to l_y if there is a sequence of locations such that $l_x \rightarrow \ldots \rightarrow l_y$.

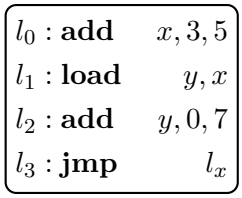


FIGURE 2.1: Basic block with instructions and a single terminator at l_3

In order to simplify reasoning about control flow, instructions can be grouped into *basic blocks*. A basic block is a pair (I, T) where $I, T \subseteq Loc$ such that I and T consist of subsequent locations. Suppose that I consists of $i_0, i_1, \ldots, i_n$ and T consists of $t_0, t_1, \ldots, t_m$, then $i_1 = i_0+1, i_2 = i_1+1, \ldots i_n = i_{n-1}+1, t_0 = i_n+1, t_1 = t_0 + 1, \ldots, t_m = t_{m-1} + 1$. Furthermore, $i_0, \ldots i_n$ each only have a single successor, and $i_1, \ldots, i_n, t_0, \ldots t_m$ only have a single predecessor. Only call instructions are allowed to have a successor that does not belong to the same basic block under the assumption that they return to the location that follows directly after the call instruction.

Therefore, a basic block contains a linear sequence of instructions I that ends with a sequence of *terminators* T that may have multiple successors. Such terminators are usually branch or jump instructions where their target location is always the entry location of another basic block. Under these assumptions, a successor relation can also be defined on the set of basic blocks as follows: $b_x \to b_y$ if and only if there is some terminator in b_x that is succeeded by the entry of b_y.

2.3.2 Control Flow Graph

The collection of basic blocks with instructions and their successor relation can be represented using a control-flow graph (CFG). We assume that a CFG has a unique entry and a unique exit. In this thesis, a graphical representation is used to describe CFGs, which will be introduced in this section. A sequence of instructions forms a basic block, which is illustrated in Figure 2.1.

The successor relation between basic blocks is represented using control-flow edges in the CFG, as illustrated in Figure 2.2. The terminator branch at l_0 has two successors, hence its block has two successor blocks. Similarly, the jump instruction at l_2 has a single successor and as a result, its basic block has a single successor. There are no branch or jump instructions in the block at location l_3. In this case, the last instruction of that block, namely the **add** instruction at l_3 can be treated as a terminator with a single successor, namely l_4. As a result, this block has a single successor block. This is called a *fallthrough*, since control flow falls through to the next block without encountering a terminator.

A graphical representation of a CFG can also be used to describe abstract control flow features without defining a concrete CFG. One example of this is given in Figure 2.3, in which there is a path from l_3 to l_x. Such path may pass through

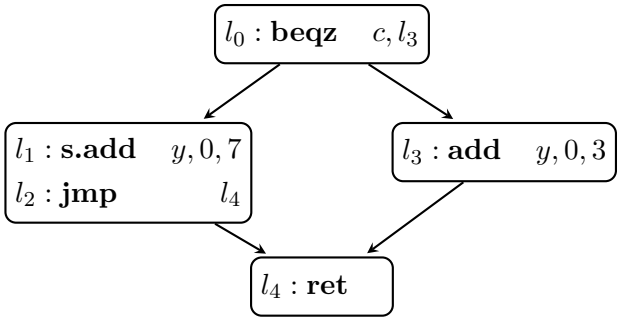


FIGURE 2.2: An example of a CFG with control-flow edges (solid line).

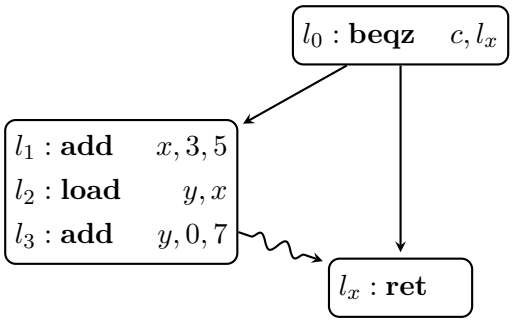


FIGURE 2.3: Example description of control flow with a path from l_3 to l_x.

several other basic blocks, but the target must be reachable through a sequence of control-flow edges.

2.3.3 Structured Control Flow

In this section we define both *structured* and *triangle-structured* control flow, based on the formalization of Sabne et al. [40]. First, the dominance and post-dominance relations are defined for both locations, edges and basic blocks. A location (edge, block) l *dominates* l' if each path from the CFG entry to l' contains l. Conversely, a location (edge, block) l *post-dominates* l' if each path from l' to the CFG exit contains l. These control flow properties are illustrated in Figure 2.4.

Both the dominance and post-dominance relations can be captured by a (post-)dominance tree in which the parent of a block in this tree (post-)dominates that block. The parent of a block in the (post-)dominance tree is called the *immediate post-dominator*. Furthermore, regions and single-entry single-exit (SESE) regions are defined. A *region* $[l, l']$ is the set of all locations (blocks) and edges present on any path from l to l'. Such region is a *single-entry single-exit (SESE) region* $[l, l']$ if the following holds:

- l dominates l'

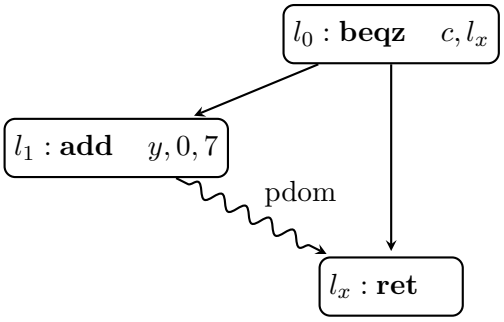


FIGURE 2.4: Example description of control flow with a post-dominance relation between l_1 to l_x.

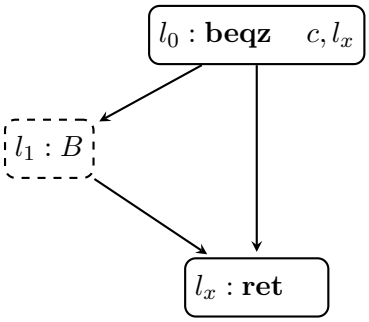


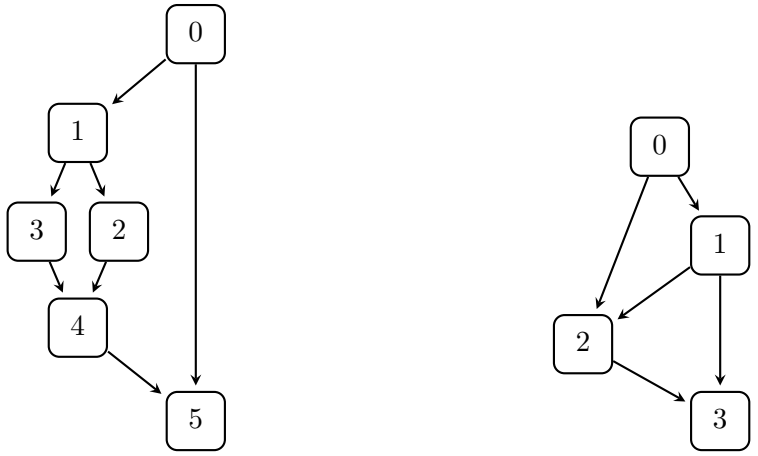
FIGURE 2.5: Example description of control flow with a region $B = [l_1, l_x]$.

- l' post-dominates l
- every cycle containing l also contains l' and vice versa

A region is represented by a node with a dashed border, as shown in Figure 2.5. Location l is the *entry* of region $[l, l']$, l' is the *exit*, and any location within the region that precedes the exit is called an *exiting* location. Similarly, a location that precedes the entry is called an *entering* location.

The notion of structured control flow has been formalized by Sabne et al [40], where the maximum in-degree and out-degree of each node is two. Any CFG with nodes of a higher in- or out-degree can trivially be converted into a CFG with maximum in- and out-degree of two. Hence, we assume that locations have at most two successors and at most two predecessors.

We adopt their definition of a structured selection condition node by defining a *structured branch* as a location for which each path from a successor of that location to its immediate post-dominator, the region between the first and last location of that edge is a SESE region. We refer to these regions as *branch regions*. Furthermore, we introduce the notion of *triangle-structured* control flow as a subset of structured control flow where a triangle-structured branch is a structured branch with its immediate post-dominator as successor.



(a) Example of structured control flow (b) Example of non-structured control flow

FIGURE 2.6: Examples of structured and non-structured control flow

Example 1. *In Figure 2.6a, block* 1 *is structured, since each path from* 1 *to its immediate post-dominator* 4 *(*$1 \to 2 \to 4$ *and* $1 \to 3 \to 4$*) spans an SESE region between its first and last edge. Block* 0 *is triangle-structured, because it is structured, and its immediate post-dominator* 5 *is also a successor of* 0*.*

Example 2. *In Figure 2.6b, block* 0 *is not structured, because for the path* $0 \to 2 \to 3$ *to its immediate post-dominator* 3*, the region between the first and last edge of this path is not SESE. Similarly, block* 1 *is not structured.*

2.3.4 Reducible Control Flow

A more common requirement for control flow structure is reducibility [21]. A CFG is *reducible* if the control-flow edges can be partitioned into *forward edges* and *back edges*. The forward edges form a directed acyclic graph (DAG) in which all locations are reachable from the entry, and for all back edges $l_a \to l_b$, l_b dominates l_a.

Each back edge of a reducible CFG unambiguously defines a *(natural) loop* [20]. Such loop is single-entry, all paths to a location within the loop must pass through the *(loop) header* (the header dominates all locations in the loop). Furthermore, we assume that all back edges enter the header of their loop [3] and that all locations in the loop are *strongly-connected*, meaning that each location must be reachable from all other locations.

We also define the following terminology that is used in the LLVM [27] compiler [1]. An *entering* block is a non-loop block that enters the header of a loop, and if

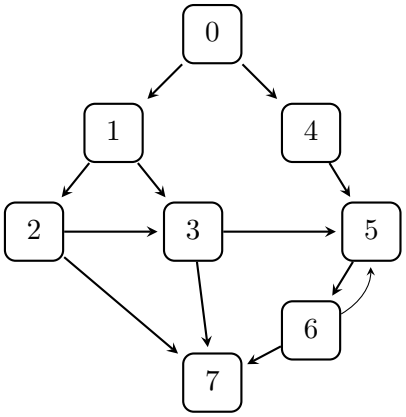


FIGURE 2.7: Example of a reducible CFG. 0 is the entry of the CFG and 7 is the exit. $6 \to 5$ is the only back edge, which defines a loop with blocks 5 and 6. 5 is the header of this loop, and 6 is both a latch and an exiting block.

this is the only edge entering the header. A *latch* is a block that has a back edge to the header, and an *exiting edge* leaves the loop from an *exiting* block to an *exit*.

> **Example 3.** *The CFG in Figure 2.6b is reducible, because the edges form a DAG. Another example of a reducible CFG is given in Figure 2.7. It is reducible because the target* 5 *of back edge* $6 \to 5$ *dominates* 6*, and all remaining edges form a DAG.*

2.4 Liveness Analysis

In order to allocate physical registers to virtual registers, *liveness analysis* is used to determine whether a register holds a value that may be needed in the future. Moreover, liveness analysis is used to determine data and control dependencies. This section discusses how the liveness of a register can be determined.

2.4.1 Def-Use Relation

Each instruction may *define* and *use* some registers. For example, the **add** x, a, b instruction uses a and b to compute $a + b$, and defines x by assigning the result to x. Such use-def relations can be defined using inference rules similar to [35], as shown in Figure 2.8, along with the successor relation. Note that in these rules, we assume that each expression is a register and not a value, since def and use are only defined on registers. These relations induce a chain on defs and uses. For example, a and b may be used by an **add** instruction to compute $x = a + b$, which defines x. Then x may be used later in the program by another instruction that might define yet another register.

$$\frac{l : \textbf{add}\ x, y, z}{\begin{array}{ll}\text{def}(l, x) & \text{use}(l, y) \\ \text{use}(l, z) & l \to l + 1\end{array}} \qquad \frac{l : \textbf{mul}\ x, y, z}{\begin{array}{ll}\text{def}(l, x) & \text{use}(l, y) \\ \text{use}(l, z) & l \to l + 1\end{array}} \qquad \frac{l : \textbf{call}\ l'}{l \to l'} \qquad \frac{l : \textbf{jmp}\ x}{\begin{array}{l}\text{use}(l, x) \\ l \to [\![x]\!]_r\end{array}}$$

$$\frac{l : \textbf{beqz}\ c, l'}{\begin{array}{c}\text{use}(l, c) \\ l \to l + 1 \\ l \to l'\end{array}} \qquad \frac{l : \textbf{bnez}\ c, l'}{\begin{array}{c}\text{use}(l, c) \\ l \to l + 1 \\ l \to l'\end{array}} \qquad \frac{l : \textbf{load}\ x, a}{\begin{array}{c}\text{use}(l, x) \\ \text{use}(l, a) \\ l \to l + 1\end{array}} \qquad \frac{l : \textbf{store}\ x, a}{\begin{array}{c}\text{use}(l, x) \\ \text{use}(l, a) \\ l \to l + 1\end{array}}$$

FIGURE 2.8: Inference rules for def, use and the successor relation

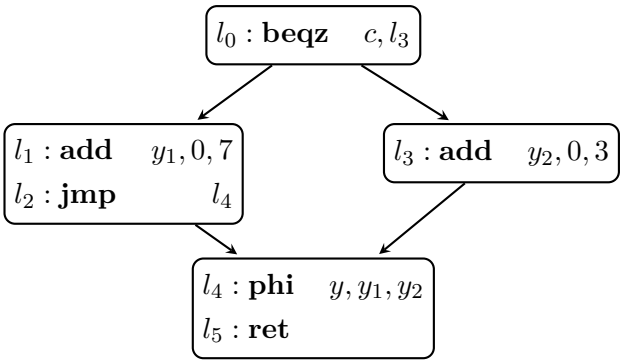


FIGURE 2.9: Example CFG in SSA

2.4.2 Static Single Assignment

Most compiler passes assume that the intermediate representation is in *static single assignment (SSA)* form [39]. This form provides a compact def-use graph that simplifies many dataflow optimizations while avoiding expensive dataflow analysis [27]. In this form, each variable or register can only be defined once in the entire program.

To enable more complex control flow in this form, "join" or ϕ-nodes (phi) can be used to choose a variable based on which predecessor block was executed before. For example, consider Figure 2.2. This program is not in SSA, since y is defined both at l_1 and at l_3. Figure 2.9 shows how this program can be written in SSA-form, by using a **phi** (ϕ) instruction, which chooses between y_1 and y_2 depending on the block that was executed before, and assigns the result to y.

2.4.3 Liveness

Liveness analysis is required for register allocation, since two variables that are in use cannot be assigned to the same physical register. A variable is *live* at a certain location if it holds a value that may be needed in the future [8]. Pfenning and

Platzer [35] define inference rules to derive liveness based on the defs and uses of instructions, and the control flow of the program. These rules are given below.

$$\frac{\text{use}(l, x)}{\text{live}(l, x)} \qquad \frac{\text{live}(l', u) \quad \text{succ}(l, l') \quad \neg\text{def}(l, u)}{\text{live}(l, u)}$$

These rules derive liveness by working backwards starting from the uses of variables. If a variable is used at a certain location, it must be live (rule 1). The variable is also live at any predecessor, unless it is defined at that location, in which case the backwards propagation of liveness stops (rule 2). Furthermore, we say that a variable is a *live-in* of a basic block if the variable is live at the entry of the block, and a variable is a *live-out* of a basic block if the variable is live at the entry of a successor block.

2.4.4 Data and Control Dependencies

Data and control dependencies have been briefly discussed in Section 2.2.2 about static taint analysis. In this section, we formalize those dependencies for AMiL programs. We say that two instructions at l and l' respectively are *data dependent* if there is a register x defined at l and used at l', and there is a path from l to l' on which x remains live. Two locations l and l' are *control dependent* if there is a path from l to l' such that each location on the path after l is post-dominated by l', but l' does not post-dominate l [16].

> **Example 4.** *An example of a data-dependency is the def-use relation in* `x = a + b`*, where* x *is marked as secret if* a *or* b *is secret. An example of a control-dependency is* `if (a == 0) { x = 2 } else { x = 5 }` *where* x *is secret-dependent if* a *is secret.*

Both control and data dependencies are transitive, hence if l_3 is dependent on l_2 and l_2 is dependent on l_1, l_3 is dependent on l_1. In the case of taint analysis, both types of dependencies are considered. We say that l' is secret if there is some secret l such that (l, l') is part of the transitive closure over all data and control dependencies.

2.5 Common Compiler Transformations

2.5.1 Transformation

Numerous compiler passes will apply transformations to the control flow graph. Such transformations may add, remove or change instructions, which may also change the successor relation in case branch instructions are modified. Register allocation is an example of such transformation.

A function takes a set of physical registers as inputs, and a set of physical registers as outputs. Furthermore, changes may be committed to the memory. A transformation should preserve program semantics. However, not all parts of the architectural configuration are considered "live". For example, a transformation may introduce spills of a register on the stack, meaning that the transformed program would write to memory locations that are left untouched by the original program. This should still be considered correct, since these memory locations are no longer in use at the end of the function.

Similarly, a transformation such as register allocation may cause the program to use different registers. However, since these registers might no longer live at the end of the function if they are not output registers, hence this transformation should still be considered correct.

2.5.2 Register Allocation

A *register allocation* is a function alloc : $\text{Regs}_v \rightarrow \text{Regs}_p$, it maps variables or virtual registers to physical registers. We say that such allocation is *valid* if and only if for each $v_1, v_2 \in \text{dom}(\text{alloc})$ where $v_1 \neq v_2$, $\text{alloc}(v_1) = \text{alloc}(v_2)$ implies that there is no location l at which both v_1 or v_2 are live. Intuitively, this means that a valid allocation will never assign two variables that are in use at the same time to the same physical register.

To compute a valid allocation, an *interference graph* can be constructed based on the liveness information [8]. An interference graph contains edges (v_1, v_2) if there is a location at which both v_1 and v_2 are live. Hence, it describes which variables cannot be assigned to the same physical register. Using this graph, finding a valid allocation can be reduced to coloring of the interference graph such that no two neighboring nodes are given the same color. Such graph coloring solution will always impose a valid allocation.

Since the number of physical registers is limited, it is desirable to use as little registers as possible in an allocation, to avoid having to spill registers on the stack. *Register pressure* is the number of physical registers needed for a valid allocation, and it can be used to describe the impact of a certain program on the number of needed registers. One of the goals of this thesis is to reduce register pressure when using AMi to linearize programs compared to state-of-the-art methods.

2.6 Architectural Mimicry

Architectural mimicry (AMi) is a novel approach that extends hardware to facilitate software-based hardening against side-channel attacks introduced by Winderix et al [46]. They propose a new hardware mechanism, called *mimic execution*, an execution mode in which instructions are executed only for their attacker observable effects, while most architectural effects are suppressed. Additionally, they propose programming models for code balancing and linearization, and implement AMi as a RISC-V ISA extension.

2.6.1 Qualified Instructions

In AMi, instructions are paired with a *qualifier*, which defines the behavior of instructions in both the standard execution mode and mimicry mode. The mimic qualifier **m** causes an instruction to always be mimicked. For example, the **m.add** instruction is always mimicked, hence it has no architectural effects, but it leaks the same observations to an adversary as a normal **add** instruction.

By default, most instructions are qualified as standard instructions, in which case the architectural behavior of the instruction depends on the execution mode of the processor at the time the instruction is being executed. In normal execution mode, all architectural effects are persistent, but in mimicry mode, no changes are committed to the architectural state. In both cases, the same information is observed by an attacker.

2.6.2 Activating Branches

To support linearization with minimal overhead, AMi provides *activating branches* **a.br** $c, target$, which conditionally activate mimicry mode until the target address is reached instead of jumping to the target address. These activating branches always fall through to the next instruction, but when mimicry mode is enabled, all instructions encountered within the branch will be mimicked. As a result, the activating branches behave the same way as a normal branch on an architectural level, but leak the same observations regardless of the branch condition.

> **Example 5.** *This is illustrated in Figure 2.10. The example program is taken from [46]. The* ***beqz*** *branches if the condition* c *is* 0*, but when paired with an activating qualifier* ***a****, it will instead enable mimicry mode. When mimicry mode is enabled in this example (i.e. when the branch condition evaluates to* 0*), the results of the two add instructions will not be made persistent, hence* v *remains untouched. Consequently, the semantics of the program are the as in the original program.*

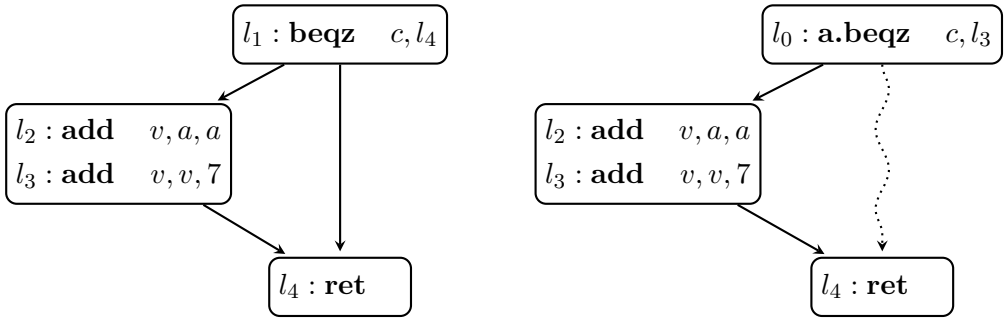
The region starting with the activating branch as entry and its target as exit is referred to as the *activating region* of that branch. The security and correctness of an activating region are formalized in [46], and they will be revisited in this thesis in Section 5.5 and Section 5.4.2 respectively.

2.6.3 Persistent Instructions

Not all architectural effects should be suppressed. For example, architectural updates that affect the non-secret-dependent control flow within an activating region, or instructions that influence the address of memory accesses should not be suppressed. It is necessary to make these effects persistent, otherwise the microarchitectural observations in mimicry mode and standard execution mode would differ,

```
if (c != 0) {
        v = 2 * a + 7;
}
```

C code



Vulnerable assembly code

Linearized assembly form

FIGURE 2.10: Branch linearization using AMi

which breaks security. For this reason, AMi provides *persistent instructions* (qualifier **p**) that are executed normally regardless of the execution mode.

2.6.4 Non-mimicable Instructions

Instructions such as system calls or store operations cannot be mimicked currently. To allow store instructions within an activating region without breaking the correctness, *ghost instructions* can be used to nullify the store operation when executed in mimicry mode. A ghost instruction is only executed in mimicry mode, and will be mimicked in standard execution mode. Consider a store instruction that stores x at address v (**store** x, v). By adding a ghost load **g.load** x, v right before the store instruction, the value that was originally stored at location v will be loaded in mimicry mode, nullifying the architectural effect of the store. Conversely, during normal execution the ghost load will be mimicked, hence x will still contain the new value, not the value that was originally stored at location v [46].

Chapter 3

Related Work

This chapter contains a short survey of different structurization (Section 3.1) and linearization (Section 3.2) techniques. We use the reducible CFG shown in Figure 3.1 as an example throughout this section.

3.1 Structurization

Several linearization methods [12, 48] require structured CFG. A method to transform reducible control flow into structured control flow has been proposed by Anantpur et al. [7]. This method is inspired by the idea of predicated/guarded execution of the basic blocks. For each basic block, a *guard block* is created which decides whether a block should be executed.

