UNIVERSITY OF TRENTO

Department of Information Engineering and Computer Science

Bachelor's Degree in
Computer Science

Final Dissertation

Confidentiality Analysis for Move

Supervisor

Marco Patrignani

Student

Luca Giacometti

Academic year 2022/2023

Contents

Abstract

In an era where blockchain technology and smart contracts are reshaping the landscape of digital transactions and decentralized applications, the need for robust security and correctness in the development of blockchain-based systems cannot be overstated.
The Move language, designed by Facebook Diem (formerly known as Libra) project, emerges as a pioneering language tailored for secure and efficient smart contract development. Move fundamental design principles, like resource types and ownership, lay the foundation for robust and reliable blockchain applications and although it addresses safety concerns through its bytecode verifier, ensuring data confidentiality remains a challenge.
This thesis is a response to the call to strengthen the security of Move-based smart contracts: at its core, this work revolves around the development of a confidentiality analysis for Move modules which aims to identify potential vulnerabilities and weaknesses that might lead to unwanted disclosure of sensitive information within the blockchain.

1 Introduction

1.1 Background and Context

2 Background

2.1 Introduction to Move

Smart contract languages confront several significant challenges:

1. **Indirect representation of assets.** An asset is encoded as an integer, a representation that does not precisely capture the essence of the asset itself, leading to potential awkwardness and errors in programming. This becomes particularly evident when attempting to pass assets in and out of procedures or store them in data structures, necessitating specialized language support.

2. **Scarcity is not extensible.** These languages are designed to represent a single scarce asset, with scarcity protections hard-coded into the language semantics. This poses a considerable difficulty for programmers seeking to create custom assets, as they must meticulously reimplement scarcity without any substantial support from the language itself.

3. **Inflexible access control.** The predominant access control policy revolves around a signature scheme based on public keys. This policy, akin to scarcity protections, is deeply entrenched in the language semantics, making it non-trivial to allow programmers to define their own custom access control policies.

The Ethereum Virtual Machine (EVM) allows programmers to publish smart contracts that interact with assets and even define new assets using a Turing-complete language. The high expressivity of the EVM introduces however a new set of risks for implementers of custom assets, as exemplified by the ERC20 standard, which lack the same protections that Ether enjoys thus demanding a meticulous approach to prevent potential bugs leading to duplication, reuse or loss of assets. This complexity is compounded by the indirect representation issue and the dynamic behavior of the EVM.
For instance, in the context of transferring Ether, a smart contract A might call external code B to complete the transaction whose behavior is not known at compile-time, considerably making this a form of dynamic dispatch. The calling contract A might expect B to behave in a certain way, but because the call is dynamic, i.e. resolved at runtime, B could execute unexpected code, including reentering A in a recursive manner before it resolves its internal state. In 2020, DeFi protocols like Uniswap and Lendf.Me lost $25 million each to such vulnerability, known as 'Reentrancy Attack'. TODO: ref

2.1.1 Key Features and Design Principles

The Move language is developed to serve as a robust, programmable bedrock for financial infrastructures, aiming to articulate governance protocols in a clear, comprehensible, and certifiable fashion. For these reasons, its design is underpinned by four pivotal goals: the creation of first-class assets, flexibility, safety and verifiability.

2.1.1.1 First-class Resources

The notion of first-class resources allows for the creation of custom types, a concept influenced by linear logic which ensure resources are neither duplicated nor inadvertently discarded, but are rather moved across program storage locations. This characteristic is upheld statically through Move type system, categorizing resources as conventional program values with the freedom to be embedded in data structures and forwarded as procedure arguments.
The architecture of Move grants programmers the ability to safeguard critical operations through modules, akin to smart contracts in other blockchain dialects, which express resource types and procedures and encapsulate the rules governing resource management, while promoting a high degree of data abstraction and operational integrity within the resource type defining module.

2.1.1.2 Flexibility

The Move language enhances flexibility through the implementation of transaction scripts and modules, which function as the backbone for customizable transaction procedures and code composition respectively. Each transaction in Move encapsulates a transaction script that can invoke multiple procedures from modules present on the blockchain and perform local computations on the yielded results, facilitating either unique one-off behaviors or reusable behaviors by calling a single procedure encapsulating reusable logic.
On the other hand, Move modules usher in a distinct form of flexibility through secure yet adaptable code composition, drawing a parallel to the interrelation among classes, objects, and methods in object-oriented programming but with a notable distinction: a Move module can declare resource types however its procedures lack a `self` or `this` value, resonating more with a restrained version of ML-style modules.

2.1.1.3 Safety

To maintain crucial properties such as resource, type and memory safety, Move adopts a pragmatic approach in choosing a representation that ensures every program executed on the blockchain adheres to these attributes. Instead of solely relying on a high-level programming language with a compiler to check these properties or a low-level untyped assembly to conduct safety checks at runtime, Move finds a middle ground: it employs a typed bytecode as its executable format, which stands at a higher level than assembly but remains lower than a source language. This typed bytecode is subjected to on-chain verification by a bytecode verifier for resource, type, and memory safety before being executed directly by a bytecode interpreter. In other words, Move achieves safety without making the system more vulnerable, by adding a source compiler to the trusted components, or slowing down transactions, by increasing the cost of compilation in the critical path.

2.1.1.4 Verifiability

Move has been designed to balance between the importance of safety assurances and the added complexity that comes with on-chain verification. The chosen strategy is to perform lightweight on-chain verification for critical safety properties while aligning the Move language with advanced off-chain static verification tools.
To make Move more amenable to static verification compared to many general-purpose languages, the following design choices have been made:

1. **No dynamic dispatch.** The target of each call site can be statically determined, which simplifies the task of verification tools, making it easier to understand the effects of function calls without complex call graph analysis.

2. **Limited Mutability.** Move embraces limited mutability, where changes to a value are channeled through references that are temporary and exist only within a single transaction script. Move bytecode verifier employs a borrow-checking mechanism inspired by Rust to ensure that at most one mutable reference to a value exists at any given time.

3. **Modularity.** As mentioned before, modules in Move serve to implement data abstraction and confine essential procedures to specific resources.

