
Improving Lazy Self-Composition for Secure Information Flow

A thesis submitted in fulfillment of the requirements

for the degree of Master of Technology

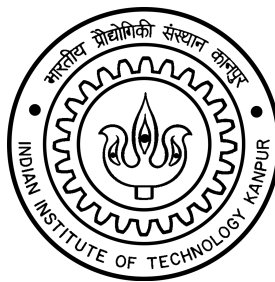
by

Deepak Sirone J

17111013

under the guidance of

Dr. Pramod Subramanyan



to the

Department of Computer Science and Engineering


Indian Institute of Technology Kanpur

June 2019

Page intentionally left blank

Statement of thesis preparation

1. Thesis title IMPROVING LAZY SELF-COMPOSITION FOR SECURE INFORMATION FLOW
2. Degree for which submitted MASTER OF TECHNOLOGY
3. The thesis guide was referred to for thesis preparation. Yes/~~No~~
4. Specifications regarding thesis format have been closely followed. Yes/~~No~~
5. The contents of the thesis were organized according to the guidelines. Yes/~~No~~

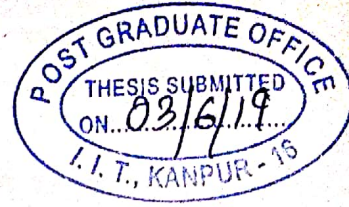

(Signature of Student)

Name : DEEPAK SIRONE J

Roll No. : 17111013


Department : CSE

Page intentionally left blank



Certificate

It is certified that the work contained in the thesis titled "**Improving Lazy Self-Composition for Secure Information Flow**" has been carried out under my supervision by **Deepak Sirone J** and that this work has not been submitted elsewhere for a degree.



Dr. Pramod Subramanyan

Professor

Department of Computer Science and Engineering

Indian Institute of Technology Kanpur

Kanpur 208016

June 2019

Page intentionally left blank

ABSTRACT

Name of student: **Deepak Sirone J** Roll no: **17111013**

Degree for which submitted: **Master of Technology**

Department: **Department of Computer Science and Engineering**

Thesis title: **Improving Lazy Self-Composition for Secure Information Flow**

Name of Thesis Supervisor: **Dr. Pramod Subramanyan**

Month and year of thesis submission: **June 2019**

Program verification of single trace properties has been well studied in literature. Programs are modelled as state transition systems and a correctness condition on all possible runs of the transition system is specified using a suitable program logic. Single trace properties are specific instances of a more general class of correctness conditions known as hyperproperties. A hyperproperty defines a correctness condition that simultaneously relates one or more traces of a program.

A variety of security properties of programs can be cast as 2-safety hyperproperties including secure information flow, determinism and commutativity. Naively, 2-safety hyperproperties of programs are verified using self-composition, an algorithm by which two copies of the program are created and then analyzed. Recent work by Yang *et al.* introduces lazy self-composition, an algorithm which attempts to reduce the computational cost of self-composition using a taint based transition system. The taint based transition system captures equalities between corresponding variables in the two copies of the self composed program. However, the taint based system relies on an over-approximation of the actual transition system and hence needs to be refined periodically to have a reasonable degree of accuracy.

In the work by Yang *et al.*, refinement was done by patching the taint transition system at the particular step in the trace where a false taint violation was detected. We propose to refine the taint transition system using predicates that are input by the user at the start of the verification procedure. Through this work it is shown that our scheme can prove program properties efficiently in comparison to the scheme by Yang *et al.* in some examples.

Page intentionally left blank

Acknowledgements

I am deeply indebted to my thesis advisor, Dr. Pramod Subramanyan for his invaluable guidance and help throughout the duration of my thesis project. At a time when I was contemplating my thesis topic and my next career move, his arrival on campus proved to be a true godsend. His energy, enthusiasm and lucid explanations about complicated topics in formal methods kept me interested and motivated for the past year. Over the course of the past year, I could feel that I have grown both as a person and as a researcher under his mentorship.

I am also especially grateful to Dr. Sandeep Shukla for introducing me to cyber security through his course and also for his expert advice regarding the type of thesis project which I should aim to do.

I extend my heartfelt gratitude to Dr. Muralikrishnan K and Dr. Sudeep K S of the National Institute of Technology Calicut for their excellent classes on the fundamentals of theoretical computer science. Their early motivation and support proved pivotal in building my resolve to pursue research.

Last but not the least I would like to thank my parents and sister for lending me immense help and support whenever I needed it.

Contents

Acknowledgements	ix
1 Introduction	4
1.1 Secure Information Flow	5
1.2 Verification of Secure Information Flow	7
1.3 Solution Summary	8
1.4 Our Contributions	8
1.5 Structure of this document	9
2 Background	10
2.1 Abstract Program Model	10
2.2 Property Specification	11
3 Verification of Secure Information Flow	17
3.1 Self Composition	17
3.2 Lazy Self-Composition	18
3.3 Dependent Taints for Lazier Self-Composition	27
3.4 Details Unclear from Previous Work	29
3.5 Limitations	30
4 Experiments	31
4.1 UCLID5	31
4.2 Implementation	32
4.3 Results	33
5 Conclusion and Future Work	39
5.1 Lambda Expressions for Array Taints	39
5.2 Converting Hyperaxioms to Array Taint Refinements	40
5.3 Investigating Z3 Inductive Trace	40

List of Figures

1.1	Secure information flow: Public outputs should not be dependent on secret inputs	5
1.2	Taint Propagation: A variable is tainted if atleast one of its dependencies is tainted	7
2.1	Trace view of systems and single trace properties	12
2.2	System S satisfies single trace property P	12
2.3	Trace view of systems and hyperproperties: Hyperproperty P is a set of set of traces	13
2.4	System S satisfies hyperproperty P	13
2.5	Non-interference: In the second trace the high security states are removed	14
2.6	Observational Determinism: The colours represent the observable output of the states	15

List of Uclid5 Examples

4.1	A UCLID5 model that computes the Fibonacci sequence	32
4.2	A UCLID5 model that demonstrates a false branch condition dependence	34
4.3	A UCLID5 model that demonstrates similar updates in both the branches	35
4.4	A UCLID5 model that demonstrates similar updates in all branches	37

Page intentionally left blank

Dedicated to my parents and sister

Chapter 1

Introduction

In a world which is increasingly run using computers, the correctness and security of computer systems is of utmost concern. Bugs in the software and the hardware can lead to security vulnerabilities in the most unanticipated of ways. An important class of vulnerabilities which we address in this thesis is related to information leaks in hardware and software designs. Informally, information leaks refer to the phenomenon by which the observable characteristics of the running system can be used to deduce the private internal state of the system. Such vulnerabilities generalize the notion of side-channel attacks on systems. Defending against side-channel attacks is often hard, as they are difficult to detect by reading through the system description. Formally verifying the system for information leaks involves reasoning about all program execution paths in presence of a powerful adversary, who is allowed to execute arbitrary actions in the background.

Verification using formal methods entails mathematically proving that a system satisfies a specification. Formally proving that a system is secure guarantees that the system is free of vulnerabilities, as foreseen by the security specification. Formal methods for program verification include static analysis and abstract interpretation as well as model checking. Static analysis and abstract interpretation involves analysing the code segment of the program to generate the control and data flow, using suitable abstractions for the system being analysed. The abstract model of the program is further analysed to verify whether the property being checked holds. In many systems whose behaviour is restricted and well-defined (e.g. finite state machines), this approach would suffice. Model checking entails exhaustively checking every program state, whether or not the property being verified is satisfied. Model checking can be either bounded or unbounded. Bounded model checking involves checking all the program paths whose length is lesser than a particular bound length for a property violation whereas unbounded model checking entails the checking of all possible paths in the program without a length bound. A modern algorithm for unbounded model checking is property directed reachability [1, 2]. Many modern PDR implementations take

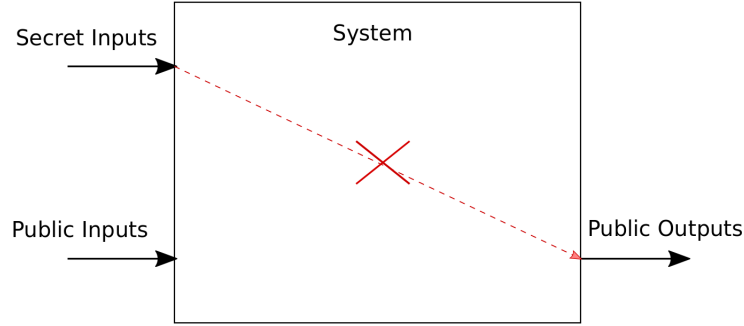


Figure 1.1: Secure information flow: Public outputs should not be dependent on secret inputs

their input a transition system specified using constrained horn clauses [3, 4, 5, 6, 7].

1.1 Secure Information Flow

Controlling how information is propagated in a system is of critical importance for information security. In earlier days, access control mechanisms were used to regulate system access to sensitive information. A program requests for some information and the request is either granted or denied. As soon as a request is granted, no further checking of how the program uses the information is done. This poses problems, as the program in question has to be fully trusted to not leak or share the sensitive information to unauthorized parties. In a running process, information flow is the transfer of information from one program variable to another and it captures the interdependence between the values of the program variables. *Information flow control* is used to ensure that a program handles the data it is provided securely[8]. An information flow control mechanism enforces information flow policies in a system. Several methods have been proposed to achieve information flow control including runtime tracking of labelled data, static analysis of programs to verify information flow and the use of security type systems. In the case of security type systems, each program variable carries both a type and a security label. This system ensures that the value of a high security variable is not leaked to a low security variable by construction.

