CMSC838L Final Report

Preventing Timing Side Channels in High Level Synthesis

28.06.2026

Sai Divvela, Danesh Sivakumar, Eric Yu
University of Maryland

Contents

1. Introduction

High-level synthesis (HLS) is an appealing alternative to designing domain specific accelerators. It allows programmers to reason about hardware behavior with a language at a higher level of abstraction, such as C or C++. Hardware accelerators have many different applications, including in machine learning, genomics, cryptography and more [1], [2], [3]. In many of these applications, the accelerators interact with a variety of data, including data that is highly sensitive and must be kept secure. A promising approach to dealing with this challenge is information flow control (IFC). Information flow control aims to solve the problem of preventing public observers and attackers from learning about sensitive or secret data during a computation. It does so by preventing flows from secret to public that are both *direct* and *implicit*. One major class of attacks that modern hardware faces due to implicit flows is *side channel attacks*. These can be difficult to reason about, as they attempt to exploit a correlation between secret data and physical device state, such as energy, time spent, or memory usage. As such, they can target a wide variety of features present on a chip or accelerator. Among these side channel attacks, timing side channels are one of the oldest and most well-known [4]. They can lead to devastating attacks such as the Spectre vulnerability, wherein an attacker exploits speculative execution in CPUs to trick programs into leaking sensitive data. Prior work on preventing these timing side channels has either occurred at a lower level of abstraction, such as at the hardware description language (HDL) level [5], [6], or relies on invisible changes and opaque static analyses that can be hard for programmers to reason about [7], [8]. In our work, we propose a type system at the HLS level to defend against these timing side channels by enforcing a property known as *time-sensitive noninterference*.

2. Motivation

To motivate our problem and provide context, we will look at a classic example of a timing side channel attack. Consider the following code snippet that implements a password checker:

```
1   string secretpassword = "PASSWORD";
2
3   bool checkPassword (string password) {
4     if (secretpassword.length != password.length)
5       return false;
6     for (i = 0; i < secretpassword.length; i++) {
7       if (secretpassword[i] != password[i])
8         return false;
9     }
10    return true;
11  }
```

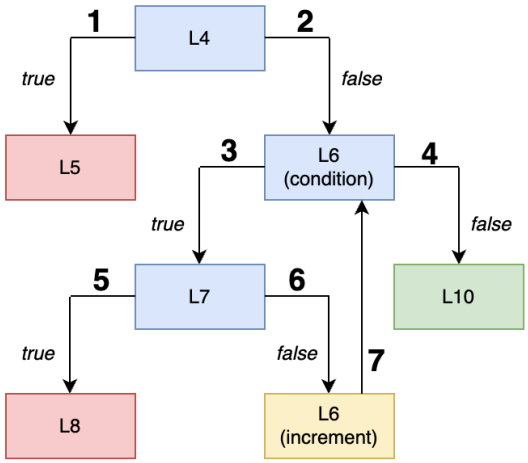


Figure 1: Control flow graph for `checkPassword`.

Observe that the time this algorithm takes to execute will vary based on the input password's similarity to the secret password, thereby revealing information about the secret password itself. More concretely, suppose the password was `PASSWORD` and an attacker supplies the strings $s_1 =$ `AAAAAAAA` and $s_2 =$ `PBAAAAAA` to `checkPassword` and measures the first output to be 0.1ms and the second output to be 0.3ms. In view of the control flow graph in Figure 1, s_1 takes the path {2, 3, 5} and s_2 takes the path {2, 3, 6, 7, 3, 5}, so the latter path has three more steps. The attacker can infer that this difference happened because the for loop executed for one more iteration with s_2, implying that the first character of the password is P. The attacker can proceed this way character-by-character, breaking the password in time linear to the password length.

One major pitfall of this password checker implementation is that it contains a branch that can reveal information about a secret variable referred to in its guard. More concretely, the vulnerability with this insecure `checkPassword` implementation is a timing side-channel, wherein an attacker exploits the time taken for a system to process different inputs to reveal secret data.

It is critical to ensure that hardware accelerator implementations of important algorithms, such as string comparison, RSA encryption, or discrete Gaussian sampling, are not susceptible to these timing attacks, since such vulnerabilities can expose secret material and compromise the fundamental security guarantees these algorithms should provide.

