PSec: A Programming Language for Secure Distributed Computing

by

Shivendra Kushwah

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Sanjit A. Seshia, Chair
Professor Ralcua Ada Popa

Spring 2020

PSec: A Programming Language for Secure Distributed Computing

PSec: A Programming Language for Distributed Secure Computing

by Shivendra Kushwah

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Sanjit A. Seshia
Research Advisor

5/26/2020

(Date)

* * * * * * *

Professor Raluca Ada Popa
Second Reader

May 13, 2020

(Date)

Abstract

PSec: A Programming Language for Secure Distributed Computing

by

Shivendra Kushwah

Master of Science in Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Sanjit A. Seshia, Chair

We introduce PSec, a programming language for secure distributed computing. PSec is a high-level language designed for Trusted Execution Environments such as Intel SGX to enable programmers to create distributed systems consisting of both trusted and untrusted machines. We design and abstract away various secure message sending and dynamic machine creation protocols. We also create a trust designation system to establish trust between different machines in the system. By combining techniques in information flow control and cryptography, we are able to prevent programmers from inadvertently leaking sensitive data and allow them to send data securely from one machine to another.

Dedicated to Mom, Dad, Arnav, and all of my family and friends for providing me tremendous amounts of love and support throughout my entire journey, and without whom I would not be here today.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank my mentors, Ankush Desai and Pramod Subramanyan, and my advisor, Professor Sanjit Seshia, for providing incredible guidance throughout the course of this project and throughout the entire Master's program. This project was designed and refined over the course of many, many discussions with these individuals and would not be possible without them. Converting P to be compatible with Intel SGX was no small feat, and was part of previous work undertaken by Ankush and Pramod. This proved to be invaluable and enabled me to have a running start. I'm thankful for the Learn&Verify group for providing helpful feedback on my project presentations as well as valuable advice and lifelong friendships. I'm grateful to Professor Alvin Cheung for helping me formulate PSec's type system as well as providing feedback regarding system performance. Finally, I want to thank Professor Raluca Popa for providing valuable feedback on my thesis and for being my second reader.

Chapter 1

Introduction

1.1 Motivation

Distributed systems are essential to modern computing. The ability to break a computationally complex task into multiple parts and divide the workload between multiple computers leads to more efficient computing as well as more modular code. Since this design is powerful and used widely in the real world, many domain-specific programming languages have been developed that enable programmers to more readily develop these kinds of systems.

Unfortunately, ensuring the resulting distributed systems are secure is a different problem entirely. Programming secure distributed systems is still an active area of research because writing secure code is innately a hard problem. While enabling programmers to write performant code, lower level programming languages such as C/C++ are susceptible to attacks such as buffer overflows. Even if using higher level languages, programmers generally have to understand the basics of cryptography to properly initialize and effectively use cryptographic code and libraries to perform sensitive operations. This can prove troublesome, as in the case of the Sony PS3 Private Signing Key leak [22] where programmers actually used the same random number for each ECDSA signature. This made cracking the private signing key much easier and enabled hackers to trick Sony PS3 machines into running malicious code that appeared to be signed by Sony. In the distributed setting, using proper cryptography is especially critical to transmit messages that may contain sensitive data. However, the problem doesn't just end there. Even if the message is sent securely, there are no guarantees that the receiving machine will not accidentally leak the underlying sensitive data to untrusted machines after decryption.

1.2 Related Work

In the past, programming language research in secure distributed systems has generally focused on preventing sensitive data from being inadvertently leaked. Works such as Jif/split [38], SIF [3], Swift [4], and Fabric [18] utilize language-based information flow control

to achieve desired security properties in the distributed setting. They accomplish this by enforcing confidentiality and integrity policies on data passed through the system.