Secure information flow comprises two broad aspects: i) information confidentiality and ii) information integrity[8]. A well studied notion of secure information flow is non-interference[9, 10] which requires that the public outputs of a program be independent of the program's secret variables. Intuitively, this provides a notion of confidentiality, as an external observer who is allowed

to access only the public variables of the system should not be able to deduce the internal state of the system. The original definition of non-interference for sequential programs was not well defined for concurrent and non-deterministic systems. Roscoe[11] and McLean[12] defined low security observational determinism as a generalization of non interference. Observational determinism states that all pairs of executions of a program must satisfy the condition that the public outputs of corresponding states are not distinguishable, in the sense that they do not give away any information about the secret internal variables in the program. In both the executions of the program the secret variables are allowed to be different.

The example below shows a program snippet consisting of two variables, *high_var* and *low_var* which are private and public variables, respectively. *low_var* depends on *high_var* through the branch condition and its value can be used to deduce the state of *high_var*.

An example of an information leak

```
int low_var;
bool high_var;
if (high_var == true)
    low_var = 10;
else
    low_var = 20;
```

In this thesis we consider programs modelled as state transition systems. A trace of a transition system is a sequence of valid states, meaning that the sequence of states is allowed by the transition system. A trace property is a program specification which is defined over the variables of a single program state. As the definitions of non-interference and observational determinism are over two traces of a program, we cannot express the property for secure information as a trace property. We cast the problem of low security observational determinism as a verification instance over two logical program traces viz. a 2-safety hyperproperty verification problem. One way to verify 2-safety hyperproperties is to self-compose[13] the program that is to combine two copies of the program, which is then analyzed. We set the secret variables in both the copies of the program to be the same at the start of the program execution and then we verify if it is possible for the public variables to have different values at any step in the execution. However, self-composition involves the exploration of a large state space and is computationally very expensive. Several methods to reduce the cost of verification have been proposed including type based analysis[14], constructing product programs with aligned fragments[15], lockstep execution of loops[16] and transforming Horn clause rules[17, 18].

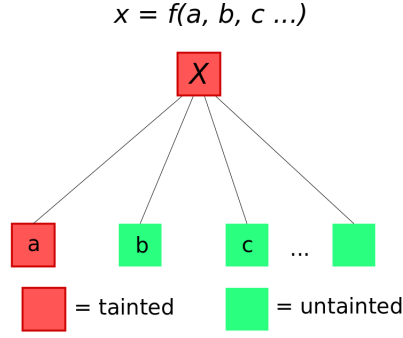


Figure 1.2: Taint Propagation: A variable is tainted if atleast one of its dependencies is tainted

1.2 Verification of Secure Information Flow

One way to verify secure information flow in a program is to use dynamic taint analysis[19, 20, 21]. Taint analysis is a method of dynamic information flow analysis, where the program in question is augmented and run to verify flow properties. Each program variable has a bit associated with it. Initially, the taint bit of the variables whose information flow in the program needs to be tracked is turned on. In subsequent program execution, a variable is tainted whenever the value stored in it depends on the value of a variable that is already tainted. For example consider the program fragment $x = y + z$. Now, the variable x becomes tainted (that is its taint bit x_t is set) if either of the variables y or z is tainted viz. $x_t = y_t \vee z_t$. Taints are sound in the sense that if there is a true interdependence between two variables where one is initially tainted, the second variable will end up tainted. However, the taints are not complete in the sense that a tainted variable need not imply an interdependence with the set of variables that was initially tainted. For example, a branch condition which is tainted will taint all the variables in both the branches even if a variable is set to the same value in both the branches.

Recent work by Yang *et al.* [22] proposed the use of taints to reduce the cost of self-composition. Every variable in the original transition system has a corresponding Boolean taint variable. If a taint variable is true at a particular step in the execution of the taint transition, then it implies an equality between the corresponding program variables in the self-composed system at that particular execution step. If a taint variable is false then the corresponding program variables may or may not be equal in both the traces. The problem of secure information flow now reduces to checking if the taint variables corresponding to the public variables are true throughout the execution of the program. In practice, the taint transition system greatly reduces the state space for finding an information flow violation.

The taint transition provides a sound over-approximation for equalities between program vari-

ables in the self-composed system. This implies that a particular taint variable can be false when in fact the corresponding variables were equal at that particular step in the self-composed system. In turn, this means that if the taint variable corresponding to a public variable turns out to be false, it need not be an actual violation of the original property. To counter this imprecision the actual property is verified using a model checking instance of the self-composed system till the length of the taint counter example. If the property does not hold, we have found an actual counter example to the property. If the property holds then the taint transition system is refined. In the paper by Yang *et al.* the refinement is done by setting the taint variables to the correct values depending on the output of the model checker till the length of the taint counter example.

The problem with this refinement strategy is that it does not generalize over the whole execution of the program. The underlying SMT solver does not get any hint as to how the taints may evolve in future execution steps.

1.3 Solution Summary

We augment the taint transition system such that the taint variables depend on the values of the program variables as well as the branch conditions. We let the user specify state predicates (i.e. Boolean formulae over of the program variables) which are used to refine the taint variables when a violation is detected. When a taint property violation is detected, we first check if the violation is spurious. If the violation is in fact spurious, then we iteratively attempt to refine the taint variable using the user supplied refinement predicate. For each predicate, we add the condition to the taint transition system that the particular taint variable being refined is true if the refinement predicate holds, else the original relation for that taint variable holds. The refinement predicate could give rise to unsoundness of the taint transition system and hence a self-composed system is used to verify that the taint variable indeed implies an equality of the variables in the two copies of the program. If the refinement is sound then we run the taint property verification again. We repeat this process until either all the refinements have been exhausted for all the taint variables or if we have managed to prove the taint property. For each variable we keep track of the refinement predicates that are yet to be tried out.

1.4 Our Contributions

In this work we extend the taint transition system introduced by Yang *et al.* to accommodate the verification of a restricted class of 2-safety hyperproperties which captures the problem of secure information flow.

1. We formalize a taint transition system that accommodates variables of type integer, bit-vector, Boolean and also multi dimensional arrays. Each variable in the taint transition system of type Boolean corresponds to a particular variable in the original transition system. Array variables also have a Boolean taint array which captures whether a particular location in the array is equal in the two traces of the program. For each type of assignment to the variables, we define the corresponding update to the taint variables.
2. We give an algorithm to convert a restricted class of 2-safety properties into a corresponding taint property. We check if the taint property is satisfied, if yes then by construction the two safety property is satisfied well. If not, then we try out functional refinements to the taint variables using lambda expressions that are input by the user.
3. We implement the taint transition system and the refinement on top of the UCLID5 verification and synthesis framework. UCLID5 uses the Z3 SMT solver from Microsoft as it's solver backend. We try our scheme of refinement over a set of examples and show that it outperforms lazy self-composition by Yang *et al.*

1.5 Structure of this document

Chapter 2 details the background material needed to describe the core ideas of the thesis. These include descriptions of the system model, the verification techniques used and details on the types of program specifications used. Chapter 3 describes the approach used by Yang *et al.* for lazy self-composition along with our improvements to the design. Chapter 4 is a collection of example UCLID5 programs on which our refinement scheme does better than the existing refinement scheme of Yang *et al.* . Chapter 4 details the concluding remarks and the future work to be done.

Chapter 2

Background

In this chapter we will discuss the main ideas and results which lead up to the core of the thesis.

2.1 Abstract Program Model

In this thesis, we consider programs modelled as state transition systems defined using first order logic modulo a theory \mathcal{T} denoted by $FOL(\mathcal{T})$. We associate a transition system with every program P , namely $M = \langle X, Init(X), Tr(X, X') \rangle$ where X is a set of uninterpreted constants representing program variables while $Init$ and Tr are quantifier-free formulas in $FOL(\mathcal{T})$ representing the conditions on the initial states and transition relation respectively. A state of the transition system is an assignment of values to all the variables in X . These states correspond to structures over a signature $\Sigma = \Sigma_{\mathcal{T}} \cup X$ where $\Sigma_{\mathcal{T}}$ is the alphabet as defined by $FOL(\mathcal{T})$. We write $Tr(X, X')$ to denote that Tr is defined over the signature $\Sigma_{\mathcal{T}} \cup X \cup X'$, where X is used to represent the pre-state of a transition, and $X' = \{a' \mid a \in X\}$ is used to represent the post-state.

We look for bugs in the program by defining formulae over runs of the transition system to capture “bad” conditions.

A trace of a program is a valid sequence of states as defined by the initial condition and the transition relation. Let $s_0 \dots s_{n-1}$ be a valid trace of a program with transition system $M = \langle X, Init(X), Tr(X, X') \rangle$. Then it must be the case that $Init(s_0) \bigwedge_{i=0}^{n-2} Tr(s_i, s_{i+1})$ evaluates to true. Traces can be of finite as well as of infinite length. Each program can be viewed as the set of its valid traces. Given a transition system M , we will write the $Traces(M)$ to denote the set of all valid traces of M . We will denote traces by π, π_1, π_2 etc. The i th element of the trace π will be referred to by $\pi[i]$.

Example Transition System

Let

- $M = \langle X, Init(X), Tr(X, X') \rangle$ where
- $X = \{x, y, z\}$
- $Init(X) = (x \geq 0) \wedge (y \geq 0) \wedge (z > 0)$
- $Tr(X, X') = (x' = x + y) \wedge (y' = y + z) \wedge (z' = z + 1)$

Then $\{(x = 2, y = 2, z = 3), (x = 4, y = 5, z = 4), (x = 9, y = 9, z = 5) \dots\}$ is a valid trace of this program. $\pi[2] = (x = 9, y = 9, z = 5)$ and $\pi[2].x = 9$.