2.1.2 Resource Types and Ownership

As introduced in Section 2.1.1, the creation of first-class resources is one of the main goals of Move. Such linear resource types allow to preserve the conservation invariants of monetary assets since they can only be moved, not copied, thus preventing the possibility of double spending and mandating that procedures consume all their resources, mitigating accidental losses. In contrast, other account-based languages like Solidity and Scilla (TODO:ref) utilize maps that associate addresses with integers to depict monetary assets, making the conservation invariants harder to ensure via ad hoc programming. For instance, consider a basic deposit function in a bank Solidity smart contract. The developer has to be particularly careful to increment the user's (TODO: allowed?) credit balance by the deposited amount only once, since failure to do so could result in either the user being cheated or the system being exploited by the user. While Solidity lacks built-in safeguards to prevent such errors, Move

embraces a more structured and secure approach. In Move, the deposit procedure mandates payment by declaring a parameter of type `money` and credits the caller by specifying a return value of type `credit`. Both `money` and `credit` are considered resources so the language type system enforces both consumption of inputs and the return of ownership of the outputs. Furthermore, representation of money as a resource unlocks programming patterns that would not be possible to achieve otherwise: (TODO: references to such patterns from move awesome repo) for example a user could grant another user permission to withdraw their funds by transferring ownership of the credit resource, a feat that would require modifying the smart contract code in Solidity.
Move, being a smart contract language, needs to enforce memory safety by design to promote deterministic execution since the blockchain is a replicated state machine and nondeterministic behaviors may prevent the validators to reach consensus and consequentially stall the entire system. To archive such goal, the following key concepts are adopted:

1. **Tree-shaped values and canonical paths.** Move uses a forest of tree-shaped values to manage memory, where each tree is rooted in an account address that identifies a local variable or a global memory location. You can create references to these values but not nest references within other values: this makes sure that every reference is clearly represented as a path starting from the root and introduces a partial order on reference values, simplifying static reasoning about code that uses them.

2. **Borrow graph.** Move introduces the idea of a directed and acyclic borrow graph (DAG) that overlays the memory structure. This DAG includes nodes that represent different things like values on the call stack, values in global memory of a specific type or references stored on the call stack or operand stack. Each connection in this graph indicates the transfer of ownership along a certain path.

3. **Ownership and borrowing discipline.** The language enforces a programming discipline based on ownership, similar to Rust, where a location in memory is considered the owner of the value it stores. This value can either be moved or modified via this location or 'borrowed', a concept that allows to create a reference without taking ownership of it, similar to having read or write access to the value without being responsible for it. When a value is borrowed, the original owner can't move or modify it until all the borrowed references to it are destroyed, ensuring that the value remains stable and safe to use and preventing issues like dangling references. The rules of ownership and borrowing are not just limited to individual variables but can be applied recursively: for example, if one variable borrows from another, which in turn has been borrowed from a third one, the original owner doesn't regain full control until all layers of borrowing are resolved.

2.1.3 Move Bytecode Verifier

The bytecode verifier has a crucial role in controlling what code can be added to the blockchain, since to publish a module it must first pass certification by the bytecode verifier itself. One important part of this verifier is the borrow analysis, which checks whether a Move bytecode program adheres to the ownership discipline introduced in the chapter before (TODO: ref to prev section), but it also relies on other static checks to enforce resource, type and memory safety:

1. **Stack usage analysis.** This analysis ensures that the stack is used consistently throughout the program, maintaining the same 'shape' or height at points where different paths in the code converge. This is crucial for ensuring that the program behaves as expected and avoids issues like stack overflows or underflows.

2. **Value type analysis.** This analysis checks whether a value has been moved when it shouldn't have been and verifies that each bytecode instruction is applied to values of the correct type.

3. **Acquires analysis.** Checks the correctness of 'acquires' annotations in procedures.

All these static analysis together ensure resource, type and memory safety and especially enforce absence of dangling references, absence of memory leaks and referential transparency for immutable references.

2.1.4 Move Language Excerpts

Move code executes on a stack machine with a distinctive approach to storage management. The architecture partitions the global storage into two specialized domains:

- **Memory.** The memory is a first-order storage where pointers, termed as *locations*, cannot be stored.
- **Globals.** Globals, on the other hand, are tailored to accommodate *locations* using a distinct indexing mechanism based on a literal (an address) and an associated data type.

This partition simplifies the operational semantics when moving values within the operand stack.
Programs in Move are organized hierarchically into modules, symbolized as Ω, that house lists of function declarations, denoted as P: each one encapsulates its input and output types along with a set of instructions, symbolized as $[i]$. The runtime state of the Move stack machine, represented as σ, is a tuple $\langle C, M, G, S\rangle$, encompassing a call stack (C), memory (M), globals (G) and operand stack (S).

- **The call stack.** The call stack C is an ordered list of triples $\langle P, pc, L\rangle$, where P is function name, pc the program counter and L a set of local variables (x) mapped to either locations (ℓ) or values (r).
- **The memory.** The memory M maps *locations* to *values* ($\ell \mapsto r$).
- **The globals.** The globals G maps resource identifiers to *locations* ($\langle a, \rho\rangle \mapsto \ell$). These resource identifiers, defined as $\langle a, \rho\rangle$, comprise an address and a type, which collectively serve to identify both the function and the record type of the global.
- **The operand stack.** The operand stack S functions as a universal repository for values that are either consumed or produced by instructions and those forwarded to (and returned by) functions.

2.1.4.1 Move Operational Semantics

The operational semantics of the Move stack machine is characterized by a small-step semantics, formally represented by the judgment

$$\Omega, P, i \vdash \sigma \to \sigma'$$

which reads *"given a module Ω, an instruction i within a function P transforms the state σ into σ'"*. This mechanism is further refined by incorporating two types of reductions:

- **Global reductions** are guided by the judgment $P, i \vdash \langle M, G, S\rangle \to_{glob} \langle M', G', S'\rangle$. This articulates the behavior of instructions i in function P that specifically modify globals through interactions with either the operand stack S or the memory M.
- **Local reductions** adhere to the judgment $i \vdash \langle M, L, S\rangle \to_{loc} \langle M', L', S'\rangle$ which delineates the impact of instructions i that modify locals only. Similar to global reductions, these changes also take place through the operand stack or the memory, but are obviously restricted to local variables and their associated values.

$$\begin{array}{rl}
\text{instrs.} & \mathbf{Call}\langle P\rangle \mid \mathbf{Ret} \mid \mathbf{BranchCond}\langle pc\rangle \mid \mathbf{Branch}\langle pc\rangle \\
\text{global instrs.} & \mathbf{MoveTo}\langle s\rangle \mid \mathbf{MoveFrom}\langle s\rangle \mid \mathbf{Exists}\langle s\rangle \\
 & \mid \mathbf{BorrowGlobal}\langle s\rangle \mid \mathbf{Pack}\langle s\rangle \mid \mathbf{Unpack}\langle s\rangle \\
\text{local instrs.} & \mathbf{MvLoc}\langle x\rangle \mid \mathbf{StLoc}\langle x\rangle \mid \mathbf{CpLoc}\langle \ell\rangle \\
 & \mid \mathbf{BorrowLoc}\langle x\rangle \mid \mathbf{Pop} \mid \mathbf{LoadConst}\langle a\rangle \mid \mathbf{Op} \\
 & \mid \mathbf{ReadRef} \mid \mathbf{WriteRef} \mid \mathbf{BorrowFld}\langle f\rangle
\end{array}$$

Figure 2.1: instructions of the Move language.