> **Example 6.** *Applying this method to the example in Figure 3.1 results in the CFG in Figure 3.2, a triangle-structured CFG. All blocks are placed in a linear*

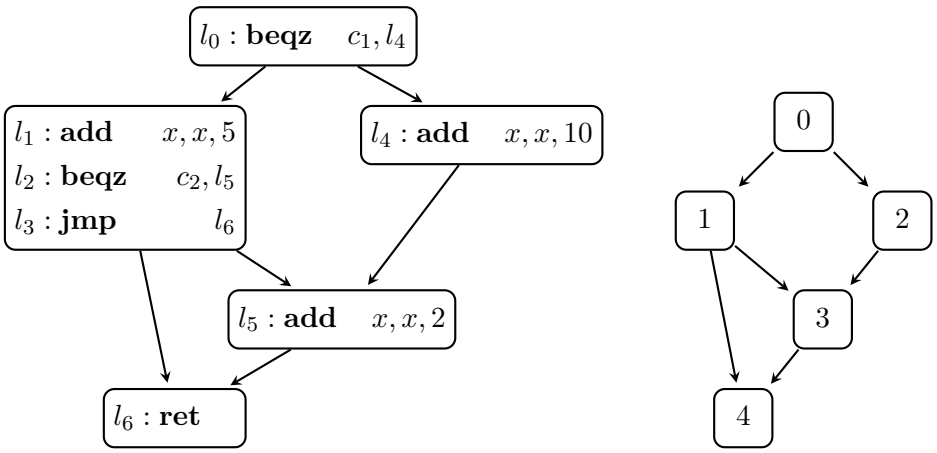


FIGURE 3.1: A simple reducible, but unstructured control flow graph (left) and a more abstract representation of this CFG (right)

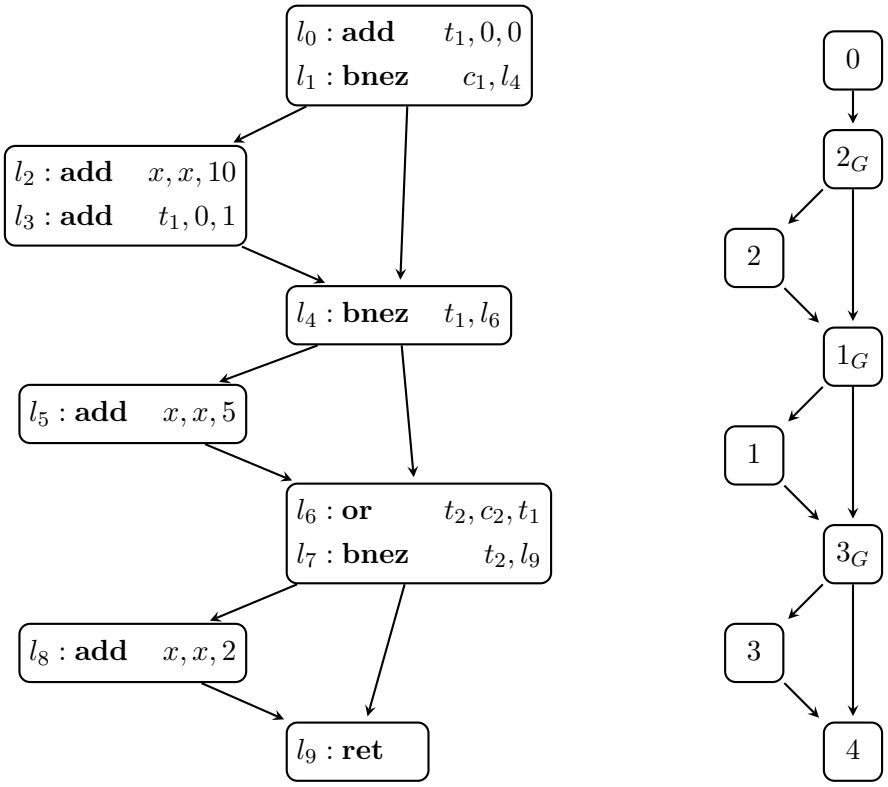


FIGURE 3.2: A triangle-structured version of the CFG in Figure 3.1

> *sequence, and each block i is preceded by a guard block i_G. We introduced auxiliary predicates t_1 and t_2 to determine which blocks should be executed, and which blocks should be skipped. Predicate t_1 is set if and only if block 2 is executed. Since paths that contain block 2 never pass through block 1, we branch over block 1 if t_1 is set. Similarly, we only execute block 3 if condition c_2 is not zero or t_1 is set (i.e. block 2 was taken).*

This approach is used to improve performance on *Single Instruction Multiple Threads (SIMT)* architectures such as GPUs [7]. Branch conditions may evaluate to different values in different threads, leading to *divergence* of threads that negatively impacts performance. Structurization is applied to reduce the impact of divergence by ensuring that divergent threads reconverge earlier. However, this approach structurizes to a CFG where each branch is triangle-structured, which is suboptimal for some applications, such as the mitigation of side-channel leakage using the methods proposed by [12] and [48]. In more recent work by Reissmann et al. [37], a similar method is applied to restructure branches, resulting in structured control flow that is not necessarily triangle-structured.

> **Example 7.** *The application of this method on the example in figure 3.1 would*

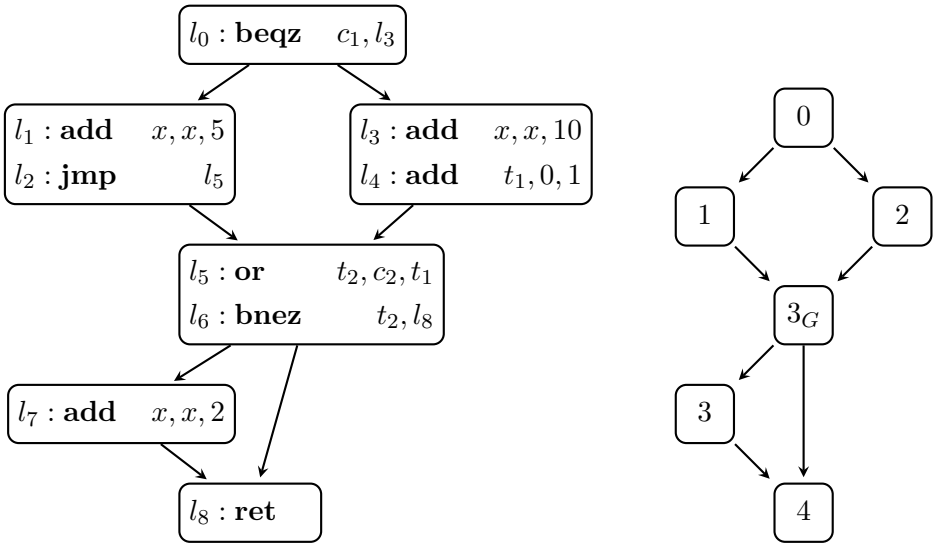


FIGURE 3.3: A structured version of the CFG in figure 3.1

> *result in the CFG shown in figure 3.3.*

Similar techniques are used in the implementation of Borrello et al. [12], in which control flow is structurized before their linearization methods are applied.

3.2 Linearization

Linearization is the process of transforming a CFG into a linear sequence of instructions (i.e. without branches) that produces the same results. This is useful on SIMT architectures, such as GPUs, where *divergent* branches may reduce performance, and it is necessary for SIMD [32]. A branch is considered divergent if the branch condition evaluates to different values on different parallel threads [7, 32]. While the structurization methods described in previous section aim to reconverge divergent threads as soon as possible, more recent work eliminates these divergent branches entirely by linearizing them [32].

3.2.1 Linearization of Structured Control Flow

Since linearization eliminates branches, it can also be used to mitigate against side-channel leakage [12, 48, 43, 15, 33]. Molnar et al. [33] introduced the notion of *PC-Security*. In this model, all control flow decisions are leaked to attackers. They implemented a transformation on C source code where both branches of an if-statement are executed, but only the results of one of those branches are retained. Later research implemented similar methods in the compiler backend [15] or middle-end [12, 48]. First the structurization methods introduced in previous section are applied such that each secret-dependent branch is structured. Then the branch regions are placed in a linear sequence, such that both regions are always executed. A

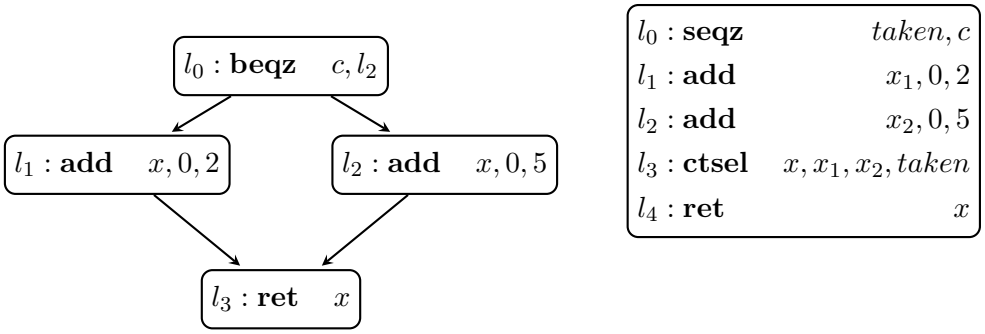


FIGURE 3.4: Program with branches (left) and linearized version (right)

constant-time selection primitive **ctsel** is used to copy either of the used registers to the defined register depending on whether a predicate is true. This predicate is set if and only if the branch in the original branch would have been taken. That way, only the results of one of those branch regions are retained.

> **Example 8.** *This is illustrated in Figure 3.4. Instead of taking a branch if c is zero, a taken predicate is set by the **seqz** (set if equal to zero) instruction.*
>
> *Then, both l_1 and l_2 are placed in a linear sequence, followed by a constant-time selection between x_1 and x_2, predicated by the taken predicate.*

On architectures such as X86, this can constant-time conditional selection can be implemented efficiently making use of the *conditional move* instruction. To implement this primitive on other architectures that do not support such instructions, the method of Molnar et al. [33] can also be applied. Different ways to instantiate **ctsel** are also discussed by [12]. When applied to a program in SSA-form, control-flow linearization can be straightforwardly implemented by replacing ϕ-nodes with **ctsel** primitives.

3.2.2 Partial Control Flow Linearization (PCFL)

A different approach to linearize secret-dependent branches has been proposed by Soares et al. [43]. Their work is based on the concept of *partial control-flow linearization (PCFL)* introduced by Moll and Hack [32]. PCFL is a method for SIMD that eliminates divergent branches, while keeping the uniform branches in place, in contrast to previous methods, in which non-divergent, or *uniform*, branches are also linearized. The only requirement for this method is reducible control flow. Soares et al. [43] note that by replacing "uniform" with "public" and "divergent" with "secret", an algorithm can be obtained to mitigate against side-channel leakage. This method relies on predication or masking *inside* the blocks, in contrast to the methods of [12] and [48] where conditional move instructions are used *after* executing the linearized blocks. In this thesis, we generalize the linearization method of AMi proposed by [46] to support reducible control flow by adapting PCFL.

a) Before linearization

b) After linearization

FIGURE 3.5: Pattern to linearize conditional branches

3.2.3 Programming Model of Architectural Mimicry

Lastly, we introduce the programming model to linearize control flow using AMi as introduced by Winderix et al. [46]. In this programming model, the linearization pattern in Figure 3.5 is applied on secret-dependent branches. A concrete example of the application of this pattern is shown in Figure 2.10. However, the applicability of this pattern is limited to triangle-structured branches. For example, in figure 2.6a, this pattern can only be applied to block 0, where $[1, 4]$ is the SESE region B. Consequently, CFGs such as those shown in Figures 2.7 and 3.1 cannot be linearized without first triangle-structurizing the secret-dependent branches, resulting in significant running time and code size overhead. This limitation originates from the hypotheses that are imposed on well-behaved activating regions. Consider the following proposition for *well-behaved activating regions* as defined by Winderix et al. [46].

Proposition 1 (Well-behaved activating region)**.** *For any activating region* $[l, l']$ *and valid configuration* σ *such that* $[l, l']\sigma \Downarrow_o \sigma'$*, if AC is set after executing the instruction at location* l *(either because it was already before, or because it was incremented at location* l*):*

1) it remains set during the execution of $[l, l']$ *(including recursive function calls but excluding* l'*), and*

2) if AC is incremented at location l*, it is restored to its initial value during the evaluation of the instruction at location* l'*.*

They express the two hypotheses under which this proposition holds. The first hypothesis assumes that the program can be partitioned into a set of distinct functions, and states that a function cannot jump to an arbitrary location in another function. A function can only be entered through its entry label, and a function call at address l always returns to $l + 1$. The second hypothesis states all activating regions are SESE regions, whereas in their notion of a CFG, a control-flow edge

$l \to l'$ is also included if l contains an activating instruction with target l'. This is in contrast with the definition of the successor relation defined in section 2.3.1 of this thesis, which does not include those edges. The programming model requires that this well-behavedness property holds for each activating region. As a result, pattern 3.5 is the only applicable linearization pattern.

Chapter 4

System and Leakage Model

This chapter introduces the system model called *AMiL*, as defined by Winderix et al [46] (Section 4.1) and its formal semantics, along with the leakage model (Section 4.2). We extend AMi semantics to enable more efficient linearization based on PCFL.

4.1 System Model

In this thesis, we assume that programs are compiled to a typical ISA with an extension for AMi, as introduced by [46]. To support the formalization of AMi, the authors of [46] introduce a simplified ISA called AMiL.

4.1.1 Syntax

We adopt the syntax of AMiL from [46], with minor additions. This syntax describes the instructions that can be interpreted by a processor, and those instructions form a program when placed in a sequence. We assume a set of program locations $Loc \subseteq \mathbb{N}$. Additionally, a set of *registers* Regs is assumed, which is further partitioned into a finite set of *physical registers* Regs_p and an infinite set of *virtual registers* Regs_v. Such registers are used to store values $v \in \mathcal{V}$. Registers are mapped to values by the register file $r : \text{Regs} \to \mathcal{V}$. Only physical registers are assumed to be present on a physical machine, hence compilation should allocate a physical register to each virtual register. Let $Instr$ be a set of *instructions*, and $P : Loc \to Instr$ a *program* that maps locations to instructions. The syntax of instructions in AMiL is given by Figure 4.1. An expression can be evaluated to a value given a certain register file r with $[\![\cdot]\!]_r$.

As defined by Winderix et al. [46], qualified instructions can be divided into three classes: mimicable, activating and always persistent. These classes define which qualifiers can be added to instructions. Table 4.1 is adopted from [46], extended to allow activating jumps and function return. We define new instructions **q.instr** for each instruction **instr** and supported qualifier **q** for that instruction (as described in Table 4.1), in addition to the instructions defined in Figure 4.1.

$\langle Expr \rangle ::= \langle Val \rangle \mid \langle Reg \rangle$

$\langle Instr \rangle ::=$ **add** $\langle Reg \rangle, \langle Expr \rangle, \langle Expr \rangle \mid$ **mul** $\langle Reg \rangle, \langle Expr \rangle, \langle Expr \rangle$
$\mid$ **beqz** $\langle Expr \rangle, \langle Loc \rangle \mid$ **bnez** $\langle Expr \rangle, \langle Loc \rangle$
$\mid$ **call** $\langle Loc \rangle \mid$ **ret** $\mid$ **jmp** $\langle Expr \rangle$
$\mid$ **load** $\langle Reg \rangle, \langle Expr \rangle \mid$ **store** $\langle Expr \rangle, \langle Expr \rangle$

FIGURE 4.1: Syntax of base instructions in AMiL where $\langle Val \rangle$ ranges over $\mathcal{V}$,$\langle Reg \rangle$ ranges over Regs and $\langle Loc \rangle$ ranges over Loc.

Class	Instructions	Qualifiers
Mimicable	**add**, **mul**, **load**	**s**, m, g, p
Activating	**call**, **ret**, **jmp**, **beqz**, **bnez**	a, **p**
Always Persistent	**store**	**p**

TABLE 4.1: Classes of qualified instructions for AMiL (default qualifier in bold)

Lastly, we adopt the notion of an *architectural configuration* and an *AMi configuration* from [46]. The former is a tuple $\langle m, r, \mathrm{pc} \rangle$ where $m : \mathcal{V} \to \mathcal{V}$ is memory and $r : \mathrm{Regs} \to \mathcal{V}$ a register file, which maps registers to values. The program counter pc is a register that stores the location of the next instruction to execute. An *AMi configuration* $\sigma = \langle m, r, \mathrm{pc}, \mathrm{AC}, \mathrm{En}, \mathrm{Ex} \rangle$ extends an architectural configuration $\langle m, r, \mathrm{pc} \rangle$ with three additional registers: an activation counter AC, which counts the number of nested activations, a mimicry entry address En that stores the location at which AC was set, and a mimicry exit address Ex, which stores the location at which AC will be restored to its initial value.

4.1.2 Semantics

Base Semantics The base semantics of AMiL can be defined as a transition system over architectural configurations $a \xrightarrow[inst]{o} a'$. Here, o is an observation trace, which defines what information leaks on microarchitectural side channels during the evaluation, as will be further elaborated upon in Section 4.2. The formal semantics of base AMiL are given by Figure 4.2.

AMi semantics In addition to the semantics of base AMiL, the authors of [46] define the semantics of the full AMiL ISA (including qualified instructions) as a relation $\sigma \overset{o}{\Rightarrow} \sigma'$ over AMi configurations, which define when instructions are mimicked or executed, and how the activation counter is updated. We adopt these semantics from [46], with the addition of evaluation rules for **a.bnez** and activating unconditional jump, as shown in Figures 4.3 and 4.4. The activating jump and **a.bnez** are added to AMiL in order to support the extended programming model introduced in this thesis.

$$\langle m, r, \mathrm{pc}\rangle \xrightarrow[\textbf{add } x,e_1,e_2]{o} \langle m, r[x \mapsto [\![e_1]\!]_r + [\![e_2]\!]_r], \mathrm{pc}+1\rangle$$

$$\langle m, r, \mathrm{pc}\rangle \xrightarrow[\textbf{mul } x,e_1,e_2]{o} \langle m, r[x \mapsto [\![e_1]\!]_r * [\![e_2]\!]_r], \mathrm{pc}+1\rangle$$

$$\langle m, r, \mathrm{pc}\rangle \xrightarrow[\textbf{load } x,e]{o} \langle m, r[x \mapsto m[[\![e]\!]_r], \mathrm{pc}+1\rangle$$

$$\langle m, r, \mathrm{pc}\rangle \xrightarrow[\textbf{store } x,e]{o} \langle m[[\![e]\!]_r \mapsto x], r, \mathrm{pc}+1\rangle$$

$$\langle m, r, \mathrm{pc}\rangle \xrightarrow[\textbf{beqz } e,l]{o} \langle m, r, l \text{ if } [\![e]\!]_r = 0 \text{ else } \mathrm{pc}+1\rangle$$

$$\langle m, r, \mathrm{pc}\rangle \xrightarrow[\textbf{bnez } e,l]{o} \langle m, r, l \text{ if } [\![e]\!]_r \neq 0 \text{ else } \mathrm{pc}+1\rangle$$

$$\langle m, r, \mathrm{pc}\rangle \xrightarrow[\textbf{call } l]{o} \langle m, r[\mathrm{ra} \mapsto \mathrm{pc}+1], l\rangle$$

$$\langle m, r, \mathrm{pc}\rangle \xrightarrow[\textbf{jmp } e]{o} \langle m, r, [\![e]\!]_r\rangle$$

$$\langle m, r, \mathrm{pc}\rangle \xrightarrow[\textbf{ret}]{o} \langle m, r, r[\mathrm{ra}]\rangle$$

FIGURE 4.2: Formal semantics of base AMiL defined as a transition system over architectural configurations

Non-activating Instructions The EXECUTE rule is applied to persistent instructions (**p**), standard instructions (**s**) in normal execution mode (AC = 0) and ghost instructions (**g**) in mimicry mode (AC > 0). In this case, the instruction is executed as defined by the transition system in Figure 4.2, and its architectural effects are persisted. Additionally, the activation counter is decremented if the mimicry exit address (Ex) is reached, as defined in the *decrAC* function (see Figure 4.4). Similarly, the MIMIC rule is applied to mimic instructions (**m**), standard instructions (**s**) in mimicry mode, and ghost instructions (**g**) in normal execution mode. This rule ignores the architectural effects of the base instruction ($\langle m, r, \mathrm{pc}\rangle \xrightarrow[inst]{o} \cdot$), the transition system is only evaluated for its microarchitectural observations o. As in the EXECUTE rule, the activation counter may be decremented.

Activating Instructions Activating instructions may increment the activation counter if it is currently set to 0, or if the activation counter was already set, but the same activating instruction is encountered again (pc = En). The latter is necessary to ensure correct behavior in the case of *nested activations* (which may occur in nested function calls). This is defined by the *incrAC* function in figure 4.4.

The ACTIVATING-CALL rule first jumps to the target location l by changing the program counter, and then enables mimicry mode unconditionally. Both the return address and the mimicry exit address Ex are set to pc+1, such that mimicry mode is enabled for the duration of the call. The ACTIVATING-BRANCH rule does not jump to the target location l. Instead, the mimicry exit address Ex is set to l, such that mimicry mode is enabled for the duration of the branch. The activation counter AC is only incremented if the branch condition evaluates to true or if it is a nested

activation. Similarly, the ACTIVATING-JUMP rule enables mimicry mode until the target location is reached, but in this case it happens unconditionally.

Big-step Evaluation Winderix et al. [46] introduce the notion of big-step evaluation of a program region $[l, l']$, denoted as $[l, l'] \ \sigma \Downarrow_o \sigma'$. It executes the program starting at location l in configuration σ until location l' is reached as part of the same function call, resulting in configuration σ', and leaking observation trace o. The semantics of big-step evaluation are given in Figure 4.5. Most instructions are handled by the STEP and INST rule, but call instructions are handled separately by the CALL rule to avoid early termination of the evaluation in the case of recursive function calls.

4.2 Leakage Model and Security Objectives

This thesis adopts the *constant-time leakage model* as defined by [6], and its application to the AMiL ISA proposed by [46]. This model defines the ability of an attacker to observe (micro)architectural state and infer architectural state. As defined in Section 4.1.2, the evaluation of an instruction in a certain configuration, given by the relation $\sigma \xrightarrow[inst]{o} \sigma'$, produces an observation $o \in \mathcal{O}$. The semantics of this leakage model are given by a set of leakage functions $\lambda_{inst} : \mathcal{A} \to \mathcal{O}$ where $\mathcal{A}$ is the set of all possible architectural configurations, and $\mathcal{O}$ the set of possible observations. Instructions **add** and **mul** each leak a fixed observation $\lambda_{\textbf{add}}$ and $\lambda_{\textbf{mul}}$ respectively, independent of the configuration. For instructions that access memory, such as **load** and **store**, the accessed memory address is leaked, for instance, through the cache. Furthermore, we assume that control flow is leaked explicitly, hence branch and jump instructions leak their target address. The leakage functions adopted from [46] are given in Figure 4.6. Note that activating instructions don't leak any observations, as defined by Figure 4.3.

Through program annotations, the developer identifies which parts of the register map and memory should remain secret. By hardening the program, we aim to avoid leakage of these secrets. This is modelled by a *security policy* $\mathcal{P}$ that maps registers and memory locations to two security levels: public and secret. Two configurations σ and σ' are considered low-equivalent under this policy, denoted as $\sigma =_{\mathcal{P}} \sigma'$, if they agree on the public part of their register file and memory map. Since programs with the same public inputs, but potentially different secret inputs, should leak the same observations, we say that a program is secure if two executions starting from low-equivalent states produce the same observations.

EXECUTE

$$\frac{P[\mathrm{pc}] = q.inst \qquad \mathrm{AC}' = decrAC(\mathrm{AC}, \mathrm{Ex}, \mathrm{pc}) \\ execute(q, \mathrm{AC}') \qquad \langle m, r, \mathrm{pc}\rangle \xrightarrow[inst]{o} \langle m', r', \mathrm{pc}'\rangle}{\langle m, r, \mathrm{pc}, \mathrm{AC}, \mathrm{En}, \mathrm{Ex}\rangle \overset{o}{\Rightarrow} \langle m', r', \mathrm{pc}', \mathrm{AC}', \mathrm{En}, \mathrm{Ex}\rangle}$$

MIMIC

$$\frac{P[\mathrm{pc}] = q.inst \qquad \mathrm{AC}' = decrAC(\mathrm{AC}, \mathrm{Ex}, \mathrm{pc}) \\ mimic(q, \mathrm{AC}') \qquad \langle m, r, \mathrm{pc}\rangle \xrightarrow[inst]{o} \cdot \qquad \mathrm{pc}' = \mathrm{pc} + 1}{\langle m, r, \mathrm{pc}, \mathrm{AC}, \mathrm{En}, \mathrm{Ex}\rangle \overset{o}{\Rightarrow} \langle m, r, \mathrm{pc}', \mathrm{AC}', \mathrm{En}, \mathrm{Ex}\rangle}$$

ACTIVATING-CALL

$$\frac{P[\mathrm{pc}] = \textbf{a.call } l \qquad \mathrm{AC}' = decrAC(\mathrm{AC}, \mathrm{Ex}, \mathrm{pc}) \\ \mathrm{AC}'', \mathrm{En}', \mathrm{Ex}' = incrAC(\mathrm{AC}', \mathrm{En}, \mathrm{Ex},\ \mathrm{pc},\ \mathrm{pc} + 1,\ \mathrm{true}) \\ r' = r[\mathrm{ra} \mapsto \mathrm{pc} + 1] \qquad \mathrm{pc}' = l}{\langle m, r, \mathrm{pc}, \mathrm{AC}, \mathrm{En}, \mathrm{Ex}\rangle \overset{\epsilon}{\Rightarrow} \langle m, r', \mathrm{pc}', \mathrm{AC}'', \mathrm{En}', \mathrm{Ex}'\rangle}$$

ACTIVATING-BRANCH

$$\frac{P[\mathrm{pc}] = \textbf{a.beqz } l \qquad \mathrm{AC}' = decrAC(\mathrm{AC}, \mathrm{Ex}, \mathrm{pc}) \\ c = (\llbracket e \rrbracket_r = 0) \qquad \mathrm{pc}' = \mathrm{pc} + 1 \\ \mathrm{AC}'', \mathrm{En}', \mathrm{Ex}' = incrAC(\mathrm{AC}', \mathrm{En}, \mathrm{Ex},\ \mathrm{pc},\ \mathrm{l},\ \mathrm{c})}{\langle m, r, \mathrm{pc}, \mathrm{AC}, \mathrm{En}, \mathrm{Ex}\rangle \overset{\epsilon}{\Rightarrow} \langle m, r, \mathrm{pc}', \mathrm{AC}'', \mathrm{En}', \mathrm{Ex}'\rangle}$$

ACTIVATING-BRANCH

$$\boxed{\frac{P[\mathrm{pc}] = \textbf{a.bnez } l \qquad \mathrm{AC}' = decrAC(\mathrm{AC}, \mathrm{Ex}, \mathrm{pc}) \\ c = (\llbracket e \rrbracket_r \neq 0) \qquad \mathrm{pc}' = \mathrm{pc} + 1 \\ \mathrm{AC}'', \mathrm{En}', \mathrm{Ex}' = incrAC(\mathrm{AC}', \mathrm{En}, \mathrm{Ex},\ \mathrm{pc},\ \mathrm{l},\ \mathrm{c})}{\langle m, r, \mathrm{pc}, \mathrm{AC}, \mathrm{En}, \mathrm{Ex}\rangle \overset{\epsilon}{\Rightarrow} \langle m, r, \mathrm{pc}', \mathrm{AC}'', \mathrm{En}', \mathrm{Ex}'\rangle}}$$