2.2 Property Specification

2.2.1 Specifying Trace Properties of Programs

A trace property is one which is specified over the variables of a single program trace. A single trace property defines a set of traces P and a system S satisfies the property if the set of all possible traces of the system is a subset of P . Trace properties are of two types:

1. Safety: Properties whose violations give a finite length trace as a counter example.
2. Liveness: Properties whose violations give an infinite length trace as a counter example.[23]

Trace Property Example

Let

- $M = \langle X, Init(X), Tr(X, X') \rangle$ where
- $X = \{x, y, z\}$
- $Init(X) = (x \geq 0) \wedge (y \geq 0) \wedge (z > 0)$
- $Tr(X, X') = (x' = x + y) \wedge (y' = y + z) \wedge (z' = z + 1)$

The safety property which specifies that $z \geq 0$ in all states is as follows:

$$\forall \pi \in Traces(M) : \forall i : \pi[i].x \geq 0$$

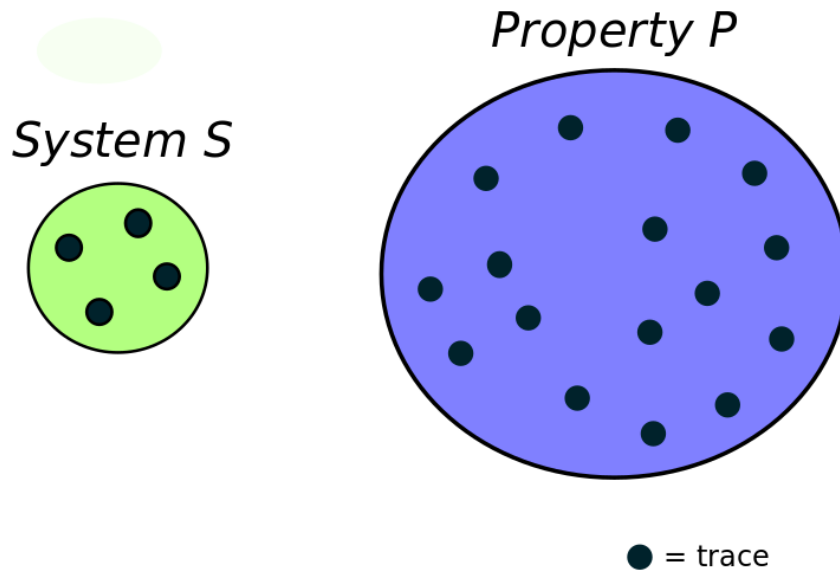


Figure 2.1: Trace view of systems and single trace properties

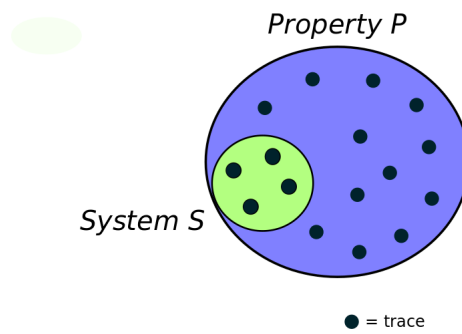


Figure 2.2: System *S* satisfies single trace property *P*

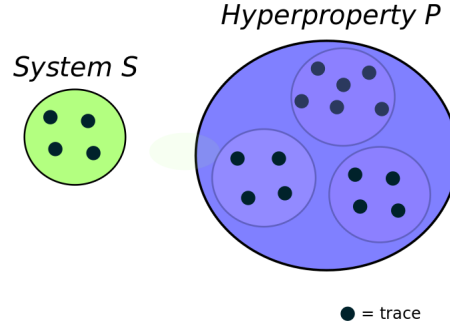


Figure 2.3: Trace view of systems and hyperproperties: Hyperproperty P is a set of set of traces

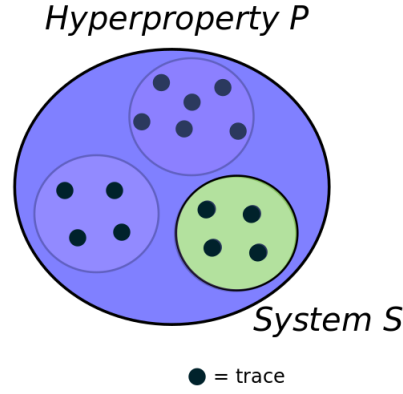


Figure 2.4: System S satisfies hyperproperty P

2.2.2 Hyperproperties

Properties which are defined over one or more traces are called hyperproperties[24]. Unlike single trace properties, a hyperproperty H defines a set of set of program traces P . A system S satisfies hyperproperty H if the set of all possible traces of the system S is an element of the set P . Similar to single trace properties, hyperproperties are of two types:

1. **Hypersafety:** If a set of traces T violates a hypersafety property P then there exists a set of traces M , a prefix of T such that any extension of M also does not satisfy the property P . A set of traces A is a prefix of a set of traces B if for each trace $t \in B$ there exists a trace $t' \in A$ such that t' is a prefix of t . For example, observational determinism and non-interference.
2. **Hyperliveness:** A hyperproperty P is a hyperliveness property if for all finite sets of finite traces, each element in the set can be extended with a suffix T and $T \in P$. For example, service level agreements like average response time.

Some examples of hyperproperties are:

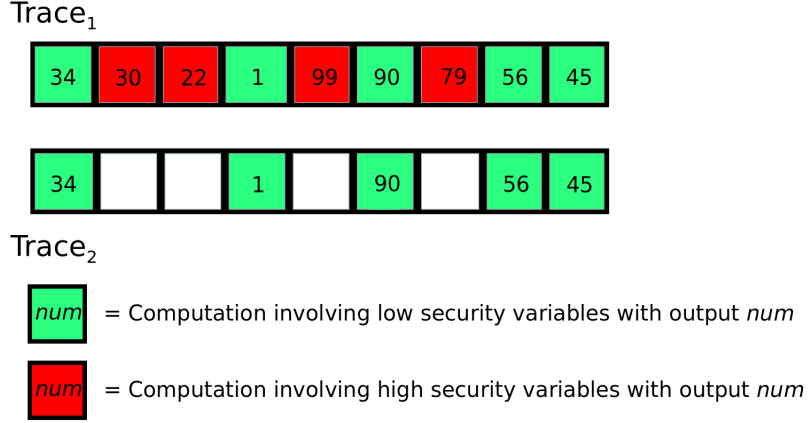


Figure 2.5: Non-interference: In the second trace the high security states are removed

1. Non interference: Non-interference for deterministic and sequential programs is a hyperproperty which defines indistinguishability between pairs of program executions in the sense that the public output of the program remains the same even when certain states pertaining to the usage and modification of the high security variables are removed from the trace. This can be stated as:

$$\forall \pi_1 \exists \pi_2 : \forall i : (ev_{low}(\pi_1[i]) = ev_{low}(\pi_2[i])) \wedge (ev_{hi}(\pi_2[i]) = \perp) \wedge (Obs(\pi_1[i]) = Obs(\pi_2[i]))$$

Here ev_{low} and ev_{hi} denote the low and high variables in a state respectively and Obs denotes the publicly observable information of a state.

2. Observational Determinism: Observational determinism is a 2-safety property which generalises the notion of indistinguishable program executions to non-deterministic and concurrent programs. Given that the secret variables in the two program traces can be initialized to different values, the public variables being the same in the initial state, we want to verify that the observable output of the program is the same in both the traces. This can be stated as:

$$\forall \pi_1 \pi_2 : ev_{low}(\pi_1[0]) = ev_{low}(\pi_2[0]) \implies \forall i : Obs(\pi_1[i]) = Obs(\pi_2[i])$$

Observational determinism can be also be viewed as a 2-player game consisting of an attacker and a defender. The goal of the attacker is to learn the value of a secret variable in the system. The defender chooses the secret value to be used during execution from a public set containing two values. The attacker can provide arbitrary inputs to the system and

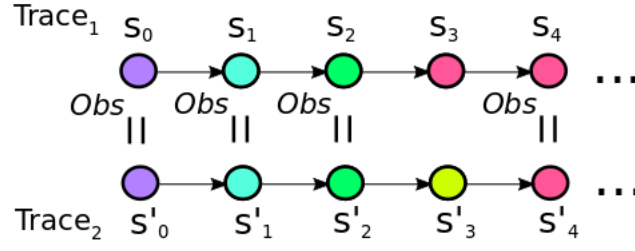


Figure 2.6: Observational Determinism: The colours represent the observable output of the states

observer its public outputs. The attacker wins if he can guess the secret which is used with a probability significantly greater than 0.5.

2.2.3 Specifying Secure Information Flow

Secure information flow in sequential programs is specified using non interference whereas for non deterministic and concurrent programs, it is specified as observational determinism. The two aspects of secure information flow are confidentiality and integrity.