The list of Move instructions is in Figure 2.1 and includes calling, returning, branching conditionally and unconditionally, moving a value from memory to the stack (and back), borrowing a global, checking

$$\frac{L(x) = \ell \quad \ell \in \texttt{dom}(M)}{\mathbf{MvLoc}\langle x\rangle \vdash \langle M, L, S\rangle \rightarrow_{\text{loc}} \langle M \setminus \ell, L \setminus x, M(\ell) :: S\rangle}\ \text{([MoveLoc])}$$

$$\frac{v \in \texttt{StorableValue} \quad \ell \notin \texttt{dom}(M) \quad M' = M \setminus L(x) \text{ if } L(x) \in \texttt{dom}(M) \text{ else } M}{\mathbf{StLoc}\langle x\rangle \vdash \langle M, L, v :: S\rangle \rightarrow_{\text{loc}} \langle M'_{C\setminus\ell}[\ell \mapsto v], L[x \mapsto \ell], S\rangle}\ \text{([StoreLoc])}$$

$$\frac{p = \langle P.\texttt{mid}, s\rangle \quad G(\langle a, p\rangle) = \ell \quad M(\ell) = v}{P, \mathbf{MoveFrom}\langle s\rangle \vdash \langle M, G, a :: S\rangle \rightarrow_{\text{glob}} \langle M \setminus \ell, G \setminus \langle a, s\rangle, v :: S\rangle}\ \text{([MoveFrom])}$$

$$\frac{p = \langle P.\texttt{mid}, s\rangle \quad \langle a, p\rangle \notin \texttt{dom}(G) \quad \ell \notin \texttt{dom}(M) \quad M' = M_{C\setminus\ell}[\ell \mapsto v] \quad G' = G[\langle a, p\rangle \mapsto \ell]}{P, \mathbf{MoveTo}\langle s\rangle \vdash \langle M, G, a :: v :: S\rangle \rightarrow_{\text{glob}} \langle M', G', S\rangle}\ \text{([MoveTo])}$$

$$\frac{\texttt{instr}(\Omega, \sigma) = i \quad i \vdash \langle M, L, S\rangle \rightarrow_{\text{loc}} \langle M', L', S'\rangle}{\Omega \vdash \langle\langle P, \text{pc}, L\rangle :: C, M, G, S\rangle \rightarrow \langle\langle P, \text{pc}+1, L'\rangle :: C, M', G, S'\rangle}\ \text{([Step-Loc])}$$

$$\frac{\texttt{instr}(\Omega, \sigma) = i \quad P, i \vdash \langle M, G, S\rangle \rightarrow_{\text{glob}} \langle M', G', S'\rangle}{\Omega \vdash \langle\langle P, \text{pc}, L\rangle :: C, M, G, S\rangle \rightarrow \langle\langle P, \text{pc}+1, L\rangle :: C, M', G', S'\rangle}\ \text{([Step-Glob])}$$

Figure 2.2: semantics of the Move language (excerpts).

the existence of a global, packing and unpacking a record, moving a value to the local stack (and back), copying it, borrowing it, popping a value, loading a constant, binary operations, reading (and writing) to memory and accessing a record field.
Figure 2.2 presents only the judgments that we actually need for our type of analysis. We indicate accessing a map, e.g. the memory, as $M(\ell)$, updating its content as $M[\ell \mapsto v]$ and its domain as $\texttt{dom}(M)$. Lists of elements K are $[K]$ and its length is denoted with $\|[K]\|$, while $P.\texttt{mid}$ is the module identifier of the procedure P, $P.\texttt{inty}$ is the list of inputs types and $P.\texttt{rety}$ is the list of return types. Function $\texttt{instr}(\Omega, \sigma)$ returns the current instruction by looking it up in the codebase Ω given the current function and the program counter from the top of the call stack in σ.

2.1.4.2 Trace Semantics

We now introduce a big-step trace semantics, building on the foundation of the previously established small-step operational semantics, which aims to capture the execution traces of a trusted code segment. The semantic is structured on three levels:

1. At the primary level, we have a single-step, single-labelled semantics, whose core responsibility is the generation of individual actions. The judgments are represented as:
$$\Omega^{\dagger} \triangleright \Omega, P, i \vdash \sigma \xrightarrow{\alpha} \sigma$$
2. In the secondary level we have a big-step, single-labelled semantics. This is essentially the reflexive-transitive closure of the preceding level and its judgment is symbolized as:
$$\Omega^{\dagger} \triangleright \Omega, P \vdash \sigma \overset{\alpha}{\Longrightarrow} \sigma$$
3. Lastly we have the big-step, trace-labelled semantics which integrates all the big-step, single-labelled steps presented before, concatenating them into a trace. The judgment here is represented as:
$$\Omega^{\dagger} \triangleright \Omega, P \vdash \sigma \overset{\bar{\alpha}}{\Longrightarrow\!\!\!\!\!\!\Rightarrow} \sigma$$

The trace semantics is conscious of the point from which the trace originates: this aspect is encapsulated in the judgments, where it builds upon the operational semantics judgment by incorporating this

$$\frac{\begin{array}{c}\textsf{(Action-No)}\\ \texttt{instr}(\Omega,\sigma)=i \quad \Omega\vdash\sigma\to\sigma' \quad \sigma=\langle C,M,G,S\rangle\\ (i\neq\textbf{Call}\ \text{ and }\ i\neq\textbf{Ret})\end{array}}{\Omega^{\dagger}\rhd\Omega\vdash\sigma\xrightarrow{[]}\sigma'}$$

$$\frac{\begin{array}{c}\textsf{(Action-Call)}\\ \texttt{instr}(\Omega,\sigma)=\textbf{Call}\langle P_0\rangle \quad \Omega\vdash\sigma\to\sigma' \quad \sigma=\langle C,M,G,S\rangle\end{array}}{\Omega^{\dagger}\rhd\Omega\vdash\sigma\xrightarrow{\texttt{call}\ P_0\ M,G}\sigma'}$$