Jif/split [38] focuses on protecting sensitive data in the context of distributed systems where entities trust different distributed host machines to varying degrees. It accomplishes this by secure program partitioning, which involves dividing a main program and a description of entity-host trust relationships into various sub-programs deployed on the distributed hosts. Through this process, each entity can be assured that sensitive data is only sent to machines they trust and not leaked to untrusted machines. It is important to note that a declaration of trust of a particular machine means that the entity trusts that machine's hardware, operating system, and Jif/split's run-time support. The adversary model is that, for a given entity, attackers may subvert any untrusted host machines. However, this should not compromise or leak that entity's confidential data. Swift [4] further explores program partitioning with a particular focus in the web application space. The partitioning procedure has additional constraints and optimizations because although client machine code is regarded as less trusted than server code, it generally yields better performance to end-users. In comparison to Jif/split, the authors state that Swift supports a richer programming language with better information flow control properties. Servlet Information Flow (SIF) [3] is another work in the web application space that focuses on language-based information flow enforcement. Rather than focusing on program partitioning, SIF is a framework that enables programmers to directly write secure server-side code. The adversary model is that attackers are assumed to be able to compromise client machines and that server-side code is potentially buggy but benign. Therefore, the confidentiality and integrity data policies need to be enforced server-side regardless of the actions of client machines or inadvertent programmer mistakes. Finally, Fabric [18] is a newer paper that continues to leverage language-based security for secure distributed computation and storage. It additionally allows new distributed hosts to join the system and supports consistent, distributed computing over shared, persistent data. Similar to previous approaches, Fabric users are able to express varying level of trust in different Fabric nodes and security guarantees are lost if adversaries compromise trusted nodes.

A main assumption behind these approaches is the correctness of the trust designation system. These approaches enable entities to specify which distributed nodes they trust to correctly run and securely execute code. However, if an entity trusts a potentially corrupted node, they lose any security guarantees provided by the system. Additionally, this trust definition often requires trust in the hardware and operating system of the distributed node. Although this may be a fair assumption in most cases, this attack surface is by no means small and stronger adversaries may be able to exploit bugs and security vulnerabilities to compromise even trusted systems.

More recently, research has leveraged hardware enforced trusted computing (such as Trusted Platform Modules [35] and Trusted Execution Environments/hardware enclaves [33]) to reduce previous trust assumptions and provide guarantees in the face of more privileged adversaries. Some recent work has focused on formally verifying security properties of programs built using trusted computing technologies in the presence of privileged adversaries.

Moat [31] is a tool that verifies confidentiality properties of hardware enclave-based program binaries (specifically Intel SGX-based [17]) in the face of an adversary that can arbitrarily modify external code. It requires developers to have knowledge of lower-level enclave specifics in order to effectively use the tool. Sinha et al. [28] provide a methodology for developing Intel SGX applications that requires placing the entire application within trusted enclave memory and having it communicate with the outside world through a narrow, trusted interface. They formalize these restrictions as a confidentiality property coined *Information Release Confinement* and create a verification tool for this property. Subramanyan et al. [34] formalize an idealized enclave platform into a trusted abstract platform (TAP) model. This work serves to create a verification methodology for the TAP model and formalizes the definition of secure remote execution for enclaves.

A different stream of work has focused on using trusted computing technologies to enable programmers to more readily create secure applications through a language-based approach. Fournet and Planul [11] develop a compiler for secure distributed information flows by leveraging Trusted Platform Modules, secure boot, and remote attestation, and combining it with information flow control techniques. They use cryptographic methods to emulate secure memory. In contrast, IMP_e [13] is an information flow control calculus that uses Intel SGX enclaves to directly provide secure memory. However, IMP_e is not tailored to the distributed setting. Patrignani et al. [21] develop a secure compilation scheme that compiles high level language code to protected module architectures (PMAs) that essentially provide memory isolation similar to enclaves. However, once again, this work is not tailored to the distributed setting and doesn't consider applications that may require multiple enclaves. One of the more recent examples, EActors [24], develops an actor-based programming language that leverages Intel SGX to enable trusted communication between multiple entities. Programmers are able to create secure distributed systems by specifying trusted/untrusted actors and having them perform computation and send messages to each other. The adversary model assumes a privileged adversary who has physical access to hardware in the system and control over the entire software stack (including the operating system kernel). The hardware trusted execution support (Intel SGX) is assumed to be correct and side-channel attacks are discounted. Trusted actors are implemented using Intel SGX enclaves, and the authors create a secure instant messaging service and a secure multiparty computation service to demonstrate the capabilities of their language. However, it is important to note that this work doesn't provide any protections against inadvertent data leakage to untrusted parties. The key focus of this line of research is the reliance on hardware for trusted computation and attestation for establishing trust between secure environments across different machines.

For our work, we seek to combine these lines of research and create a high-level programming language that provides information flow guarantees while leveraging hardware enclaves for trusted computation in the secure distributed systems setting. Our adversary model includes a powerful adversary that has privileged access to host machines in our system (similar to EActors) and is able to snoop on network requests. For the purposes of this thesis, side channel attacks are out of scope. We detail our entire adversary model in Section 3.3.1.