3. Background

3.1. HLS and Dahlia

HLS is an alternative approach to designing accelerators and specialized ASICs. Rather than creating a design in a low-level HDL such as Verilog, HLS instead treats HDLs as a compilation target, allowing for the programmer to reason about hardware at a higher level of abstraction. However, using modern HLS tools can be very unpredictable, as they often have silent and implicit rules, and can produce nonintuitive results. To combat this issue, the authors of [9] created a new HLS language called Dahlia.

Dahlia is a high level language that compiles down to predictable hardware implementations. In order to achieve predictability, Dahlia introduces a novel time-sensitive affine type system. Specifically, the type system allows the programmer to separate code into regions known as "logical time steps." This separation is achieved through an operator called "ordered composition", denoted by `---`. Within a logical time step, resources may only be used at most once, as the type system enforces an affine restriction on memory. Due to this affine restriction, statements within a logical time-step can be freely parallelized and pipelined, as there is no possibility of aliasing or data races. To compose statements inside of a logical time-step, programmers can use "unordered composition", denoted by a semicolon.

3.2. SecVerilog and SpecVerilog

We take inspiration from both SecVerilog and SpecVerilog to design our type system. Both prior works create a type system for HDLs to prevent timing side channel attacks through the use of IFC. SecVerilog [5] is one of the first major works that incorporates time-sensitive IFC in its type system. Additionally, it incorporates dynamic labels for registers that can change at runtime depending on the data that is stored. These labels are useful because they allow for the reuse of hardware components across security levels. However, naively using dependent labels can lead to a security vulnerability known as *implicit declassification*. If the value of a variable changes, it can cause labels for other variables to change, potentially declassifying information.

To combat this issue, SpecVerilog [6] is a follow-up work that introduces the notion of *erasure labels*, which allow programmers to specify under what conditions a declassification can occur. Assuming these conditions are consistent, whenever a condition occurs, the registers that contain the variable's memory will be cleared to prevent information leakage.

3.3. ASSURE and SecHLS

We are not the first to consider time-sensitive IFC for HLS. ASSURE [7] pioneered the idea of comprehensive timing-sensitive IFC enforcement. Similar to our proposed design, it annotates variables with IFC labels, and statically enforces flows between them. The major difference between our approaches is the handling of timing side channels. The approach taken by ASSURE is to generate a second machine which handles all observable I/O, therefore making the entire module appear time-invariant. In fact, we rely on a similar principle: attackers can only view public, low secrecy registers, and modifications to them should be time-invariant.

Independently, but building on ASSURE, the SecHLS [8] system was developed. The main objective of SecHLS is to identify key locations in existing synthesis algorithms at which security properties may be enforced, but doing so as discreetly as possible. They specify a property called the "Secure Input" constraint, which ensures something reminiscent of information flow. It specifies that nodes connected to insecure inputs do not share hardware resources with those connected to secure inputs. The authors later generalize this property to stratify multiple levels of security, rather than just secure/insecure. In contrast to SecHLS, our system aims to provide time-sensitive IFC guarantees at the type level.

4. Approach

4.1. Overview

We implement static enforcement of both time-sensitive and information noninterference in the Dahlia HLS. Our implementation is available as a fork of Dahlia at https://github.com/zbrachinara/dahlia/. Two different systems are added to the language: an information flow checker that enforces correct information flow, and a branch balancing pass that enforces time-sensitive noninterference. These two systems interact at the branch condition, where, if secret data is used as the condition for a branch, we run a branch balancing pass. Since Dahlia's `for` loops do not depend on data, we only consider `if` commands and `while` loops.

4.2. Language Changes

We first introduce labels to the type system. Labels annotate variables (which in Dahlia can either represent single registers or entire memories), denoting their security level.

```
decl foo : bit<32> <H> = 1234;
```

We take inspiration from the label system of SecVerilog, which has three kinds of labels. The first two, `H` and `L` correspond to just the top and bottom of the IFC lattice (secret and public data). The third kind of label is a dependent label. These vary based on the values of variables at runtime. In the notation of Dahlia, dependent labels are introduced via a boolean-valued function and a call to that function.

```
boolean label(a : boolean, b : boolean) { return a && b }
...
decl x : boolean <L> = false; decl y : boolean <H> = true;
decl foo : bit<32> <label(x, y)>;
```

In this example, the annotation `<label(x, y)>` means that the label of `foo` *varies* with the values of `x` and `y`.

A formal description of the language syntax and our changes can be found in the appendix.