Specifying Confidentiality

Confidentiality in sequential and non deterministic programs can be specified using observational determinism. The goal is to verify that the program does not leak any information about its secret variables through its observable public outputs. In addition to having secret variables in a program, a program may also take in untrusted inputs at every state in the execution. The high security variables in both the traces are set to different values while the public values in the initial state are set to identical values. The property to be proved is that the observable values at each step of execution of both the traces are identical. More formally:

$$\begin{aligned}
 & \forall \pi_1 \pi_2 : (ev_{low}(\pi_1[0]) = ev_{low}(\pi_2[0])) \wedge \\
 & \forall i : (inp_{untrusted}(\pi_1[i]) = inp_{untrusted}(\pi_2[i])) \\
 & \implies \forall i : Obs(\pi_1[i]) = Obs(\pi_2[i])
 \end{aligned}$$

Specifying Integrity

Integrity in the context of secure information flow is the dual of confidentiality. The objective of verifying integrity in a program is that an untrusted program variable should not affect the values of a trusted program variable. If initially the trusted values are equal in both the traces and the

trusted input to the program is the same in all the steps then it should be the case that the trusted variable has the same values in both the traces. More formally:

$$\begin{aligned} & \forall \pi_1 \pi_2 : (ev_{trusted}(\pi_1[0]) = ev_{trusted}(\pi_2[0])) \wedge \\ & \forall i : (inp_{trusted}(\pi_1[i]) = inp_{trusted}(\pi_2[i])) \\ & \implies \forall i : ev_{trusted}(\pi_1[i]) = ev_{trusted}(\pi_2[i]) \end{aligned}$$

2.2.4 Verification Techniques

There are several techniques by which system properties can be verified. In this thesis we use Bounded Model Checking(BMC) and Unbounded Model Checking using Property Directed Reachability.

Algorithm 2 Algorithm for Bounded Model Checking

```

 $F_0(s_0) \leftarrow Init(s_0)$ 
 $i \leftarrow 0$ 
while  $i \leq k$  do
  /* Check if the property holds for the current state */
  if  $\neg \phi(s_i) \wedge F_i$  is SATISFIABLE then
    return UNSAFE
  else
    /*If property holds then add the transition predicate to  $F_{i+1}$ */
     $F_{i+1} \leftarrow F_i \wedge Tr(s_i, s_{i+1})$ 
     $i \leftarrow i + 1$ 
  end if
end while
return UNKNOWN

```

1. Bounded Model Checking(BMC): In bounded model checking[25], we expand the transition system for as many states as specified by the bound of the verification. On expanding each new state, we check if the invariant holds till that many steps using an SMT solver. More formally, given a program P with transition system $M = \langle X, Init(X), Tr(X, X') \rangle$, a bound k and an invariant ϕ , we define the algorithm for Bounded Model Checking(BMC) as shown in Algorithm 2.

At each iteration in the while loop, if the condition for safety is not violated then we expand the next state. Else we have found a counter example in the i^{th} step.

2. Unbounded model checking using Property Directed Reachability: Unbounded model checking[2, 3] does not impose a bound on the number of execution steps of the program. Property directed reachability maintains a list of facts called a trace and then derives new facts based on queries on a single transition. There is no unrolling of the transition system as in BMC.

Chapter 3

Verification of Secure Information Flow

This thesis focuses on the verification of secure information flow properties. A standard technique for this is known as self-composition [13], which we describe first. We then describe two enhancements to self-composition: lazy self-composition and the dependent taint abstraction.

3.1 Self Composition

A program is self composed by making an identical copy of itself and renaming its variables [13]. Both the copies of the transition system are then bound together such that their transitions “move” together. In other words, given a state machine we are constructing the product state machine containing the cross product of all the states of both the copies.

Consider a program P with transition system $M = \langle X, \text{Init}(X), \text{Tr}(X, X') \rangle$. Let $X_1 = \{a_1 \mid a \in X_1\}$, $X'_1 = \{a'_1 \mid a_1 \in X_1\}$, respectively. X_2 and X'_2 are defined analogously. $M_{sc} = \langle Z, \text{Init}_{sc}, \text{Tr}_{sc}(Z, Z') \rangle$ is defined as the self-composition of M where:

$$\begin{aligned} Z &= X_1 \cup X_2 \\ \text{Init}_{sc}(Z) &= \text{Init}(X_1) \wedge \text{Init}(X_2) \\ \text{Tr}_{sc}(Z, Z') &= \text{Tr}(X_1, X'_1) \wedge \text{Tr}(X_2, X'_2) \end{aligned} \tag{3.1}$$

Barthe *et al.* [13] proposed the self-composition of programs as a general technique to the verification of secure information flow. Terauchi and Aiken [14] used self-composition for generalized 2-safety properties. The following condition captures observational determinism where Obs_x

defines whether or not the variable x is publicly observable in a particular state, and $L \subset X$ is the set of publicly observable variables :

$$Bad_{sc}(Z) = \bigvee_{x \in L} Obs_x(X_1) \wedge Obs_x(X_2) \wedge \neg(x_1 = x_2) \quad (3.2)$$

Observational determinism is violated when there is an execution of M_{sc} such that there exists a particular variable x which is public and observable in the original transition system but has different values in both the traces of the self-composed system M_{sc} .

After the construction of M_{sc} either BMC or the horn solver is used to check if the property $Bad_{sc}(Z)$ is violated or not.

Self-Composition Example

Let

- $M = \langle X, Init(X), Tr(X, X') \rangle$ where
- $X = \{x, y, z\}$
- $Init(X) = (y = x + 1) \wedge (x = z + 1) \wedge (z > 0)$
- $Tr(X, X') = (y' = x + 1) \wedge (z' = z + 1) \wedge (x' = x)$

Then the self-composition of the system is defined as $M_{sc} = \langle Z, Init_{sc}, Tr_{sc}(Z, Z') \rangle$ where

- $Z = \{x_1, y_1, z_1, x_2, y_2, z_2\}$
- $Init_{sc} = (y_1 = x_1 + 1) \wedge (x_1 = z_1 + 1) \wedge (z_1 > 0) \wedge (y_2 = x_2 + 1) \wedge (x_2 = z_2 + 1) \wedge (z_2 > 0)$
- $Tr_{sc}(Z, Z') = (y'_1 = x_1 + 1) \wedge (z'_1 = z_1 + 1) \wedge (x'_1 = x_1) \wedge (y'_2 = x_2 + 1) \wedge (z'_2 = z_2 + 1) \wedge (x'_2 = x_2)$

A valid trace of the self-composed system is as follows $\{((z_1 = 1, x_1 = 2, y_1 = 3), (z_2 = 2, x_2 = 3, y_2 = 4)), ((z_1 = 2, x_1 = 2, y_1 = 3), (z_2 = 3, x_2 = 3, y_2 = 4)), \dots\}$. In this case all the variables are equal and hence the property of observational determinism is satisfied irrespective of the public variables.

3.2 Lazy Self-Composition

Verifying the self-composition of a program is computationally very expensive as the SMT solver has to reason over two independent executions of the same program. In the subsequent sections we introduce techniques to reduce the computation cost of self-composition. The first is lazy

self-composition introduced by Yang *et al.* [22] and the second is our approach.

3.2.1 Taints for Self-Composition

Let P be a program with a transition system $M = \langle X, \text{Init}(X), \text{Tr}(X, X') \rangle$.

Let $X_s \subseteq X$, $X_t = \{x_t \mid x \in X_s\}$ and $Y = X_s \cup X_t$. Each x_t in X_t is a Boolean variable. Lazy self-composition defines an augmented transition system M_t over the variables Y . This augmented transition system M_t has two characteristics, first it is a bisimulation of M [26, 27], and second for each variable x_t , M_t maintains the invariant that $x_t \implies x_1 = x_2$ where $x_1 \in Z$ and $x_2 \in Z$ are the two copies of the variable x in the self-composition $M_{sc} = \langle Z, \text{Init}_{sc}, \text{Tr}_{sc}(Z, Z') \rangle$.

$M_t = \langle Y, \text{Init}_t, R_t(Y, Y') \rangle$ is defined as the taint transition system of M where:

$$\begin{aligned} \text{Init}_t(Y) &= \text{Init}(X_s) \wedge \text{Init}_{x_t}(X_t) \\ R_t(Y, Y') &= \text{Tr}(X_s, X'_s) \wedge R_{x_t}(Y, X'_t) \end{aligned}$$

In the definition above $R_{x_t}(Y, X'_t)$ is the state update of the taint variables.

Here all the pairs of corresponding variables which are equal in the initial state of the self-composed system have their corresponding taint variables set to true in Init_t . A taint variable is updated by ANDing all the taint variables of its dependent variables in that step. Dependencies include branch conditions as well as the right hand sides of direct assignment statements.