1.3 Overview

In order to solve the aforementioned problems, we propose PSec, a language that abstracts security details and implementations away from programmers and enables them to easily and effectively write secure distributed systems code. We will be building on top of P [9], an existing actor-based programming language for creating distributed systems. By leveraging Trusted Execution Environments such as Intel SGX [17] and combining them with secure machine creation and message exchange protocols, we can enable programmers to design secure distributed systems that provide specific guarantees. Programmers are able to mark certain messages and variables as `secure` to prevent them from being inadvertently leaked and can use our `send` construct to securely send sensitive messages from one machine to another.

We structure the remainder of this thesis as follows: Chapter 2 will describe relevant background information on both P and Intel SGX; Chapter 3 will provide the PSec language design, formalisms, implementation, and an evaluation of our system; and Chapter 4 will conclude and discuss future work.

1.4 Contributions

The core contributions of this thesis are:

1. We design a programming language for creating secure distributed systems and an information flow control type system to prevent secure data from being leaked to untrusted systems or maliciously corrupted. We formalize the type system as well as the operational semantics of our programming language.

2. We design security protocols and develop a secure runtime to enable the secure creation of machines and the ability to send messages securely. We also create a trust designation system in order to establish trust between machines.

3. We present initial performance metrics on an implementation of our language and system (located at `https://github.com/ShivKushwah/PSec`). We create a One Time Passcode and a Secure Voting example to demonstrate language expressivity. Finally, we give formal proofs for the confidentiality and integrity properties provided by our programming language.

Chapter 2

Background

2.1 The P Programming Language

P is a programming language co-developed by Microsoft and UC Berkeley to address the issues with asynchronous programming [10]. Due to the nature of asynchronous programs, problems such as race conditions and Heisenbugs arise and are often very difficult to spot and debug. By using P, users are able to model and specify protocols in this space. P has been used in the development of the USB 3.0 driver inside the Windows Phone and Windows 8.1. It is also being used to develop cloud infrastructure inside Azure and is currently run on hundreds of millions of devices all over the world.

The programming model is actor-based and consists of concurrently executing state machines communicating via events with payloads. Programmers can easily write state machines in P and specify transitions based on the results of different events. The P compiler provides automated testing for concurrency-related race conditions and generates executable C code for running the protocols. This allows P to meet the difference between a high-level model and the low-level implementation, and lends itself to be more readily accepted by programmers than traditional formal modeling. The P memory management system is based on linear typing and unique pointers (prevents race conditions associated with concurrent access of data) which enforces safe memory practices. It is important to note that P does not readily enable different state machines to be deployed on different physical hosts. Rather, all state machines run on 1 physical host and P ensures that the various state machines correctly model distributed behavior. Overall, P enables users to model concurrency, specify safety and liveness properties and has the ability to check that the program satisfies its specification by using systematic search.

In terms of an actual P program, PingPong.p (Figure 2.1) touches on many major aspects of P. In this program, we have a Server P machine and a Client P machine. The Client machine creates a new instance of the Server machine and transitions to the SendPing state. In this state, the Client sends the Server a Ping event (containing a reference to itself) and goes to the WaitPong state. Upon receiving the Ping event, the Server transitions to the

```
1  event PING: machine;
2  event PONG;
3
4  machine Client
5  {
6      var server: machine;
7
8      start state Init {
9          entry {
10           server = new Server();
11           goto SendPing;
12         }
13     }
14
15     state SendPing {
16         entry {
17             send server, PING, this;
18             goto WaitPong;
19         }
20     }
21
22     state WaitPong {
23         on PONG goto SendPing;
24     }
25 }
```

(a) Client

```
1  machine Server
2  {
3      start state WaitPing {
4          on PING goto SendPong;
5      }
6
7      state SendPong {
8          entry (payload: machine) {
9              send payload, PONG;
10             goto WaitPing;
11         }
12     }
13 }
```

(b) Server

Figure 2.1: P Client Server Example

SendPong state, and sends the Client machine a Pong event. The Client receives this Pong event and transitions back to the SendPing state, where it sends the Server a Ping message again. These set of states cycle as the Client and Server continually exchange Ping and Pong events in a never ending loop. This example highlights the simplicity and expressivity of the P syntax.