4.3. Information Flow Noninterference

Dependent labels are used to allow the same module to be reused for operations which optionally depend on secret data, provided that secret and public data are provably quarantined from each other. More specifically, suppose that a dependent label was updated – then for any read of the variable this label annotates, there must be a write between the update to the dependent label and this read.

```
decl t : bit<1> = 1;
decl x : bit<32> <l(t)>;
x := secret;
t := 0; // x contains secrets!
public := x; // failure
```

```
decl t : bit<1> = 1;
decl x : bit<32> <l(t)>;
x := secret;
t := 0; // x contains secrets!
x := 0; // no longer
public := x; // success
```

Listing 1: A demonstration of implicit upgrade and mitigation.

This condition is used to prevent what is referred to as "implicit upgrades", where data inside a variable can become upgraded by virtue of remaining in place (while its type changes around it). In order to get around this issue, [6] introduces *erasure labels*, which are programmer specified labels that specify when it is safe for a variable to be erased. This would be implemented by discharging the conditions to an SMT solver in order to check that the conditions can be met safely.

Labels are then given a partial lattice structure $\sqsubset, \sqcup$. By definition, `L` and `H` are the bottom and top of the lattice, so for all labels λ,

$$\mathtt{L} \sqsubset \lambda \sqsubset \mathtt{H}, \quad \mathtt{L} \sqcup \lambda = \lambda, \quad \mathtt{H} \sqcup \lambda = \mathtt{H}$$

Two dependent labels are not necessarily comparable, though they are certainly comparable if all their conditions are known.

We then use this lattice structure to check information flows. As usual, for x labeled with λ_x and e labeled with λ_e, $x \leftarrow e$ type checks if $\lambda_e \sqsubset \lambda_x$. In addition, for expressions which outside of some branch have label λ_e, their label inside the branch body is upgraded to $\lambda_e \sqcup \lambda_c$, where λ_c is the label of the condition. This is sufficient to guarantee that no information flows are violated [10]. The typing rules can be found in the appendix to this report.

4.4. Time-sensitive Noninterference

Having guaranteed information flow integrity, we move on to time-sensitive integrity. We hijack existing Dahlia infrastructure, ordered composition, in order to make timing guarantees. Dahlia's ordered composition separates programs into regions of code called "logical time steps", in each of which a memory can be accessed at most once. In order to preserve this guarantee, Dahlia guarantees that commands cannot move past ordered compositions [9]. In fact, if we consider logical time steps to be the only kind of command, then a Dahlia program is just an in-order execution of commands. Motivated by this insight, we impose the following restriction: **A branch with secret condition balances the number of logical time steps on each side.**

We begin to define this restriction by example. Consider the following branch in Dahlia with a secret condition:

```
---
if (secret) {
  if (public) {a1() --- a2() } else { b1() } --- b2();
} else {
  c1() --- c2()
}
---
```

If `secret` is `true` and `public` is `false`, then we first execute `b1`, then `b2`, in sequence. If `secret` was `false`, then we execute `c1`, then `c2`. Therefore, if all logical time steps took the same amount of time to execute, then these two states are indistinguishable from each other. However, these states are both distinguishable from when `secret` is `true` and `public` is `true`, which would take three logical time steps (by executing `a1`, then `a2`, then `b2`). Therefore, an adversary could be able to tell the state of `secret` by observing an irregularly high number of logical time steps. This issue could be corrected, for example, by removing the ordered composition on the left side of the branch on `public`.

So returning to our definitions:

- A branch is balanced iff there exists n such that for any program state, executing the branch observes n ordered compositions.
- Our restriction is that any branch with a secret condition is also balanced.

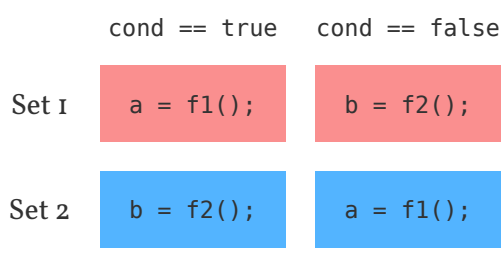
In order for this restriction to mean anything, the adversary must only able to observe up to logical time steps. However, we would expect a competent adversary to be able to at least observe cycle-level behavior. To get around this in the explanation above, we assume that every logical time step takes the same amount of time to execute. This assumption is far too strong, and in fact a weaker assumption will still admit noninterference with this restriction.