Taint Example

Let

- $M = \langle X, \text{Init}(X), \text{Tr}(X, X') \rangle$ where
- $X = \{x, y, z\}$
- $\text{Init}(X) = (y = x + 1) \wedge (x = z + 1) \wedge (z = 0)$
- $\text{Tr}(X, X') = (y' = x + 1) \wedge (z' = z + 1)$

Then the taint transition system is defined as $M_t = \langle Y, \text{Init}_t, R_t(Y, Y') \rangle$ where

- $Y = \{x, y, z, x_t, y_t, z_t\}$
- $\text{Init}_t(Y) = x_t \wedge y_t \wedge z_t \wedge (y = x + 1) \wedge (x = z + 1) \wedge (z = 0)$
- $R_t(Y, Y') = (y' = x + 1) \wedge (z' = z + 1) \wedge (y'_t = x_t) \wedge (x'_t = z_t)$

3.2.2 Converting a 2-Safety Hyperproperty into a Taint Property

In this thesis we will restrict our attention to 2-safety hyperproperties which are *taint convertible*. A 2-safety property ϕ is said to be taint convertible if it is derivable using the following rules:

$$\begin{aligned} e &::= v \mid op(v_1, \dots, v_n) \\ t1 &::= e \mid t1 \wedge t1 \\ t2 &::= eqTr(t1) \mid t2 \wedge t2 \\ \phi &::= t2 \implies t2 \end{aligned}$$

Here op represents any quantifier-free formula in $FOL(\mathcal{T})$ over variables $v_1 \dots v_n$ of the transition system M and $eqTr$ is a boolean predicate which structurally equates every variable in a t_1 expression in both the traces. For example, $eqTr(x + y - z)$ would translate to $\forall \pi_1 \pi_2 : \forall i : (\pi_1[i].x = \pi_2[i].x) \wedge (\pi_1[i].y = \pi_2[i].y) \wedge (\pi_1[i].z = \pi_2[i].z)$

For each equality in a taint convertible 2-safety property we replace the same with the corresponding taint variable for the variable involved in the equality. For example if the taint convertible 2-safety property is as follows:

$$\begin{aligned} \forall \pi_1 \pi_2 \forall i : (\pi_1[i].x = \pi_2[i].x) \wedge (\pi_1[i].y = \pi_2[i].y) \wedge (\pi_1[i].z = \pi_2[i].z) \wedge f(\pi_1[i].x, \pi_1[i].y, \pi_1[i].z, \dots) \implies \\ (\pi_1[i].mem[\pi_1[i].x] = \pi_2[i].mem[\pi_2[i].x]) \wedge (\pi_1[i].d = \pi_2[i].d) \end{aligned}$$

Then the corresponding taint property is:

$$\forall \pi \in Traces(M_t) : \forall i : (\pi[i].x_t) \wedge (\pi[i].y_t) \wedge (\pi[i].z_t) \wedge f(\pi[i].x, \pi[i].y, \pi[i].z, \dots) \implies \pi[i].mem_t[\pi[i].a] \wedge \pi[i].d_t \dots$$

By this construction it can be safely concluded that whenever the taint property holds the original property holds as well but not vice-versa.

3.2.3 Lazy Self-Composition using BMC

The vanilla version of self-composition mandates that all the variables be duplicated in each state of unrolling in the case of unbounded model checking. Yang *et al.* [22] use the taint transition system to only selectively duplicate variables thereby reducing the load on the underlying SMT solver.

The following algorithm formalizes this:

Algorithm 3 *LazySC_BMC*(M, ϕ, k)[22]

Input: Transition system M , a 2-safety property ϕ , bound k
Output: UNSAFE if the program does not satisfy the property ϕ else UNKNOWN
 $i \leftarrow 0$
 $F_0(s_0) = \text{Init}(s_0)$
while $i \leq k$ **do**
 $M_t(i) \leftarrow \text{EncodeTaint}(M(i))$ // Compute the taint transition function till the i^{th} step
 /* Get the unrolled transition system of the self-composed system till the i^{th} step */
 TR of $M_{sc}(i) \leftarrow \text{LazySC}(F_i, M_t(i))$
 $\text{result} \leftarrow \text{SolveSMT}(\text{query } M_{sc}(i) \wedge \neg\phi(i))$
 if $\text{result} = \text{counterexample}$ **then**
 return UNSAFE
 else
 $i \leftarrow i + 1$
 $F_{i+1} \leftarrow F_i \wedge \text{Tr}(s_i, s_{i+1})$
 end if
end while
return UNKNOWN

Algorithm 4 *LazySC*(M_t, M)[22]

Input: Transition system M and the corresponding taint transition system M_t
Output: Transition relation of the self-composed program M_{sc}
/* Walk through each state update in the transition system and construct the transition system of M_{sc} */
for each state update $x \leftarrow \delta$ **do**
 Add state update $x_1 \leftarrow \delta$ to M_{sc}
 $\text{tainted} \leftarrow \text{SolveSMT}(\text{query } x_t \text{ on } M_t)$
 if $\text{tainted} = \text{true}$ **then**
 Add state update $x_2 \leftarrow x_1$ to M_{sc}
 else
 Add state update $x_2 \leftarrow \text{duplicate}(\delta)$ to M_{sc}
 end if
end for
return M_{sc}

In the previous taint example, the taints remain true for each variable and while unrolling the self-composed system the expressions in the right hand side of assignments in the second copy are not duplicated. Instead the variables in the second copy are assigned to be equal to those in the first copy.

Lazy Self-Composition using BMC Example

Let

- $M = \langle X, \text{Init}(X), \text{Tr}(X, X') \rangle$ where
- $X = \{x, y, z\}$
- $\text{Init}(X) = (y = x + 1) \wedge (x = z + 1) \wedge (z = 0)$
- $\text{Tr}(X, X') = (y' = x + 1) \wedge (z' = z + 1) \wedge (x' = x)$

Then the taint transition system is defined as $M_t = \langle Y, \text{Init}_t, R_t(Y, Y') \rangle$ where

- $Y = \{x, y, z, x_t, y_t, z_t\}$
- $\text{Init}_t(Y) = x_t \wedge y_t \wedge z_t \wedge (y = x + 1) \wedge (x = z + 1) \wedge (z = 0)$
- $R_t(Y, Y') = (y' = x + 1) \wedge (z' = z + 1) \wedge (x' = x) \wedge (y'_t = x_t) \wedge (x'_t = x_t) \wedge (z'_t = z_t)$

Let var_{ij} denote an unrolled version of variable var of the i^{th} copy of variables in the self composition M_{sc} and in the j^{th} step of unrolling the corresponding transition system M . For BMC bound $k = 2$, the unrolled transition system for M_{sc} using vanilla self-composition will be:

$$\begin{aligned}
 & (y_{11} = x_{11} + 1) \wedge (x_{11} = z_{11} + 1) \wedge (z_{11} = 0) \wedge \\
 & (y_{12} = x_{11} + 1) \wedge (z_{12} = z_{11} + 1) \wedge (x_{12} = x_{11}) \wedge \\
 & (y_{21} = x_{21} + 1) \wedge (x_{21} = z_{21} + 1) \wedge (z_{21} = 0) \wedge \\
 & (y_{22} = x_{11} + 1) \wedge (z_{22} = z_{11} + 1) \wedge (x_{22} = x_{21}) \wedge
 \end{aligned}$$

The transition system constructed using lazy self composition using a BMC bound of $k = 2$ will be:

$$\begin{aligned}
 & (y_{11} = x_{11} + 1) \wedge (x_{11} = z_{11} + 1) \wedge (z_{11} = 0) \wedge \\
 & (y_{12} = x_{11} + 1) \wedge (z_{12} = z_{11} + 1) \wedge (x_{12} = x_{11}) \wedge \\
 & (y_{21} = y_{11}) \wedge (x_{21} = x_{11}) \wedge (z_{21} = z_{11}) \wedge \\
 & (y_{22} = y_{12}) \wedge (z_{22} = z_{12}) \wedge (x_{22} = x_{12}) \wedge
 \end{aligned}$$

In the case of lazy self-composition using BMC, at each step of unrolling the self composed system M_{sc} , for each state update, the taint transition system is queried to find if the corresponding variable is tainted at that particular step. If yes, then we do not duplicate the right-hand-side expression of the state update in the second copy as we have proved that the variables are equal in both the traces at that particular step. If not, then we duplicate the state update in the second copy.

In the example above, all the variables have their corresponding taint variables constrained to be true in all the steps. Therefore, all variables in the second copy are assigned to those in the first copy.

3.2.4 Lazy Self-Composition using Property Directed Reachability

A useful property of a Property Directed Reachability based solver is that it gives an inductive trace in case it finds a counter example. When a taint property counter example is found, the inductive trace which the solver provides can be used to refine the self-composed model. Also for each violation of a taint property we verify for each untainted variable if the violation is spurious or not by constructing a BMC instance of the self-composed program. Based on the trace generated by the BMC instance, we also modify the taint transition system to correct the taint values of variables which are equal at a particular step but have a taint violation in that step.

Roughly the algorithm for lazy self-composition using PDR is as follows:

1. Construct a horn clause model of the transition system as well as the taint transition system
2. Given a taint property, we query the horn solver to see if it is violated
3. If the taint property is not violated then we return safe
4. Else we rewrite the self-composed transition system as was done in LazySC_BMC by setting the corresponding variables in the two copies to be equal if the taint variable in a particular step is true. Then we construct a self-composed bounded model checking(BMC) instance of the transition system till the length of the taint counter example and we query the solver if the original property is violated or not
5. If the original property is violated we return unsafe
6. Else we use the trace provided by the BMC query to refine the taint transition system by setting the taint variables to true at their respective steps where their corresponding variables are equal
7. Goto step 1

Algorithm 5 *LazySC_Horn*(M, ϕ)[22]

Input: Transition system M and a taint convertible 2-safety property ϕ **Output:** SAFE, UNSAFE or UNKNOWN $M_t \leftarrow \text{ConstructTaintModel}(M, \phi)$ $M_d \leftarrow \text{ConstructSelfComposition}(M)$ $G \leftarrow [G_0 = \text{Init}_t]$ // The inductive trace of the taint transition system $H \leftarrow [H_0 = \text{Init}_d]$ // The inductive trace of the self-composed system**repeat** $(G, R_{\text{taint}}, \pi) \leftarrow \text{solveHorn}(M_t, G)$ **if** R_{taint} is SAFE **then**

return SAFE

else

/* Rewrite the inductive trace of the self-composed system by replacing state updates with equalities between the corresponding variables if the taint variable at that step is true */

 $H \leftarrow \text{Rewrite}(G, H)$ $M_c \leftarrow \text{Constraint}(M_d, \pi)$ $(\mathbf{H}, R_s, \pi) \leftarrow \text{solveBMC}(M_c, H)$ **if** $R_s = \text{UNSAFE}$ **then**

return UNSAFE

else

/* Rewrite the inductive trace of the taint transition system by setting taint variables to true if the corresponding variables are proved to be equal in the self-composed system */

 $G \leftarrow \text{Rewrite}(H, G)$ $M_t \leftarrow \text{Refine}(M_t, G)$ // Use the new taint trace to update the taint transition system **end if****end if****until** infreturn UNKNOWN

The horn solver version of self-composition attempts to reduce the search space to find a counter example. The taint properties which are queried are stronger conditions than the original property. However, the refinement strategy used does not generalize over the program execution. The problem with this refinement strategy is that it does not generalize to future steps in the program execution. In many cases the refinement loop has been shown to not terminate as the actual property may hold but the taint property is violated at every step leading to a refinement and a restart of the verification loop.