2.2 Intel SGX

Intel Software Guard Extensions (SGX) allow for application developers to create secure applications that may reside in untrusted host machines [17]. Intel SGX takes advantage of a specialized instruction set in newer Intel CPUs that allows developers to create enclaves, or secure areas of execution in memory. The content of these enclaves is protected from all other processes (including higher-level operating system processes) and is only decrypted on the fly by the actual CPU to run the commands. Intel also provides a way to perform attestation of these enclaves to verify their identity and that they have not been tampered with. By using all of these features, we can leverage Intel SGX to perform secure computation.

Enclave applications have a unique architecture. They are divided into an untrusted portion and a trusted portion. The trusted portion runs inside the enclave and is responsible for executing secure code. The untrusted host serves as a wrapper for the enclave to make external calls because many syscalls aren't allowed/implemented within the enclave. Therefore, the enclave must rely on the untrusted host to make these calls and return the correct results back. However, enclave programmers must not assume that any code running outside the enclave will execute correctly and must design their applications accordingly.

More specifically, as in Figure 2.2, the enclave can make an `ocall` to pipe requests (such as syscalls) to outside the enclave. In a similar fashion, the outside world can make `ecall`s to execute trusted code. A general enclave application flow might look as follows.

1. A network request comes in and is processed by the untrusted host
2. The untrusted host makes an `ecall` to the enclave to forward the contents of the request
3. The enclave runs its secure application code and makes appropriate `ocall`s as needed
4. The enclave returns and the untrusted host sends back the results of the computation over the network

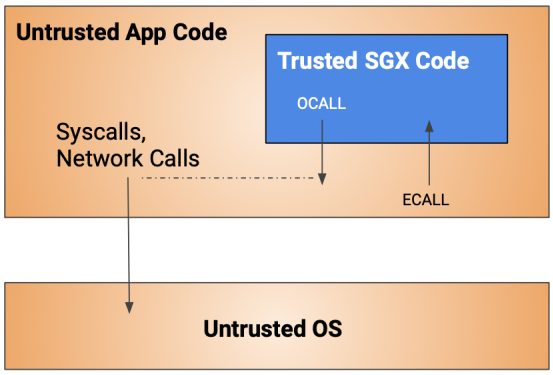

Figure 2.2: Enclave Architecture

An important consideration is that enclaves are inherently vulnerable to a Denial of Service attack by the untrusted host. For example, in the flow mentioned above, the untrusted host can choose to not forward any of the network requests to the enclave. Since the enclave can't open ports and process networks requests directly, the enclave will be prevented from doing anything useful in this case. However, it is important to note that the security of the system isn't impacted from this type of attack.

Unfortunately, writing Intel SGX code is a very non trivial task. For example, a bare bones version of attestation using the Intel SGX SDK requires 1000+ lines of C++ code.

In addition to this, code executed within the enclave has only access to a restricted set of LibC++, which makes converting normal programs/libraries to "enclave" programs/libraries an arduous task. Further wrappers on top of the Intel SGX SDK do exist (such as Microsoft Open Enclave SDK [19]), but these also require extensive knowledge of basic enclave primitives. In addition to these limitations, recent research has demonstrated that enclaves are vulnerable to a wide variety of side channel attacks that may yield unwanted leakage of sensitive knowledge. These side channels include page-fault attacks [36, 30], cache-timing attacks [14, 1, 26], and speculative execution attacks [2]. Correspondingly, there has been a lot of recent work in providing defenses [30, 29, 12, 15, 25, 20] with varying degrees of success.

Despite these challenges, enclaves are being increasingly used in a wide range of applications due to the security guarantees they provide. They are often used in cloud applications where we need trusted code to run in the cloud but we can't trust the cloud provider to not snoop on our data. In summary, enclaves provide confidentiality, integrity, and security for the application running within. In a modern world that is becoming increasingly security and privacy driven, this is an extremely valuable technology.

Chapter 3

PSec

While designing PSec, we wanted to be able to create language constructs to enable programmers to implement real-world systems. Secure distributed systems, which consist of trusted machines, may have to interact with external untrusted machines for input or to execute non-sensitive tasks. As a result, we want to incorporate the concepts of both trusted and untrusted state machines in PSec. In the following sections, we will describe the various additions to the P language to support this new programming paradigm.