By definition, any execution of a balanced branch must see exactly n ordered compositions. Ordered compositions cannot be reordered with respect to each other. Therefore, fixing some execution, we can label **some of** the ordered compositions in the branch from 1 to n based on the order in which they are executed. In fact, an ordered composition's label is well defined even without reference to the execution that encounters it. To see this, fix some ordered composition. Without loss of generality, suppose it is the last ordered composition seen by some execution. Then for any execution that sees this ordered composition, it must also be the last ordered composition of *that* execution, or the branch is not balanced. Thus we can show inductively (on n) that if an ordered composition is labeled i by some execution, then it must be labeled i by every execution that labels it.

We can now define the notion of logical time step sets. A logical time step is labeled i if it is between an ordered composition labeled i and an ordered composition labeled $i + 1$. Using the beginning of the branch as the 0th ordered time step, and the end of the branch as the $n + 1$th ordered time step, then we have $n + 1$ sets of logical time steps, where the ith set contains all of the logical time steps labeled i.

Finally, we arrive at our weaker assumption: **For any two logical time steps in the same set, their execution takes the same time.**

```
if (cond) {
  a = f1();
  ---
  b = f2();
} else {
  b = f2();
  ---
  a = f1();
}
```



Listing 2: Logical time step sets of a simple balanced branch

For this assumption to be meaningful, it has to be true. But as Dahlia is implemented, there are no guarantees about the real time a logical time step takes. Thankfully logical time step sets are easily computable (using the same procedure which type checks balanced branches), so enforcing the logical time step assumption should be more or less the same as enforcing the non-reorderability of ordered compositions. In fact, if the backend is timing-aware, then this may be possible in some cases without any code generation. This is the case for a particular backend of Dahlia called Calyx, which has the ability to fold dynamic modules into static ones [11].

5. Evaluation

Our evaluation consists of two main parts: concrete testing of components that we have implemented, as well as a qualitative discussion about the complications and considerations with our approach.

5.1. Testing

We added syntactical changes to the Dahlia AST and parser to account for security labels and three additional passes to the typechecker: a `SecurityCheck` to enforce normal IFC noninterference, a `TimeChecker` to enforce time-sensitive noninterference, and an `IntegrityCheck` that cleans up security labels so that the AST can be processed by Dahlia's existing passes downstream.

In order to gain confidence that these components worked as intended, we wrote unit tests for each of the new components we added. For the parser and AST, we wrote unit tests to ensure the parser successfully captured high, low and dependent labels. For the `SecurityCheck`, we added unit tests to make sure that explicit flows from secret to public data are prohibited. For the `IntegrityCheck`, we wrote tests that ensure the Dahlia AST is free of the augmented security labels. The tests for these passes are mostly straightforward and can be found in the codebase.

More interestingly, the unit tests for the `TimeChecker` have much more room for flexibility. We have tested several different types of branching to gain confidence in our system and implementation. Aside from the verifications of basic `if` statements with balanced/unbalanced branches whose conditions depend on secret/public variables, we also tested examples of `if` statements whose branches themselves contain `if` statements. Examples of accepted and rejected programs can be seen in Listing 3, and further examples can be found in the codebase. In the future, we seek to use randomized testing and fuzzing to generate more complex inputs for each of the passes to gain more confidence in correctness.

```
let a: bit<32> <H> = 1;
let b: bit<32> <L> = 2;
if (a == 1) {
  a;
  ---
  a;
}
else {
  if (b == 2) {
     b;
  }
  else {
     b;
  }
  ---
  a;
}
// accepted by TimeChecker
```

```
let a: bit<32> <H> = 1;
let b: bit<32> <L> = 2;
if (a == 1) {
 a;
 ---
 a;
}
else {
 if (b == 2) {
    b;
    ---
    a;
 }
 else {
    b;
 }
 ---
 a;
} // rejected by TimeChecker
```

Listing 3: Example programs from `TimeChecker` testing file

Based on our testing, we are confident that these three passes work as intended: in particular, we believe that only programs that are free of timing side-channels with our abstraction level will be accepted by the typechecker.

These guarantees only apply at the logical time-step level, whereas in practice an attacker usually is assumed to be able to measure timing behavior at a more granular level (i.e. number of cycles). However, as we argued in Section 4.4, we can compute logical time step sets and pass this information to the backend, which can then ensure that corresponding logical time step sets take the same amount of time. Assuming a backend that can perform this task, our guarantees at the HLS level will still hold at the HDL level.

5.2. Complications

There are several complications with our intended approach of ensuring time-sensitive noninterference that are worth discussing.