Non-Termination of LazySC using PDR

Let

- $M = \langle X, Init(X), Tr(X, X') \rangle$ where
- $X = \{x, y\}$
- $Init(X) = (y > 0) \wedge (x = 0)$
- $Tr(X, X') = (y' = y) \wedge (x' = ite(y > 0, x + 1, y + 3))$

Then the taint transition system is defined as $M_t = \langle Y, Init_t, R_t(Y, Y') \rangle$ where

- $Y = \{x, y, x_t, y_t\}$
- $Init_t(Y) = x_t \wedge \neg y_t \wedge (y > 0) \wedge (x = 0)$
- $R_t(Y, Y') = (y' = y) \wedge (x' = ite(y > 0, x + 1, y + 3)) \wedge (y'_t = y_t) \wedge (x'_t = y_t \wedge ite(y > 0, x_t, y_t))$

In the transition system above, the values taken by the variable x are equal in both the traces of the self composed system as $y > 0$ for all steps. However, x becomes untainted from the second step onwards because the branch condition depends on y which is untainted in all the steps. Starting from the second step, using BMC queries we find that the variable x is equal and the variable y to be unequal in both the traces. Following the refinement strategy of Yang *et al.*, we set x_t to be true in the second step of the taint transition system. However, y_t still remains false in the second and this makes x_t to be false in the third step. After refining x_t in third step, the fourth step is still faulty as y_t is always false. This continues for every state and hence this refinement strategy never terminates.

3.3 Dependent Taints for Lazier Self-Composition

We modify the refinement step used by Yang *et al.* [22] by letting the user specify a set of Boolean refinement predicates at the start of the verification procedure. The predicates are defined over the variables of the transition system M viz. X . Whenever there is a spurious counter example wrt. a taint variable, the taint transition system is modified so that the variable is true whenever the refinement predicate is true and if the refinement predicate is false then the original taint update to the variable holds. To ensure the approach remains sound, we introduce the refinement for a variable by first checking the refinement using a horn solver query over the self-composed system. Finally, the algorithm gives up once every refinement predicate has been tried out for each taint variable with a violating counter example.

Algorithm 6 *LazierSC*(M, ϕ, ψ)

Input: Transition system M , a taint convertible 2-safety property ϕ and a set of refinement predicates ψ

Output: SAFE, UNSAFE or UNKNOWN

$M_t \leftarrow \text{ConstructTaintModel}(M, \phi)$

$M_d \leftarrow \text{ConstructSelfComposition}(M)$

$G \leftarrow [G_0 = \text{Init}_t]$ // The inductive trace of the taint transition system

$H \leftarrow [H_0 = \text{Init}_d]$ // The inductive trace of the self-composed system

$\text{refinements} \leftarrow \{\}$

for x_t in X_t **do**

$\text{refinements} \leftarrow \text{refinements} \cup \text{initializeRefinementSet}(x_t)$

end for

repeat

$(G, R_{\text{taint}}, \pi) \leftarrow \text{solveHorn}(M_t, G)$

if R_{taint} is SAFE **then**

 return SAFE

else

 /* Rewrite the inductive trace of the self-composed system by replacing state updates with equalities between the corresponding variables if the taint variable at that step is true */

$H \leftarrow \text{Rewrite}(G, H)$

$M_c \leftarrow \text{Constraint}(M_d, \pi)$

$(H, R_s, \pi) \leftarrow \text{solveBMC}(M_c, H)$

if $R_s = \text{UNSAFE}$ **then**

 return UNSAFE

else

 /* Rewrite the inductive trace of the taint transition system using the user supplied refinement predicates */

$G \leftarrow \text{RewriteRefinement}(H, G, \text{refinements}, M, \psi)$

$M_t \leftarrow \text{Refine}(M_t, G)$

end if

end if

until inf

return UNKNOWN

Algorithm 7 *RewriteRefinement*($H, G, refinements, M, \psi$)

Input: An inductive trace H of the self-composed system of M , an inductive trace of the taint transition system of M , the set of refinements tried out for each variable $refinements$ and the set of user supplied refinement preficated ψ

Output: Rewritten taint trace G'

```

 $k \leftarrow length(G)$ 
/* We check the last state of the taint trace as once a taint variable becomes false, there is no
way that it can become true again */
for each false taint variable  $y_t$  in  $G_k$  do
   $M_c \leftarrow constructSelfComposition(M)$ 
  Bad of  $M_c \leftarrow \neg(y_1 = y_2)$ 
   $r \leftarrow solveBMC(M_c, k)$ 
  if  $r = \text{SAFE}$  then
    /* Get the next unused refinement predicate associated with variable  $y_t$  */
     $ref \leftarrow getNextRefinementt(\psi, refinements, y_t)$ 
    while  $ref \neq \text{NIL}$  do
       $M_{sc} \leftarrow constructSelfComposition(M)$ 
      Bad of  $M_{sc} \leftarrow \neg(ref_1 \implies (y_1 = y_2))$ 
      /*  $ref_1$  is a Boolean predicate over the variables of the transition system  $M$  */
       $t \leftarrow solveHorn(M_{sc})$ 
      if  $t = \text{SAFE}$  then
        break
      else
         $ref \leftarrow getNextRefinement(\psi, refinements, y_t)$ 
      end if
    end while
    if  $ref = \text{NIL}$  then
      continue
    else
      /* Add the predicate  $y_t = ite(ref_1, true, previousRHS)$  to each state in the taint inductive trace */
       $G \leftarrow Rewrite(G, ref_1)$ 
    end if
  end if
end for
return  $G$ 

```

Taint Refinement Example

Let

- $M = \langle X, Init(X), Tr(X, X') \rangle$ where
- $X = \{x, y\}$
- $Init(X) = (y > 0) \wedge (x = 0)$
- $Tr(X, X') = (x' = ite(y > 0, x + 1, y + 1) \wedge (y' = y))$
- User supplied refinement $\psi = \{y > 0\}$
- Property to be verified $\phi = \forall \pi_1 \pi_2 \in Traces(M) : \forall i : \pi_1[i].x = \pi_2[i].x$

The taint transition system for the system defined above is $M_t = \langle Y, Init_t, R_t(Y, Y') \rangle$ where

- $Y = \{x, y, x_t, y_t\}$
- $Init_t(Y) = x_t$
- $R_t(Y, Y') = (x'_t = y_t \wedge ite(y > 0, x_t, y_t) \wedge (y' = y))$
- The taint property $\phi_t = (x_t = true)$

Although the values of x is the same in both the copies, x_t is false from the second step onwards as the variable y is untainted and the branch condition depends on it. The refinement strategy used by Yang *et al.* fails in this case as y remains tainted for all steps beyond any counterexample that is found which in turn taints the variable x . Using our refinement strategy, the state update to x_t when using the first refinement predicate will be $x'_t = ite(y > 0, true, y_t \wedge ite(y > 0, x_t, y_t))$. As $y > 0$ is always true, x_t is always true and the taint property holds.

3.4 Details Unclear from Previous Work

The paper by Yang *et al.* [22] does not explain how arrays should be tainted. We introduce two type of taint variables for arrays: arr_{addr} and arr_{loc} .

Let arr be a variable of type $Array(inType, outType)$. arr_{addr} is of type Boolean whereas arr_{loc} is of type $Array(inType, Boolean)$. arr_{addr} captures whether the same array location is written in both the traces while arr_{loc} captures whether the same value is written to the array at both the traces.

Array Taint Examples

Let $x = arr[y + 1]$; then $x'_t = arr_{loc}[y + 1] \wedge arr_{addr} \wedge y_t$

The example above captures the fact that x is equal in both the traces if the location $y + 1$ is tainted implying that the value at that address is equal in both the traces, all the addresses written to thus far are equal in both the traces and the address being accessed is the same in both the trace

Let $arr' = arr[a \leftarrow b]$; then

$arr'_{addr} = arr_{addr}(arr) \wedge a_t$ and

$arr'_{loc} = arr_{loc}[a \leftarrow b_t]$

The example above illustrates the taint computation on an array store operation. The address taint is tainted if all the locations written thus far were equal in both the arrays and the address a is equal in both the traces. The location taint is updated by the taint value of the variable b indicating that the same value was stored in both the traces.

The taints for arrays are computed recursively for nested stores and complex expressions. They are also extended to support multidimensional arrays. If two arrays are equal in the initial step then arr_{addr} is initialized to true and all cells of arr_{loc} are set to true. Else arr_{addr} is initialized to false and all cells of arr_{loc} are set to false.