ACTIVATING-JUMP

$$\boxed{\frac{P[\mathrm{pc}] = \textbf{a.jmp } l \qquad \mathrm{AC}' = decrAC(\mathrm{AC}, \mathrm{Ex}, \mathrm{pc}) \\ \mathrm{pc}' = \mathrm{pc} + 1 \\ \mathrm{AC}'', \mathrm{En}', \mathrm{Ex}' = incrAC(\mathrm{AC}', \mathrm{En}, \mathrm{Ex},\ \mathrm{pc},\ \mathrm{l},\ \mathrm{true})}{\langle m, r, \mathrm{pc}, \mathrm{AC}, \mathrm{En}, \mathrm{Ex}\rangle \overset{\epsilon}{\Rightarrow} \langle m, r, \mathrm{pc}', \mathrm{AC}'', \mathrm{En}', \mathrm{Ex}'\rangle}}$$

FIGURE 4.3: Evaluation rules for qualified instructions, taken from [46] with minor additions marked with boxes

$$
\begin{aligned}
\mathit{execute}(q, \mathrm{AC}) &\triangleq q = \mathbf{p} \vee (q = \mathbf{s} \wedge \mathrm{AC} = 0) \\
&\quad \vee (q = \mathbf{g} \wedge \mathrm{AC} > 0) \\
\mathit{mimic}(q, \mathrm{AC}) &\triangleq q = \mathbf{m} \vee (q = \mathbf{s} \wedge \mathrm{AC} > 0) \\
&\quad \vee (q = \mathbf{g} \wedge \mathrm{AC} = 0) \\
\mathit{decrAC}(\mathrm{AC}, \mathrm{Ex}, \mathrm{pc}) &\triangleq \begin{cases} \mathrm{AC} - 1 & \text{if } \mathrm{AC} = 0 \wedge \mathrm{pc} = \mathrm{Ex} \\ \mathrm{AC} & \text{otherwise} \end{cases} \\
\mathit{incrAC}(\mathrm{AC}, \mathrm{En}, \mathrm{Ex}, \mathrm{pc}, \mathrm{Ex}', c) &\triangleq \begin{cases} \mathrm{AC} + 1, \mathrm{pc}, \mathrm{Ex'} & \text{if } \mathrm{AC} = 0 \wedge c = \mathit{true} \\ \mathrm{AC} + 1, \mathrm{En}, \mathrm{Ex} & \text{if } \mathrm{AC} > 0 \wedge \mathrm{pc} = \mathrm{En} \\ \mathrm{AC}, \mathrm{En}, \mathrm{Ex} & \text{otherwise} \end{cases}
\end{aligned}
$$

FIGURE 4.4: Functions used by evaluation rules in Figure 4.3

INST

$$
\frac{l = l' \qquad \sigma.\mathrm{pc} = l \qquad P[l] \neq q.\mathit{call}\ f \qquad \sigma \overset{o}{\Rightarrow} \sigma'}{[l, l']\ \sigma \Downarrow_o \sigma'}
$$

CALL

$$
\frac{\begin{array}{c} l = l' \qquad \sigma.\mathrm{pc} = l \qquad P[l] = q.\mathit{call}\ f \\ \sigma \overset{o_c}{\Longrightarrow} \sigma_f \qquad [\mathit{entry}(f), \mathit{exit}(f)]\ \sigma_f \Downarrow_{o_f} \sigma' \qquad o = o_c \cdot o_f \end{array}}{[l, l']\ \sigma \Downarrow_o \sigma'}
$$

STEP

$$
\frac{\begin{array}{c} l \neq l' \qquad \sigma.\mathrm{pc} = l \qquad [l, l]\sigma \Downarrow_{o''} \sigma'' \\ l'' = \sigma''.\mathrm{pc} \qquad [l'', l']\ \sigma \Downarrow_{o'} \sigma' \qquad o = o'' \cdot o' \end{array}}{[l, l']\ \sigma \Downarrow_o \sigma'}
$$

FIGURE 4.5: Big-step evaluation of region $[l, l']$, taken from [46]

$$
\begin{aligned}
\lambda_{\mathbf{add}}(\langle m, r, \mathrm{pc}\rangle) &= \mathit{add} && \text{if } P[\mathrm{pc}] = \mathbf{add}\ x, e_1, e_2 \\
\lambda_{\mathbf{mul}}(\langle m, r, \mathrm{pc}\rangle) &= \mathit{mul} && \text{if } P[\mathrm{pc}] = \mathbf{mul}\ x, e_1, e_2 \\
\lambda_{\mathbf{load}}(\langle m, r, \mathrm{pc}\rangle) &= \mathit{load}\ a && \text{if } P[\mathrm{pc}] = \mathbf{load}\ x, e \text{ and } [\![e]\!]_r = a \\
\lambda_{\mathbf{store}}(\langle m, r, \mathrm{pc}\rangle) &= \mathit{store}\ a && \text{if } P[\mathrm{pc}] = \mathbf{store}\ e_1, e_2 \text{ and } [\![e_2]\!]_r = a \\
\lambda_{\mathbf{call}}(\langle m, r, \mathrm{pc}\rangle) &= \mathit{call}\ l && \text{if } P[\mathrm{pc}] = \mathbf{call}\ l \\
\lambda_{\mathbf{jmp}}(\langle m, r, \mathrm{pc}\rangle) &= \mathit{jmp}\ l && \text{if } P[\mathrm{pc}] = \mathbf{jmp}\ e \text{ and } [\![e]\!]_r = l \\
\lambda_{\mathbf{beqz}}(\langle m, r, \mathrm{pc}\rangle) &= \mathit{br}\ l' && \text{if } P[\mathrm{pc}] = \mathbf{beqz}\ e, l \text{ and } l' = \begin{cases} \mathrm{l} & [\![e]\!]_r = 0 \\ \mathrm{pc} + 1 & [\![e]\!]_r \neq 0 \end{cases}
\end{aligned}
$$

FIGURE 4.6: Leakage functions of AMiL taken from [46] where *add*, *mul*, *load*, *store*, *call*, *jmp*, *br* are leakage identifiers.

Chapter 5

Linearization with Architectural Mimicry

This chapter presents several patterns to linearize secret-dependent branches to mitigate leakage of secrets on side channels based on control flow. Along with linearization patterns, additional criteria are specified to ensure both correctness and security of the hardened program according to the constant-time leakage model.

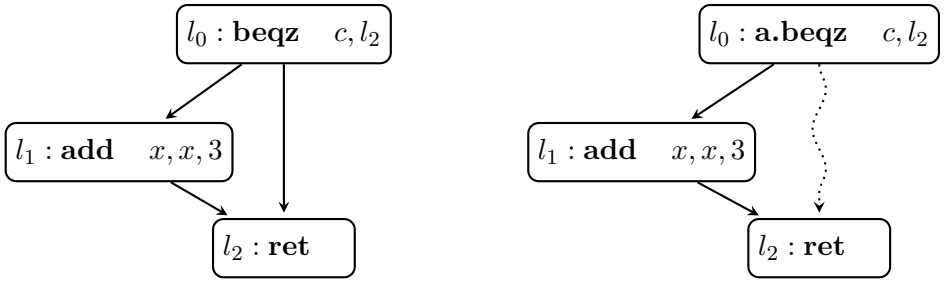
Section 5.1 introduces key concepts used to describe linearization methods that make use of AMi. These methods are explained in Section 5.2, and Section 5.3 describes how suitable activating instructions can be chosen in accordance to the linearized control flow graph. The conditions for a linearization to be correct are given in Section 5.4, which explains how to nullify persistent side-effects and describes the need for additional register allocation constraints. Finally, Section 5.5 discusses the security of the linearized program. It elaborates on the need to make architectural effects of some instructions persistent to ensure that the leakage trace does not depend on secrets.

5.1 Key Concepts

In this section, we describe several concepts, such as *activating edges*, *ghost edges* and the notion of an *activating region*.

Definition 1 (Activating edge, Activating region)**.** An *activating edge* is a pair (l_x, l_y) where l_x is the location of an activating instruction with target l_y. Contrary to related work, we do not include this edge in the CFG. The region $[l_x, l_y]$ is called the *activating region* of this edge.

Activating regions are pairs of locations that depict the region for which mimicry mode remains active. Although activating edges are not part of the CFG, we represent them visually with dotted arrows.



(a) Original CFG (b) Linearized CFG

FIGURE 5.1: A simple CFG with a single branch (left) and a linearized CFG that produces the same result (right)

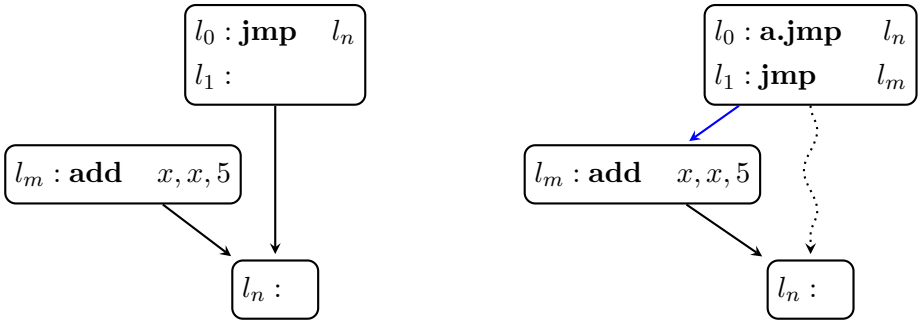
> **Example 9.** *Figure 5.1 gives an example of a CFG and a linearized version. Here,* (l_0, l_2) *is an activating edge, and* $[l_0, l_2]$ *an activating region. If* $c = 0$ *in the linearized program, mimicry mode will be enabled at* l_0*, and it remains enabled until* l_2 *is reached. Similarly, in the original program, the instructions within this region will not be executed if the branch is taken.*

Under the assumption that l_1 does not affect the architectural state, both programs will produce the same architectural results. However, since control flow does not depend on c, the linearized program will produce the same microarchitectural results, independent of the secret c. Hence, a normal control-flow edge may be replaced with an activating edge without changing the architectural semantics of the program, in order to mitigate leakage of secrets.

In the linearized program we assume that for each activating region $[l_x, l_y]$, l_y post-dominates l_x. This is necessary to ensure that after enabling mimicry mode, it will eventually be disabled when reaching l_y before returning from the function. This condition are formalized as part of a revised notion of a *well-behaved* activating region in section 5.4.1.

Definition 2 (Ghost edge)**.** A control-flow edge from block B to block B' is a *ghost edge* if and only if there is an activating jump in B that unconditionally enables mimicry mode.

Ghost edges are edges that may be present in a linearized CFG, while not being present in the original CFG. We require that such edges are only taken in mimicry mode, hence they do not change the semantics of the program, as will be formalized in section 5.4. The introduction of such edges in the linearized CFG may be useful to ensure that the exit of an activating region post-dominates the entry, as it enables the changing the control flow in mimicry mode. Using a ghost edge to divert control flow to the exit of another activating region may be useful to ensure that the entry of this region is post-dominated by its exit.



(a) Original CFG (b) Linearized CFG

FIGURE 5.2: A CFG with a jump (left) and a linearized CFG that produces the same result (right)

Example 10. *Changing control flow in mimicry mode using ghost edges is illustrated in figure 5.2. The original program never executes the instruction at location l_m, because the jump at l_0 is succeeded by l_n. On the other hand, in the linearized CFG, control flow does go through l_m. However, since $[l_0, l_n]$ is an activating region, and l_0 unconditionally enables mimicry mode, we know that l_m will always be mimicked when reached from l_0. Hence, $l_1 \to l_m$ is a ghost edge induced by the jump at l_1, and the linearized program produces the same architectural results as the original program, despite reaching different locations in the control flow. This is useful when l_m is the target of an activating branch earlier in the program, as this change ensures that control flow will always pass through l_m before reaching l_n, which will be further illustrated in section 5.2.*

Based on these concepts, we introduce a class of transformations called *AMi linearizations*. Intuitively, such linearization only replaces control-flow edges with activating edges, and each edge that is added to the CFG must be a ghost edge. Section 5.4 formalizes the correctness of these transformations.

Definition 3 (AMi linearization)**.** Let P be a program. An *AMi linearization* is a transformation T that

- replaces some non-activating instructions in P with their activating counterpart (i.e. replaces control-flow edges with activating edges), or
- adds jump instructions such that for each edge $B \to B'$ in $T(P)$, $B \to B'$ is either part of P, or it is a ghost edge (i.e. adds ghost edges)

The transformations shown in Figures 5.1 and 5.2 are basic examples of AMi linearizations, but an AMi linearization can replace multiple edges with activating edges, and create multiple ghost edges. For any AMi linearization it holds that edges that are removed from the CFG are activating edges, while any edges that are added

are ghost edges. Hence, given the original CFG and a linearized form, the sets of activating edges and ghost edges can unambiguously be determined. However, both activating edges and ghost edges will be emphasized in illustrations throughout this chapter for clarity. In the next section, several methods are introduced to construct a linearized CFG. The corresponding transformations that result from these methods will conform with the definition of an AMi linearization.

5.2 Linearization Methods

This section proposes three methods to linearize control flow making use of AMi. Most existing techniques to linearize control flow in order to mitigate side channel leakage are similar to *if-conversion* [13], and require transformation to structured control flow [12, 48]. More recent studies, however, have introduced the concept of *PCFL* [32] as an alternative to if-conversion which avoids such structurization, which can result in a significant reduction of code size and execution time. These techniques can also be applied to mitigate side channel leakage, as shown by Soares et al. [43].

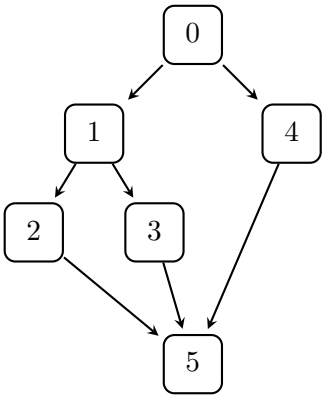
Both of these methods make use of predication or conditional selection. In this section, we apply these techniques to AMi, and propose three linearization methods. Each of these methods is an AMi linearization as defined in Definition 3. To facilitate the description of linearization methods, instructions will be omitted from CFGs. Instead, each linearization method will be characterized by the set of activating edges and the set of ghost edges they produce, which can be represented visually.

5.2.1 Method 1: Structurization to Triangle-Structured Control Flow

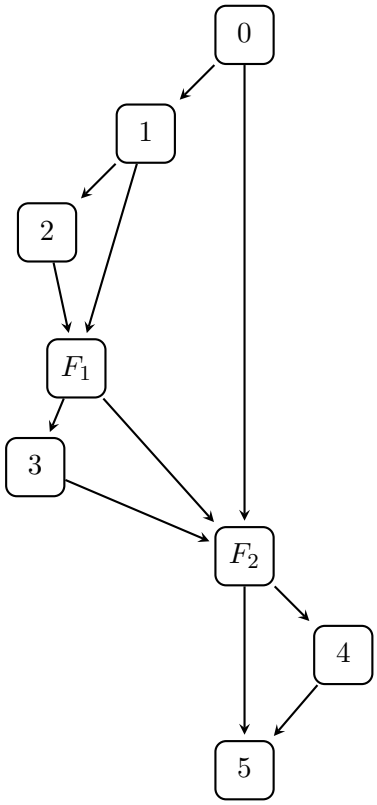
The linearization pattern introduced by Winderix et al. [46] can only be applied to triangle-structured control flow, which can be achieved by structurizing the original CFG to a triangle-structured CFG.

> **Example 11.** *Method 1 is illustrated in Figure 5.3, where the CFG is first structurized to a triangle-structured CFG, after which several edges are replaced with activating edges. The resulting activating regions* $[0, F_2]$, $[1, F_1]$, $[F_1, F_2]$ *and* $[F_2, 5]$ *are well-behaved, since they are SESE regions.*

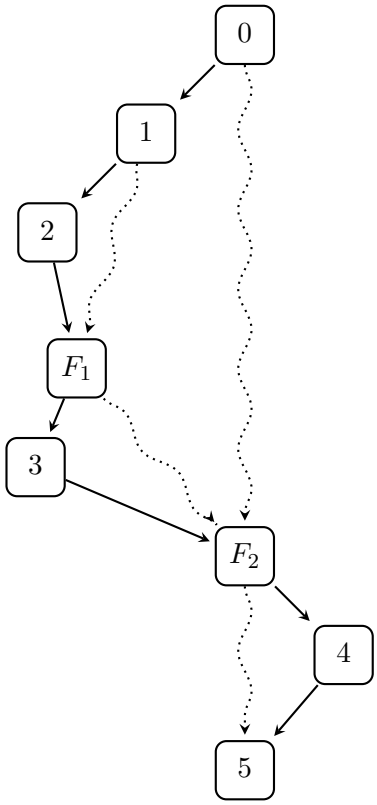
This method has two main drawbacks. The introduction of another conditional branch requires that the branch condition outlives the first branch region, which introduces additional register pressure. Furthermore, in the structurized CFG of Example 11, any variable that lives from block 2 or 3 to block 5 is live at block 4, while this is not the case in the original CFG. Hence, the structurized CFG has higher register pressure than the original CFG. However, in practice, either the region of block 1, or the region of block 4 is executed, but not both. Consequently, such register allocation constraints are not necessary to maintain correctness.



(a) Original CFG



(b) Triangle-structured CFG



(c) Linearized CFG

FIGURE 5.3: Example of triangle structurization and linearization of structured control flow graph

5.2.2 Method 2: Linearization with Ghost Edges

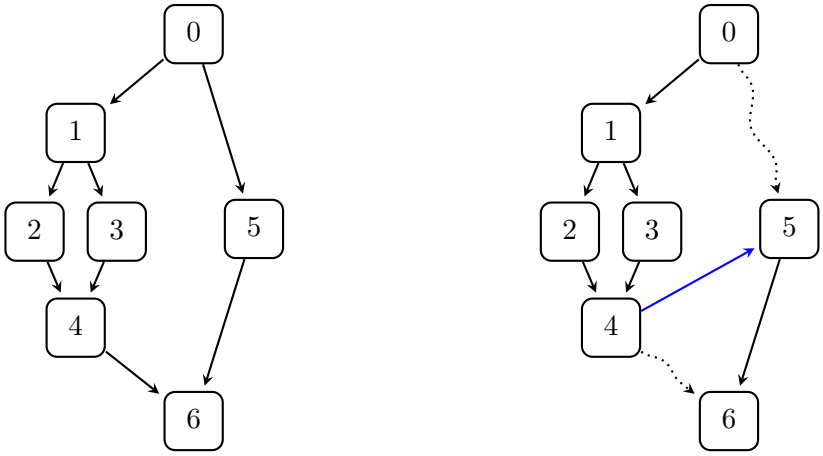
If all conditional branches are structured, the regions that follow the conditional branch will be SESE regions. In this case, the program structure tree [22] can be built to identify all branch regions. This is a hierarchical representation of the control structure of the program where each node is a SESE region. By using ghost edges, control can flow from one branch region to another branch region, by creating ghost edges from the exit of a branch region to the target of that branch.

Either way, only one of the branch regions is executed, and the other one is mimicked. This matches the behavior of the original program where control flow passes through only one of those regions. However, the linearized form produces the micro-architectural effects of both regions, hence leakage of the outcome of this branch condition is mitigated. To make sure that the newly added edges are considered ghost edges, one must ensure that mimicry mode is enabled, requiring other outgoing edges of the exit block to become activating edges. Furthermore, this ensures that branch target post-dominates the activating branch, which is a requirement for the well-behavedness of an activating region. However, the activating regions are no longer considered well-behaved by the original formalization by Winderix et al. [46], since they are not SESE. For this reason, a relaxation of this notion of well-behavedness is introduced in Section 5.4.1.

Example 12. *This is illustrated in Figure 5.4, where we assume that block* 0 *contains a secret-dependent branch. The branch regions of this branch are determined, and we find regions* $[1, 4]$ *and* $[5, 5]$*. The exit of* $[1, 4]$ *is* 4*, hence we create ghost edges from* 4 *to the entry of* $[5, 5]$*. To ensure that this is a ghost edge, we need to make sure that mimicry mode is enabled before taking this edge. As a result, the edge from* 4 *to* 6 *becomes an activating edge. The reason why this is necessary is as follows. When taking the ghost edge* $4 \rightarrow 5$*, there are two options. Either mimicry mode was already enabled for activating region* $[0, 5]$*, in which case blocks* 1*,* 2 *or* 3 *and* 4 *are mimicked, any activating instructions in block* 4 *have no effect, and after taking the ghost edge, block* 5 *is executed normally.*

If mimicry mode was not enabled for region $[0, 5]$*, it will be enabled for region* $[4, 6]$*, because of the outgoing activating edge in block* 4*. Since there is an outgoing activating edge and an outgoing ghost edge, we make sure that mimicry mode is enabled unconditionally, by using an activating jump instruction in block* 4*. This will be elaborated upon in Section 5.3. In this case,* 1*,* 2 *or* 3 *and* 4 *are executed normally, and block* 5 *is mimicked.*

Note that in the linearized CFG, the targets of activating branches post-dominate that branch, but the activating regions $[0, 5]$ *and* $[4, 6]$ *are not SESE, meaning that those regions are not well-behaved according to the well-behavedness proposition of Winderix et al. [46]. As a result, the well-behavedness proposition needs to be relaxed in order to support this linearization method.*



(a) Before linearization (b) After linearization

FIGURE 5.4: Example of linearization of the branch in block 0 using ghost edges

5.2.3 Method 3: Linearization of Reducible Control Flow

The method described in the previous section can be generalized to non-structured secret-dependent branches, assuming that the CFG is reducible. Although the regions of a non-structured branch are not SESE, similar methods can be applied to all exiting blocks of the region in such a way that the resulting activating regions are well-behaved, by ensuring that the exit of each activating region post-dominates the entry.

Partial Control Flow Linearization

Moll and Hack [32] proposed a method for partial control-flow linearization (PCFL) of reducible control flow in the context of vectorization of a SIMD program while preserving uniform branches. Soares et al. [43] show that this method can also be applied to eliminate side channels. Assuming a reducible CFG with back edges removed, a *topological ordering* of such DAG only visits a node after all its predecessors have been visited. We adopt the definition of a *compact topological ordering* as defined by [43] and [32].

Definition 4 (Compact Topological Ordering)**.** An n-sequence of blocks $B_1, \ldots, B_n$ is dominance compact if whenever B_1 dominates B_n then B_1 dominates every B_i, $1 < i < n$. Similarly, an n-sequence of blocks $B_1, \ldots, B_n$ is loop compact if whenever B_1 and B_n belong to a loop L then every B_i, $1 < i < n$, belong to L as well. A topological ordering of a CFG is compact if it is both dominance and loop compact with respect to all dominance sets and loops.

Both Moll and Hack [32] and Soares et al. [43] show that when their methods are applied using a compact topological ordering, uniform branches [32] or branches that don't depend on secrets [43] will be preserved. With this notion of compactness,

we give Algorithm 1 based on [32], which computes the set of activating edges and the set of ghost edges for a reducible CFG with back edges removed. Given these sets, branch terminators can be rewritten to linearize the program accordingly, as will be explained in section 5.3.

The main idea behind this algorithm is the use of *deferral edges* D throughout the algorithm. When an edge (b, s) is removed from the CFG, a deferral edge (next, s) is added for each remaining successor "next". Such deferral edge guarantees that the target of the edge post-dominates the source in the linearized program. Throughout the algorithm, these deferral edges are moved to successor blocks while maintaining this post-dominance relation. This ensures that even though some successors have been removed from secret-dependent branches, those successors will still be reached eventually. The changes introduced in this thesis make sure that mimicry mode is enabled until this successor is reached, to ensure that the program behaves correctly. Furthermore, if a deferral edge is materialized into a control-flow edge in the linearized program that was not already present in the original CFG, we ensure that this is a ghost edge. Under these conditions, this linearization is an AMi linearization, which is correct, as will be explained in Section 5.4.4.

The original algorithm constructs a set E_l of control-flow edges in the linearized CFG, creating a directed acyclic graph (DAG), after which back edges B are reinserted. Instead of computing set E_l, we mark which control-flow edge in the original CFG should be replaced by activating edges by adding them to set A. Similarly, any new edges will become ghost edges in the linearized CFG, and they are added to the set G. One can show that $G = E_l \setminus E$ and $A = E \setminus (E_l \cup B)$. Here E_l refers to the set of edges in the linearized CFG, as constructed by the original algorithm from [32] and shown in Algorithm 1 in comments. Hence, $E_l = (E \setminus (A \cup B)) \cup G$, meaning that the linearized DAG can be constructed by removing activating edges and back edges from the original CFG, and adding ghost edges. To obtain the linearized CFG with loops from the original CFG, we reinsert back edges B.