In the Dahlia language, `while` loops are only sequential. This means that in order to prove that every execution will observe the same number of ordered compositions, we need to show that the loop will terminate after a fixed number of steps regardless of the program state. While this is possible for some programs, we do not attempt it.

For time sensitive noninterference to actually be satisfied, we have to make sure that writes are not visible at inconsistent times. Technically, this is still possible with our assumption and restriction from Section 4.4, because a write can occur

at any time in a logical time step. This problem can be sidestepped by requiring that memories be written only in their own time step.

In the definition of logical time step sets from Section 4.4, we assumed that the first ordered composition is at the beginning of a branch, and the last ordered composition is at the end. However, time steps don't have to be entirely contained within a branch or entirely contain a branch. An ordered composition can be before a branch point, and then the next ordered composition can be in the branch body itself, like so:

```
c1();
if (b) {
  c2() --- c3()
} else {
  c4()
}
```

The solution here is to treat the branch point as if it were at the beginning of the logical time step, and duplicate all the previous commands into each of the bodies:

```
if (b) {
  c1(); c2(); --- c3();
} else {
  c1(); c4();
}
```

Now there is an unbalanced branch, which the `TimeChecker` can detect.

6. Conclusion

In our report, we presented a type system for enforcing *time-sensitive non-interference* to prevent timing side channels from being introduced when designing accelerators using HLS. However, we still have many future avenues of work. A major challenge for this project was performing our implementation. Since we were extending the Dahlia language, it took us a long time to understand the intricacies of the codebase. Additionally, the Dahlia compiler is written in Scala, which none of the group members had experience in. This meant that we had to learn a new language while figuring out how to work on and modify the compiler. As such, many of our future goals are implementation-oriented. While we have proposed the usage of dependent labels to minimize hardware duplication, due to time constraints and some of the challenges mentioned earlier, we were unable to actually implement them. We want to fully implement dependent labels and erasure policies. Our strategy for implementing them would be to generate preconditions for these dependent labels, similar to Hoare Logic, and allowing

an SMT solver to check them. In addition, the Dahlia language provides a lot of syntax sugar for interfacing with memories, and expressing loops (including allowing programmers to manually specify loop unroll factors and providing special syntax for common code patterns). We would like to extend our implementation to support more of this syntactical sugar and therefore a larger subset of the Dahlia language.

Furthermore, we want to extend our front-end reasoning about timing all the way to the backend. While we are able to reason about logical timesteps, it is a very high level abstraction and is not really true to the actual behavior of the hardware. As mentioned previously in our report, we currently provide hints to the backend to alert it to the portions of our program that need to have timing behaviors enforced. Actually performing that enforcement is a natural next step.

Finally, we want to prove that our system actually enforces time-sensitive noninterference. This seems to be a difficult property to formally state. However, doing so will allow us to gain confidence that our system performs as expected. While we have tested many examples, we cannot know that the rules are sound until we have formally proved that our type system upholds the guarantees we want it to.

Bibliography

[1] S. Sarkar, T. Majumder, A. Kalyanaraman, and P. P. Pande, "Hardware accelerators for biocomputing: A survey," in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, 2010, pp. 3789–3792. doi: 10.1109/ISCAS.2010.5537736.

[2] D. Boruah and M. Saikia, "A review on hardware accelerators for elliptic curve cryptography," in *National Conference On Computing, Communication and Information Processing (NCCCIP-2015)*, 2015.

[3] T. Mohaidat and K. Khalil, "A survey on neural network hardware accelerators," *IEEE Transactions on Artificial Intelligence*, vol. 5, no. 8, pp. 3801–3822, 2024.

[4] X. Lou, T. Zhang, J. Jiang, and Y. Zhang, "A Survey of Microarchitectural Side-channel Vulnerabilities, Attacks, and Defenses in Cryptography," *ACM Comput. Surv.*, vol. 54, no. 6, July 2021, doi: 10.1145/3456629.

[5] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A Hardware Design Language for Timing-Sensitive Information-Flow Security," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, in ASPLOS '15. Istanbul, Turkey: Association for Computing Machinery, 2015, pp. 503–516. doi: 10.1145/2694344.2694372.

[6] D. Zagieboylo, C. Sherk, A. C. Myers, and G. E. Suh, "SpecVerilog: Adapting Information Flow Control for Secure Speculation," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, in CCS '23. Copenhagen, Denmark: Association for Computing Machinery, 2023, pp. 2068–2082. doi: 10.1145/3576915.3623074.