3.1 Language Design

In PSec, programmers can choose between 2 types of state machines. We have Secure State Machines (SSMs), which the programmer can denote by defining the state machine code within an enclosing `secure_machine` block. We also have Untrusted State Machines (USMs), denoted using an enclosing `machine` block as in regular P. SSMs are trusted to run sensitive code and handle secret data while USMs run in the untrusted world. As in P, State Machines can receive and send events with payloads and take different actions based on events received. PSec is built as an extension to the P language, and we outline notable differences in the following sections. One important note is that all commands are assumed to be atomic and state machines do not execute the next command until the current command terminates.

3.1.1 Machine Creation

Rationale

Since PSec machines are created dynamically, we need a way to create state machines on the fly in a secure manner. In addition to this, we need a way to designate trust so that state machines can be confident they are sending sensitive data to and receiving correct data from trusted entities. Our initial approach involved having SSMs trust all other SSMs running valid PSec code. However, we determined this designation of trust to be too broad because

adversaries could spin up their own SSMs (running valid PSec code) and convince our SSMs to send them sensitive data. Although this is not a problem in itself (our implementation of SSMs prevent them from leaking data to their untrusted host machines), the adversaries would be able to more readily perform denial of service attacks on these SSMs and prevent them from doing any useful work since they reside on systems local to the adversary. This would result in our machines waiting on responses that will never come.

A more sensible designation of trust relies on having trust chains centered around machine creation. In this scheme, the primary driver of trust is that SSMs trust any SSMs they have created and these child SSMs trust their parent SSM. The secondary driver of trust is through trust designation. Child SSMs additionally trust SSMs deemed by their parent to be trustworthy (as well as SSMs deemed by those SSMs and so on). The last point is an important distinction because this enables trust to flow in a chain rather than a tree, which is important in enabling us to express real-world applications. Initial trust is bootstrapped by having the first SSM in the system be created by a trusted host machine that performs the necessary initial setup. If this trusted host machine is corrupted, the worst case scenario is again a denial of service (the host can refuse to create this first SSM or can relay incorrect information, preventing anyone from communicating with it). However, we argue that this point of failure is smaller than the point of failure discussed in the previous approach. With this approach in mind, we design our machine creation as follows.

Design

Machine creation falls in the following cases:

1. `Trusted Create`: SSM1 creates SSM2
2. `Untrusted Create`: SSM1 creates USM1 `or` USM1 creates USM2 `or` USM1 creates SSM1

In both of these cases, machine creation is denoted using the `new` command as in:

```
machineHandle = new StateMachine();
```

Invoking the `new` command actually utilizes the PSec Runtime to send a state machine creation request across the network to one of the distributed host machines in our system. Upon successful machine creation, a machine handle is returned that can be used to send events to the newly created machine.

In the `Untrusted Create` case, the parent machine receives the handle of the newly created child machine so that it knows where to send future messages. This is reflected by the type of the returned handle, which is `machine_handle`;

```
machine ParentUSM {
    var machineHandle : machine_handle;
    ...
    entry {
        machineHandle = new ChildSSM();
        ...
    }
}
```

In the `Trusted Create` case, the parent SSM receives the handle of the newly created child SSM as well as the `capability` to send trusted data to it. This type of the returned handle is `secure_machine_handle` which reflects the necessary `capability` of the child machine. The parent SSM can share this `capability` by simply sending this `secure_machine_handle` to other trusted SSMs.

```
secure_machine ParentSSM {
    var machineHandle : secure_machine_handle;
    ...
    entry {
        machineHandle = new ChildSSM();
        ...
    }
}
```

Once created, a PSec state machine can access its own machine handle by using the `this` keyword. A USM can access its own `machine_handle`, and a SSM can additionally access its `secure_machine_handle`.

3.1.2 Message Sending

Rationale

We need to define multiple message sending types in order to properly capture the various interactions between USMs and SSMs. First of all, SSMs should be able to send trusted messages containing sensitive data to other SSMs they trust as well as receive data they know not to be corrupt/malicious from these entities. In addition to this, they should also be able to send and receive messages from USMs as well as untrusted SSMs (one simple use case being to accept input from user systems and sending back computed output). SSMs should be able to differentiate between these trusted and untrusted interactions so that they can respond accordingly. USMs, in general, should be able to send messages to both USMs and SSMs.