3.5 Limitations

The limitations of lazier self-composition are:

1. Both the array taints are either all initialized to true or to false. This causes imprecision as ranges of an array maybe equal in both the traces.
2. Currently the refinement step is implemented only for one variable and only works on programs whose taint property can be expressed as the truth value of a single taint variable
3. Refinements are limited to non-conditional expressions over the program variables. This does not allow the refinement of the taint array arr_{loc} . Taint arrays need to be refined using lambda variables

Chapter 4

Experiments

We implemented support for hyperproperty verification using Bounded Model Checking (BMC) and Property Directed Reachability (PDR) in the the UCLID5 synthesis and verification framework.

4.1 UCLID5

UCLID5 is an open-source program verification and synthesis framework written in Scala. The framework defines its own language using which programs can be modelled as state transition systems. The language supports a variety of datatypes including integers, multi-dimensional arrays and lambdas. Along with a system description, the properties to be checked and the algorithm to be used for verification are taken as inputs.

Systems modelled using UCLID5 are parsed to a solver agnostic Abstract Syntax Tree (AST). This AST is then converted to the appropriate form for the backend SMT solver. The properties to be proved are converted to solver queries. Currently UCLID5 uses the Z3 SMT solver from Microsoft.

The system variables are declared initially with their respective datatypes. The initial constraints and the transition relation are then defined for the system in the *init* and the *next* blocks respectively. A *control* block defines the verification technique along with its parameters. The following is an example program in UCLID which models the Fibonacci series and checks whether the second number is always lesser than or equal to the first number using Bounded Model Checking (BMC).

```

1 module main
2 {
3   var a, b : integer;
4   init {
5     a = 0; b = 1;
6   }
7
8   next {
9     b' = a + b; a' = b;
10  }
11
12  invariant eq_s : a <= b;
13
14  control {
15    v = unroll(5);
16    check;
17    print_results;
18  }
19 }

```

Example 4.1: A UCLID5 model that computes the Fibonacci sequence

4.1.1 BMC in UCLID5

BMC in UCLID5 is implemented by unrolling the transition system till the number of steps as specified by the bound. For each step in the transition system, a new set of SMT variables are created which is then constrained using the values of the variables in the previous step. The property being checked is expanded in each state and converted to an SMT query. Each query is a negation of the property being verified, if the negation is unsatisfiable then the property holds. All the constraints and queries are fed to an SMT solver which returns if the property holds. If the property does not hold, a counterexample leading to the property violation is returned.

4.1.2 Unbounded Model Checking in UCLID

Unbounded model checking is implemented in UCLID using k-induction and the Property Directed Reachability (PDR) engine. In the case of k-induction, the transition system is unrolled for k steps and the property to be proved is assumed to hold for $k - 1$ steps. Then the property for the k^{th} step is given as an SMT query to the solver.

The Property Directed Reachability (PDR) engine is invoked by framing the program to be verified as a set of horn clauses. The horn clauses inductively capture the execution of the program and the property to be verified is negated and fed to the solver's PDR engine.

4.2 Implementation

Lazy self composition using BMC and PDR was implemented in UCLID5. To facilitate this, support for hyperproperties and hyperaxioms were added as well. The overall implementation took over 4000 lines of Scala code.

4.2.1 Hyperproperty Support in UCLID

Support for k -safety properties were added to UCLID5. In the case of BMC this was done by creating k copies of every step of the transition system on the same solver object. In the case of the horn solver, the horn clauses were modified to have k copies of the variables. Hyperaxioms refer to axiom constraints which relate the variables of more than one copy while hyperinvariants refer to the k -safety property which is to be verified.

4.2.2 Transition Relations

The program transition relation and the initial state condition were reimplemented as lambda functions. This was done to facilitate transition function rewriting and substitutions.

4.2.3 Taints

Taints are computed for all constraints which are equalities. The support set of the RHS of an equality is used to compute the final taint updates of each variable.

4.2.4 Challenges in Implementation

The challenges faced during implementation were:

1. UCLID5 has a large codebase of over 25000 lines of code. Analysing and understanding the relevant parts of the codebase took a considerable amount of time
2. Implementing the lambda based transition system and the conversion required for the Horn clause solver was challenging as the existing code was tightly coupled
3. Debugging was challenging as test programs create relatively large transition systems and analysing them manually for errors in the constraints took a lot of time and energy

4.3 Results

To evaluate our scheme we used three representational examples where the tainting and refinement scheme used by Yang *et al.* does not fare well as the refinement used only depends on the length of the counter example and not based on any fact about the program execution. In our examples, we use refinements which have been shown to be sound using self composition. The time taken to verify the soundness of the refinements using PDR is negligible compared to the runtime of the examples.

4.3.1 Example 1: False Branch Condition Dependence

Consider the following program.

```

1 module main {
2
3   var x, y : integer;
4
5   init {
6     x = 0;
7     assume(y > 0);
8   }
9
10  next {
11    x' = if (y > 0) then x + 1 else y + 1;
12  }
13
14  hyperinvariant[2] eq_x: true ==> (x.1 == x.2);
15
16  control {
17    v = sygus_refine(y > 0);
18    check;
19    print_results;
20  }
21 }

```

Example 4.2: A UCLID5 model that demonstrates a false branch condition dependence

Analysis of Example 1

Here $x.1$ and $x.2$ denote the copies of x in the first trace and the second trace respectively.

The taint transition system of the program is as follows:

$$Z = \{x, y\}$$

$$Init_t = x_t \wedge step_t \wedge \neg y_t$$

$$Tr_t = (y'_t = y_t) \wedge (x'_t = y_t \wedge ite(y > 0, x_t, y_t))$$

The taint property to be verified is $x_t = true$. Here x_t remains false from the second step onwards as y_t is always false. However, the value of x in both the traces are equal as $y > 0$ and x is always incremented by one. Our algorithm finds that this is a false taint violation in the second step of execution and applies the refinement $y > 0$ to the taint variable x_t .

The taint update of x_t after refinement is $x_t = ite(y > 0, true, y_t \wedge ite(y > 0, x_t, y_t))$. This refinement ensures that x_t is always true as y is never updated in any of the states after the initial state.

4.3.2 Example 2: Similar Updates in Both Branches

Consider the following program.

```

1 module main {
2
3   var stored_pwd : [integer]integer;
4   var input_pwd : [integer]integer;
5   var stored_pwd_len : integer;
6   var input_pwd_len : integer;
7   var match : boolean;
8
9   var idx : integer;
10  var step : integer;
11
12
13  init {
14    idx = 0;
15    step = 0;
16    match = true;
17  }
18
19  next {
20    if (idx < input_pwd_len) {
21      if (idx < stored_pwd_len && stored_pwd[idx] == input_pwd[idx]) {
22        step' = step + 1;
23      }
24      else {
25        step' = step + 1;
26        match' = false;
27      }
28      idx' = idx + 1;
29    }
30  }
31
32  hyperaxiom[2] eq_inp : input_pwd.1 == input_pwd.2;
33  hyperaxiom[2] eq_len : (input_pwd_len.1 == input_pwd_len.2) && (input_pwd_len.1 > 0);
34  hyperaxiom[2] pos_len : (stored_pwd_len.1 > 0 && stored_pwd_len.2 > 0);
35
36  hyperinvariant[2] eq_step: true ==> step.1 == step.2;
37
38  control {
39    v = sygus_refine(idx <= input_pwd_len || step == 0 || previous_inp_char == previous_pwd_char);
40    check;
41    print_results;
42  }

```

Example 4.3: A UCLID5 model that demonstrates similar updates in both the branches

Analysis of Example 2

The program above compares a user input password with a stored password. The property we want to verify is that the number of steps taken for execution should be the same in both the traces if the input password has the same length in both the traces even though the password in both the traces can have different lengths. By verifying this property, we can ensure that an adversary does not learn any information about the stored secret password by observing the number of steps of execution of the program. The taint property to be verified in this program is $step_t = true$ in all the steps of the taint transition system.

In this program, we can see that the number of steps taken for execution only depends on the length of the input password. As the input password and its length is constrained to be equal in both the traces, the number of steps of execution is the same as well in both the traces.

The taint update of the variable $step_t$ is as follows:

$$step'_t = idx_t \wedge stored_pwd_len_t \wedge stored_pwd_addr \wedge stored_pwd_loc[idx] \wedge input_pwd_addr \wedge input_pwd_loc[idx] \wedge ite((idx < stored_pwd_len) \wedge (stored_pwd[idx] = input_pwd[idx]), step_t, step_t)$$

In this program the variable $step$ does not depend on the inner branch condition. However, the taint variable $step_t$ becomes false from the second state onwards because it depends on the $stored_pwd$ array. The $stored_pwd$ variable is unequal in both the traces and hence both the taint variables associated with it are in turn set to false in the initial state.