[7] Z. Jiang, S. Dai, G. E. Suh, and Z. Zhang, "High-level synthesis with timing-sensitive information flow enforcement," in *Proceedings of the International Conference on Computer-Aided Design*, in ICCAD '18. San Diego, California: Association for Computing Machinery, 2018. doi: 10.1145/3240765.3243415.

[8] S. Shi, N. Pundir, H. M. Kamali, M. Tehranipoor, and F. Farahmandi, "SecHLS: Enabling Security Awareness in High-Level Synthesis," in *2023 28th Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan. 2023, pp. 585–590.

[9] R. Nigam *et al.*, "Predictable accelerator design with time-sensitive affine types," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, in PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 393–407. doi: 10.1145/3385412.3385974.

[10] D. E. Denning, "A lattice model of secure information flow," *Commun. ACM*, vol. 19, no. 5, pp. 236–243, May 1976, doi: 10.1145/360051.360056.

[11] C. Kim, P. Li, A. Mohan, A. Butt, A. Sampson, and R. Nigam, "Unifying Static and Dynamic Intermediate Languages for Accelerator Generators," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA2, Oct. 2024, doi: 10.1145/3689790.

[12] R. Nigam *et al.*, "Predictable Accelerator Design with Time-Sensitive Affine Types," *CoRR*, 2020, [Online]. Available: https://arxiv.org/abs/2004.04852

[13] S. Hunt and D. Sands, "On flow-sensitive security types," in *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, in POPL '06. Charleston, South Carolina, USA: Association for Computing Machinery, 2006, pp. 79–90. doi: 10.1145/1111037.1111045.

7. Appendix

We first present the syntax for our core language which is inspired by the core language of Dahlia [12]:

$$x \in \text{variables} \quad a \in \text{memories} \quad n \in \text{numbers } b ::= \text{true} \mid \text{false}$$

$$v ::= n \mid b$$

$$e ::= v \mid v_1 \text{ bop } v_2 \mid x \mid a[e]$$

$$c ::= e \mid \text{let } x = e \mid c_1 \text{ --- } c_2 \mid c_1; c_2 \mid \text{if x then } c_1 \text{ else } c_2 \mid \text{while x } c \mid x := e \mid a[e_1] := e_2$$

We also have the following syntax for types in our language, where type is a base datatype found in the Dahlia language, and x is a variable:

$$\begin{aligned} &\mathsf{ty} := \mathsf{type} \ \langle \mathsf{secLabel} \rangle \\ &\mathsf{secLabel} := \mathsf{H} \mid \mathsf{L} \mid \mathsf{dependentLabel} \\ &\mathsf{dependentLabel} := \mathsf{f(x)} \end{aligned}$$

In words, our possible types are a base datatype augmented with either the base labels H, L or a dependent label, which is a function from variables to labels.

For our typing rules, we extend the typing rules found in the Dahlia technical report [12]. Dahlia provides two different typing contexts: Γ maps non-affine variables to types, and Δ maps affine memories to types. Our first set of typing rules are similar to the standard presentation of flow-sensitive information flow control [13]. With these rules, our goal is to prevent secret dependent control and data flow. To achieve this, we first modify the original type contexts Γ, Δ to allow standard datatypes from the Dahlia language to be augmented also with a security label λ. Types in the language look like $\langle \tau, \lambda \rangle$. A memory in Dahlia can be viewed as an array since it can be indexed with arbitrary expressions. As such, we only need to have a security label for the declared memory segment that is being accessed. The rules include the program context Ctx := H | L.

$$\textsf{E-Val} \ \frac{}{\Gamma,\ \Delta \vdash \mathrm{v} : \langle\ \tau,\ \mathrm{L}\ \rangle \dashv \Delta}$$

$$\textsf{E-Var} \ \frac{\Gamma\ (x) = \langle\ \tau,\ \lambda\ \rangle}{\Gamma,\ \Delta \vdash \mathrm{x} : \langle\ \tau,\ \lambda\ \rangle \dashv \Delta}$$

$$\textsf{E-BinOp} \ \frac{\begin{array}{c}\Gamma,\ \Delta_1 \vdash e_1 : \langle\ \tau,\ \lambda_1\ \rangle \dashv \Delta_2 \\ \Gamma,\ \Delta_2 \vdash e_2 : \langle\ \tau\ ,\ \lambda_2\ \rangle \dashv \Delta_3\end{array}}{\Gamma,\ \Delta_1 \vdash e_1 \text{ bop } e_2 : \langle\ \tau,\ \lambda_1 \sqcup \lambda_2\ \rangle \dashv \Delta_3}$$