Example 13. *This algorithm is illustrated on an example in Figure 5.5. Figure 5.5a shows the original CFG where the blocks are ordered in a compact topological order as signified by their number.*

First, block 0 *is visited. We will assume that* 0 *contains a secret-dependent branch. We choose block* 1*, the successor with the lowest number, as the single successor in the linearized CFG ("next" in Algorithm 1). Then we add activating edges to each other successor, and we add deferral edges from block* 1 *to each other successor or deferral target in* T*, but note that* T *is currently empty. This results in Figure 5.5b.*

Next, we visit block 1*. Since* 1 *does not contain a secret-dependent branch, we iterate over all its successors, and check if there is a deferral target in* T *with a lower number than the successor. This is not the case for both successors, hence no ghost edges are created. We also create new deferral edges from each successor to the current deferral targets in* T*, and clean up any of the*

Algorithm 1: Partial Linearization with AMi

input : F, a function with edges E (with all backedges removed)
B_{sd}, set of secret-dependent branch blocks
output: A, set of activating edges
G, set of ghost edges

$D \leftarrow \emptyset$; `// Set of deferral edges`
foreach $b \in$ `compactOrder`(F) **do**
  $T \leftarrow \{s \mid (b, s) \in D\}$; `// Current deferral targets`
  if $b \notin B_{sd}$ **then**
    foreach $s \in \mathrm{succ}_E(b)$ **do**
      $\mathrm{next} \leftarrow \min(T \cup \{s\})$;
      if $\mathrm{next} \neq s$ **then**
        $G \leftarrow G \cup (b, \mathrm{next})$; `// Deferral edge becomes ghost edge`
        $A \leftarrow A \cup (b, s)$; `// Add this outgoing edge to A`
      end
      `//` $E_l \leftarrow E_l \cup \{(b, \mathit{next})\}$`;`

      `// Add deferral edges from next to this successor or deferral targets`
      $D \leftarrow D \cup \{(\mathrm{next}, t) \mid t \in (T \cup \{s\}) \setminus \{\mathrm{next}\}\}$;
    end
  else
    $S \leftarrow \mathrm{succ}_E(b)$;
    $\mathrm{next} \leftarrow \min(T \cup S)$;
    if $\mathrm{next} \in T \setminus S$ **then**
      $G \leftarrow G \cup (b, \mathrm{next})$; `// Deferral edges become ghost edges`
    end
    `//` $E_l \leftarrow E_l \cup \{(b, \mathit{next})\}$`;`

    `// Add other outgoing edges to A`
    $A \leftarrow A \cup \{(b, s) \mid s \in S \setminus \{\mathrm{next}\}\}$;
    `// Add deferral edges from next to other successors or deferral targets`
    $D \leftarrow D \cup \{(\mathrm{next}, t) \mid t \in (T \cup S) \setminus \{\mathrm{next}\}\}$;
  end
  `// Remove all outgoing deferral edges`
  $D \leftarrow D \setminus \{(b, s) \mid (b, s) \in D\}$;
end
assert $T \neq \emptyset$;

outgoing deferral edges of block 1 *before moving on to the next block, resulting in Figure 5.5c.*

The next block we visit is block 2*. We also assume that this block does not contain a secret-dependent branch. However, deferral target* $4 \in T$ *has a lower number than the successor* 5 *of block* 2*. This means that deferral target* 4 *will be chosen as successor "next", and a ghost edge will be created from* 2 *to* 4*. Any other outgoing edge to a successor with a higher number will become an activating edge. We also create a deferral edge from* 4 *to* 5*. This results in Figure 5.5d.*

When visiting block 3 *and block* 4*, we find that the single successor of the branch matches the deferral target. In this case, nothing is changed, and we can simply remove the deferral edge.*

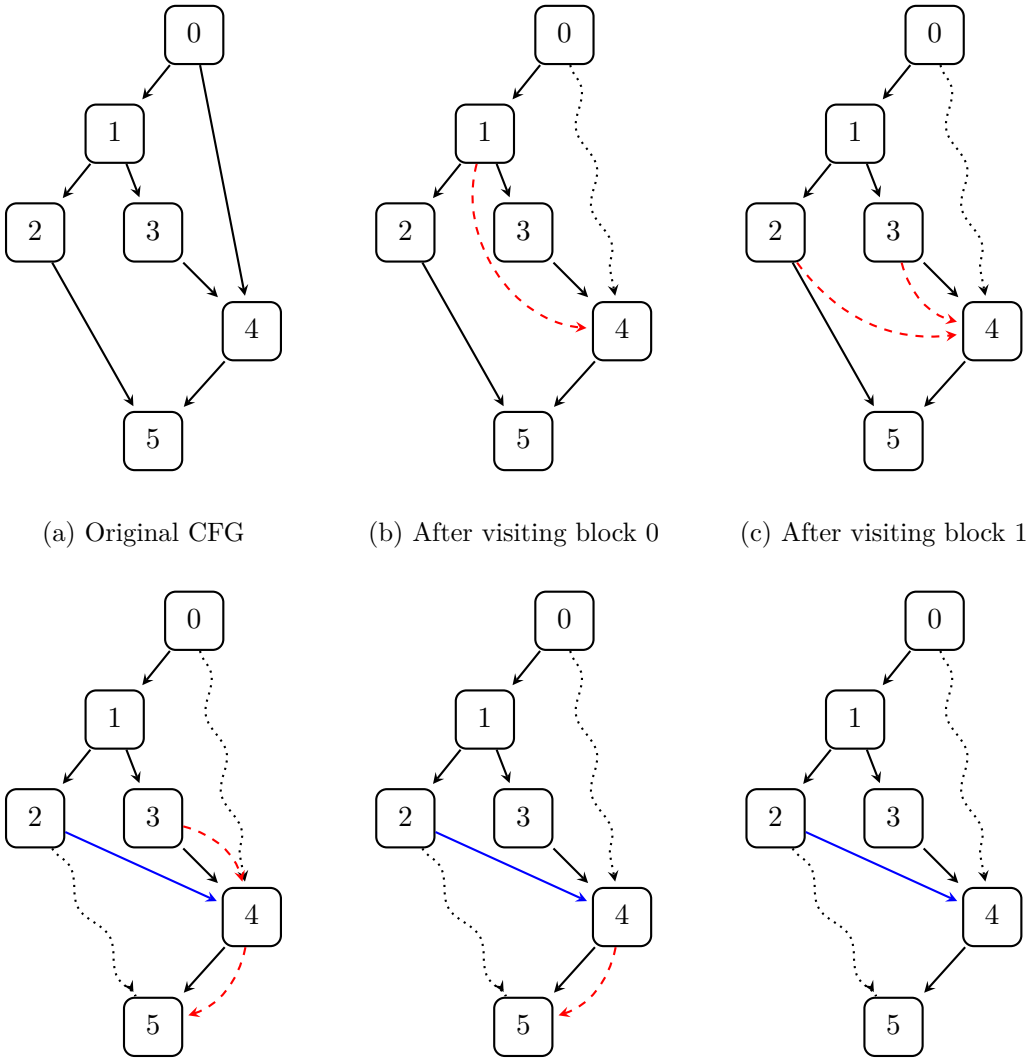
5.3 Rewriting Branch Terminators

The methods described in the previous section determine which edges should be removed from the CFG (activating edges), and which edges should be added to the CFG (ghost edges) in order to linearize the program. This section describes how given a CFG, a set of activating edges A, and a set of ghost edges G, appropriate activating instructions and branches can be inserted in the linearized program. We recall that for each activating edge (B, B'), there must be some activating instruction in B with B' as target. Additionally, for each ghost edge (B, B'), B unconditionally enables mimicry mode on exit.

With this in mind, activating instructions for each block can be determined. Figures 5.6 and 5.7 give an exhaustive overview of how the terminators must be chosen given sets A and G. In some cases of this exhaustive overview, side channel leakage of the branch condition is mitigated, while in other cases this is not possible due to the requirements of a ghost edge. In general, leakage is mitigated if the conditional branch of a block is replaced with an activating branch, which is the case if the block has a single successor in the linearized CFG. All methods introduced in the previous section ensure that this holds if the block contains a secret-dependent branch, to ensure that leakage of secret branch conditions is always mitigated.

For certain instantiations of A and G, **beqz** (branch if equal to zero) must be replaced with **bnez** (branch if not equal to zero) and the targets of the conditional branch and the jump need to be swapped. For example, if a **beqz** instruction at l_0 has l_a as target, but $(l_0, l_b) \in A$, we cannot replace this branch with an activating branch **a.beqz**, since we do no want $l_0 \to l_a$ to become an activating edge. By replacing **beqz** c, l_a and **jmp** l_b with **bnez** c, l_b and **jmp** l_a, program semantics are preserved, and **bnez** can be replaced with an activating branch.

Example 14. *Recall the linearized program from the example in Figure 5.5f and assume that the terminators in the original CFG shown in Figure 5.5a are given by Figure 5.8a. For blocks* 1*,* 3 *and* 4*, no outgoing edges changed*



(a) Original CFG (b) After visiting block 0 (c) After visiting block 1

(d) After visiting block 2 (e) After visiting block 3 (f) After visiting block 4

FIGURE 5.5: Example of algorithm 1. Dotted edges are activating edges added to A, and will no longer be part of the CFG. Red dashed edges are used to represent the deferral edge. Any other edge that is added to the CFG is a ghost edge, as marked in blue (in this example, the edge from 2 to 4 is a ghost edge).

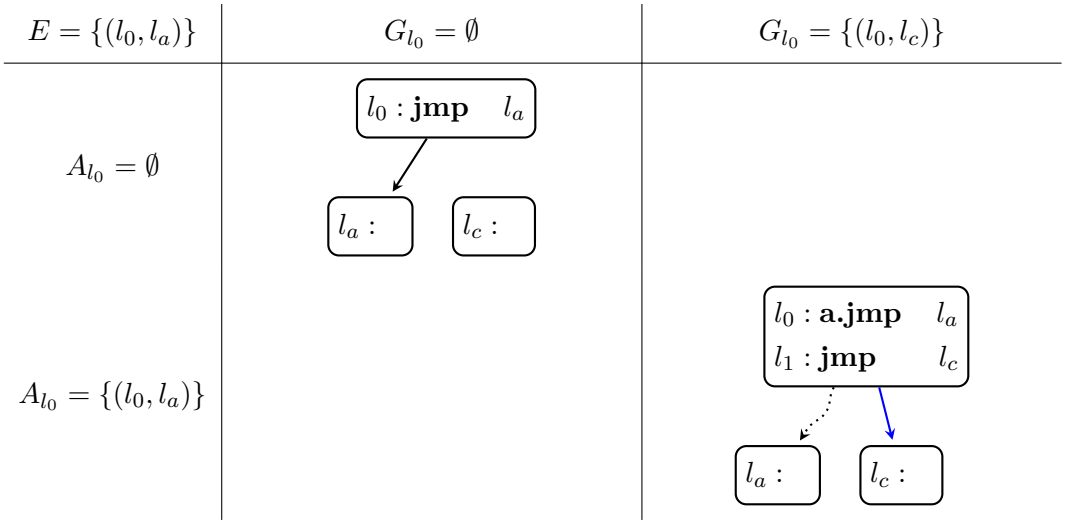


FIGURE 5.6: Overview of activating instructions and branches that must be chosen for the given sets of activating edges A and ghost edges G if the original branch block contains a single unconditional jump.

> *compared to the original CFG, hence we can use the same terminators as in the original CFG.*
>
> *Block* 0 *has a single outgoing control-flow edge, and a single outgoing activating edge. As given by the first column in Figure 5.7, we need an activating branch, followed by an unconditional jump. Since the target of the activating edge is also the target of the* ***beqz*** *instruction in the original CFG, we replace the* ***beqz*** *instruction with an activating branch. Because we have replaced* ***beqz*** c_1 *with* ***a.beqz*** c_1*, leakage of* c_1 *through control flow is mitigated against.*
>
> *Block* 2 *has an outgoing ghost edge, and an outgoing activating edge. This case is described by Figure 5.6, hence we replace the existing* ***jmp*** *instruction with an activating jump, and add another jump instruction for the ghost edge.*

Some instantiations of the sets A and G are invalid, in which case no appropriate instructions can be chosen. For example, if there are no activating edge, but an edge is added to the linearized CFG, this edge can never be a valid ghost edge, because ghost edges require that mimicry mode is enabled. Similarly, if all outgoing edges of a block in the original CFG are removed from the CFG, because they are marked as activating edges, and no ghost edges are added, there is no valid instantiation of terminators, since the block in the linearized CFG would have no successors, yet there are outgoing activating edges whose target must be reached eventually. Such cases of invalid instantiations of sets A and G do not occur when applying the linearization methods described in section 5.2.

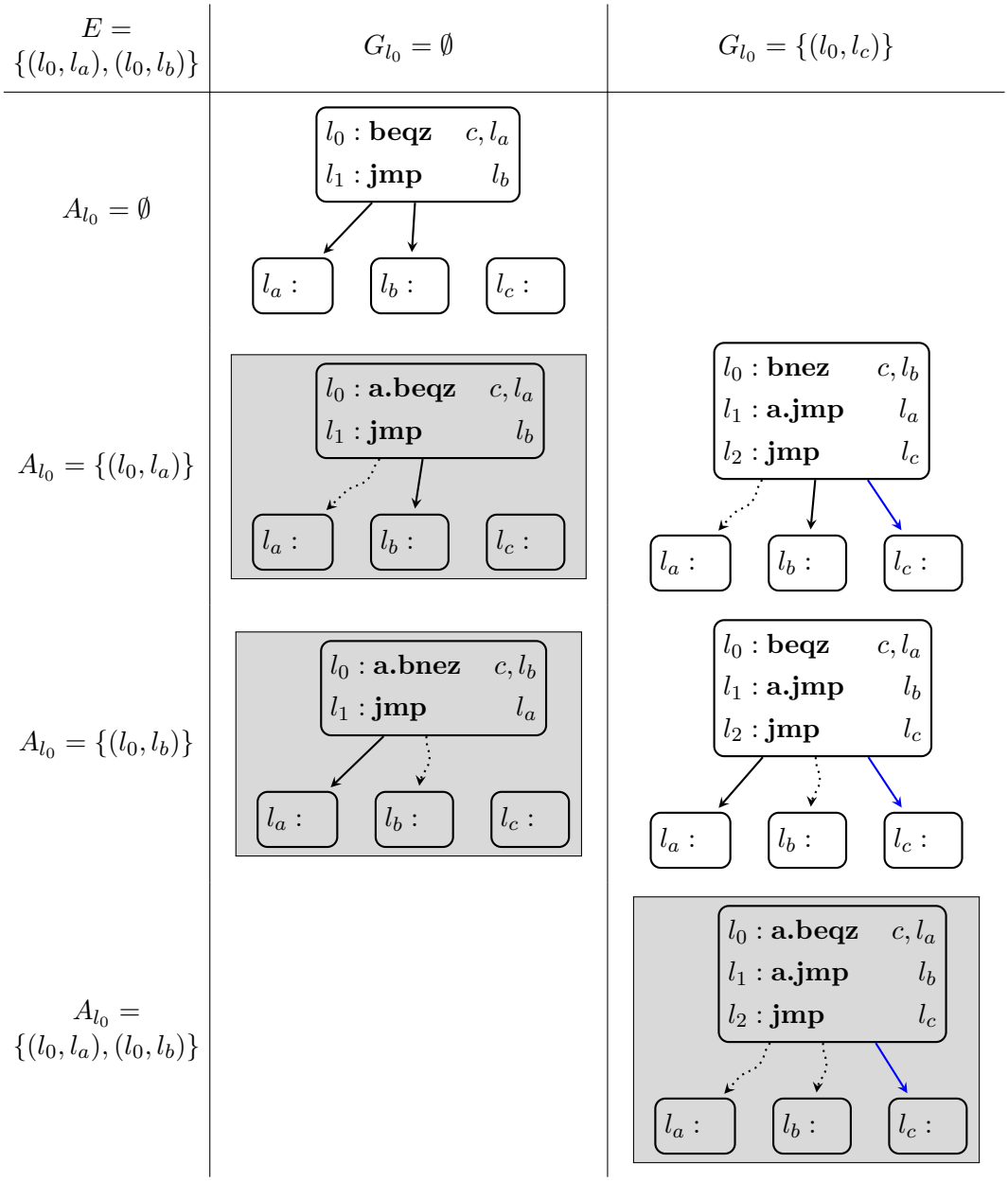
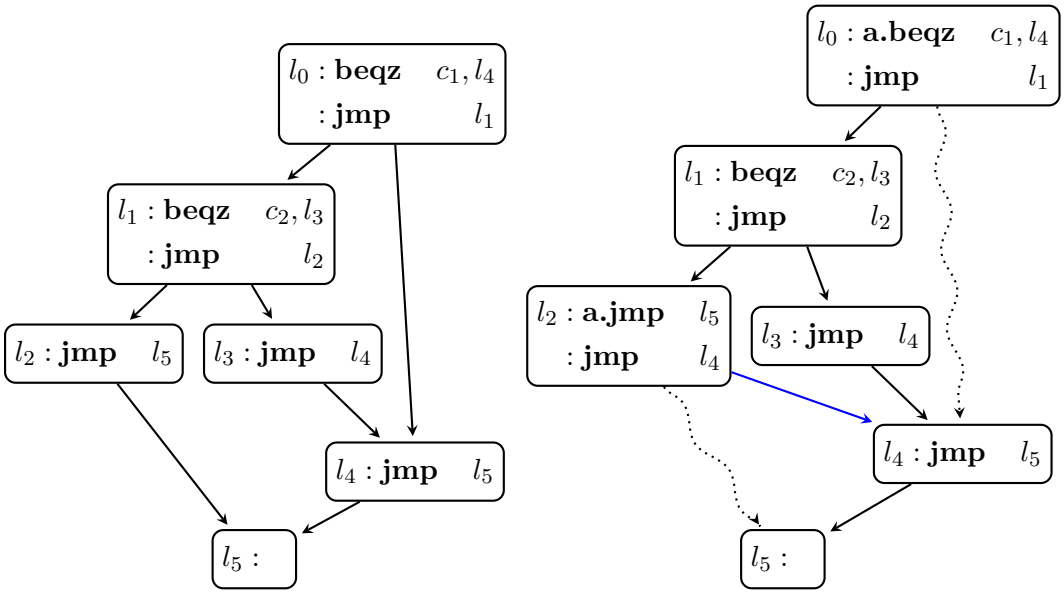


FIGURE 5.7: Overview of activating instructions and branches that must be chosen for the given sets of activating edges A and ghost edges G when the branch block contains a conditional branch (assuming **beqz** without loss of generality). The cases in which side-channel leakage of the branch is mitigated is shown with boxes.



(a) Original CFG (b) Linearized CFG

FIGURE 5.8: Example of a CFG and its linearized version with terminators

5.4 Correctness

This section formalizes the correctness of the linearization methods described in Section 5.2. First, Section 5.4.1 relaxes the well-behavedness proposition of [46]. Section 5.4.2 formalizes the correctness of the regions over which activating branches jump. To achieve correct regions, register allocation needs to be constrained, as explained in Section 5.4.3. Finally, Section 5.4.4 formalizes and proves the correctness of the linearization methods under the assumption that all activating regions are well-behaved and correct.

5.4.1 Well-behaved Activating Region

Replacing branches with activating branches is only valid if the region spanned between the successors and the target of the activating branch is SESE, as described in the well-behavedness proposition in [46]. This limits the applicability of the linearization pattern discussed in previous sections to triangle-structured control flow, since method 1 is the only linearization method that results in well-behaved activating regions. In order to support the linearization methods introduced in this thesis, the well-behavedness proposition and the hypotheses under which it holds, as defined by [46], must be relaxed, as motivated by the following example.

Example 15. *Recall the linearized CFG from Figure 5.5f (linearization method*

3), which contains two activating regions, namely $[0, 4]$ *and* $[2, 5]$*. The well-behavedness proposition from [46] only holds for activating regions that are SESE, where they consider activating edges to be part of the CFG. The activating regions in this example are not SESE, since* $[0, 4]$ *has a second exiting edge* $2 \to 5$*, and* $[2, 5]$ *can also be entered through* $3 \to 4$ *and* $0 \to 4$*. However, this program is still correct, and therefore it would be desirable to relax the conditions for the well-behavedness of an activating region, such that these regions would also be considered well-behaved*

We relax the second hypothesis from [46] that states that activating regions are SESE by (1) not including activating edges in the CFG, and (2) removing the assumption that the entry of an activating region dominates the exit. As a consequence, the well-behavedness proposition from [46] no longer holds. Hence, we relax this proposition and adapt the proof accordingly.

To achieve this relaxation, we change the notion of a CFG used by [46], and replace it with the semantics of the successor relation defined in section 2.3.1, which does not consider activating edges to be control-flow edges. The motivation for this change is that activating edges do not affect control flow, and execution will always continue at the subsequent location in the program as stated in figure 4.3. This change weakens the second hypothesis, which states that each activating region is SESE. To give an example of (1), consider the linearized program in figure 5.4b from method 2. The activating region $[0, 5]$ would not be considered well-behaved by the original formulation of the proposition, since it is not SESE if activating edges are considered control-flow edges. If we don't consider them as part of the CFG, the activating region is considered SESE, and hence well-behaved. Furthermore, an activating region does not need to be single-entry (2), since entering the region through paths that don't contain the entry (i.e. the activating instruction) does not affect the behavior of the activating region. However, to ensure that nested activations behave well, each cycle containing the entry of the activating region must also contain the exit. We replace the second hypothesis of [46] with the following:

Hypothesis 2 (Well-behavedness Criterion)**.** *For each activating region* $[l, l']$ *in the program, the following holds:*

- l' *post-dominates* l
- *each cycle containing* l *also contains* l' *and vice-versa*

As a consequence of these relaxations, proposition 1 (the well-behavedness proposition of [46]) no longer holds under this altered version of the second hypothesis, since early termination of an activating region is possible.

Example 16. *Consider the linearized CFG in Figure 5.5f. Hypothesis 2 holds for activating region* $[2, 5]$*. However, Proposition 1 does not hold.*

Suppose that the activating branch in block 0 *enabled mimicry mode, setting the mimicry exit address* $Ex = 4$*, and block* 2 *is reached during mimic execution.*

> *The activating jump in block* 2 *does not affect the mimicry exit address Ex, and the ghost edge from* 2 *to* 4 *is taken. Proposition 1 states that AC must remain set for the duration of* $[2, 5]$*. However, since Ex is set to* 4*, AC will be unset when reaching block* 4*, before* 5 *is reached, resulting in early termination of the activating region.*

Such early termination of an activating region is explicitly disallowed by the formalization of [46]. However, early termination does not necessarily lead to incorrect behavior of the linearized program, hence it should be allowed. To rectify this limitation, the well-behavedness proposition can be revised such that it only applies to regions $[l, l']$ where AC was incremented by the activating instruction at l.

Proposition 2 (Well-behaved Activating Region (revised))**.** *For any activating region* $[l, l']$ *and valid configuration* σ *such that* $[l, l']\sigma \Downarrow_o \sigma'$*, if AC is incremented at location* l*:*

1) it remains set during the execution of $[l, l']$ *(including recursive function calls but excluding* l'*), and*

2) it is restored to its initial value during the evaluation of the instruction at location l'*.*

This revised proposition for a well-behaved activating region enables the linearization methods for structured and reducible control flow introduced in this thesis. The proof of [46] is adapted to this proposition in appendix section A.2.

The following proposition states that the application of algorithm 1 results in only well-behaved activating regions under the assumption that the source CFG is reducible, and each exiting block of a loop does not contain a secret-dependent branch.

Proposition 3 (Linearization Results in Well-behaved Activating Regions)**.** *Algorithm 1 (method 3) results in well-behaved activating regions.*

A proof is given in Appendix A.3 of Appendix A, and we note that a similar argument can be made for the other two linearization methods.

5.4.2 Correct Region

The activating regions that result from the application of these linearization methods are only correct under the assumption that the live state is unaffected when executed in mimicry mode. The live state is a partition of the register bank and the memory map. We define this partition as follows.

Definition 5 (Live state)**.** Given a program P, the *live state* of the program at location l, denoted as $\mathcal{L}_P^l$ is a function that maps an AMi configuration (or architectural configuration) σ to a set of live registers and memory locations such that

- For each $(x, v) \in \sigma.r$, $(x, v) \in \mathcal{L}_P^l$ if and only if live(l, x) in P.

- We assume a liveness partition for memory locations.

In other words, a register is part of the live state if it is live according to the liveness defined in Section 2.4.3. Based on this notion of the live state of the program, two configurations can be compared.

Definition 6 (Live-equivalence)**.** Two configurations σ and σ' in respective programs P and P' are *live-equivalent* if $\mathcal{L}_P^{\sigma.\mathrm{pc}}(\sigma) = \mathcal{L}_P^{\sigma.\mathrm{pc}}(\sigma')$ and $\mathcal{L}_{P'}^{\sigma'.\mathrm{pc}}(\sigma) = \mathcal{L}_{P'}^{\sigma'.\mathrm{pc}}(\sigma')$. Notation: $\sigma{=_{\mathcal{L}}}\sigma'$.

This definition states that when comparing two configurations in two different programs, registers or memory locations must have the same values if they are considered live by one of the programs at the respective program counters in each program. Note that P may be equal to P', in which case the live state at different locations in the same program can be compared.

Some instructions have architectural side effects and may affect the live state, even in mimicry mode, notably persistent instructions and ghost instructions. In particular, store operations are always persistent, meaning that they will always write to memory. In mimicry mode, such undesired side effects should be prevented.

Recall that Proposition 2 relaxes the conditions under which an activating region is considered well-behaved. Similarly, the correctness definition from [46] can be relaxed such that it only applies to a region $[l_1, l_n]$ if AC was incremented right before entering this region.

Definition 7 (Correct region)**.** Let $[l_0, l_n]$ be a well-behaved, terminating activating region. The terminating region $[l_1, l_n]$ is correct if the following holds for any (m, r). If AC was incremented at l_0 and $[l_1, l_n]\langle m, r, l_1, 1, l_0, l_n\rangle \Downarrow_o \langle m', r', l_n, 0, -, -\rangle$, then $\langle m, r, l_1\rangle{=_{\mathcal{L}}}\langle m', r', l_n\rangle$.

Intuitively, this states that if mimicry mode was enabled at the start of an activating region, the live state of the program should not be affected by the mimic execution of this region. The following proposition gives a criterion for an activating region to be correct, which we state without proof.