The refinement which we introduce sets $step_t$ to true whenever the predicate is satisfied. As updates to $step$ only depends on the condition $idx < input_pwd_len$, the taint property passes.

```

1 module main {
2
3   var powers : [integer]integer;
4   var base : integer;
5   var power : bv4;
6   var step : integer;
7   var result : integer;
8
9   init {
10     step = 0;
11     result = 1;
12     powers[0] = base;
13     powers[1] = powers[0] * powers[0];
14     powers[2] = powers[1] * powers[1];
15     powers[3] = powers[2] * powers[2];
16   }
17
18   next {
19     case
20     (step == 0) : {
21       if ((power & 1bv4) == 1bv4) {
22         result' = result * powers[step];
23         step' = step + 1;
24       }
25       else {
26         result' = result * 1;
27         step' = step + 1;
28       }
29     }
30
31     (step == 1) : {
32       if ((power & 2bv4) == 1bv4) {
33         result' = result * powers[step];
34         step' = step + 1;
35       }
36       else {
37         result' = result * 1;
38         step' = step + 1;
39       }
40     }
41
42     (step == 2) : {
43       if ((power & 4bv4) == 1bv4) {
44         result' = result * powers[step];
45         step' = step + 1;
46       }
47       else {
48         result' = result * 1;
49         step' = step + 1;
50       }
51     }
52
53     (step == 3) : {
54       if ((power & 8bv4) == 1bv4) {
55         result' = result * powers[step];
56       }
57       else {
58         result' = result * 1;
59       }
60     }
61   esac
62 }
63
64 hyperinvariant[2] eq_step : true ==> step.1 == step.2;
65 control {
66   v = sygus_refine(step <= 3);
67   check;
68   print_results;
69 }
70 }

```

Example 4.4: A UCLID5 model that demonstrates similar updates in all branches

4.3.3 Example 3: Similar Updates in All Branches

Example 3 Analysis

The program above implements fast exponentiation. The *powers* array is populated with the powers of the base by repeated squaring. From the second state onwards, starting from the least significant bit of the exponent, each bit is examined and if it is set, then the appropriate power is multiplied with the *result* variable. If the bit is unset then the *result* variable is multiplied with one.

Similar to the previous example, the property to be proved here is that the number of steps taken is the same in both the traces of the program even when the base and the exponent is different. This property captures the notion that an adversary learns nothing about the base or the exponent by observing the number of steps taken by the program to finish execution. The taint property which has to be verified is $step_t = true$.

The *step* variable is incremented only when the current value of *step* is lesser than or equal to 3 and does not depend on the *power* variable. Hence the refinement which we supply to our algorithm is $step \leq 3$ which causes the property to be discharged.

Chapter 5

Conclusion and Future Work

In this thesis we implemented and evaluated the algorithms used by Yang *et al.* to reduce the cost of lazy self composition. We saw how taints are implemented as an augmentation to the program transition system and how it acts as a sound over-approximation to equalities in both the copies of the self composed system. We also saw how the refinement strategy that was used by Yang *et al.* does not scale well in certain examples. As an improvement to the refinement strategy, we let the user specify a set of refinement predicates which is used to refine individual variables once they are proved to be sound for that particular variable. We show that our refinement strategy works where the previous strategy failed and has negligible overhead in proving the soundness of the refinements. In the following sections we detail three avenues for improvement which are worth exploring.

5.1 Lambda Expressions for Array Taints

The present scheme of refinement of variables does not allow for the refinement of the taint location array in the case of array variables. The refinement can only be applied on the address taint variable of an array variable. This poses a problem as a taint array initialized to false will remain so till the end of the verification procedure. In most cases, not all array locations are untainted during the execution of the program. Associating each array taint variable with a lambda expression along with support for inputting lambda expressions as refinements can help providing better precision for array taints. For example, consider the assignment expression $x = arr[y]$, we propose that this should be reimplemented as $x = (lambda(mem, idx) : mem[idx])(mem, y)$. Now suppose that the refinement predicate is $lambda(mem, idx) : ite((idx < 10) \wedge (idx \geq 0), true, mem[idx])$. Then the rewritten expression for the variable x would be: $x = lambda(mem, idx) : ite((idx < 10) \wedge (idx \geq 0), true, mem[idx])(mem, y)$.

5.2 Converting Hyperaxioms to Array Taint Refinements

The present implementation of the taint transition system does not take the hyperaxioms into account when initializing the taint array for an array variable. Either a taint array is initialized to be entirely false or entirely true. Creating a syntax for hyperaxioms involving arrays could help in initializing the taint array to more accurate values based on lambdas. For example a hyperaxiom of the form $\forall \pi_1 \pi_2 : \forall (a : \text{addr}_t)(a \geq 0) \wedge (a < 100) \implies \pi_1.\text{mem}[a] = \pi_2.\text{mem}[a]$ can be parsed and used to initialize the location taint array to an application of the following lambda: $x = \text{lambda}(\text{mem}, \text{idx}) : \text{ite}((\text{idx} < 100) \wedge (\text{idx} \geq 0), \text{true}, \text{false})$.

5.3 Investigating Z3 Inductive Trace

Currently, we are not able to extract the values of all the variables from the counter example provided by the Z3 horn solver as it is giving us only those variables on which the counter example depends on. Getting all the variables from the horn solver is crucial for implementing the full refinement strategy for all the variables.

Bibliography

- [1] N. Een, A. Mishchenko, and R. Brayton, “Efficient implementation of property directed reachability,” in *Formal Methods in Computer-Aided Design (FMCAD 2011)*, 2011.
- [2] A. R. Bradley, “Sat-based model checking without unrolling,” in *12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2011)*, 2011.
- [3] K. Hoder and N. Bjørner, “Generalized property directed reachability,” in *SAT*, 2012.
- [4] A. Komuravelli, A. Gurfinkel, and S. Chaki, “Smt-based model checking for recursive programs,” in *Computer Aided Verification* (A. Biere and R. Bloem, eds.), (Cham), pp. 17–34, Springer International Publishing, 2014.
- [5] N. Bjørner and A. Gurfinkel, “Property directed polyhedral abstraction,” in *Proceedings of the 16th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume 8931*, VMCAI 2015, (New York, NY, USA), pp. 263–281, Springer-Verlag New York, Inc., 2015.
- [6] A. Komuravelli, N. Bjørner, A. Gurfinkel, and K. L. McMillan, “Compositional verification of procedural programs using horn clauses over integers and arrays,” in *Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design*, FMCAD ’15, (Austin, TX), pp. 89–96, FMCAD Inc, 2015.
- [7] N. Bjørner, A. Gurfinkel, K. McMillan, and A. Rybalchenko, *Horn Clause Solvers for Program Verification*, pp. 24–51. Cham: Springer International Publishing, 2015.
- [8] D. Hedin and A. Sabelfeld, “A perspective on information-flow control,” 2011.
- [9] J. A. Goguen and J. Meseguer, “Security policies and security models,” *1982 IEEE Symposium on Security and Privacy*, pp. 11–11, 1982.
- [10] J. A. Goguen and J. Meseguer, “Unwinding and inference control,” *1984 IEEE Symposium on Security and Privacy*, pp. 75–75, 1984.
- [11] A. W. Roscoe, “Csp and determinism in security modelling,” in *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, SP ’95, (Washington, DC, USA), pp. 114–, IEEE Computer Society, 1995.
- [12] J. McLean, “Proving noninterference and functional correctness using traces,” *Journal of Computer Security*, vol. 1, pp. 37–58, 1992.
- [13] G. Barthe, P. D’Argenio, and T. Rezk, “Secure information flow by self-composition,” *Mathematical Structures in Computer Science*, vol. 21, pp. 1207–1252, 12 2011.
- [14] T. Terauchi and A. Aiken, “Secure information flow as a safety problem,” in *Proceedings of the 12th International Conference on Static Analysis*, SAS’05, (Berlin, Heidelberg), pp. 352–367, Springer-Verlag, 2005.
- [15] G. Barthe, J. M. Crespo, and C. Kunz, “Relational verification using product programs,” in *FM 2011: Formal Methods* (M. Butler and W. Schulte, eds.), (Berlin, Heidelberg), pp. 200–214, Springer Berlin Heidelberg, 2011.
- [16] M. Sousa and I. Dillig, “Cartesian hoare logic for verifying k-safety properties,” *ACM SIGPLAN Notices*, vol. 51, pp. 57–69, 06 2016.

- [17] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti, “Solving horn clauses on inductive data types without induction,” *Theory and Practice of Logic Programming*, vol. 18, 04 2018.
- [18] D. Mordvinov and G. Fedyukovich, “Synchronizing constrained horn clauses,” in *LPAR*, 2017.
- [19] A. Youssef and A. F. Shosha, “Quantitative dynamic taint analysis of privacy leakage in android arabic apps,” in *Proceedings of the 12th International Conference on Availability, Reliability and Security*, ARES ’17, (New York, NY, USA), pp. 58:1–58:9, ACM, 2017.
- [20] M. Gyung Kang, S. McCamant, P. Poosankam, and D. Song, “Dta++: Dynamic taint analysis with targeted control-flow propagation,” 01 2011.
- [21] E. J. Schwartz, T. Avgerinos, and D. Brumley, “All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask),” in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP ’10, (Washington, DC, USA), pp. 317–331, IEEE Computer Society, 2010.
- [22] W. Yang, Y. Vitzel, P. Subramanyan, A. Gupta, and S. Malik, “Lazy self-composition for security verification,” in *Computer Aided Verification* (H. Chockler and G. Weissenbacher, eds.), (Cham), pp. 136–156, Springer International Publishing, 2018.
- [23] B. Alpern and F. B. Schneider, “Defining liveness,” *Information Processing Letters*, vol. 21, no. 4, pp. 181 – 185, 1985.
- [24] M. R. Clarkson and F. B. Schneider, “Hyperproperties,” *J. Comput. Secur.*, vol. 18, pp. 1157–1210, Sept. 2010.
- [25] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, “Bounded model checking using satisfiability solving,” *Formal Methods in System Design*, vol. 19, pp. 7–34, Jul 2001.
- [26] R. Milner, “Calculi for synchrony and asynchrony,” *Theoretical Computer Science*, vol. 25, no. 3, pp. 267 – 310, 1983.
- [27] D. Park, “Concurrency and automata on infinite sequences,” in *Theoretical Computer Science* (P. Deussen, ed.), (Berlin, Heidelberg), pp. 167–183, Springer Berlin Heidelberg, 1981.