$$\frac{\begin{array}{c}\textsf{(Action-Return)}\\ \texttt{instr}(\Omega,\sigma)=\textbf{Ret} \quad \Omega\vdash\sigma\to\sigma' \quad \sigma=\langle C,M,G,S\rangle\end{array}}{\Omega^{\dagger}\rhd\Omega\vdash\sigma\xrightarrow{\textsf{ret}\ M,G}\sigma'}$$

$$\frac{\begin{array}{c}\textsf{(Single)}\\ \Omega^{\dagger}\rhd\Omega\vdash\sigma\xRightarrow{[]}\sigma''\\ \sigma''=\langle P,pc,L\rangle::C,M,G,S \quad \Omega^{\dagger}\rhd\Omega\vdash\sigma''\xrightarrow{\alpha}\sigma'\end{array}}{\Omega^{\dagger}\rhd\Omega\vdash\sigma\xRightarrow{\alpha}\sigma'}$$

$$\frac{\begin{array}{c}\textsf{(Trace-Both)}\\ \Omega^{\dagger}\rhd\Omega\vdash\sigma\overset{\bar{\alpha}}{\Longrightarrow\!\!\!\!\Rightarrow}\sigma''\\ \Omega^{\dagger}\rhd\Omega\vdash\sigma''\xRightarrow{\alpha?}\sigma''' \quad \Omega^{\dagger}\rhd\Omega\vdash\sigma'''\xRightarrow{\alpha!}\sigma'\end{array}}{\Omega^{\dagger}\rhd\Omega\vdash\sigma\overset{\bar{\alpha}::\alpha?::\alpha!}{\Longrightarrow\!\!\!\!\Rightarrow}\sigma'}$$

$$\frac{\begin{array}{c}\textsf{(Trace-Single)}\\ \Omega^{\dagger}\rhd\Omega\vdash\sigma\overset{\bar{\alpha}}{\Longrightarrow\!\!\!\!\Rightarrow}\sigma''\\ \Omega^{\dagger}\rhd\Omega\vdash\sigma''\xRightarrow{\alpha?}\sigma''' \quad \neg(\Omega^{\dagger}\rhd\Omega\vdash\sigma'''\xRightarrow{\alpha!}\sigma')\end{array}}{\Omega^{\dagger}\rhd\Omega\vdash\sigma\overset{\bar{\alpha}::\alpha?}{\Longrightarrow\!\!\!\!\Rightarrow}\sigma'}$$

Figure 2.3: trace semantics for Move programs (excerpts).

context, denoted by the $\Omega^{\dagger}$ on the left.

Action-No indicates instructions that do not need to be traced, unlike a **Call**, a **Ret** or a jump not confined within the bounds of trusted code. Action-Call traces a function call and Action-Return captures control flow transfer between them. Finally rule Single concatenates a series of empty steps followed by an action as a single action that is then used by rules Trace-Both and Trace-Single to generate a trace.

2.2 Non-interference and Confidentiality

In fields where guarantees of safe manipulation of private data is needed, the standard way to protect confidentiality is access control, demanding special privileges to parties requesting read access to such data. This method however cannot control propagation of information.

Non-interference, on the other hand, addresses the critical domain of information flow. It operates on a foundational principle: the actions, decisions or data access patterns of one user should never inadvertently influence, or become observable to, another user, especially if the two users are classified under different security levels.

For instance, consider a system where users A and B have high (considered as secret) and low (considered as public) security clearances, respectively. If user A engages in specific activities or accesses classified data, non-interference dictates that such actions should neither be discernible to user B nor should they alter the data user B can access.

2.2.1 Ellis Cohen's Strong Dependency

Since information flow in sequential programs happens via variables, the problem of confidentiality is strictly related to the dependencies among data, introduced by the notion of strong dependency formalized by Ellis Cohen [TODO: ref].

Definition 1. Variables L are strongly dependant on H, meaning the program P is not secure, is defined as:

$$H \rhd^P L \overset{\text{def}}{=} \exists\, \sigma_1, \sigma_2 : \ \sigma_1^L = \sigma_2^L \wedge [\![P]\!]\,(\sigma_1)^L \neq [\![P]\!]\,(\sigma_2)^L$$

Where:

- L is the set of public variables and H is the set of private ones.
- σ_1 and σ_2 are states of P, namely tuples of values for the variables in P.
- σ^L is the tuple of values in σ for the variables in L.

This informally means that if the variable A can have different initial values and this leads to variable B having different values after running the program P, then we can say that B strongly depends on A. To show that information transmission is possible, only two different input values for A that yield different values for B after the execution of P are needed.

2.2.2 Denning's Lattice Model

Denning's lattice model offers a comprehensible mathematical basis for secure information flow analyses, where the arrangement of the lattice defines the authorized information flows. The lattice consists of a partially ordered set of security levels, with some of them being more restrictive, i.e. higher, than others.
The information flow model F is defined by five primary elements: $\langle N, P, SC, \oplus, \rightarrow\rangle$.

- N represents a collection of objects, denoted as $\{a, b, \ldots\}$. Depending on the level of detail needed, these objects may represent files, segments or program variables, as in our case.
- P is the set of processes, symbolized as $\{p, q, \ldots\}$, the active agents causing the flow of information.
- SC is the lattice of *security classes* expressed as $\{A, B, \ldots\}$, which represents the security tier of the objects within the system. Every unique object, say a, is intrinsically linked to a security level, termed A, that indicates the protection level of the data held within. By definition, the lattice has a *least upper bound* - the join - and a *greatest lower bound* - the meet - operator, the former symbolized by '$\oplus$' and the latter by '$\otimes$'.
- '$\oplus$' is the join, a class-combining operator that defines the security class for the outcome of any binary function.
- '$\rightarrow$' is the flow relation among classes. For instance, if we have two classes, A and B, the expression $A \rightarrow B$ would mean that data from class A can be channeled into class B. In simpler terms, data 'flows' from A to B when the data tied to A influences or alters the value of the data linked to B.

The integrity of model F is upheld only if, during any operation sequence, the data flow abides by the defined '$\rightarrow$' relation, avoiding any breaches.