Proposition 4 (Region Correctness Criterion)**.** *A region $[l_0, l_n]$ is correct if the following holds for any location l in the region.*

- *if l contains a store instruction* ***store*** *v, a, then $l - 1$ contains a ghost load* ***g.load*** *v, a*
- *if l contains a persistent instruction or a ghost instruction that defines x, then x is not part of the live state at l.*

The pattern in Figure 5.9 can be applied to nullify a store operation, as is also illustrated by Winderix et al. [46]. By adding a ghost load before a store operation of register x to location v, we ensure that in mimicry mode, the original value stored at location v is loaded into x before storing x back to v, nullifying the effect of the store.

l_0 :		l_0 : **g.lw**	x, v
l_1 : **p.sw**	x, v	l_1 : **p.sw**	x, v
l_2 :		l_2 :	

a) Original instruction b) After adding ghost load

FIGURE 5.9: Ghost load pattern

l_0 :		l_0 : **s.add**	$y, x, 0$
l_1 :		l_1 : **g.lw**	y, v
l_2 : **p.sw**	x, v	l_2 : **p.sw**	y, v
l_3 :		l_3 :	

a) Original instruction b) After adding ghost load

FIGURE 5.10: Ghost load pattern (with free register y)

Note that since x may be part of the live state, it would not always be correct to define this register with a ghost load. Therefore, it may be necessary to use a free register y that is not part of the live state of the activating region, as illustrated in Figure 5.10. Here, we copy register x to free register y (**add** $y, x, 0$) in standard execution mode, while loading from location v to y in mimicry mode. Instead of storing register x, we instead store register y, which contains either the value stored in x, or the value stored at location v, depending on the mode of execution.

However, program hardening is applied after registers have already been allocated, meaning that such free register y may not be available. To address this, Section 5.4.3 describes ways to constrain the register allocation to ensure that such register is reserved, or to ensure that register x used by a store operation is not part of the live state. Similarly, register allocation constraints are used to ensure that persistent instructions don't define registers that are part of the live state in the linearized program.

5.4.3 Register Allocation Constraints

As mentioned in the previous section, persistent instructions or ghost instructions within an activating region may break the correctness of that activating region, because linearization is applied after register allocation. First we motivate why

linearization cannot be applied before register allocation. Usually, compilers use SSA form before register allocation. Unfortunately, it is not possible to express an AMi-linearized program in SSA form, since ϕ-nodes select between results depending on the predecessor location during execution, but they cannot be used to select between results from different activating regions. Blocks in a linearized program only have a single predecessor, since activating edges are not considered to be part of the CFG. If one would try to express the linearized program before register allocation without using SSA form, liveness analysis would still yield undesirable results. Consequently, register allocation would lead to an incorrect program. This is illustrated by the following example.

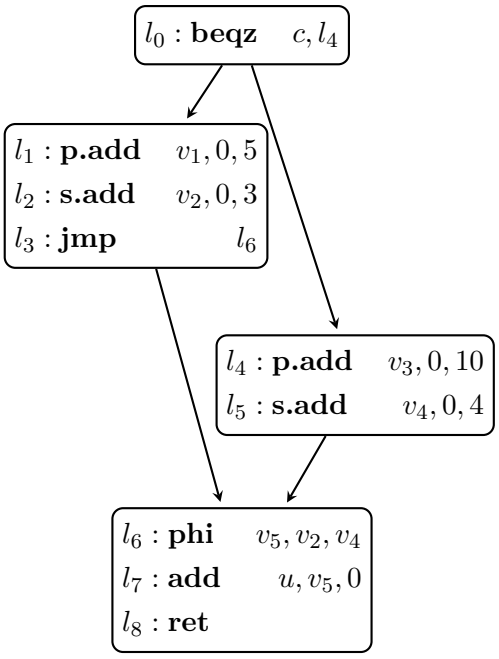
Example 17. *Consider the CFG in Figure 5.11a where a ϕ-node is used to select between v_2 and v_4 depending on the predecessor during execution. Before register allocation is applied, ϕ-nodes are eliminated, resulting in Figure 5.11b.*

Suppose that one of the linearization methods would be applied, resulting in Figure 5.11c. When computing the liveness of variables, the first definition of v_2 at l_2 would be considered dead, since there are no uses before the next definition of v_2 at l_5. On each path from l_2 to the function exit, l_5 is reached, and there is no use of v_2 before l_5. As a consequence, the register allocator may assign the same physical register to v_2 and v_3, since v_2 is not live at l_4 where v_3 is live, resulting in Figure 5.11d. This program is incorrect, since physical register x is overwritten by the persistent instruction at l_4.

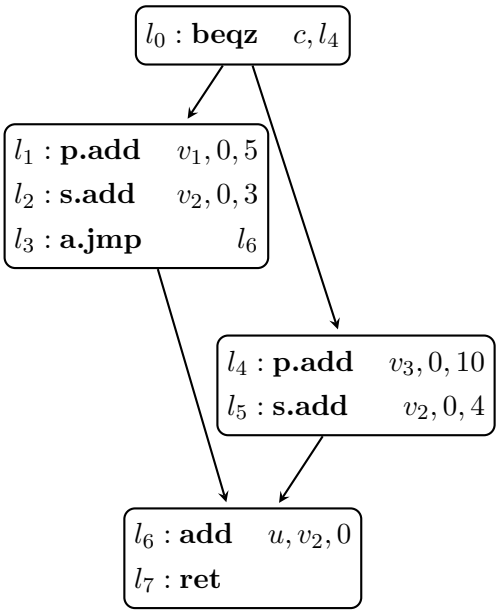
The intended behavior of the program is that either the instruction l_2 or the one at l_5 is executed while the other is mimicked. That way, only one of those instructions would define v_2, or x. But this cannot be expressed using the classical notion of liveness, where the first definition of v_2 at l_2 is considered dead, since it would always be redefined at l_5. This is not how standard AMi instructions behave in practice.

As a result, linearization must be applied after register allocation if we do not integrate AMi semantics in the compiler. However, linearization after register allocation may still break the correctness of the program, as illustrated by the following example.

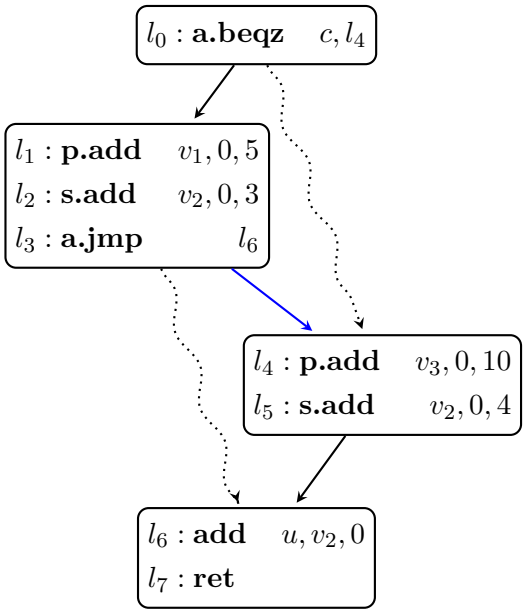
Example 18. *Figure 5.12 illustrates how linearization can lead to an incorrect program. In this example, the regions that follow the activating regions are not correct, because register x is part of the live state of these regions. Indeed, register x is live out of the block at l_2 and live into the block at l_4. The persistent instructions at l_1 and l_4 define register x, and this effect is also applied in mimicry mode. Hence, the correctness of the activating region is broken, since the live state of the activating region is modified in mimicry mode. In this example, this means both linearized forms will return* 18 *or* 19 *depending on c, while the original program returns either* 8 *or* 14.



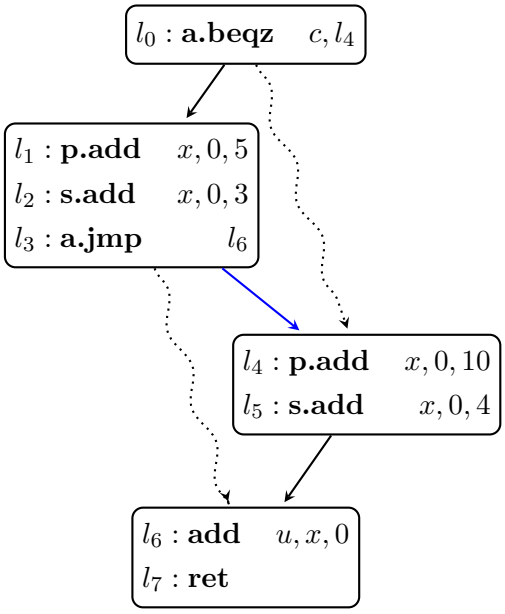
(a) A CFG before register allocation



(b) A CFG before register allocation after eliminating ϕ-nodes

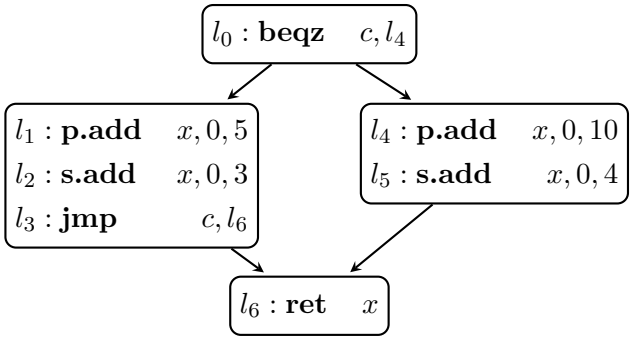


(c) A linearized CFG before register allocation where $x \in \text{Regs}_p$

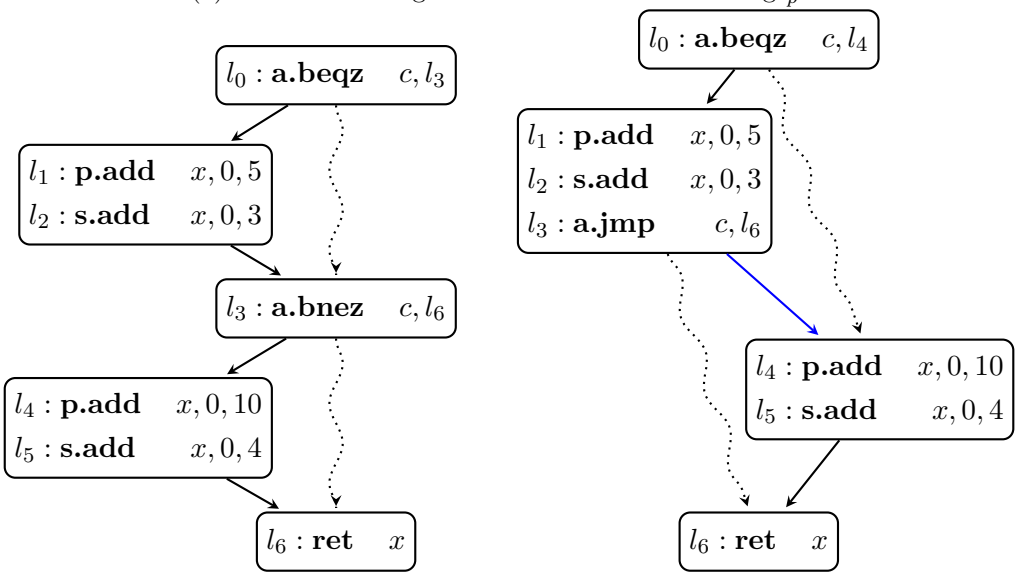


(d) A linearized CFG after register allocation

FIGURE 5.11: Illustration of problems with AMi and register allocation ($c, u \in \text{Regs}_p$ are input and output registers respectively)



(a) A CFG after register allocation where $x \in \text{Regs}_p$



(b) Two linearized versions of the CFG

FIGURE 5.12: A CFG after register allocation (a), and two linearized versions of this CFG (b)

Originally, the interference graph only includes interference edges between virtual registers if there is a location at which both are live. To solve the issue described in this section, additional constraints must be added to the interference graph of the original CFG. Specifically, every variable that is live on an activating edge in the original CFG (live-out of the branch block and live-in of the target block) conflicts with every variable defined by a persistent instruction or ghost edge in the activating region of that edge, in addition to the existing constraints. This ensures that the instructions within the region don't affect the live state at the boundaries of the region, hence the region is correct according to Definition 7.

Example 19. *We illustrate how additional constraints are added on the exam-*

ples in Figure 5.11, and will apply register allocation on Figure 5.11a. Here, the interference graph initially has no constraints between the virtual registers, due to the minimalistic nature of the example. In more realistic examples, the interference graph will already include constraints between virtual registers that are live at the same location. Applying register allocation on this CFG without additional constraints leads to incorrect programs after linearization, as illustrated in Figure 5.12.

To find the constraints that need to be added, we first determine how we will linearize after register allocation. That way, we can identify all activating edges and activating regions, as shown in Figure 5.11c, without committing this linearization to the CFG. Based on this information, we determine additional constraints as follows. Consider the control-flow edge $l_3 \to l_6$ in Figure 5.11b, which will become an activating edge in the linearized program. Since v_2 is defined at l_2 and used at l_6, it is live on this edge, hence it conflicts with every variable defined by a persistent instruction in the activating region of this edge. The activating region is $[l_3, l_6]$, which contains a persistent instruction at l_4 that defines v_3, thus we add a constraint between v_2 and v_3 to the interference graph. No variables are live on the control-flow edge $l_0 \to l_4$, hence there are no registers that interfere with the persistent instruction at l_1.

Applying register allocation while taking these additional constraints into account will ensure that different physical registers are assigned to v_2 and v_3, resulting in correct activating regions.

A shortcoming of Definition 7 is that anything defined persistently within a correct region cannot outlive this region, otherwise it would be live at the exit, and hence the initial configuration would not be live-equivalent with the configuration after executing the region.

5.4.4 Linearization Correctness

This section described the correctness of applying the linearization methods described in Section 5.2. First, we can define the correctness of a transformation.

Definition 8 (Correct program transformation)**.** A program transformation T is correct if the following holds for any program P and configuration σ. If $P\ \sigma \Downarrow_o \sigma_n$ and $T(P)\ \sigma \Downarrow_o \sigma'_n$, then $\sigma_n =_{\mathcal{L}} \sigma'_n$.

Intuitively, this states that both the original program P and the transformed program $T(P)$ have the same effect on the live state. The linearization methods described in Section 5.2 are AMi linearizations, as defined by Definition 3. We state when AMi linearizations are correct program transformations.

Theorem 1 (Correctness of AMi Linearization)**.** Let P be a program and T an AMi linearization as defined by Definition 3. Then T is correct if for each activating region $[l_x, l_y]$ in $T(P)$, $[l_x, l_y]$ is well-behaved (Proposition 2), and $[l_{x+1}, l_y]$ is correct (Definition 7).

A sketch of a proof by induction on the amount of big-step evaluation steps is given in Appendix A.1. The idea behind this inductive proof is that we take single evaluation steps in the case the instruction at $T(P)[l_n]$ is not activating, since the instructions in both programs will be equal. If the instruction at $T(P)[l_n]$ is activating, we either don't enable mimicry mode, in which case the instruction behaves identically to the one at $P[l_n]$, or we enable mimicry mode for the duration of the activating region. If mimicry mode is enabled, we apply the definition of a correct region (Definition 7) to skip to the end of the activating region where mimicry mode is disabled again (since the activating region is assumed to be well-behaved as part of the induction hypotheses), hence a large step is taken during the induction. In the original program, the branch at $P[l_n]$ will be taken, and execution continues at the target location. In both cases the live state is unaffected, and we continue the induction at the target location. This means that at any inductions, mimicry mode is disabled, and the inductive argument still holds regardless of how many ghost edges are added by the linearization, since the induction always jumps past the ghost edges. This way the intuitive idea of "ghost edges don't break correctness" is formalized into a proof.

Linearization Methods are AMi Linearizations

Finally, we show that the methods described in Section 5.2 are AMi linearizations. When combined with the result of Proposition 3, the correctness theorem applies.

Recall Definition 3, which states that an AMi linearization can only (1) replace control-flow edges with activating edges or (2) add ghost edges. On the one hand, when linearizing triangle-structured control flow (method 1), no additional edges are created, but some edges are replaced with activating edges. On the other hand, when linearizing structured control flow (method 2), we make use of ghost edges from the exiting blocks of one branch region to the branch target. Additionally, we make sure that all other outgoing edge of the exiting block are replaced with activating edges. As can be seen in the lower right cases in Figures 5.6 and 5.7, this results in the creation of an activating jump, hence mimicry mode is enabled unconditionally, and the newly added edges are valid ghost edges.

A similar argument can be made when linearizing reducible control flow (method 3) with Algorithm 1. Activating edges (b, s) are only added if $b \to s$ is already a control-flow edge of the original CFG. Whenever ghost edges are added to set G, we ensure that all other outgoing edges become activating edges by adding them to set A. Hence, mimicry mode will always be enabled when leaving a block through a ghost edge.

5.5 Security

In order to satisfy the security objective defined in Section 4.2, execution of an activating region in mimicry mode must be indistinguishable from normal execution of said region, meaning that both executions must leak the same observations. We adopt the notion of a *secure region* as defined by Winderix et al.

Definition 9 (Secure region)**.** Let $[l_0, l_n]$ be a well-behaved, terminating activating region. The terminating region $[l_1, l_n]$ is secure if the following holds for any low-equivalent $(m, r){=_\mathcal{P}}(m', r')$, and configurations $\sigma_0 = \langle m, r, l_1, 0, -, - \rangle, \sigma_0' = \langle m', r', l_1, 0, -, - \rangle, \sigma_0'' = \langle m', r', l_1, 1, l_0, l_n \rangle$:

1. if $[l_1, l_n]\sigma_0 \Downarrow_o \sigma_1$ and $[l_1, l_n]\sigma_0' \Downarrow_{o'} \sigma_1'$ then $o = o'$ and $\sigma_1{=_\mathcal{P}}\sigma_1'$
2. if $[l_1, l_n]\sigma_0 \Downarrow_o \sigma_1$ and $[l_1, l_n]\sigma_0'' \Downarrow_{o''} \sigma_1'$ then $o = o''$ and $\sigma_1{=_\mathcal{P}}\sigma_1''$

The first condition states that $[l_1, l_n]$ is secure by itself, meaning that it should not leak any secret into the public part or as observations. Taint analysis can be used to determine the flow of secret data, as explained in Section 6.6. If through data dependencies or control dependencies, secret data flows to a certain register of memory location, this register or memory location should also be considered secret. Since secret data should not be leaked through microarchitectural observations, a secure program must not use a secret register to compute a memory address, or to compute a branch condition. To mitigate against such leakage, we reject programs with secret-dependent memory accesses, and we eliminate all secret-dependent branches by linearizing them.

The second condition states that the region produces the same microarchitectural observations regardless of whether mimicry mode is enabled. To achieve this, the same memory locations must be accessed regardless of the state of AC, and the same path in the control flow graph must be taken in each case. As described by Winderix et al. [46]: "if a value can leak to the microarchitectural state, any architectural update that this value depends on should always be executed regardless of the processor mode." In the following section, we elaborate on this and describe which instructions need to be executed persistently.

5.5.1 Persistent Computation and Taint Analysis

To determine which branches need to be linearized, we apply taint analysis to identify secret-dependent branches. Data dependencies must certainly be taken into account, since most operations can be inverted by an attacker. For example, if x is secret, and y is defined by an instruction that adds 5 to x, leakage of y enables an attacker to derive x by subtracting 5 from the leaked value. As a result, y should be considered secret.

Similarly, *indirect flows* should be taken into account, since registers defined by an instruction that is control-depedent on a branch may leak information about the value of the branch condition. For example, if x gets value 5 only when a secret-dependent branch is taken, leakage of x enables an attacker to determine whether this secret-dependent branch was taken, hence the attacker learns something about a secret value. Unfortunately, considering taint flow through all indirect flows has undesirable consequences, as is illustrated by the following example.

Example 20. *Consider the program in Figure 5.13a, and suppose that the branch at l_1 is secret-dependent. This branch needs to be linearized using an activating branch, resulting in activating region $[l_1, l_4]$. Within this region is a loop with a latch and exiting branch at l_3.*

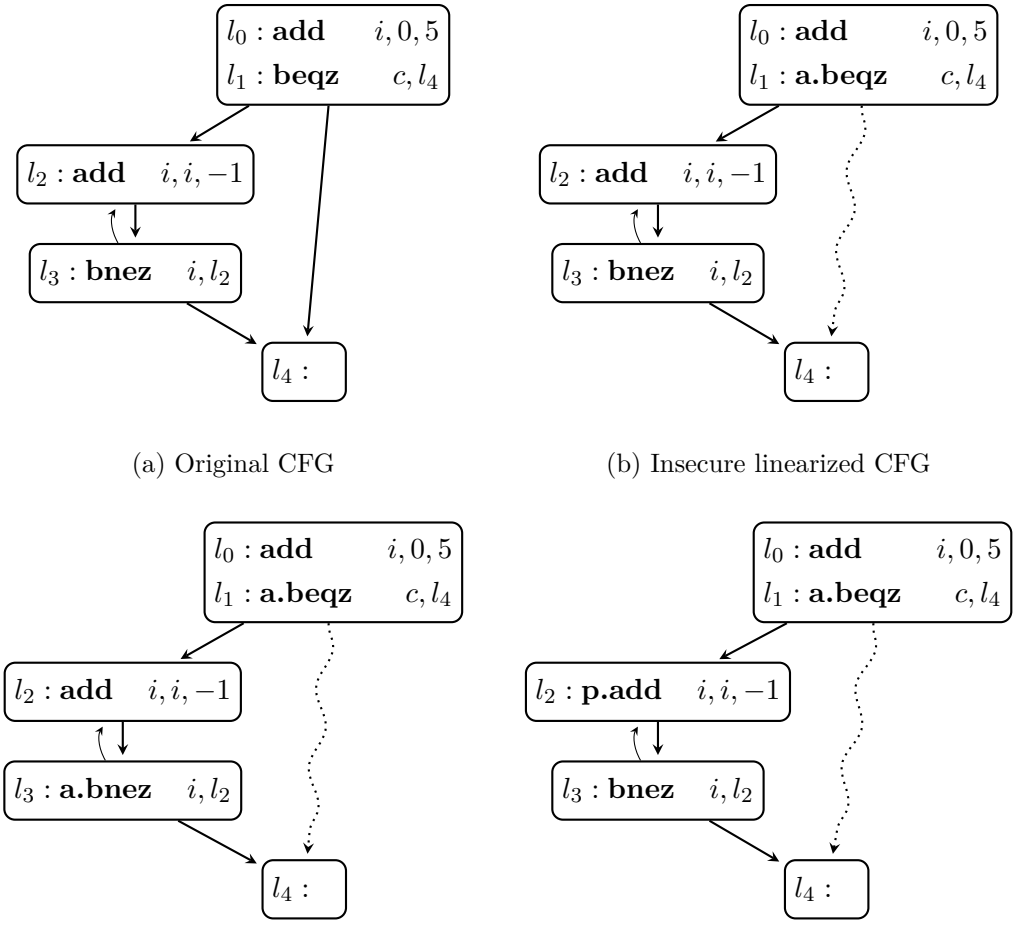
Simply replacing the branch at l_1 with an activating branch results in the insecure program in Figure 5.13b. If the activating region is executed normally, i will be decremented five times before exiting the loop. However, if the activating region is mimicked, the instruction at l_2 will always be mimicked, hence i is never decremented and the program does not terminate. An attacker can observe this, since we assume control flow leaks on side channels. Note that our current formalization does not state whether this program is secure, nor if it is correct, since we only define when a terminating region is secure or correct.

To avoid such leakage through control flow, there are two options. First, suppose that we took all indirect flows into account. In this case, i would be control-dependent on the secret-dependent branch at l_1, since its value depends on whether this branch is taken. Consequently, the loop latch at l_3 would be considered secret-dependent, but linearizing this branch as shown in Figure 5.13c is incorrect, since its target l_2 does not post-dominate l_3, and hence the activating region is not well-behaved.

Second, suppose that we do not consider i to be secret, but instead we make sure that the branch condition of the latch at l_3 is computed persistently, as illustrated in Figure 5.13d. We now know that the activating region will terminate regardless of whether mimicry mode is activated, since i is decremented in both modes of execution. Furthermore, the same amount of iterations is executed in both mimicry mode and standard execution mode. Even if i is leaked to an attacker each iteration, the attacker is unable to determine whether mimicry mode is active, hence the secret c is not leaked.

Any instruction within an activating region that affects the condition of a persistent branch or the memory address of a load or store operation should be executed persistently, to ensure that the region is secure. However, if part of this computation takes place in another activating region that precedes this persistent branch or memory operation, the correctness of that region will be broken. The following example illustrates why we need to take indirect flows into account during taint analysis.