$$\textsf{E-MemoryRead}\ \frac{\Gamma,\ \Delta \vdash \mathrm{e} : \langle\ \mathrm{bit}\langle\ \mathrm{n}\ \rangle\ ,\ \lambda_1\ \rangle \dashv \Delta_2 \quad \Delta_2 = \Delta_3 \cup \{a \mapsto \langle \textsf{mem}\ \tau[n], \lambda_2 \rangle\}}{\Gamma,\ \Delta_1 \vdash \mathrm{a}[e] : \langle\ \tau,\ \lambda_1 \sqcup \lambda_2\ \rangle \dashv \Delta_3}$$

$$\textsf{C-MemoryWrite}\ \frac{\begin{array}{c}\Gamma,\ \Delta, \vdash e_1 : \langle\ \mathrm{bit}\langle\ \mathrm{n}\ \rangle, \lambda_1 \rangle \dashv \Delta_2 \\ \Gamma,\ \Delta_2, \vdash e_2 \colon \langle\ \tau\ ,\ \lambda_2\ \rangle \dashv \Delta_3 \\ \Delta_3 = \Delta_4 \cup \{a \mapsto \langle \textsf{mem}\ \tau[n_1], \lambda_2 \rangle\} \\ \mathrm{Ctx} \sqcup \lambda_1 = \lambda_3,\ \lambda_3 \sqsubset \lambda_2\end{array}}{\Gamma,\ \Delta_1,\ \mathrm{Ctx} \vdash \mathrm{a}[e_1] \coloneqq e_2 \dashv \Gamma,\ \Delta_4}$$

$$\textsf{C-UnorderedSeq}\ \frac{\Gamma_1,\ \Delta,\ \mathrm{Ctx} \vdash c_1 \colon \mathrm{ok} \dashv \Gamma_2,\ \Delta_2 \quad \Gamma_2,\ \Delta_2,\ \mathrm{Ctx} \vdash c_2 \colon \mathrm{ok} \dashv \Gamma_3,\ \Delta_3}{\Gamma,\ \Delta_1,\ \mathrm{Ctx} \vdash c_1;\ c_2 \dashv \Gamma,\ \Delta_3}$$

For the C-UnorderedSeq rule, even though the compiler is free to re-order and parallelize statements, we follow the original Dahlia paper and enforce an evaluation order on the program statements. This is because at the level of the type system, we only care about checking equivalence across logical timesteps, so fixing an evaluation order doesn't matter. We leave to the backend to check and enforce the timing equivalence of statements within a logical timestep.

$$\textsf{C-OrderedSeq}\ \frac{\Gamma_1,\ \Delta_1,\ \mathrm{Ctx} \vdash c_1 : \mathrm{ok} \dashv \Gamma_2,\ \Delta_2 \quad \Gamma_2,\ \Delta_2,\ \mathrm{Ctx} \vdash c_2 : \mathrm{ok} \dashv \Gamma_3,\ \Delta_3}{\Gamma_1,\ \Delta_1 \vdash c_1 \text{ --- } c_2 \dashv \Gamma_3,\ \Delta_2 \cap \Delta_3}$$

$$\textsf{C-While}\ \frac{\Gamma,\ \Delta_1,\ \mathrm{Ctx} \vdash \mathrm{x} \colon \langle\ \tau,\ \mathrm{L}\ \rangle \dashv \Delta_2 \qquad \Gamma,\ \Delta_2,\ \mathrm{L} \vdash \mathrm{c} \dashv \Gamma,\ \Delta_3}{\Gamma,\ \Delta_1,\ \mathrm{Ctx} \vdash \text{while x c} \dashv \Gamma,\ \Delta_2 \cap \Delta_3}$$

For the C-While rule, the body must be sequential - meaning only consisting of logical timesteps. However, we require that any commands run are done in a low context, as we cannot guarantee how many times the loop will run.

Otherwise, the Dahlia language only supports bounded loops. Therefore, we can simply unroll any loop and treat it as a series of nested branches. In the next section, we will present our rules for enforcing timing guarantees of branches.