2.2.2.1 Explicit and Implicit Flow

Detecting all flow causing operations is not trivial, since not all of them are explicitly typed in a function or even execute. Explicit flows are those caused by statements that directly transfer information among variables (e.g. assignments), while implicit flows are often associated with conditional statements where the choice of which path to take can be influenced by sensitive data (e.g. a secret guard), leading to flow of information if the attacker can observe these choices. Consider the code `if a = 0 then b = 0`; if $b \neq 0$ initially, then testing $b = 0$ after the execution gives us information whether $a = 0$ or not, irrespective of the `then` clause being executed or not.
To further consolidate this notion, consider the following example which creates both explicit and implicit flow: `if a = 0 then b = c`;

- **Explicit flow.** The instruction `b = c` causes an explicit flow from variable `c` to `b` when the condition is true, since the assignment *directly* transfers information.
- **Implicit flow.** An implicit flow occurs from variable `a` to variable `b` irrespective of whether the condition is true or false. The implicit flow arises from the fact that the value of `a` influences the behavior of the program, and this influence can potentially reveal information about it even when it is not directly assigned to `b`. It is implicit because it leaks information *indirectly* through control flow.

2.2.2.2 Security Enforcement

In [TODO: ref], Denning recursively defines an abstract program S as:

1. S is an elementary statement or instruction, e.g. an assignment.
2. There exists S_1 and S_2 such that $S = S_1; S_2$. This declares sequences of simpler programs as abstract programs.
3. There exists $S_1, \ldots, S_n$ and a n-valued variable c such that $S = c : S_1, \ldots, S_n$. This declares conditional structures, in which the value of the variable c is selected among alternative programs, i.e. different decision paths. This structure represents all conditionals and iterative statements found in programming languages and is where implicit flows happens.

Given this definition, Denning defines the security requirements as follows.

1. An elementary statement S is secure if any explicit flow caused by S is secure. In particular if S replaces the contents of any object b with a value derived from $a_1, \ldots, a_n$, then it is required that $a_1 \oplus \ldots \oplus a_n \to b$ holds after execution.
2. $S = S_1; S_2$ is secure if both S_1 and S_2 are secure, given the transitivity of '$\to$'.
3. $S = c : S_1, \ldots, S_n$ is secure if each S_i is secure and each implicit flow from c is secure. In particular, let $b_1, \ldots, b_n$ be the objects into which S specifies explicit flows, then all implicit flows are secure if $c \to \; b_1 \otimes \ldots \otimes b_n$ holds after execution, where '$\otimes$' is the greatest lower bound operator.

2.3 Static Code Analysis and Principles of Dataflow Analysis

Static analysis refers to the examination of source code or compiled bytecode without actually executing the program. This technique is employed to uncover various properties of the code such as potential errors, security vulnerabilities or deviations from coding standards. One notable instance of a static analysis tool is *Lint* for C programming, which performs rudimentary checks for syntax and logical errors (TODO: Johnson - Lint, a C program checker) or *FindBugs* for Java, which identifies potential bugs like null pointer exceptions and infinite loops (TODO: Finding bugs is easy. ACM Sigplan Notices).
Static analysis has also been extended to more complex evaluations such as dataflow analysis, which aims to gather information about the possible set of values that variables can take and how these values propagate through a program and is typically accomplished by examining the control-flow graph (CFG) of the program.

2.3.1 Control-Flow Graphs

Control-flow graphs provide a structural representation trough a directed graph that encapsulates the potential execution paths through a program or a particular function. In a CFG, nodes correspond to basic blocks, i.e. contiguous sequences of code with a single entry and a single exit point, while edges signify possible transitions between these basic blocks due to program control statements like loops, conditionals and jumps. This graphical representation allows for a more intuitive understanding of the structure of a program, and it is instrumental in a myriad of applications ranging from compiler optimizations to various forms of static and dynamic analysis. For instance, dataflow analysis techniques operate on CFGs to propagate information along the edges.

```
fun test(): u64 {
    // Block 1: Initialize variables
    let i = 0;
    let total = 0;
    // Block 2: Check condition
    while (i < 10) {
      // Block 3: Execute loop
      total = total + i;
      i = i + 1;
    };
    // Block 4: End loop and return
    return total;
}
```

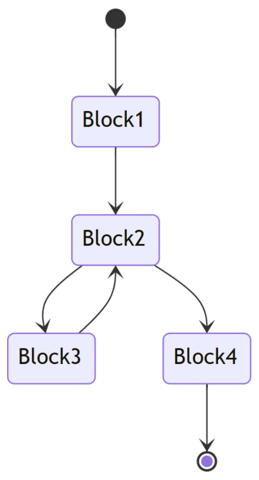


Figure 2.4: possible CFG of a Move function.

Despite their apparent simplicity, CFGs encapsulate a level of complexity when loops or recursion are present, as these create cycles in the graph that must be carefully managed with techniques, like loop unrolling or strongly connected components (SCCs) decomposition, that simplify the CFG without altering the essential semantics of the original program.

2.3.2 Kildall's Approach to Dataflow Analysis

Gary Kildall's algorithm for dataflow analysis employs the fixed-point iteration method to reach a stable solution for a set of dataflow equations. The approach works by approximating the dataflow values for the entry block of the CFG, often setting it to an extreme value like 'empty' or 'universal set' depending on the problem at hand, and then enters an iterative process that aims to compute the successor blocks. During each iteration, the algorithm traverses the basic blocks and updates the dataflow values based on predefined transfer and confluence functions.

- **Transfer function**. The transfer function models how the basic block transforms its input dataflow values into output values, based on the instructions it contains.
- **Confluence function**. The confluence function is used to combine dataflow values from multiple incoming edges into a single value at the join points in the CFG.

The notion of a fixed point comes into play when these iterative updates no longer result in any changes to the approximated dataflow values, i.e. the application of the transfer and confluence functions repeatedly does not lead to any further changes. In dataflow analysis based on the Denning's model (Section 2.2.2), the set of possible dataflow values is designed to form a complete lattice $(L, \leq)$, where L is the set of all possible dataflow values, and $\leq$ is a partial order that satisfies the following properties:

- **Reflexivity**: $\forall a \in L, a \leq a$.
- **Antisymmetry**: $\forall a, b \in L, a \leq b \wedge b \leq a \implies a = b$.
- **Transitivity**: $\forall a, b, c \in L, a \leq b \wedge b \leq c \implies a \leq c$.

We also need to enforce the following:

- **Monotonicity.** The transfer function $f : L \to L$ must be monotonic which formally means that $\forall a, b \in L, a \leq b \implies f(a) \leq f(b)$.
- **Completeness.** The lattice must be complete, meaning all its subsets have both a least upper bound and a greatest lower bound. This is trivial in our case, since every non-empty finite lattice is complete, like the one presented in section (TODO:ref) for our dataflow analysis. Moreover, another property known as boundedness, inherently linked to the completeness property, ensures that the iterative process starts and ends at a well-defined value.