Example 21. *Suppose that a (persistent) branch is data-dependent on some register x, and x is computed within an activating region, as illustrated in Figure 5.14b. If we would not take indirect flows into account, x would not be considered secret and the branch remains persistent. Therefore, we must ensure that x (i.e. the condition of this branch) is computed persistently, resulting in the persistent instruction at l_2. However, this persistent instruction breaks the*



(a) Original CFG

(b) Insecure linearized CFG

(c) Incorrect linearized CFG

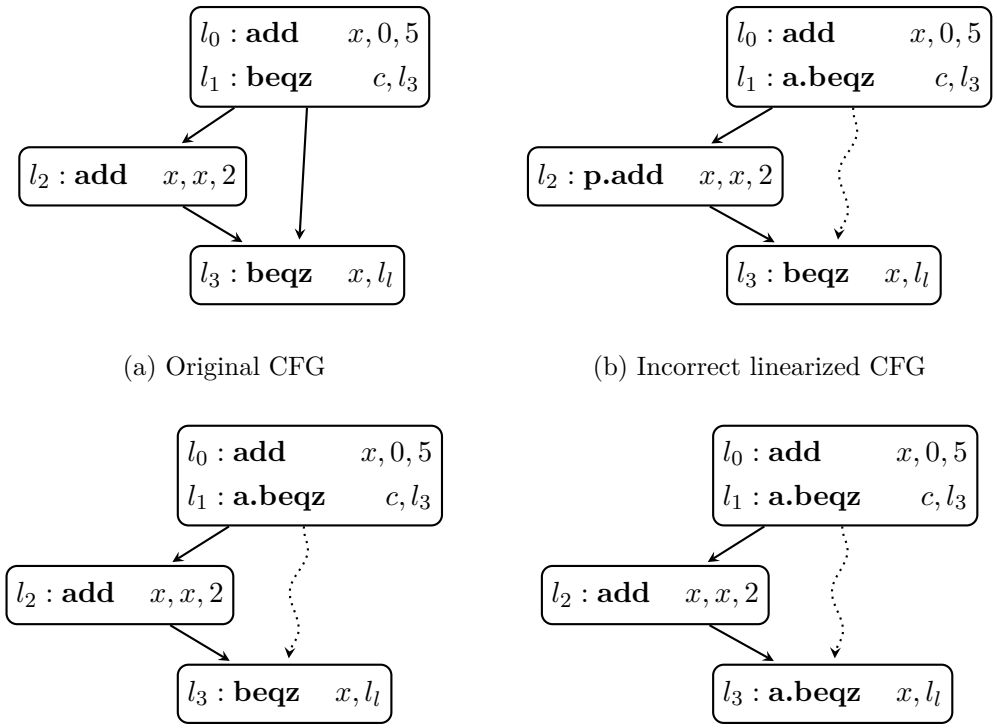
(d) Correct and secure linearized CFG

FIGURE 5.13: Example program with a loop that is control dependent on a secret-dependent branch.

correctness of the region $[l_1, l_3]$, since x outlives this region.

Suppose that we would not compute x persistently in order to avoid breaking the correctness of this region, as shown in Figure 5.14c. In this case, the program is insecure, because the branch at l_3 leaks x, and the value stored in x leaks whether the instruction at l_2 was executed. Consequently, we leak whether mimicry mode was enabled, which leaks information about the secret stored in c.

When taking indirect flows into account, x is control-dependent on the secret-dependent branch at l_1, hence it is considered secret. This implies that the persistent branch at l_3 is also secret-dependent, in which case it should be eliminated and replaced with an activating branch, as illustrated in Figure 5.14d. As a result, x should not be computed persistently and the correctness of the first region is not broken.



(a) Original CFG

(b) Incorrect linearized CFG

(c) Insecure linearized CFG

(d) Correct linearized CFG

FIGURE 5.14: Example of insecure and incorrect program (left) and the corrected program (right)

To conclude, these examples illustrate that indirect flows are not trivial to handle. On the one hand, Example 20 illustrates that not all indirect flows should be considered during taint analysis as this limits the amount of programs that can be linearized, since secret-dependent loop latches are not allowed. On the other hand, Example 21 illustrates that indirect flows must be considered for register that outlive an activating region, in order to not miss secret-dependent branches and memory accesses. To solve this problem, we only propagate indirect flows for registers that outlive an activating region. For example, in Figure 5.14, we consider an implicit flow from l_1 to its immediate post-dominator l_3, since x is defined after the branch at l_2, and it is live at l_3. However, in Figure 5.13, no implicit flow is considered, even though i is control-dependent on l_1, because it is no longer live at the immediate post-dominator l_4. Proving that this approach results in correct and secure programs is left for future work.

Chapter 6

Implementation in LLVM

This chapter introduces an extension of LLVM, a compiler framework designed to support lifelong program analysis and transformation [27], to support control-flow linearization using AMi [46]. To increase security guarantees, all analyses and transformations are implemented in the compiler backend of the RISC-V target.

A brief overview of LLVM is given in Section 6.1, in which the different stages of the compilation pipeline are described. Section 6.2 explains how the RISC-V AMi target can be described in the backend. Section 6.3 describes the program annotations used to annotate secret data in the source program, and the assumptions formulated about the high-level type system. The propagation of these program annotations to the compiler backend is explained in Section 6.4. In Section 6.5, we give an overview of the hardening passes we implemented in the backend. We use static taint analysis to track the flow of secret data within a single function based on the annotations. This is explained in Section 6.6. Finally, we describe how we implemented control-flow linearization in Section 6.7.

6.1 LLVM Overview

The core of LLVM provides a target-independent code optimizer, a collection of compiler passes that analyze and transform an intermediate code representation (*IR*). This representation is in *SSA* form, which means that each *virtual register* is written exactly once, and each register use is dominated by its definition [27]. This stage of the compilation pipeline is sometimes referred to as the *middle-end* since it is positioned between the frontend and the backend. The *frontend* parses code and emits IR. Different programming languages each have their own compiler frontend, such as clang for C and C++, and rustc for Rust.

In addition, LLVM offers code generation support for numerous CPU architectures. These components are often called *backends*, which transform target-independent IR into target-specific machine code. Optimizations are implemented in the middle-end (all targets can benefit), backends convert IR to machine code. The backends make use of a different intermediate representation called *Machine IR (MIR)*, which can contain target-specific instructions. Middle-end IR is lowered

into MIR by constructing a directed acyclic graph called the *SelectionDAG*, which describes the dependencies between instructions while they are gradually replaced with target-specific instructions in a process called legalization.

It is common to implement hardening passes in the middle-end [12, 48]. Some optimizations are implemented in the backend, since they might rely on target-specific features, or because they cannot be expressed as an SSA-based optimization. Unfortunately, optimizations may break security properties of hardened code, for example, by introducing secret-dependent branches, which may result in the manifestation of a timing side channel, as noted by Simon et al. [42]. For this reason, the transformations described in this thesis are implemented in the backend of RISC-V.

6.2 Architectural Mimicry Target Description

To minimize the need for extensive target-specific C++ code, LLVM employs a domain-specific language known as *TableGen*. This domain-specific language the ISA, instructions and registers specific to a given target architecture. The compiler backends can generate substantial portions of code from this description.

The description of the AMi instructions is implemented by Winderix et al. [46] in the `RISCVInstrInfoXAMi.td` TableGen file, but we have extended this file to support more AMi instructions. Additionally, we describe the different classes of AMi instructions, such as `Mimicable`, `Activating` and `AlwaysPersistent` in this file. Furthermore, `QualifiedInstr` tables are generated to map instructions to their AMi-qualified variants.

We implemented additional target description in the `RISCVInstrInfo` C++ class. First, we implement a method that describes the transfer of variables marked as secret through an instruction, which is used by taint analysis. Second, we provide a method that returns the operands of an instruction that leak on side channels, such as the operands of a branch instruction and the address used by memory operations. This method describes the leakage model, and can be seen as a form of hardware/software contract [19] embedded in the LLVM compiler.

6.3 Program Annotations and Type System Assumptions

Due to the impact of linearization on running time [46, 48, 12], it is essential to limit the number of linearized branches. Therefore, we allow the programmer to add annotations to the source program. In this thesis, function arguments, return values and global variables can be annotated. An example of such annotations is given in Listing 6.1.

```
#define _secret_ [[clang::annotate_type("secret")]]

static volatile int _secret_ values[256];

int _secret_ challenge(int _secret_ p, int i, int _secret_ a) {
```

```
  // Start of secret-dependent branch region
  if (p > 0) {
    values[i] = 42 + a;
  }

  return a;
}

int _secret_ * challenge2(int _secret_ * p1, int * _secret_ p2) {
  // [..]
}
```

Listing 6.1: Example of program annotations

Note the difference between `int _secret_ *` and `int * _secret_`, the former describes a pointer to a secret integer and the latter describes a secret pointer to a public integer.

This thesis assumes that the annotated source program adheres to a certain notion of well-typedness. Existing research has investigated such type systems, in which a "trusted" is a subtype of "untrusted", or similarly "secret" is a subtype of "public". Additionally, such type system may enable explicit coercion of a public type into a secret type [34, 45]. In this thesis, we aim to support such coercion through type casts. While these casts are currently not implemented, explicit type coercion is also supported by returning from functions. For example, if a function with a public return value in its signature returns a secret, said secret will be declassified and becomes a public value.

Consider the following function signature.

```
int _secret_ challenge(int _secret_ a, int _secret_ * p);
```

This function may be called with a secret value as first argument, but due to subtyping, a public value may also be used. Furthermore, a pointer to a secret integer is passed as second argument. It is not secure to pass a pointer to a public value. This is because pointer types can be seen as output parameters, in which this subtyping relation is contravariant. Indeed, the implementation may assume that this argument points to a secret value, hence it is valid to store another secret value at this location. If a pointer to a public value would be passed to this function, the secret value would leak to a public memory location, breaking the security of the program.

On the other hand, it is also not allowed to pass a pointer to a secret value to a function that expects a pointer to a public value, because a pointer can also be used as an input, in which case the subtyping is covariant. Hence, we cannot allow subtyping on pointer types, unless additional annotations are added to specify if a pointer can be used for input or output.

These typing rules are not checked, and this thesis assumes that any IR emitted by the frontend is well-typed according to the security type system. There exist implementations of extended type systems for taint analysis in clang, the LLVM compiler frontend for C/C++, such as Quala [41].

6.4 Propagation of Annotations

Since the linearization passes are implemented in the backend, the annotations must propagate to the compiler backend, to determine secret-dependent branches.

Lowering from C to IR Source level annotations are lowered to LLVM IR, requiring minor modifications to clang. Specifically, `CGCall` and `CodeGenModule` are changed to emit an IR attribute for each function argument, return value and global variable.

Lowering from IR to MIR To lower IR to the intermediate representation of the backend known as *Machine IR* or *MIR*, a *SelectionDAG* is constructed as an intermediate step. A SelectionDAG is a directed acyclic graph (DAG) where each node represents a certain operation on variables, such as an addition, and each edge represents a dependency between operations. Leaf nodes contain operations that copy values from input registers, according to the calling convention. Root nodes can be references to other basic blocks that use certain values, return nodes or operations with side effects, such as store operations. Between leaf and root nodes are operations that describe how to compute the root values from the leaf nodes.

Selection patterns are applied to lower generic DAG nodes into target-specific nodes that represent target-specific instructions. After several transformation passes on the SelectionDAG, Machine IR instructions are emitted, a process called *instruction selection*. During instruction selection, function parameters are lowered according to the calling convention, by copying to and from registers and stack slots. As a consequence, information about which function argument is stored in which register or stack slot is no longer available after instruction selection, making it difficult to infer secrecy annotations in MIR code.

This thesis solves this by lowering annotations on function parameters to custom SelectionDAG, by inserting a custom node between the operations that result from lowering the calling convention and the remainder of the DAG. These custom DAG nodes are then lowered into MIR pseudo-instructions, and the input variable is mapped to the output variable. This approach is that it is independent of the calling convention.

6.5 Hardening Pipeline

Figure 6.1 gives an overview of the backend passes that implement AMi linearization. The hardening pipeline can be divided in two stages. In the first stage, analysis passes run to determine the register allocation constraints needed to ensure that the linearized program will be correct. In the second stage, after register allocation, the same analysis passes run once again to determine how to apply the linearization. These results are then applied by changing branch terminators as described in Section 5.3.

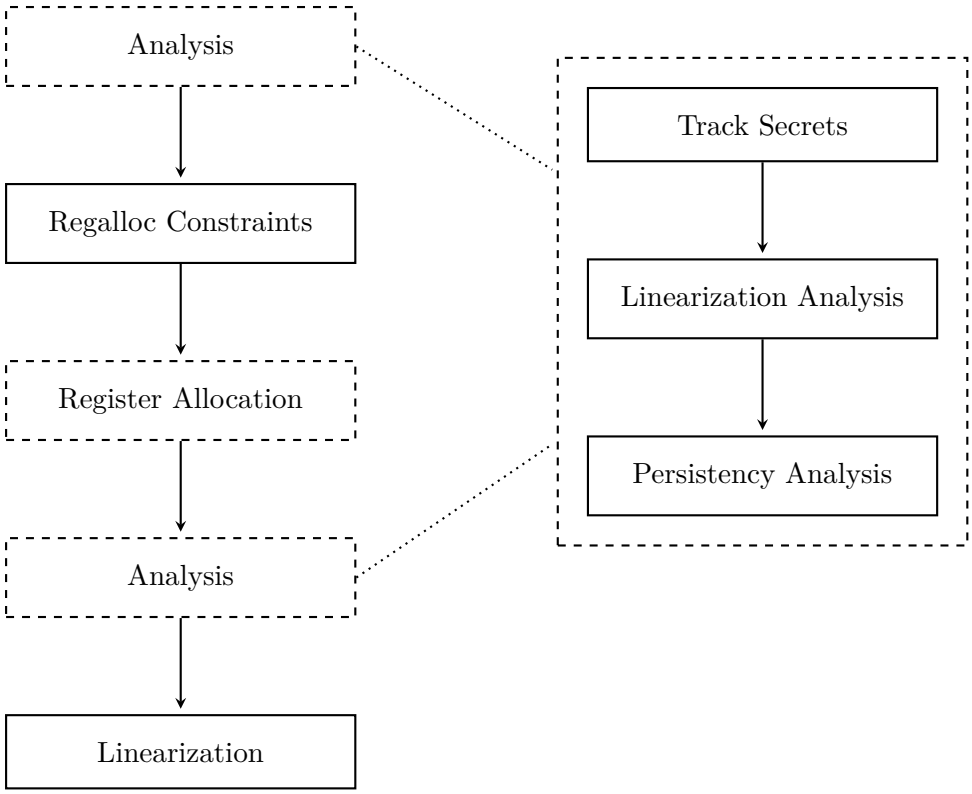


FIGURE 6.1: Overview of backend passes used to implement AMi linearization in LLVM. Solid borders represent a single pass, while dashed borders are used to represent a collection of passes.

The analysis passes include a taint analysis pass called “Track Secrets”, a “Persistency Analysis“ pass that determines which instructions need to be executed persistently to reach the security objective, and a “Linearization Analysis“ which implements the methods described in Section 5.2 to determine the sets of activating edges and ghost edges. The “Linearization Analysis” pass implements two linearization methods. The “SESE” variant implements both method 1 and method 2 described in Section 5.2, since applying method 2 to triangle-structured control flow yields the same results as method 1. The third method is implemented by the “PCFL” variant.

6.6 Static Taint Analysis

To identify secret-dependent branches, static taint analysis tracks the flow of secrets from annotated function arguments and return values to branch instruction operands. Data-flow equations are set up for each instruction, and those can be solved by repeatedly calculating the secrecy of the defs of an instruction given the secrecy of the uses. This method is also known as Kildall’s method [23]. This is described in Algorithm 2, which we have implemented in the compiler backend.

Algorithm 2: Static taint analysis algorithm used to propagate secrecy

```
input  : S, list of secrets
output: B, list of secret-dependent branches
while S ≠ ∅ do
    i ← pop(S);
    foreach u ∈ uses(i) do
        if u is a branch instruction then
            add u to B
        end
        foreach d ∈ transfer(u) do
            add d to S
        end
    end
end
```

The *transfer* function mentioned in Section 6.2 returns the list of secret registers defined by the instruction that uses a certain secret operand. For most instructions, these are all defined registers, given that one of the operands is secret. However, memory operations behave differently. For example, a load instruction that uses an address register with secrecy label "pointer to secret value" will cause the defined register to be labeled as a "secret value". If this address operand would be a secret value itself, the code is rejected, as this would result in a secret-dependent memory access.

6.7 Control Flow Linearization

In this thesis, control-flow linearization is applied to secret-dependent branches to remove leakage of secrets through control flow. Section 6.7.1 explains how existing infrastructure can be used to structurize IR code. Section 6.7.2 describes how linearization can be applied using AMi. Finally, Section 6.7.3 describes how we implemented a method based on the approaches presented by Borrello et al. [12] and Wu et al. [48], used to compare its performance against the implementation that makes use of AMi.

6.7.1 Structurization and Analysis

Control Flow Structurization To facilitate the linearization of secret-dependent branches, some approaches require code to be structurized, such as the first two methods described in Section 5.2 and the work of Borrello et al. [12] and Wu et al. [48]. LLVM provides the *StructurizeCFG* pass [1], which structurizes a CFG to triangle-structured control flow. Although this pass was written with SIMD op-

[1] `https://llvm.org/doxygen/StructurizeCFG_8cpp.html`

timizations in mind, it has also been used by linearization methods that mitigate against side-channel leakage, such as the work of Borrello et al. [12].

Unfortunately, it is not always desirable to structurize to a triangle-structured CFG, since this could impact the performance and register pressure of the linearized CFG. For methods that only require structured control flow, Borrello et al. [12] implemented a *BranchEnhance* pass, which tries to undo some structurizations, resulting in structured control flow that is not necessarily triangle-structured. In this thesis, we make use of these passes in case the chosen linearization method requires (triangle-)structured control flow. For the linearization method that only requires reducible control flow, we make use of built-in LLVM passes, such as FixIrreducible [2] and LoopSimplify [3]. These passes remove irreducible control flow and transform loops to a canonical form respectively.

SESE Region Analysis The goal of this compiler pass is to identify structured secret-dependent branches, along with their SESE regions by building a program structure tree [22]. We implement a pass similar to the RegionInfo [4] pass of LLVM, but out pass only constructs the regions that follow a secret-dependent branch, and we assume that all regions of a branch have the immediate post-dominator of the branch as exit.

6.7.2 Architectural Mimicry

After taint analysis, the AMi linearization analysis pass determines how the program should be linearized by identifying all edges that must become activating edges, and the set of ghost edges that should be added to the CFG. Based on this information, the activating regions of the program can be determined. Subsequently, the persistency analysis determines which instructions within each activating region need to be made persistent to ensure that the resulting program is secure. The next pass creates additional register allocation constraints to ensure that the activating regions of the resulting program will be correct, after which register allocation takes place. After registers have been allocated, the results of the linearization analysis are used to apply the linearization to the program, resulting in a secure and correct program.

AMi Linearization Analysis

As explained in Section 5.2, a linearization of a CFG can be found by replacing control-flow edges with activating edges, and by introducing ghost edges. The *AMi Linearization Analysis* pass analyzes the CFG and determines which edges should become activating edges, and which ghost edges need to be added. This pass consists of two different implementations. The "SESE" implementation makes use of the results of the SESE region analysis to find the exiting blocks of one branch region, and

[2] `https://llvm.org/doxygen/FixIrreducible_8cpp.html`
[3] `https://llvm.org/doxygen/classllvm_1_1LoopSimplifyPass.html`
[4] `https://llvm.org/doxygen/classllvm_1_1RegionInfoBase.html`

create ghost edges to the next branch region, as explained in Section 5.2.2. PCFL as described in Algorithm 1 from method 3 (Section 5.2.3) is implemented as the "PCFL" implementation of this pass. This method requires a compact topological ordering, hence we implemented a CompactOrder pass based on the code of Soares et al. [43].

Region Security: Persistency Analysis

As explained in Section 5.5, each instruction that affects the condition of a persistent branch or the memory address of a load or store operation must be executed persistently. To find this set of instructions, backwards dataflow analysis through the def-use chains can be used starting from instructions that leak one of their operands. Such static dataflow analysis is similar to the taint analysis described in section 6.6. Algorithm 3 runs for each activating region and returns a set of instructions that should be executed persistently. The `constantTimeLeakage` function determines which operands are leaked by the given instruction, as defined by the constant-time leakage model.

Algorithm 3: Persistency analysis algorithm

```
input  : R, set of instructions part of an activating region
output: P, set of instructions that should be executed persistently
foreach i ∈ R do
    Ops ← constantTimeLeakage(i);
    foreach o ∈ Ops do
        S ← {o};
        while S ≠ ∅ do
            o ← pop(S);
            foreach d ∈ defs(o) do
                // d is an instruction that defines operand o
                add d to P;
                foreach u ∈ uses(i) do
                    // u is an operand used by i
                    add u to S;
                end
            end
        end
    end
end
```

Region Correctness: Register Allocation Constraints

The linearization passes can also be used to identify activating regions without applying the changes to the CFG. As described in Section 5.4.3, every variable that is live on an activating edge conflicts with every variable defined by a persistent

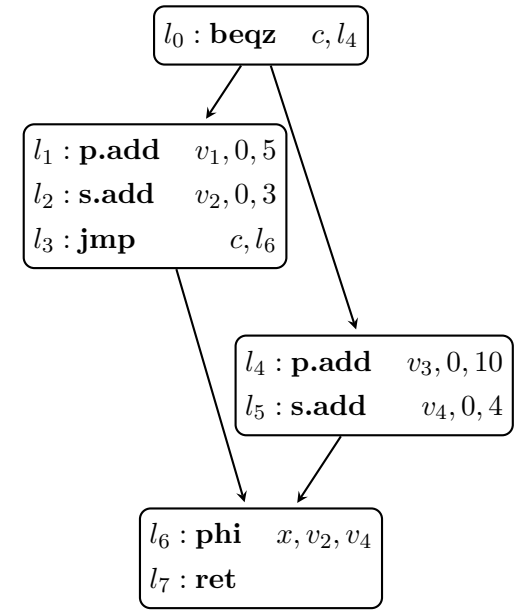
instruction or a ghost instruction within the activating region of that edge. These constraints are added to the original CFG, since the original CFG more closely resembles the behavior of the resulting program, as opposed to the linearized CFG, which better describes the microarchitectural behavior of the program. Furthermore, it is currently not feasible to describe the linearized program in SSA form, therefore we apply the linearization after register allocation.

We constrain register allocation based on the results of the linearization pass, to ensure that the activating regions will be correct in the resulting program, as defined in Section 5.4.2. Our implementation in LLVM constrains register allocation by creating a new block at each activating edge. In this block, pseudo-instructions are inserted for each register that is defined by a persistent instruction or a ghost instruction in the activating region of this activating edge, which defines the same registers as that instruction. Another pseudo-instruction is inserted that uses the same registers, to ensure that those definitions are not considered dead. For each of these pseudo-instructions a new segment is added to the live interval of the defined register, which overlaps with the interval of variables that are live on the activating edge. This way, additional constraints are added to the interference graph.

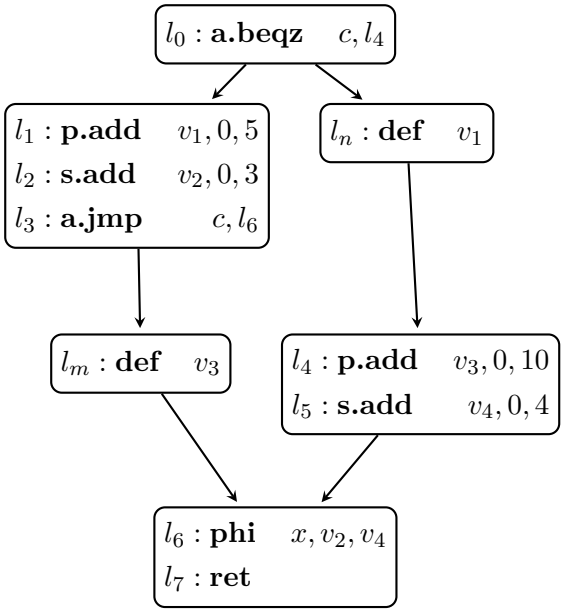
Example 22. *An example of constrained register allocation is given in Figure 6.2. We assume that linearization analysis has determined that* $l_0 \rightarrow l_4$ *and* $l_3 \rightarrow l_6$ *will become activating edges in the linearized CFG, along with their respective activating regions* $[l_0, l_4]$ *and* $[l_3, l_6]$ *. Moreover, we assume that persistency analysis has determined which instructions need to be computed persistently, and marked them with qualifier p as shown in Figure 6.2a. Next, two temporary basic blocks are created on edges* $l_0 \rightarrow l_4$ *and* $l_3 \rightarrow l_6$*, as illustrated in Figure 6.2b. These blocks contain instructions that define the registers that are defined persistently in the matching activating region.*

Then register allocation takes place, resulting in Figure 6.2c. Physical register a is assigned to most virtual registers, but it cannot be assigned to both v_3 *and* v_2*, since both are live at the temporary block that defines* v_3*. Hence, a different free physical register b is assigned to* v_3*. This ensures that in the linearized form shown in Figure 6.2d, the persistent instruction at location* l_3 *does not overwrite live register a. However, the standard instruction at location* l_4 *can overwrite a, since it will be mimicked if a is live at* l_4*. This way, careful constraining of the register allocation can significantly reduce the impact on register pressure, since the allocation is based on the liveness analysis of the original CFG instead of the linearized CFG.*

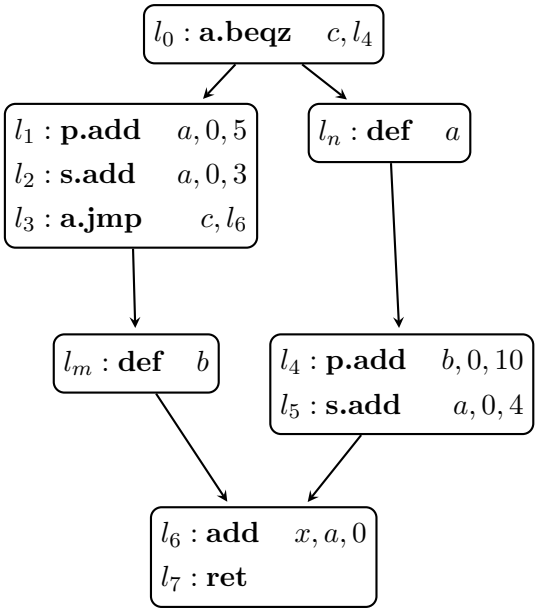
Furthermore, this compiler pass applies the ghost load pattern shown in Figure 5.10 by inserting pseudo-instructions that define a new virtual register and use the value that was originally stored. The stored virtual register in the store operation is replaced with this newly defined virtual register. We also modified the register spilling code of the register allocator such that ghost loads are emitted for register spills, and additional constraints are added for these ghost loads.