After looking at all the possible scenarios, we decided to define 2 types of message sending: Trusted Message Sending and Untrusted Message Sending. Trusted Messaging is used to exchange messages whose contents can be trusted between SSMs while Untrusted Messaging is used for all other use cases. We are assuming that all SSMs and USMs are distributed, and any messages they want to send to each other need to be sent across the network. Since P innately requires messages to be encapsulated by Event objects, we also define two types

of events: Trusted Events and Untrusted Events. On the sending side, programmers can use Trusted Message Sending to send messages encapsulated in Trusted Events and Untrusted Message Sending for messages in Untrusted Events. On the receiving side, programmers can specify different behavior for state machines when they receive a Trusted Event versus an Untrusted Event.

Trusted Sending

Trusted Sending is inferred by the compiler based on the specified handle of the receiving machine (needs to be `secure_machine_handle`). This type of sending can only occur between 2 SSMs, and the machine that is sending must have the `capability` of the receiving SSM to do so (and hence must possess the `secure_machine_handle`). These messages are sent over a secure channel across the network and are resistant to Man-in-the-Middle attacks. The PSec Type Checker enforces that Trusted Events can only be sent using Trusted Message Sending so that their contents are never leaked to the untrusted world. On the receiving machine's side, the PSec Runtime ensures that Trusted Events can only be received from other trusted SSMs through secure channels. If an adversary sends a Trusted Event without proving they possess the necessary `capability`, the receiving machine's PSec Runtime doesn't forward the event to the receiving SSM. The command template is the following:

```
send receivingSSMSecureMachineHandle, TrustedEvent, securePayload;
```

Untrusted Sending

Untrusted Sending is inferred by the compiler if the type of the receiving machine's handle is `machine_handle`. Both SSMs and USMs can use this command to send Untrusted Events with payloads to other machines without needing any sort of `capability`. These messages are sent securely as well but since these messages may originate from the untrusted world, the content of these messages are untrusted. The command template is:

```
send receivingMachineHandle, UntrustedEvent, payload;
```

3.2 Language Formalisms

In this section, we formalize the operational semantics as well as the type system of PSec. We will use these formalisms in later sections in order to be able to construct proofs regarding our language. In terms of the operational semantics, we build on top of the operational semantics of the P Programming language, denoted in [8] and [9].

3.2.1 Notation

1. Let $\mathcal{E}$ represent the set of names of all the Events

- Let $\mathcal{E}_\mathcal{T}$ represent the set of names of all the Trusted Events, which is a subset of $\mathcal{E}$
- Let $\mathcal{E}_\mathcal{U}$ represent the set of names of all the Untrusted Events, which is a subset of $\mathcal{E}$ and disjoint to $\mathcal{E}_\mathcal{T}$

2. Let $\mathcal{M}$ represent the set of names of all the state machines
   - Let $\mathcal{M}_\mathcal{T}$ represent the set of names of all the SSMs, which is a subset of $\mathcal{M}$
   - Let $\mathcal{M}_\mathcal{U}$ represent the set of names of all the USMs, which is a subset of $\mathcal{M}$ and disjoint to $\mathcal{M}_\mathcal{T}$
3. Let $\mathcal{Z}$ be the set of all machine identifiers that uniquely identify each state machine in our system, which is $\mathcal{M} \times \mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers. This is important because there can be multiple state machines created of the same type (with the same name), and we need a way to distinguish between them
4. Let $\mathcal{H}$ be the set of all machine handles, which is $\mathcal{M} \times \mathbb{N} \times \mathbb{X}$, where $\mathbb{N}$ is the set of natural numbers and $\mathbb{X} \in \{0, 1\}$. This is important because handles are used to send events to state machines and we need a way to distinguish between handles that contain Trusted Event capabilities and those that do not
   - Let $\mathcal{H}_\mathcal{T}$ represent the set of trusted machine handles that indicate that the machine that possesses the trusted handle has the `capability` to send Trusted Events to the corresponding SSM. This is represented as $\mathcal{M} \times \mathbb{N} \times 1$, where $\mathbb{N}$ is the set of natural numbers
   - Let $\mathcal{H}_\mathcal{U}$ represent the set of untrusted machine handles that indicate that the machine that possesses the handle can send Untrusted Events to the handle's corresponding machine. This is represented as $\mathcal{M} \times \mathbb{N} \times 0$, where $\mathbb{N}$ is the set of natural numbers
5. Let $\mathcal{S}$ represent the set of all possible local state for a state machine. This local state contains everything needed for execution of the machine, including data structures and control stack
6. - Let $\mathcal{V}_\mathcal{T}$ represents the set of all possible payloads that may be encapsulated in a PSec Trusted Event. These payloads must be derived from trusted variables stored in local state of machines
   - Let $\mathcal{V}_\mathcal{U}$ represents the set of all possible payloads that may be encapsulated in a PSec Untrusted Event. These payloads must be derived from untrusted variables stored in local state of machines
   - Let $\mathcal{V}$ represents the union of $\mathcal{V}_\mathcal{T}$ and $\mathcal{V}_\mathcal{U}$