The monotonic transfer function applied to the complete lattice will always allow for at least one fixed point, but in our dataflow abstraction we only consider the *least fixed point* which is the least upper bound of the set of fixed points, proved to be a complete lattice, non-empty by definition, by the *Knaster-Tarski theorem.* (TODO: ref)

2.3.3 The Worklist Approach

The first and most basic implementation of the Kildall's method is the *round-robin iterative algorithm*, shown below.

```
for i in 1..N
  initialize block i in-state
while (states are still changing)
  for i in 1..N
    recompute out-state of block i
```

It starts with an estimated input state for each block in the system, commonly referred to as the 'in-state', then computes 'out-states' by applying the transfer functions to these in-states. Subsequently, these newly computed states are used to update the original in-states through the confluence function or 'join function'. The process of calculating out-states and updating in-states through joins is iteratively performed until no further changes are observed in any state, indicating that the fixed-point has been reached.
This algorithm has a rigid structure since it visits every block in a fixed order, making it suboptimal in practice. Nowadays most dataflow analysis are implemented via a more efficient algorithm based on the worklist approach, where the worklist consists of either a stack or a queue of blocks to be processed.

```
Worklist = ∅

for i in 1..N
  initialize block i
  add i to the worklist

while (Worklist ≠ ∅)
  pop block i from the worklist
  recompute out-state of block i
  if (out-state of block i changed)
    add successors of i to Worklist, uniquely
```

The algorithm enters into a loop where it continuously picks in-states of blocks from the worklist to compute their out-state. Crucially, if any block experiences a change in its out-state due to this update, all its successor blocks are added back to the worklist for re-evaluation. Compared to the round-robin algorithm, which cycles through all blocks irrespective of whether they've changed or not, this approach is more efficient since it narrows down the areas of the CFG that require re-computation, focusing only on the blocks where information has actually changed.

3 Architecture and Components

3.1 Design of confidentiality analysis for Move

Presented the above notions, crucial for the understanding of the confidentiality analysis implementation, this section dives into the security classes lattice and the judgments design.

3.1.1 Lattice of Security Classes

In accordance with the Denning's lattice model (Section 2.2.2), the lattice of security classes serves as an abstraction mechanism for local variables, stack locations and the program counter (PC). It is composed of two values, *Public* and *Secret*, low security and high security respectively, while its ordering is $Public \sqsubseteq Secret$. The need to abstract the PC value comes from the existence of implicit flows, described in (TODO: ref), which may inadvertently disclose sensitive information via control flow structures. The security level of the PC is elevated to a high-security state during conditional branching operations that rely on a high-security condition guard, restricting the execution of operations that may enable data flow, such as function calls and returns.
To put it simply, the goal of this analysis is to prevent:

- **Explicit flows** by flagging returns of *Secret* values or forwarding *Secret* arguments to function call.
- **Implicit flows** by flagging any return or call to function while PC is *Secret*.

3.1.2 Analysis Judgments

The confidentiality analysis Ξ executes on a module Ω by traversing all its functions P and verifying each instruction i accordingly to judgments formally defined as:

$$\Omega, P, i \vdash \left\langle \hat{pc}, \hat{L}, \hat{S} \right\rangle \rightsquigarrow \left\langle \hat{pc'}, \hat{L'}, \hat{S'} \right\rangle$$

which reads *"instruction i (in function P of module Ω) consumes abstract program counter $\hat{pc}$, abstract locals $\hat{L}$ and abstract globals $\hat{S}$ and produces $\hat{pc'}$, $\hat{L'}$ and $\hat{S'}$"*. Note that the hat $\hat{}$ on top of locals, globals and PC indicates their security class, either *Public* or *Secret* as defined by the lattice.
Rule Ξ|BranchCond applies to conditional branches and changes the abstract value of the PC based on the least upper bound of the current PC and the security class of the condition guard, while Ξ|Return and Ξ|Call ensure that both return statements and calls to functions are executed exclusively during a *Public* PC state and that all variables returned and function arguments are correspondingly mapped to a *Public* abstract value.

3.2 Implementation of confidentiality analysis for Move

The Move Prover incorporates an abstract dataflow analysis framework, used for other forms of static code analysis such as liveness analysis, borrow analysis, and escape analysis (TODO: ref mp's paper), that translates a given Move module into its corresponding bytecode, which is subsequently segmented into basic blocks, defined as contiguous sequences of code that terminate in a control transfer instruction, such as a return, break, continue or branch statement. These basic blocks are then assembled to construct the Control Flow Graph (CFG) for each function within the module. Finally, the analysis iteratively evaluates each CFG using the worklist algorithm to identify any undesirable data flow.
Given that the foundational structure for this type of analysis in the Move Prover is well-established, this paper will focus on three key aspects that are essential for developers, including myself, who seek to implement additional dataflow analyses.

$$\frac{(\Xi|\mathsf{BranchCond})}{\boldsymbol{BranchCond}\langle pc\rangle \vdash \left\langle \hat{pc}, \hat{L}, \hat{v} :: \hat{S} \right\rangle \rightsquigarrow \left\langle \hat{v} \oplus \hat{pc}, \hat{L}, \hat{S} \right\rangle}$$

$$\overset{(\Xi|\mathsf{Return})}{\frac{\|\Omega(P).3\| = n \quad \forall i \in 1..n.\hat{v_i} \neq \mathit{Secret} \quad \hat{pc} = \mathit{Public}}{\boldsymbol{Ret} \vdash \left\langle \hat{pc}, \hat{L}, \hat{v_1}...\hat{v_n} :: \hat{S} \right\rangle \rightsquigarrow \left\langle \hat{pc}, \hat{L}, \hat{v_1}...\hat{v_n} :: \hat{S} \right\rangle}}$$

$$\overset{(\Xi|\mathsf{Call})}{\frac{\|\Omega(P_0).type\| = n \quad \forall i \in 1..n.\hat{v_i^a} \neq \mathit{Secret} \quad \hat{pc} = \mathit{Public}}{\boldsymbol{Call}\langle P_0\rangle \vdash \left\langle \hat{pc}, \hat{L}, \hat{v_1^a}...\hat{v_n^a} :: \hat{S} \right\rangle \rightsquigarrow \left\langle \hat{pc}, \hat{L}, \hat{v_1^r}...\hat{v_n^r} :: \hat{S} \right\rangle}}$$

Figure 3.1: Ξ confidentiality analysis excerpts.