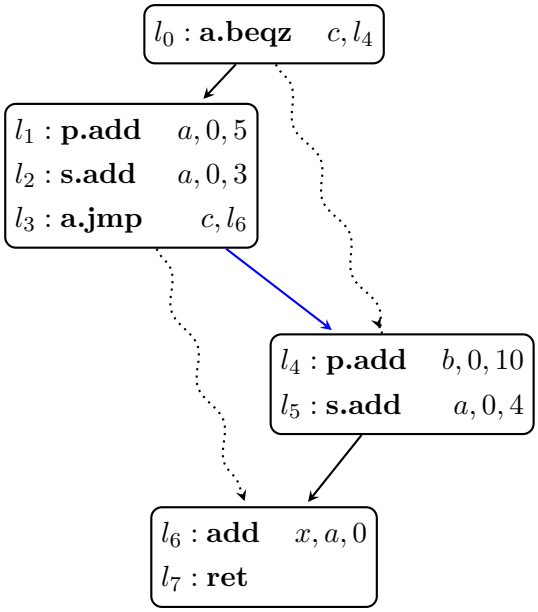
(a) Example CFG before register allocation where some instructions are marked as persistent



(b) After adding additional blocks to constrain register allocation



(c) After register allocation



(d) After linearization

FIGURE 6.2: Example of constrained register allocation

AMi Linearization

After having determined how the CFG needs to be linearized, which instructions need to be made persistent to ensure the regions are secure, and the register allocation has been constrained appropriately such that the activating regions of the linearized CFG will be correct, this linearization pass replaces certain instructions with one of their AMi-qualified counterparts.

On the one hand, the results from the persistency analysis are applied by making instructions persistent. On the other hand, we rewrite branch terminators as described in Section 5.3. The target description described in Section 6.2 is used to find qualified variants of instructions. Specifically, the table generated from the `QualifiedInstr` records (as described in Section 6.2) is used to determine the mimic, persistent, ghost or activating counterpart of a given instruction.

Furthermore, this pass changes the successors of basic blocks accordingly. Even though activating edges should not be considered as part of the CFG, this pass does add activating edges to the CFG, due to technical limitations of the instruction emitter. Program labels are only emitted for a basic block if there is some predecessor block that does not fall through to this block. As a result, labels for activating instruction targets would not be emitted, unless activating edges are included in the CFG.

6.7.3 Conditional Execution

In addition to the linearization methods that make use of AMi, we provide an implementation of the linearization methods based on conditional execution introduced by [48] and [12]. The existing implementations are written as middle-end passes, and are designed for an architecture such as X86 that provides conditional move instructions. When using these implementations on the RISC-V architecture used in this thesis, which does not provide conditional move instructions, such conditional selection would be lowered to branches, resulting in leakage. The methods proposed by Simon et al. [42] could be adapted to implement a constant time selection primitive that is lowered to branchless RISC-V code. However, in this thesis, the linearization is implemented in the backend, and conditional selection is lowered immediately into branchless code.

Taken Predicate In this implementation, we make use of a "taken" predicate, which is stored in a fixed memory location. This predicate is updated throughout the function at every control flow decision of a secret-dependent branch. We assume that 32-bit integers are used, represented in two's complement (e.g. the integer -1 is represented as 0xffffffff), and the taken predicate is represented as a mask that can take value 0x00000000 or 0xffffffff.

Each ϕ-node at the immediate post-dominator of a secret-dependent branch is replaced with a constant-time conditional selection that chooses one of the operands based on this taken predicate. To improve performance, this predicate is loaded into

a register on function entry, and stored to its global memory location before each function call.

Materializing Branch Conditions Secret-dependent branches are eliminated and replaced by an assignment to the taken variable. First, the branch condition is materialized. For example, **bge** x_1, x_2, l (branch if x_1 is greater than x_2) is replaced with **slt** c_1, x_1, x_2 (set c_1 if x_1 less than x_2) and **xori** $c, c_1, 1$ (exclusive or of c_1 and the value 1, inverting the boolean value in c_1). Second, the boolean value $c \in \{0, 1\}$ is converted to a mask $c_m \in \{0\text{x}00000000, 0\text{xffffffff}\}$ with **sub** $c_m, 0, c$. Then, we update the taken mask t by setting it only if it was already set, and the branch condition is true, e.g. **and** t, t, c_m.

Constant Time Conditional Selection Regions of secret-dependent branches are placed in a linear sequence, and ϕ-nodes are replaced with constant-time selection. Constant-time selection **ctsel** x, x_1, x_2, t is lowered to the following code:

l_0 : **xori**	$t', t, -1$
l_1 : **and**	v_1, t, x_1
l_2 : **and**	v_2, t', x_2
l_3 : **or**	x, v_1, v_2

First, we invert the incoming taken mask t using an exclusive or with -1, flipping all bits in the mask since $-1 = 0\text{xffffffff}$, and storing the result in t'. Next, we apply mask t to the first incoming value x_1, and apply the inverted mask to the second incoming value x_2, storing the result in v_1 and v_2 respectively. That way, either v_1 or v_2 contains the respective incoming value, whiles the other contains 0, since either mask t or its inverse t' will be 0. As a result, applying an "or" operation on v_1 and v_2 will store one of the incoming values to x, depending on the incoming mask t.

Store Instructions A store instruction in the linearized program should only be executed conditionally. We implemented the method described by [48], where we conditionally select between the value that is already stored and the new value that would be stored. This is similar to a ghost load in AMi, but it is predicated by the taken predicate. For example, a store operation of x to location v (**store** x, v) is replaced with the following, where we assume **ctsel** can be lowered as described in the previous section:

l_0 : **load**	y, v
l_1 : **ctsel**	z, x, y, t
l_2 : **store**	z, v

Chapter 7

Evaluation

In this chapter, we evaluate the execution time and code size overhead of automatically linearized programs using the different methods described in this thesis through the following research questions:

RQ1: Are the automated linearization methods implemented in this thesis correct and secure?

RQ2: What is the performance of our linearization techniques in terms of code size and execution time overhead?

RQ3: How do our methods compare against each other?

RQ4: How do our methods compare against manually linearized code?

RQ5: How do our methods compare against linearization methods of related work?

Methodology A benchmark of C programs with annotated secrets are compiled without linearization to provide a baseline. The same programs are then structurized by applying LLVM middle-end passes as described in section 6.7.1, resulting in three versions of the original program in LLVM IR: (1) a reducible program, (2) a structured program and (3) a triangle-structured program. Then the different linearization methods are applied, starting from a sufficiently structured IR program, depending on the requirements of each method. This results in three programs in RISC-V assembly: (1) PCFL applied to reducible IR, and the linearization method for structured control flow applied to (2) structured IR and (3) triangle-structured IR. The resulting programs are executed on the in-order pipeline of *Proteus* [11], an extendable RISC-V core implemented in SpinalHDL, with an implementation of AMi by [46] as a plugin.

Evaluation A modified version of the evaluation setup from [46] is used to validate the correctness and security of the program (RQ1). The correctness is validated by comparing the values of registers at fixed locations in the program with the expected values, by analyzing waveform files that represent both the architectural and microarchitectural state of the processor. The security is validated by comparing the trace of the program counter for different instantiations of secret inputs.

To evaluate the execution time and size overhead (RQ2), we compare the amount of cycles needed to finish execution of the linearized programs with the baseline program. To answer our remaining research questions, we evaluate all three linearization methods that make use of AMi (RQ3), the manually hardened code of Winderix et al. [46] (RQ4), and our implementation of linearization using conditional execution (RQ5).

Benchmarks The benchmark suite used for the evaluation consists of synthetic programs that feature a variety of control flow patterns and have different levels of structurization to evaluate the potential benefits of different linearization methods. This suite includes the benchmarks used by [46], to compare the performance of the compiler-assisted linearization with their manual hardening efforts.

Experimental Setup The experiments are conducted on a desktop running Fedora 38, where our modified version of LLVM 16 is used to compile and linearize programs. The programs are then linked using version 12.2.0 of the GCC RISC-V toolchain, and executed on Proteus with the AMi implementation provided by [46].

7.1 RQ1: Security and Correctness

All the benchmarks evaluated presented in this chapter pass our security and correctness evaluation. However, we have identified some cases where correctness or security may be broken by our linearization methods. For example, linearized programs where the stack counter is incremented persistently, but not decremented accordingly in mimicry mode due to limitations of our persistency analysis may be incorrect.

7.2 RQ2: Performance Results

The code size overhead for different linearization methods is given in Table 7.1, along with the results of the manual linearization by Winderix et al. [46]. We find a mean overhead ranging between 5% and 10% relative to the original program, depending on the used linearization method. The execution times of the benchmarks are given in Table 7.2. We find a mean overhead ranging between 25% and 28% depending on the linearization method.

7.3 RQ3: Comparison of Our Methods

Most of the examples used by [46] are already triangle-structured, or can be structurized without little to no size overhead. As a result, the different linearization methods yield very similar code sizes and execution times.

When trying benchmarks of non-structured, but reducible control flow, differences in code size for different linearization methods become apparent. We find that

Benchmark	Original			Manual		Compiler Linearized			
	Baseline	SESE	Triangle	Molnar	AMi	Molnar	PCFL	SESE	Triangle
triangle	136	1.00	1.00	1.15	1.00	1.26	1.00	1.00	1.00
ifthenloop	204	1.00	1.00	1.20	1.00	1.51	1.04	1.06	1.06
modexp2	308	1.00	1.00	1.16	1.06	1.10	1.00	1.00	1.00
mulmod16	264	1.00	1.08	1.23	1.03	1.17	1.03	1.03	1.06
keypad	504	1.00	1.00	1.02	0.94	1.47	1.03	1.03	1.03
bsl	368	1.00	1.00	1.00	0.92	1.09	1.00	1.00	1.00
fork	136	1.00	1.00	1.15	1.00	1.26	1.00	1.00	1.00
reducible	244	1.00	1.03	-	-	1.97	1.13	1.13	1.16
largecfg	484	1.13	1.34	-	-	3.20	1.25	1.34	1.54
highpressure	784	1.00	1.07	-	-	1.54	1.01	1.01	1.07
indirectflow	196	1.00	1.00	-	-	1.43	1.04	1.04	1.04
mean	-	1.01	1.05	1.13	0.99	1.55	1.05	1.06	1.09

TABLE 7.1: Size overhead of structurization and linearization. The baseline size is given in bytes, other values are relative to the baseline. "Triangle" and "SESE" refer to (linearization of) triangle-structured (method 1) and structured code (method 2) respectively, and "PCFL" refers to method 3.

Benchmark	Original			Manual		Compiler Linearized			
	Baseline	SESE	Triangle	Molnar	AMi	Molnar	PCFL	SESE	Triangle
triangle	86	1.00	1.00	1.07	0.95	1.27	0.95	0.95	0.95
ifthenloop	204	1.00	1.00	1.50	1.38	2.09	1.47	1.48	1.48
modexp2	12256	1.00	1.00	1.84	1.75	1.78	1.72	1.72	1.72
mulmod16	322	1.00	1.11	1.44	1.28	1.37	1.34	1.19	1.20
keypad	4708	1.00	1.00	1.59	1.18	2.22	1.57	1.57	1.57
bsl	1982	1.00	1.00	1.16	0.86	1.03	1.00	1.00	1.00
fork	86	1.00	1.00	1.07	0.95	1.27	0.95	0.95	0.95
reducible	231	1.00	1.04	-	-	2.03	1.22	1.22	1.26
largecfg	633	1.20	1.39	-	-	2.99	1.57	1.69	1.85
highpressure	1280	0.97	1.04	-	-	1.45	1.00	0.98	1.05
indirectflow	174	1.00	1.00	-	-	1.49	1.01	1.01	1.01
mean	-	1.02	1.05	1.38	1.19	1.73	1.26	1.25	1.28

TABLE 7.2: Execution time overhead of structurization and linearization. The baseline is given in number of cycles, other values are relative to the baseline. "Triangle" and "SESE" refer to (linearization of) triangle-structured (method 1) and structured code (method 2) respectively, and "PCFL" refers to method 3.

an increased size due to structurization is reflected in the resulting program when linearizing a (triangle-)structured program with a similar increase. Hence, being able to apply PCFL to reducible control flow graphs is useful to reduce code size overhead, as additional branches and predicate variables resulting from structurization are avoided.

The largecfg benchmark contains a variety of branches, but not all depend on secrets. Some of those branches result in non-structured control flow, hence structurization results in significant running time and code size overhead. For such benchmarks, our PCFL method results in programs with lower execution time than methods that require structured control flow, because reducible parts of the program that don't depend on secrets are left as in the original program, and only the branches that depend on secrets are linearized.

For certain benchmarks, such as mulmod16, the PCFL method performs worse

than the linearization of structured control flow. However, the original program is already structured, as reflected by there being no overhead on the SESE structured version. In this case, one would expect PCFL to produce the same linearized program as the other method. The disparity in execution time can be attributed to variances in the successor that is chosen for a secret-dependent branch during linearization, combined with suboptimal block placement heuristics. This could be rectified by adapting block placement to take the compact topological ordering into account. Such improvements are considered future work.

7.4 RQ4: Comparison with Manual Linearization

The manual efforts of Winderix et al. [46] do not add any code size overhead, in contrast to our automatic linearization. One reason for this discrepancy is the presence of redundant move instructions that result from applying the pattern in Figure 5.10. Another reason is that we do not support inlining annotated functions, which explains the difference in the keypad example. Moreover, when hardening manually, the programmer can perform additional optimizations. For example, when incrementing a global variable, one can avoid adding a ghost load and use a persistent load instead, since the stored value must also be loaded in standard execution mode.

Furthermore, we find a similar difference in execution time overhead. This is to be expected considering that the code size of the automatically linearized code is higher, and in a linearized program, all instructions will be executed (assuming all branches are linearized) to ensure an attacker cannot distinguish different executions that have different secret inputs.

7.5 RQ5: Comparison with Related Work

When automatically linearizing benchmark programs using our implementation based on the methods of Molnar et al. [33], Borrello et al. [12] and Wu et al. [48], we find a mean code size overhead of 55% and a mean execution time overhead of 73%. This is significantly higher than the overhead of linearization using AMi.

The size and execution time overhead when automatically applying the method of Molnar et al. [33] differs from the overhead when done manually by Winderix et al. [46]. This can be explained by small differences in the methods used by [46], and the methods used in this thesis based on [12] and [48]. Moreover, when manually linearizing code, some assumptions can be made about functions and the context in which they are called. That way, unnecessary store and load operations for the taken predicate can be avoided.

Chapter 8

Conclusion

This thesis implements automatic linearization passes in the LLVM compiler making use of the AMi ISA. The main contributions of this thesis can be summarized as follows. We extended the programming model for linearization proposed by Winderix et al. [46], and propose an algorithm based on PCFL to automatically linearize reducible control flow. These methods are implemented in the backend of the LLVM compiler, along with an implementation of static taint analysis. Section 8.1 discusses the main results of this thesis and its limitations, and avenues for future work are explored in Section 8.2.

8.1 Discussion and Limitations

8.1.1 Secret-Dependent Loops

Similar to most related work, one of the main limitations of this thesis is the lack of support for secret-dependent loops. If the source program contains a secret-dependent loop, the program will be rejected. Most methods unroll secret-dependent loops by finding an upper bound for the amount of iterations using overly conservative static analysis [48], or may result in programs that do not terminate [43]. The authors of [12] instead adaptively updates the upper bound of a loop if a program wishes to iterate more than the current bound in what they call "just-in-time linearization". However, secret information is still leaked when the loop bound changes. We propose that additional program annotations could be introduced that enable the programmer to inform the compiler of an upper bound for a loop. This way secret-dependent loops can be supported, without resulting in side-channel leakage, potentially non-terminating programs, or overly conservative loop bounds.

8.1.2 Program Annotations and Taint Analysis

The current implementation for program annotations is very limited, as there is no support for annotating struct fields. Moreover, the current taint analysis approach is not field sensitive and unable to track taint through stack slots. As a consequence, compilation of properly annotated programs may yield a program that leaks secrets

in certain cases. Currently, the frontend does not check whether calls of functions with secret arguments are valid. For example, we do not reject programs where a function that expects a public value is called with a secret. The compiler frontend should be extended to support such type-checking.

Moreover, a drawback of implementing taint analysis in the compiler backend is that although secret-dependent memory accesses can be detected, it is difficult to provide meaningful messages to guide the programmer towards fixing the leakage. Frontend support would help the programmer resolve such issues.

Lastly, we do not support other LLVM compiler frontends such as the Rust compiler, as no support for secrecy annotations is implemented. Extension to other frontends would be an interesting area of future work.

8.1.3 Correctness and Security Guarantees

Although this thesis proves that the linearization methods preserve program semantics (Theorem 1), this only holds under the assumption that the activating regions are correct (Definition 7). We state concrete approaches to ensure regions are correct and secure, but do not provide a proof. This may need to be refined to deal with potential edge cases that this thesis did not yet consider. One example of such edge case are stack pointer computations. Using persistency analysis, we may conclude that incrementing the stack pointer to reserve a stack slot must be done persistently since a store operation that stores to this stack slot is data-dependent on the stack pointer. If the stack pointer is decremented again within the same activating region, this should also be done persistently. The correctness proposition states that this should be the case since the stack pointer is part of the live state, and hence it should be restored to its original value when mimicking the region. However, the implementation currently does not guarantee this, resulting in incorrect code.

Furthermore, the current implementation in the compiler backend cannot guarantee the correctness and security of the resulting programs since standard compiler passes run throughout the code generation process. These passes have no knowledge of the semantics of AMi instructions, or any hardware/software security contract. Hence, they are prone to break the correctness or security of the program.

8.1.4 Comparison with Conditional Move

Since the RISC-V ISA does not provide conditional move instructions, the implementation based on [12] and [48] uses a conditional selection primitive that may be less efficient than a conditional move instruction. Comparing the AMi ISA extension with a RISC-V ISA extension that provides conditional move instruction may yield a more fair comparison between our method and state-of-the-art automated linearization methods.

8.2 Future Work

8.2.1 Security Type Systems

In section 6.3, we implemented basic support to annotate secrets and make several assumptions about typing rules. In future work, this type system could be extended to support static type-checking of security labels to ensure functions that expect public inputs are not called with secret values. By implementing a type checker for these security labels in the compiler frontend, secret-dependent loops could be detected earlier, and more useful error messages could be provided to the programmer. Furthermore, security types of local variables could be inferred by the frontend, instead of relying on user annotations. This information could be propagated to the middle-end and the backend by implementing these security types in the intermediate representation of the compiler.

8.2.2 AMi Semantics in Compiler Backends

Section 5.4.3 explained how the inability to express the semantics of AMi in compiler representations leads to problems regarding register allocation. Although we proposed a method to constrain register allocation resulting in a valid allocation of physical registers, there are still a few issues with this approach. On the one hand, this solution is not idiomatic as it introduces temporary basic blocks with dummy instructions, and it takes place after ϕ-node elimination, but before register allocation. Usually, no passes run in this phase, hence register allocators may have implicit assumptions about the code that is used as input for register allocation.

On the other hand, any passes that run after linearization have no knowledge of these semantics either. For example, assignments to registers by standard instructions may be considered dead, hence removing such instructions would be considered correct by compiler passes, even if it is incorrect under AMi semantics. Although this could be alleviated by scheduling linearization passes as late as possible, this creates another problem. If other passes are scheduled between register allocation and linearization, the control flow of the program can change, in which case the register allocation constraints may no longer be valid for the resulting program.

These issues would need to be resolved before automatic AMi linearization can be used in practice. To achieve this, different ways to express the semantics of AMi in compiler backends can be explored. For example by introducing a notion of *conditional liveness*, where the liveness of a register defined by a standard instruction is predicated by the condition of an activating branch. However, it would be preferable to reduce the impact on existing compiler infrastructure, as it would be undesirable to rewrite numerous existing compiler passes.

8.2.3 Contract-Aware Code Generation

Based on the idea of Contract-Aware Secure COmpilation (CASCO) [19], a contract-aware code generator could be written to ensure that an AMi linearized program

leaks no secrets. Information about the leakage model could be provided by target-specific descriptions of LLVM backends, similar to the `constantTimeLeakage` function that was briefly mentioned in section 6.7.2. This could be extended such that different hardware/software security contracts can be specified, and backend passes could be modified such that they satisfy such contracts.

8.2.4 Secure Compilation and Formal Verification

In this thesis, the security and correctness of programs are experimentally evaluated using a limited set of benchmark programs. In future work, these hardening passes could be implemented in a formally verified secure compiler such as CompCert [28], or binary analysis tools such as BinSec [17] could be extended to support programs with AMi instructions to improve the validation of hardening passes.

Appendices

Appendix A

Proofs of Theorems

A.1 Correctness of AMi Linearization

Lemma 1. *Let σ_1 and σ_2 be two live-equivalent configurations for respective programs P_1 and P_2. If $P_1[\sigma_1.pc] = P_2[\sigma_2'.pc]$, $\sigma_1 \overset{o_1}{\Longrightarrow} \sigma_1'$ and $\sigma_2 \overset{o_2}{\Longrightarrow} \sigma_2'$, then $\sigma_1' =_{\mathcal{L}} \sigma_2'$.*

Proof. This follows from the fact that in the transition system in figure 4.2, the updates to the configuration only depend on the instruction and its used operands (figure 2.8).

Since the instruction is identical in both cases, and each used operand is live according to section 2.4.3, we know that these operands will be mapped to the same values by live-equivalent configurations σ_1 and σ_2.

Hence, both instructions will have the same effect on the live state of σ_1 and σ_2, and the resulting configurations σ_1' and σ_2' will also be live-equivalent. □

Theorem 1 (Correctness of AMi Linearization)**.** Let P be a program and T an AMi linearization as defined by Definition 3. Then T is correct if for each activating region $[l_x, l_y]$ in $T(P)$, $[l_x, l_y]$ is well-behaved (Proposition 2), and $[l_{x+1}, l_y]$ is correct (Definition 7).

Proof. (sketch). We assume that there is a bijection between the locations of P and the locations of $P' = T(P)$. If there is no such bijection, locations present in $T(P)$ can be added to P, shifting other locations and changing branch targets accordingly if necessary. Furthermore, we assume that both P and $T(P)$ have a single entry l_0 and a single exit l_1.

Let σ_0 be an arbitrary configuration with $\sigma_0.\text{pc} = l_0$ and evaluate both programs in this configuration. We prove by induction on n that after n steps of big-step evaluation, the resulting configurations σ_n and σ_n' for P and P' respectively are live-equivalent, that $\sigma_n.\text{pc} = \sigma_n'.\text{pc}$ (under the aforementioned bijection) and mimicry mode is disabled in both configurations.

Induction hypothesis: during the n-th induction step with respective configurations σ_n and σ_n', the following holds:

- $\sigma_n = \sigma_n'$

- $\sigma_n.\text{pc} = \sigma'_n.\text{pc} = l_k$
- $[l_0, l_k]\sigma \Downarrow_o \sigma_n$ and $[l_0, l_k]\sigma' \Downarrow_{o'} \sigma'_n$
- $\sigma_n.\text{AC} = \sigma'_n.\text{AC} = 0$

Base step: Since we are using the same initial configuration for both P and P', $\sigma_0 = \sigma'_0$, hence they are also live-equivalent, and $\text{pc} = l_0$ in both cases.

Inductive step: Assume that $\sigma_n =_{\mathcal{L}} \sigma'_n$ and let $l_n = \sigma_n.\text{pc} = \sigma'_n.\text{pc}$. We consider the following cases:

- If $P[l_n] = P'[l_n]$, it follows from lemma 1 that there must be some live-equivalent configurations σ_{n+1} and σ'_{n+1} for which $\sigma_n \overset{o}{\Rightarrow} \sigma_{n+1}$ and $\sigma'_n \overset{o'}{\Rightarrow} \sigma'_{n+1}$, and through the application of the rules in 4.5, $\sigma_n \Downarrow_o \sigma_{n+1}$ and $\sigma'_n \Downarrow_{o'} \sigma'_{n+1}$.
- If $P[l_n] \neq P'[l_n]$, then we consider the following cases based on the definition of AMi linearization.
  - (control flow edge to activating edge) There is some branch or jump instruction *instr* at $P[l_n]$ such that $P'[l_n] = a.instr$. Then there must be some target location l' such that $[l_n, l']$ is a well-behaved activating region of P'. From the semantics of AMi shown in figures 4.3 and 4.4, we know that $\sigma'_n \overset{o'}{\Rightarrow} \sigma''_n$ such that $\sigma'_n =_{\mathcal{L}} \sigma''_n$ and $\sigma''_n.\text{pc} = l_{n+1}$.

    If AC was incremented at l_n, there is some σ'_{n+1} for which $\sigma'_n =_{\mathcal{L}} \sigma'_{n+1}$ and $\sigma_{n+1}.\text{pc} = l'$, since $[l_{n+1}, l']$ is correct (def. 7). In that case, the branch at $P[l_n]$ will be taken. Let σ_{n+1} be a configuration for which $\sigma_n \overset{o}{\Rightarrow} \sigma_{n+1}$. From the semantics of AMiL, it follows that $\sigma_{n+1} =_{\mathcal{L}} \sigma_n$. Hence, we conclude that $\sigma_{n+1} =_{\mathcal{L}} \sigma'_{n+1}$, and furthermore, both configurations have $\text{pc} = l'$.