When the branch label is low (and the context is low), we do not care about timing information, so any branch is accepted as long as it passes information flow checks.

$$\textsf{C-If}\ \frac{\Gamma, \Delta_1, \mathsf{L} \vdash x : \langle \mathsf{bool}, \mathsf{L} \rangle \dashv \Delta_2 \qquad \Gamma, \Delta_2, \mathsf{L} \vdash c_1 : \mathsf{ok} \dashv \Gamma, \Delta_3 \qquad \Gamma, \Delta_3, \mathsf{L} \vdash c_2 : \mathsf{ok} \dashv \Gamma, \Delta_4}{\Gamma, \Delta_1, \mathsf{L} \vdash \mathsf{if}\ x\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \dashv \Delta_3 \cap \Delta_4}$$

Otherwise, we introduce the notation: $\Gamma, \mathsf{Ctx} \vdash_{\mathsf{basic}} c$ for checking that c is a command without ordered composition, and check timing. Our convention is that λ_e refers to ambient context, and λ_c is the label of the condition.

$$\textsf{C-If-Base}\ \frac{\Gamma, \Delta_1, \lambda_e \vdash \mathsf{b} : \langle \mathsf{bool}, \lambda_c \rangle \dashv \Delta_2 \qquad \Gamma, \Delta_2, \lambda_e \sqcup \lambda_c \vdash_{\mathsf{basic}} c_1 : \mathsf{ok} \dashv \Delta_3 \qquad \Gamma, \Delta_3, \lambda_e \sqcup \lambda_c \vdash_{\mathsf{basic}} c_2 : \mathsf{ok} \dashv \Delta_4}{\Gamma, \Delta_1, \lambda_e \vdash \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \dashv \Delta_4}$$

$$\textsf{C-If-Ind}\ \frac{\Gamma, \Delta_1, \lambda_e \vdash b : \langle \mathtt{bool}, \lambda_c \rangle \dashv \Delta_2 \qquad \Gamma, \Delta_1, \lambda_e \sqcup \lambda_c \vdash_{\mathsf{basic}} c_1 \dashv \Delta_3 \qquad \Gamma, \Delta_1, \lambda_e \sqcup \lambda_c \vdash_{\mathsf{basic}} c_2 \dashv \Delta_4 \qquad \Gamma, \Delta_3 \cap \Delta_4, \lambda_e \sqcup \lambda_c \vdash \mathsf{if}\ b\ \mathsf{then}\ c_3\ \mathsf{else}\ c_4 \dashv \Delta_5}{\Gamma, \Delta_1, \lambda_e \vdash \mathsf{if}\ b\ \mathsf{then}\ c_1\ \text{---}\ c_3\ \mathsf{else}\ c_2\ \text{---}\ c_4 \dashv \Delta_5}$$

$$\textsf{C-If-Nested}\ \frac{\begin{array}{c}\Gamma, \Delta_1, \lambda_e \vdash b_1 : \langle \mathtt{bool}, \lambda_{c_1} \rangle \dashv \Delta_{01} \\ \Gamma, \Delta_1, \lambda_e \vdash b_2 : \langle \mathtt{bool}, \lambda_{c_2} \rangle \dashv \Delta_{02} \\ \Gamma, \Delta_1, \lambda_e \sqcup \lambda_{c_1} \sqcup \lambda_{c_2} \vdash \mathsf{if}\ b_2\ \mathsf{then}\ l_1\ \mathsf{else}\ r \dashv \Delta_a \\ \Gamma, \Delta_1 \lambda_e \sqcup \lambda_{c_1} \sqcup \lambda_{c_2} \vdash \mathsf{if}\ b_2\ \mathsf{then}\ l_2\ \mathsf{else}\ r \dashv \Delta_b\end{array}}{\Gamma, \Delta_1, \lambda_e \vdash \mathsf{if}\ b_1\ \mathsf{then}\ (\mathsf{if}\ b_2\ \mathsf{then}\ l_1\ \mathsf{else}\ l_2)\ \mathsf{else}\ r \dashv \Delta_a \cap \Delta_b}$$

$$\textsf{C-If-Comm}\ \frac{\Gamma, \Delta_1, \lambda \vdash \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \dashv \Delta_2}{\Gamma, \Delta_1, \lambda \vdash \mathsf{if}\ b\ \mathsf{then}\ c_2, \mathsf{else}\ c_1 \dashv \Delta_2}$$

The rule C-If-Ind admittedly doesn't cover all cases, since the inner branch could have a statement appended on either side as discussed in the evaluation.