3.2.1 Transfer function

The dataflow analysis framework introduced before exposes a generic trait of the transfer function, a mathematical abstraction that models the effect of a program statement or instruction on the dataflow state. Developers are required to implement the logic responsible for computing intermediate states that serve as fundamental steps for propagating changes from the initial in-state to the final out-state, based on the sequence of instructions present within the block under analysis. Essentially, this component enforces the judgments previously discussed - in practical terms, this has been achieved by matching on the Move bytecode instructions and modifying the dataflow domain accordingly.

```rust
impl TransferFunctions for ConfidentialityAnalysis {

  fn execute(&self, state: &mut State, instr: &Bytecode, offset: CodeOffset) {
    use Bytecode::*;
    use Operation::*;

    match instr {
        Call(..) => (),
        Ret(..) => (),
        Branch(..) => (),
        Assign(..) => (),
        Load(..) => (),
        Jump(..) | Label(..) | Abort(..) | Nop(..)
        | SaveMem(..) | SaveSpecVar(..) | Prop(..) => (),
    }
  }
}
```

3.2.2 Analysis domain

As noted in Section 3.2.1, the analysis needs to keep track of intermediate states, composed by the abstract value of the program counter (PC) and the taint of both locals and globals.
While the latter is pretty straightforward, since it only needs to map each local to a security level and propagate the information to the next iteration, the former is more complex as it needs to retain a history of previous PC states. In fact, this could be compared to the concept of lexical scoping in programming languages, a feature present from as early as the 1960s with ALGOL 60, and commonly implemented using a technique known as 'stack-based scoping'. As the name suggests, upon entering a new block, e.g. the 'then' block of an 'if' statement, a new scope is pushed onto a stack and modifications made within it are isolated from the parent scope, i.e. the previous one. Once the block execution is complete, the scope is popped off the stack, and control returns to the parent, effectively reverting the changes made in the child scope. This scoping mechanism closely resembles the behavior we want to mimic: it allows to recover the previous PC states upon exiting a block while

accommodating the complexities introduced by nested block structures.
Intuitively, the domain of our analysis is a tuple composed of a *StackDomain* for the abstract value of the PC and a *MapDomain* for locals taints: under the hood, the latter is unsurprisingly a map, where each local is uniquely indexed by a number, while the former is vector. Note that to mitigate potential performance issues, both data structures come from the `im` crate which introduces structural sharing, a feature where if two data structures largely replicate each other, the majority of their memory usage is commonly shared, allowing for fast and efficient copies. TODO: ref

```
  struct AnalysisState<K,V>(StackDomain<V>, MapDomain<K,V>);

  struct StackDomain<E>(Vector<E>);
  struct MapDomain<K,V>(OrdMap<K,V>);
```

3.2.3 Confluence function

As introduced by both Kildall's dataflow approach (Section 2.3.2) and the Denning's lattice model (Section 2.2.2), the confluence function, or join function, combines the newly generated out-state of the current block to the in-state of its successor to check for any changes in the program counter or local variables taints. If an alteration is observed, the in-state of the succeeding block is subsequently re-enqueued into the worklist for further analysis.
The following code is the confluence function of the *StackDomain*, which basically strips out the last PC abstract value from the stack.

```
fn join(&mut self, other: &Self) -> JoinResult {
  // self is successor block pre state, other is current block post state
  let prev_state = if other.len() > 1 {
      // get prev state in other if exists - safe unwrap
      other.get(other.len() - 2).unwrap()
  } else {
      // current one otherwise
      other.back().unwrap()
  };
  // remove one layer from self stack
  if self.len() > 1 {
      self.pop_back();
  }
  // compute join result
  if prev_state == self.back().unwrap() {
      return JoinResult::Unchanged;
  }
  JoinResult::Changed
}
```

In the transfer function of the *MapDomain*, the in-state of the succeeding block inherits the abstract values obtained from the least upper bound of itself and the current block out-state.

```
fn join(&mut self, other: &Self) -> JoinResult {
  let mut change = JoinResult::Unchanged;
  for (k, v) in other.iter() {
    change = change.combine(self.insert_join(k.clone(), v.clone()));
  }
  change
}

// Join ‘v‘ with self[k] if ‘k‘ is bound, insert ‘v‘ otherwise
fn insert_join(&mut self, k: K, v: V) -> JoinResult {
  let mut change = JoinResult::Unchanged;
  self.0
    .entry(k)
    .and_modify(|old_v| {
      change = old_v.join(&v);
    })
    .or_insert_with(|| {
      change = JoinResult::Changed;
      v
    });
  change
}
```

Lastly, the transfer function of the whole analysis state is a basic domain combinator.

```
fn join(&mut self, other: &Self) -> JoinResult {
    let stack_join = self.0.join(&other.0);
    let map_join = self.1.join(&other.1);
    stack_join.combine(map_join)
  }
```

4 Tests and Results

In this chapter we present some test cases for our confidentiality analysis, used to measure its precision. In spite of the few test scenarios, they represent the majority of the ways in which the flow of data occurs in reality, thus making it effective.
We have successfully integrated the confidentiality analysis Ξ into the Move Prover analysis framework with a Rust code implementation spanning approximately 300 lines. The framework contains libraries for Move bytecode generation, CFG construction and abstract dataflow analysis via fixed point iteration that are not included in the total above. The confidentiality analysis is available at: https://github.com/lucagiacometti19/move

4.1 Description of test cases

We will first focus on the two ways data can leak, implicit and explicit flow, with tests aimed at triggering each judgment of the analysis. Lastly, we will present a few more complex ones to prove the ability of our analysis to correctly manage nested block structures.

4.1.1 Explicit Flow Tests

The following four cases have been defined:

```
1    module 0xCAFE::ExplicitFlow {
2
3      fun dummy_fn(arg: u64) : u64 {
4        return 0;
5      }
6
7      fun case1(secret_val: u64) : u64 {
8        dummy_fn(secret_val)
9        return 0;
10     }
11
12     fun case2(secret_val: u64) : u64 {
13       let secret_val_2 = secret_val;
14       dummy_fn(secret_val_2);
15       return 0;
16     }
17
18     fun case3(secret_val: u64) : u64 {
19       return secret_val;
20     }
21
22     fun case4(secret_val: u64) : u64 {
23       let secret_val_2 = secret_val;
24       return secret_val_2;
25     }
26   }
```

Case 1 should be flagged at line 8 because of the call to a user defined function - in our case the function `dummy_fn` - while forwarding a secret variable to it - `secret_val` is mapped as secret since it is an argument of the test.