    On the other hand, if AC was not incremented, the branch at $P[l_n]$ will not be taken. Hence, the behavior of the instructions at $P[l_n]$ and $P'[l_n]$ is identical according to AMiL semantics. Hence, we can find two live-equivalent configurations σ_{n+1} and σ'_{n+1} with $\text{pc} = l_{n+1}$.
  - (add ghost edge) There is no instruction at $P[l_n]$, but there is an instruction at $P'[l_n]$. This can only be the case if $P'[l_n] = \textbf{jmp}\ l'$ such that $l_n \to l'$ is a ghost edge. Since $l_n \to l'$ is a ghost edge, mimicry mode must be enabled in σ'_n, but the induction hypothesis states that mimicry mode is disabled. Hence, this case cannot occur during the induction.

To finish proof, we show that there is some n during the induction for which $\sigma_n.\text{pc} = \sigma'_n.\text{pc} = l_1$. This follows from the fact that the induction reaches each configuration that can be reached through single evaluation steps $\Downarrow_o$, and since each program is assumed have a single exit l_1, we know that there is some σ_n reached by the induction for which $\sigma_n.\text{pc} = l_1$. The correctness of T then follows from def. 8. □

A.2 Well-behavedness of Activating Region

This section adapts the sketch of proof provided in Appendix C of [46] to the revised well-behavedness proposition stated in section 5.4.1.

Proposition 2 (Well-behaved Activating Region (revised))**.** *For any activating region* $[l, l']$ *and valid configuration* σ *such that* $[l, l']\sigma \Downarrow_o \sigma'$*, if AC is incremented at location* l*:*

1) it remains set during the execution of $[l, l']$ *(including recursive function calls but excluding* l'*), and*

2) it is restored to its initial value during the evaluation of the instruction at location l'*.*

As stated by Winderix et al., it follows from the evaluation rules of AMi in figures 4.3 and 4.4 that:

Proposition 5. *A step from a configuration where* $AC > 0$ *decrements AC only if* $pc = Ex$ *and increments AC only if* $pc = En$*. Additionally, En and Ex are only modified if AC is set to* 0*.*

Furthermore, for any *valid* configuration it follows from the evaluation rules that:

Proposition 6. *For any valid configuration* σ*, if* $\sigma.AC > 0$*, then* $[\sigma.En, \sigma.Ex]$ *is an activating region.*

Next, the authors of [46] prove that nested SESE regions behave as intended. We adopt this lemma, but in contrast to [46] we don't require that the nested region $[l_2, l_2']$ is SESE. Instead, we require that this region adheres to the well-behavedness criterion stated in proposition 2. We state this lemma with changes marked in blue.

Lemma 2. *During the evaluation of an SESE region* $[l_1, l_1']$ *in a configuration* σ *that has mimicry mode activated for* $[l_2, l_2']$ *and where* $[l_2, l_2']$ *is contained in* $[l_1, l_1']$ *and adheres to the well-behavedness criterion (proposition 2), assuming function calls in* $[l_1, l_1']$ *preserve the activation configuration and do not deactivate mimicry mode, we have that if* $[l_1, l_1']\sigma \Downarrow_o \sigma'$*, then* $\sigma.\langle AC, En, Ex\rangle = \sigma'.\langle AC, En, Ex\rangle$ *and mimicry mode is not deactivated during the evaluation of* $[l_1, l_1']$ *.*

With *contained in a SESE region* we mean that each location in $[l_2, l_2']$ (all locations reachable from l_2 and post-dominated by l_2') is contained within that SESE region. As defined by [46], a location l is contained within a SESE region $[l_1, l_1']$ if l_1 dominates l and l_1' post-dominates l.

The proof of above lemma stated by Winderix et al. still holds, since the SESE requirement of $[l_2, l_2']$ is only used at the end of the proof. Here, they require that any cycle containing l_2 also contains l_2', which is also provided by proposition 2.

We adopt the following lemma from [46] which shows that big-step evaluation of functions preserves activation configurations.

Lemma 3. *For any function f, activating region $[l, l']$, and configuration σ that has mimicry mode enabled for $[l, l']$, if $[l_f, l'_f]\sigma \Downarrow_o \sigma'$ then $\sigma.\langle AC, En, Ex\rangle = \sigma'.\langle AC, En, Ex\rangle$ and mimicry mode is not disabled during the evaluation of f.*

The proof provided by Winderix et al. still holds when using lemma 2 instead of lemma 3 defined by [46].

Finally, we adapt the proof for the well-behavedness proposition, showing modifications in blue.

Proposition 2 (Well-behaved Activating Region (revised))**.** *For any activating region $[l, l']$ and valid configuration σ such that $[l, l']\sigma \Downarrow_o \sigma'$, if AC is incremented at location l:*

1) it remains set during the execution of $[l, l']$ (including recursive function calls but excluding l'), and

2) it is restored to its initial value during the evaluation of the instruction at location l'.

Proof. (sketch) Consider a valid configuration σ such that $[l, l']\sigma \Downarrow_o \sigma'$ and mimicry mode is set after executing the instruction at location l. Let $\langle \mathrm{AC}', \mathrm{En}', \mathrm{Ex}'\rangle$ be the corresponding activation configuration where $\mathrm{AC}' > 0$. We may assume that $[l, l']$ adheres to proposition 2; let f be the function that contains $[l, l']$. From 3, we know that evaluation of nested functions during the evaluation of $[l, l']$ (including recursive function calls) do not reset AC and restore its value before returning. Therefore, we can ignore the evaluation of nested functions and, because we assume a sound CFG, we can reason about the big-step evaluation of $[l, l']$ as a sequence of small steps following a path in $cfg(f)$, starting at l and ending as soon as l' is reached. Assume that AC is incremented at location l. We need to show that: 1) AC is not reset during the whole execution of $[l, l']$ (l' excluded), 2) it is restored to its previous value when executing the instructions at location l'. Notice that, if we show that AC is not reset during the whole execution of $[l, l']$, we can also consider than En and Ex are not modified and remain set to En' and Ex' (cf. proposition 5). Since AC is incremented at location l, we know that $\mathrm{En}' = l$. Because σ is a valid configuration (and so is its successor), we know from Proposition 4 that $[\mathrm{En}', \mathrm{Ex}', i]$ s an activating region and therefore $[\mathrm{En}', \mathrm{Ex}'] = [l, l']$. In this case, from the evaluation rules of AMi we know that $\mathrm{AC}' = \sigma.\mathrm{AC} + 1$. Therefore, it is sufficient to show that:

1. there is no path in the region $[l, l']$ that goes through l twice, hence AC is not incremented again,
2. the evaluation of the instruction at location l' decrements AC to its previous value, $\sigma.\mathrm{AC}$.

The first point follows from the fact that $[l, l']$ adheres to proposition 2 and thus the program has no cycle containing l which does not also contains l'; hence the evaluation of $[l, l']$ cannot go through l twice before ending in l'. The second point

follows from the fact that during the final step, (i.e., the evaluation of the instruction at location l), because $\mathrm{Ex}' = l'$, AC is decremented and therefore restored to its previous value. □

A.3 Well-behavedness of Activating Regions after Linearization

In this section we prove that the application of algorithm 1 results only in well-behaved activating regions.

Proposition 3 (Linearization Results in Well-behaved Activating Regions)**.** *Algorithm 1 (method 3) results in well-behaved activating regions.*

On the one hand, l' must post-dominate l. We show that this holds in lemma 4.

Lemma 4. *For each activating region $[l, l']$ (or activating edge $(l, l') \in A$) in the linearized program, l' post-dominates l.*

Proof. Since there is an edge $l \to l'$ and blocks are visited in topological ordering, l will be visited before l'. When visiting the secret-dependent branch at l in algorithm 1, deferral edges are added from the single successor "next" to each other successor, including l'. At a later iteration of the algorithm, this successor "next" of l will be visited, and there will be a deferral edge to l' ($l' \in T$). It follows from Lemma B.3 from [32] that l' post-dominates the single successor of l, and hence l' post-dominates l. □

On the other hand, each cycle in $[l, l']$ that contains l must also contain through l'. This is trivially satisfied in any cycle-free control flow graph. Since back edges are removed before applying algorithm 1, the resulting activating regions will initially be well-behaved. However, we still need to show that the reinsertion of back edges does not break the well-behavedness of the activating regions. In reducible control flow with back edges we require that an exiting block of a loop cannot contain a secret-dependent branch.

Lemma 5. *The following holds when applying algorithm 1 to any reducible CFG where exiting blocks of loops don't contain secret-dependent branches. For each activating region $[l, l']$ in the linearized program, each cycle that contains l must also contain l', and vice-versa.*

Well-behavedness of each activating region $[l, l']$ in the linearized program follows from both lemmas. A similar argument can be made for the linearization methods that require structured control flow.

List of Figures and Tables

List of Figures

List of Tables

Acronyms

AMi Architectural mimicry 2–5, 17–19, 23, 24, 26, 27, 36, 41, 51, 52, 60, 61, 63–66, 70–73, 75–79, *see* architectural mimicry

CFG Control-flow graph 3, 10, 12–14, 20–22, 24, 33–36, 38–40, 42–44, 46–48, 51–55, 65–68, 70, 85, *see* control-flow graph

IR Intermediate representation 8, 60, 63, 72, *see* intermediate representation

ISA Instruction set architecture 2, 5, 8, 9, 17, 26, 27, 29, 76, 77, *see* instruction set architecture

PCFL Partial control-flow linearization 3, 23, 26, 36, 39, 66, 72, 74–76, *see* partial control-flow linearization

SESE Single-entry single-exit 13, 38, 39, 46, 47, 75, *see* SESE region

SSA Static single assignment 15, 50, 51, 60, 61, 67, *see* static single assignment

Symbols

Regs	set of registers 26, 27
$\mathbf{Regs}_v$	set of virtual registers 26
$\mathbf{Regs}_p$	set of physical registers 26
$\mathcal{V}$	set of values 26, 27
r	register file 26, 27
$Instr$	set of instructions 26
Loc	set of locations 26, 27
m	memory map 27
pc	program counter 27, 28
AC	activation counter 24, 27, 28, 47–49, 56, 83, 84
En	mimicry entry address 27, 28
Ex	mimicry exit address 27, 28, 47
σ	symbol used to represent an AMi configuration 27
$\mathcal{L}$	live state 48, 49, *see* live state
$=_{\mathcal{L}}$	live-equivalence relation 49, *see* live-equivalent
$=_{\mathcal{P}}$	low-equivalence relation 29, 55, *see* low-equivalent

Glossary

activating branch An instruction that conditionally enables mimicry mode until the target location is reached. 18, 35, 38, 42, 44, 46, 47, 56, 78

activating edge A pair (l_x, l_y) where l_x is the location of an activating instruction with target l_y (see definition 1). 33–36, 38–40, 42–44, 46, 47, 51, 54, 55, 66–68, 70, 85

activating instruction An activating branch or activating jump. 47

activating jump An instruction that unconditionally enables mimicry mode until the target location is reached. 3, 34, 38, 44, 47, 55

activating region Region spanned between an activating branch and its target (see definition 1). 18, 24, 25, 33–36, 38, 39, 46–51, 53–57, 66–68, 81, 83–85

AMi configuration A tuple $\langle m, r, \mathrm{pc}, \mathrm{AC}, \mathrm{En}, \mathrm{Ex} \rangle$ that represents the state of the processor with AMi extension. 27, 48

AMi linearization A type of linearization that only replaces control-flow edges with activating edges, and introduces ghost edges (see definition 3). 35, 36, 40, 54, 55, 81

AMiL Formal abstraction of a typical ISA with the AMi extension. 5, 26–29, 32

architectural configuration A tuple $\langle m, r, \mathrm{pc} \rangle$ that represents the state of the processor. 27, 48

architectural mimicry A novel approach that extends hardware to facilitate software-based hardening against side-channel attacks introduced by Winderix et al [46]. 2, 17,

back edge Edge $l_a \to l_b$ of a reducible CFG that is not a forward edge, and l_b dominates l_a. 13, 14, 39, 40, 85

basic block A linear sequence instructions followed by a sequence of terminators. 10, 68, 70

block . 10, 14, 38–40, 42, 44, 48, 51, 66, 68, 70, 75, 85, *see* basic block

compact topological ordering A topological ordering that is both dominance and loop compact (see defintion 4). 39, 66, 75

constant-time leakage model A leakage model under which control flow and data flow is leaked. 29, 67

constant-time programming A programming discipline that forbids secret-dependent control flow, secret-dependent memory accesses, and secret operands of instructions with variable latency. 1, 7, 9

control flow The order in which instructions can be executed. 9

control-flow edge An edge in the CFG, indicating that the target of the edge is a successor of the source. 10, 11, 13, 24, 34, 35, 40, 42, 47, 54, 55, 66

control-flow graph The collection of basic blocks and the successor relation between locations. 3, 10,

control-flow linearization A program transformation that mitigates against leakage of control flow (program counter). 2, 3, 7, 23, 60, 65

correct region A region is correct if the live state remains unaffected when executing the region in mimicry mode (see defintion 7). 49, 51, 53, 54, 57, 81

correct transformation A program is correct if it has the same effect on the live state of a program (see definition 8). 40, 48, 55

deferral edge A temporary edge $l_a \to l_b$ used by the PCFL algorithm to ensure l_b post-dominates l_a in the linearized program. 40, 43, 85

dominance A location l dominates l' if each path from the unique CFG entry to l' contains l. 11, 14, 47

entering Location or block that is succeeded by an entry or header. 12, 13

entry Given a region $[l, l']$, l is the entry. 12, 14, 34, 47

exit Given a region $[l, l']$, l' is the exit. 12, 14, 34, 38, 39, 47, 54, 66

exiting Location or block that is succeeded by the exit. 12, 14, 39, 48, 55, 56, 66, 85

fallthrough The single successor of a basic block with no terminators. 10, 70

forward edge Edges of a reducible CFG that form an acyclic graph. 13

ghost edge A control-flow edge that is only taken in mimicry mode (see definition 2). 33–36, 38–40, 42–44, 47, 53, 55, 66

ghost instruction An instruction that is only executed in mimicry mode, and mimicked in standard execution mode. 19, 49, 50, 67, 68

hardening Transforming a program in order to mitigate against side channel leakage. 5, 7, 50, 61, 73, 75

header A header of a loop dominates all locations within the loop. 13, 14

immediate post-dominator The first block or location encounter on each path starting from a location to the exit of the CFG. 11–13, 66, 70

instruction set architecture The set of machine language instructions that a processor can execute, which serves as an interface between the hardware and software. 2, 8,

intermediate representation Code used internally by a compiler to represent machine code at an abstract level. 8,

latch A location or block with an outgoing back edge to the header. 14, 56

leakage model A model that defines the ability of an attacker to observe (micro)architectural state. 5, 26

linearization . 4, 5, 7, 20, 22, 23, 36, 40, 44, 46, 48, 50, 51, 54, 55, 61, 63–67, 70, 72, 73, 75, 85, *see* control-flow linearization

linearization . *see* control-flow linearization

live A variable is live if it holds a value that may be needed in the future, see section 2.4.3. 15, 48, 49, 54

live state The live state of an architectural configuration is the set of live registers and memory locations (see definition 5). 48–51, 53–55

live-equivalent Two architectural configurations are live-equivalent if they agree on their live state (see definition 6). 54

LLVM A compiler framework designed to support lifelong program analysis and transformation. 2–5, 8, 9, 60, 61, 63, 64, 68, 72, 73, 76, 77, 79

loop A single-entry cycle in a CFG. 13, 14, 40, 48, 56, 85

low-equivalent Two configurations are low-equivalent under a policy if they agree on the public part of their register file and memory map. 29, 55

mimicry mode An execution mode in AMi in which standard instructions are mimicked. 2, 3, 18, 19, 28, 29, 33–35, 38, 40, 42, 44, 47–49, 51, 54, 55, 57

partial control-flow linearization A control-flow linearization technique. 3, 23, 39,

path Sequence of successive locations. 9, 12, 47

persistent instruction An instruction that is executed normally regardless of the execution mode. 18, 49–51, 53, 54, 67, 68

policy . *see* security policy

post-dominance A location l *post-dominates* l' if each path from l' to the unique CFG exit contains l. 11, 12, 34, 40, 47, 85

predecessor Relation that describes which locations or blocks can be executed right before a certain location or block in control flow. 9, 12, 70

qualifier A prefix for an instruction in the AMi ISA that defines the behavior of instructions in standard execution mode and mimicry mode. 18, 61, 70

reducible A CFG is reducible if the edges can be partitioned into forward edges and back edges. 3, 4, 13, 14, 20, 23, 38, 39, 48, 55, 72–74, 76, 85

region A region $[l, l']$ is the set of locations present on any path from l to l'. 11, 12, 33, 36, 38, 39, 47–49, 51, 53–55, 66, 68, 71

RISC-V An open standard ISA based on reduced instruction set computer (RISC) principles. 2–5, 8, 17, 60, 61, 70, 72, 73, 77

secure region A region is secure if it does not leak secrets on side channels (see defintion 9). 55

security policy A partitioning of memory locations into public and secret values. 29

SESE region A region $[l, l']$ where l dominates l', l' post-dominates l, and every cycle containing l contains l' and vice-versa. 11–13, 24, 36, 38, 66

side channel A channel not designed for communication between the sender and receiver where the sender does not intend to leak information, which can be exploited by an attacker at the receiving end. 1–3, 5–7, 9, 33, 42, 61

standard execution mode An execution mode in AMi in which standard instructions are executed normally. 18, 19, 49, 57

static single assignment A property of the intermediate representation of some compilers that requires each variable to be assigned exactly once and defined before it is used. 15,

structured A CFG is structured if each branch is a structured branch. 3, 11–13, 20–22, 36, 48, 55, 65, 66, 72, 74, 75, 85

structured branch A branch is structured if for each path from a successor to its immediate post-dominator, the region between the first and last location of the path is SESE. 12

structurization A correct transformation that transforms a program into structured program. 4, 20–22, 36, 65, 72–74

successor Relation that describes which locations or blocks can follow a certain location or block in control flow. 9, 10, 12–14, 25, 40, 42, 44, 46, 47, 70, 85

taint analysis A security technique that tracks how tainted data flows through a program. 8, 56, 60, 66

terminator Branch or jump instruction at the end of a basic block. 10, 46

topological ordering An ordering of a DAG which only visits a node after all its predecessors have been visited. 39, 85

triangle-structured Subset of structured control flow where branches are succeeded by a single SESE region. 3, 11–13, 20, 21, 24, 36, 46, 55, 65, 66, 72, 73

well-behaved A property of an activating region that ensures nested activations behave as expected, see propositions 1, 2 and 2. 24, 25, 36, 38, 39, 46–49, 54, 55, 57, 81, 85

Bibliography

[1] LLVM Loop Terminology (and Canonical Forms) — LLVM 17.0.0git documentation. https://llvm.org/docs/LoopTerminology.html.

[2] J. Agat. Transforming out timing leaks. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 40–53, Boston MA USA, Jan. 2000. ACM.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.

[4] N. J. Al Fardan and K. G. Paterson. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In *2013 IEEE Symposium on Security and Privacy*, pages 526–540, Berkeley, CA, May 2013. IEEE.

[5] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P.-Y. Strub. Jasmin: High-Assurance and High-Speed Cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 1807–1823, New York, NY, USA, Oct. 2017. Association for Computing Machinery.

[6] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi. Verifying Constant-Time Implementations. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 53–70, 2016.

[7] J. Anantpur and G. R. Taming Control Divergence in GPUs through Control Flow Linearization. In A. Cohen, editor, *Compiler Construction*, Lecture Notes in Computer Science, pages 133–153, Berlin, Heidelberg, 2014. Springer.

[8] A. W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, first edition, Dec. 1997.

[9] G. Barthe, B. Grégoire, and V. Laporte. Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time". In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 328–343, July 2018.

[10] D. J. Bernstein, T. Lange, and P. Schwabe. The Security Impact of a New Cryptographic Library. In D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg,

F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, A. Hevia, and G. Neven, editors, *Progress in Cryptology – LATINCRYPT 2012*, volume 7533, pages 159–176. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[11] M. Bognar, J. Noorman, and F. Piessens. Proteus: An extensible RISC-V core for hardware extensions. In *RISC-V Summit Europe '23*, June 2023.

[12] P. Borrello, D. C. D'Elia, L. Querzoni, and C. Giuffrida. Constantine: Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 715–733, Virtual Event Republic of Korea, Nov. 2021. ACM.

[13] C. Bruel. If-Conversion. In F. Rastello and F. Bouchez Tichadou, editors, *SSA-based Compiler Design*, pages 269–283. Springer International Publishing, Cham, 2022.

[14] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 249–266, 2019.

[15] B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter. Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors. In *2009 30th IEEE Symposium on Security and Privacy*, pages 45–60, Oakland, CA, USA, May 2009. IEEE.

[16] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '89*, pages 25–35, Austin, Texas, United States, 1989. ACM Press.

[17] L.-A. Daniel, S. Bardin, and T. Rezk. Binsec/Rel: Symbolic Binary Analyzer for Security with Applications to Constant-Time and Secret-Erasure. *ACM Transactions on Privacy and Security*, 26(2):11:1–11:42, Apr. 2023.

[18] Q. Ge, Y. Yarom, D. Cock, and G. Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 8(1):1–27, Apr. 2018.

[19] M. Guarnieri and M. Patrignani. Contract-Aware Secure Compilation. *CoRR*, abs/2012.14205, Dec. 2020.

[20] M. S. Hecht and J. D. Ullman. Characterizations of Reducible Flow Graphs. *Journal of the ACM*, 21(3):367–375, July 1974.

[21] J. Janssen and H. Corporaal. Making graphs reducible with controlled node splitting. *ACM Transactions on Programming Languages and Systems*, 19(6):1031–1052, Nov. 1997.

[22] R. Johnson, D. Pearson, and K. Pingali. The program structure tree: Computing control regions in linear time. *ACM SIGPLAN Notices*, 29(6):171–185, June 1994.

[23] G. A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages - POPL '73*, pages 194–206, Boston, Massachusetts, 1973. ACM Press.

[24] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–2:70, Feb. 2014.

[25] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, Oct. 2009. Association for Computing Machinery.

[26] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, May 2019.

[27] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, San Jose, CA, USA, 2004. IEEE.

[28] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, July 2009.

[29] M. Lipp, A. Kogler, D. Oswald, M. Schwarz, C. Easdon, C. Canella, and D. Gruss. PLATYPUS: Software-based Power Side-Channel Attacks on x86. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 355–371, San Francisco, CA, USA, May 2021. IEEE.

[30] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, Jan. 2018.

[31] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622, May 2015.

[32] S. Moll and S. Hack. Partial control-flow linearization. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 543–556, Philadelphia PA USA, June 2018. ACM.

[33] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks. In D. H. Won and S. Kim, editors, *Information Security and Cryptology - ICISC 2005*, Lecture Notes in Computer Science, pages 156–168, Berlin, Heidelberg, 2006. Springer.

[34] P. Ørbæk and J. Palsberg. Trust in the λ-calculus. *Journal of Functional Programming*, 7(6):557–591, Nov. 1997.

[35] F. Pfenning and A. Platzer. Lecture Notes on Liveness Analysis. 2014.

[36] A. Rane, C. Lin, and M. Tiwari. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. 2015.

[37] N. Reissmann, T. L. Falch, B. A. Bjørnseth, H. Bahmann, J. Christian Meyer, and M. Jahre. Efficient control flow restructuring for GPUs. In *2016 International Conference on High Performance Computing & Simulation (HPCS)*, pages 48–57, July 2016.

[38] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, pages 199–212, Chicago Illinois USA, Nov. 2009. ACM.

[39] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '88*, pages 12–27, San Diego, California, United States, 1988. ACM Press.

[40] A. Sabne, P. Sakdhnagool, and R. Eigenmann. Formalizing Structured Control Flow Graphs. In C. Ding, J. Criswell, and P. Wu, editors, *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 153–168, Cham, 2017. Springer International Publishing.

[41] A. Sampson. Quala: Type Qualifiers for LLVM/Clang, Mar. 2023.

[42] L. Simon, D. Chisnall, and R. Anderson. What You Get is What You C: Controlling Side Effects in Mainstream C Compilers. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 1–15, Apr. 2018.

[43] L. Soares, M. Canesche, and F. M. Q. Pereira. Side-Channel Elimination via Partial Control-Flow Linearization. 1(1).

[44] J. Szefer. Survey of Microarchitectural Side and Covert Channels, Attacks, and Defenses. *Journal of Hardware and Systems Security*, 3(3):219–234, Sept. 2019.

[45] D. Volpano, G. Smith, and C. Irvine. A Sound Type System For Secure Flow Analysis. *Journal of Computer Security*, 4, Aug. 2000.

[46] H. Winderix, M. Bognar, J. Noorman, L.-A. Daniel, and F. Piessens. Hardware Support to Accelerate Side-channel Resistant Programs.

[47] H. Winderix, J. T. Muhlberg, and F. Piessens. Compiler-Assisted Hardening of Embedded Software Against Interrupt Latency Side-Channel Attacks. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 667–682, Vienna, Austria, Sept. 2021. IEEE.

[48] M. Wu, S. Guo, P. Schaumont, and C. Wang. Eliminating timing side-channel leaks using program repair. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 15–26, New York, NY, USA, July 2018. Association for Computing Machinery.

[49] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pages 305–316, Raleigh North Carolina USA, Oct. 2012. ACM.

[50] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche. HACL*: A Verified Modern Cryptographic Library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1789–1806, Dallas Texas USA, Oct. 2017. ACM.