Case 2 should be flagged at line 14 for the same reason as before. This time the local `secret_val_2` is flagged as secret because of the assignment at line 13: note that although this is a basic test, any other instruction on the right-hand side of the assignment, like binary operations, packing and unpacking or borrowing of fields, is correctly managed by the analysis in its previous steps and will necessarily fall under this same case.

Case 3 is basically Case1 applied to the judgment about return instructions and should be flagged at line 19 because of the return of the secret local `secret_val`.

Case 4 is Case2 applied to the judgment about return instructions and should be flagged at line 24 because of the return of the secret local `secret_val_2`, which should inherit its abstract value from the assignment with `secret_val` at line 23.

4.1.2 Implicit Flow Tests

The following three cases have been defined:

```
 1  module 0xCAFE::ImplicitFlow {
 2
 3    fun dummy_fn(arg: u64) : u64 {
 4      return 0;
 5    }
 6
 7    fun case1(secret_val: u64): u64 {
 8      if (secret_val < 0) {
 9        dummy_fn(0);
10      };
11      return 0;
12    }
13
14
15    fun case2(secret_val: u64): u64 {
16      if (secret_val < 0) {
17        return 1;
18      };
19      return 0;
20    }
21
22    fun case3(secret_val: u64): u64 {
23      let secret_val_2 = 0;
24      if (secret_val < 0) {
25        secret_val_2 = 1;
26      };
27      return secret_val_2;
28    }
29  }
```

Case 1 should be flagged at line 9 because of the call to the function `dummy_fn` while the PC is secret. The PC value should be raised by the evaluation of the condition guard - since it includes `secret_val` - and should be kept as such during the whole 'then' block execution of the conditional statement. Upon exiting the branch, it should be lowered so that the return statement at line 11 does not get flagged.

Case 2 should be flagged at both line 17 and line 19 because of the return during the secret PC state. Note that the second return statement should be flagged as well, even though it is outside of the secret 'then' branch, since not executing the first one means that the second return will inevitably execute, causing indirect flow of information about the value of the condition guard `secret_val` based on which statement gets executed.

Case 3 should be flagged at line 27 because of the return of a secret local. Although this is exactly the definition of explicit flow, it has been included here because of implicit information propagation. Note, in fact, that at line 25 there is an assignment to the local `secret_val_2` while the PC is raised to secret: returning its value indirectly tells whether the conditional statement executed or not, sharing a piece of information about the value of the secret condition guard.

4.1.3 Nested Blocks Tests

The following two cases have been defined:

```
 1    fun case1(secret_val: u64) : u64 {
 2      let x = 7;
 3      if (x == 7) {
 4        if (secret_val == 0) {
 5          if (x == 7) {
 6            return 0
 7          };
 8        } else {
 9          return 2
10        }
11      };
12      return 1
13    }
```

Case 1 should be flagged at both line 12 and line 15 for implicit flow via return statement. When the execution progresses to the second conditional statement at line 10, the PC should be raised to a secret state. This state should be maintained as the analysis enters the third conditional at line 11, despite its condition guard being public.

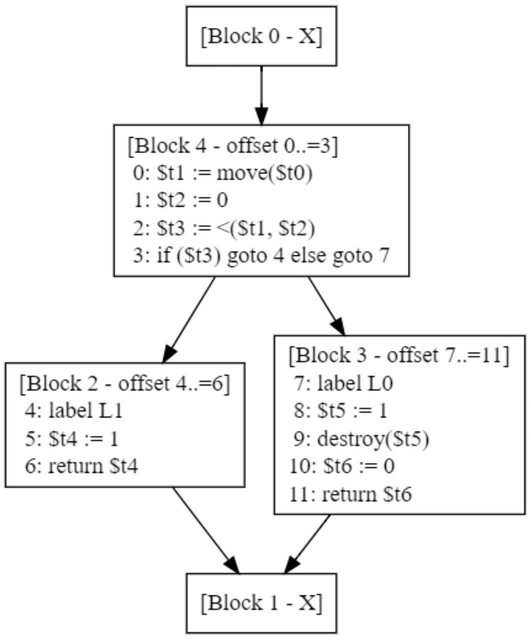
```
fun case2(secret_val: u64) : u64 {
  let i = 10;
  while (i > 0) {
      i = i - 1;
      dummy_fn(i);
      if (secret_val == 0) {
          break;
      };
      dummy_fn(i);
  };
  return i
}
```

Case 2 should be flagged at line 29 because of the call to the function `dummy_fn` while the PC is secret. This scenario parallels the second case in the implicit flow module (Section 4.1.2). However, the focal point of interest in this context is that our analysis correctly handles loops: at the bytecode level, loops basically consists of conditionals and the looping mechanism is enforced through the manipulation of edges of the CFG.

4.2 Interpretation of Results

The confidentiality analysis effectively identified and flagged the precise lines of code responsible for information leakage outside of the current module. Importantly, this analysis did not overlook any such instances, ensuring comprehensive detection.
Before moving on, let us briefly expand on probably the most interesting scenario generated by the previous tests: case 2 of the implicit flow module (Section 4.1.2). I have already mentioned the not-so-obvious reason why both return statements are correctly flagged, however note that, following the rule BranchCond, the PC should be lowered to a public state after exiting the 'then' branch, meaning that the second one should not be detected. This particular test case has been appositely chosen to illustrate the non-trivial nature of identifying implicit data flows. In fact, its code could be refactored - without altering its essence - as in Figure 4.1, where the second statement resides within the 'else' branch of the conditional. This also demonstrates how case 2 is effectively compiled into Move bytecode due to code optimizations and shows how the analysis handles such scenarios by leveraging the inherent structure of the CFG generated by the compiler, obviating the necessity to design and implement any unique behavior.

```
fun case2(secret_val: u64): u64 {
   if (secret_val < 0) {
     return 1;
   } else {
     return 0;
   };
 }
```



```
[Block 0 - X]

[Block 4 - offset 0..=3]
 0: $t1 := move($t0)
 1: $t2 := 0
 2: $t3 := <($t1, $t2)
 3: if ($t3) goto 4 else goto 7

[Block 2 - offset 4..=6]
 4: label L1
 5: $t4 := 1
 6: return $t4

[Block 3 - offset 7..=11]
 7: label L0
 8: $t5 := 1
 9: destroy($t5)
10: $t6 := 0
11: return $t6

[Block 1 - X]
```

Figure 4.1: refactor of case 2 on the left - CFG of the bytecode of case 2 on the right.

5 Conclusions

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Bibliography

Appendix A Attachment

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.