7. Let $\mathcal{B}$ represent the set of all possible values for the input buffer for state machines. Upon receiving events with payloads, state machines enqueue these messages into their input buffer. This input buffer is a sequence of $(e, v) \in \mathcal{E} \times \mathcal{V}$ pairs

3.2.2 Operational Semantics

Definitions

We define various transition relations and functions below:

1. We define the $Local \subseteq \mathcal{S} \times \mathcal{H} \times \mathcal{S} \times \mathcal{H}$ transition relation that represents the various internal transitions of a state machine. $(s, id, s', id') \in Local(m)$ means that the machine m transitions from local state s to s' and can model the movement of handles between these local states

2. We define the $Enq \subseteq \mathcal{S} \times \mathcal{H} \times \mathcal{E} \times \mathcal{V} \times \mathcal{S}$ transition relation that represents message sending from one machine to another. $(s, id, e, v, s') \in Enq(m_s)$ means that the sending machine m_s changes local state from s to s' and event e with payload v is sent to the receiving machine with handle id

3. We define the $Rem \subseteq \mathcal{S} \times \mathcal{B} \times \mathbb{N} \times \mathcal{S}$ transition relation that represents a state machine dequeuing and handling an event from its input buffer. $(s, b, n, s') \in Rem(m)$ means that the machine m dequeues the nth event from its input buffer b and changes local state from s to s'

4. We define the $New \subseteq \mathcal{S} \times \mathcal{M} \times \mathcal{S}$ transition relation that represents new state machine creation. $(s, m_c, s') \in New(m_p)$ means that the parent machine m_p moves local state from s to s' after creating a child machine m_c

5. - We define a function $uids$ such that $uids(s)$ is the set of all untrusted machine handles embedded in state s and $uids(v)$ is the set of all untrusted machine handles embedded in value v
   - We define a function $tids$ such that $tids(s)$ is the set of all trusted machine handles embedded in state s and $tids(v)$ is the set of all trusted machine handles embedded in value v
   - We define ids to be a function that is the union of $uids$ and $tids$

6. - We define a function $uvals$ that returns a map of non-sensitive variables to their values in state s
   - We define a function $tvals$ that returns a map of secret variables to their values in state s
   - We define $vals$ as a function that returns a map of all variables to their values in state s

7. We define a helper function $containsAll(A, B)$ that returns true if the map A contains all of the keys and values that are present in map B

8. We define a function $IFA(s, s')$ that returns true if s to s' represents a valid transition for a state machine after our information flow analysis type checking rules (depicted in later sections) have successfully terminated. We have 3 possible valid transitions: state remains the same, or either the untrusted state or the trusted state increases

   - $IFA(s, s')$ returns true if
     $(uvals(s) = uvals(s') \wedge tvals(s) = tvals(s'))$
     or $(uvals(s) = uvals(s') \wedge containsAll(tvals(s'),\ tvals(s)))$
     or $(tvals(s) = tvals(s') \wedge containsAll(uvals(s'),\ uvals(s)))$

Machine Handles Cannot be Created "Out of Thin Air"

Recall that an SSM can only send a Trusted Event to another SSM if it possesses its trusted handle (`secure_machine_handle`). The `capability` to send these Trusted Events to a particular SSM can be modified as these trusted handles can be transmitted from one SSM to another. Untrusted Events can be sent to any state machines as long as the sending machine has the receiving machine's untrusted handle (`machine_handle`). We need to formalize the concept that these machine handles cannot be created "out of thin air" [8] and must be present in the local state of a state machine before they can be used. State machines can get access to these handles by either creating a new machine (New) or by receiving the handle from another state machine and dequeuing the event (Rem). We formalize this as follows:

For all $m \in \mathcal{M},\ id, id' \in \mathcal{H},\ s, s' \in \mathcal{S},\ e \in \mathcal{E},\ v \in \mathcal{V},\ n \in \mathbb{N},\ b \in \mathcal{B}$

1. $(s, id, s', id') \in Local(m) \Rightarrow ids(s') \cup id' \subseteq ids(s) \cup \{id\}$
2. $(s, b, n, s') \in Rem(m) \Rightarrow ids(s') \subseteq ids(s) \cup \{ids(v)\ j\ \exists e.b[n] = (e, v)\}$
3. $(s, id, e, v, s') \in Enq(m) \Rightarrow ids(v) \cup ids(s') \subseteq ids(s)$
4. $(s, m', s') \in New(m) \Rightarrow ids(s') \subseteq ids(s)$

Propagation of Secret State

In addition to this, we need to make sure that local state flows correctly propagates sensitive labels and correctly accounts for transitions between secret and non-sensitive state. We leverage our information flow type system to guarantee information flow properties and formalize the following based on that assumption:

For all $m \in \mathcal{M},\ id, id' \in \mathcal{H},\ s, s' \in \mathcal{S},\ e \in \mathcal{E},\ v \in \mathcal{V},\ n \in \mathbb{N},\ b \in \mathcal{B}$

1. $(s, id, s', id') \in Local(m) \Rightarrow\ IFA(s, s')$
2. $(s, b, n, s') \in Rem(m) \Rightarrow containsAll(vals(s'),\ vals(s).put(\ (x, v)\ j\ \exists e.b[n] = (e, v)))$
   where x is defined to be a new variable

3. $(s, m', s') \in New(m) \Rightarrow\ IFA(s, s')$

Setup

The state of a state machine is represented as $(s, id) \in (\mathcal{S}, \mathcal{H})$ where s is the local state of the machine, and id is a placeholders used to store the target of a send command or the handle of a newly created machine.

The configuration of our system is the following tuple: (S, B, C) where S is a partial map of $\mathcal{Z}$ to $\mathcal{S} \times \mathcal{H}$, B is a partial map of $\mathcal{Z}$ to $\mathcal{B}$, and C is a partial map of $\mathcal{M}$ to $\mathbb{N}$. Essentially, $S[m, n]$ represents the state of the nth instance of machine m where $m\ \epsilon\ \mathcal{M}$ and $n \in \mathbb{N}$. $B[m, n]$ represents the input buffer of this machine. $C[m]$ represents the number of instances of machine m that have been created so far.

Internal Rules

These rules represent the internal state transitions of the state machines.

For the Local Transition, an observer is assumed to be able to infer the new untrusted state from this transition, and we indicate this in the label of the transition.

$$\text{Local Transition}\ \frac{\begin{array}{c} m \in \mathcal{M} \quad n \in \mathbb{N} \\ S[m, n] = (s, id) \quad (s, id, s', id') \in Local(m) \end{array}}{(S, B, C) \xrightarrow{uvals(s')} (S[(m, n) \mapsto (s', id')],\ B,\ C)}$$

For the Dequeue Event transition, an observer is assumed to be able to infer the new untrusted state from this transition as well as the content of any Untrusted Event that is being processed. Also, recall that dequeuing an event can add a handle to the current machine's local state (if another machine has sent it), and that is defined in the "Machine Handles Cannot be Created Out of Thin Air" formalization for the *Rem* rule in the previous section.

$$\text{Dequeue Event}\ \frac{\begin{array}{c} m \in \mathcal{M} \quad n, pos \in \mathbb{N} \quad S[m, n] = (s, id) \quad B[m, n] = b \\ (s, b, pos, s') \in Rem(m) \quad b' = remove(b, pos) \quad (e, v) = b \end{array}}{(S, B, C) \xrightarrow{uvals(s'), e,\ (vje \in \mathcal{E}_{\mathcal{U}})} (S[(m, n) \mapsto (s', id)],\ B[(m, n) \mapsto b'],\ C)}$$

Creation Rules

In the Trusted Create rule, we have a parent SSM (denoted with subscript p) creating a new child SSM (denoted with subscript c). The parent SSM will receive the trusted handle of the newly created child SSM. This is a labeled transition because an outside observer can see which child machine is created as well as which machine sent the creation request. We

denote the Trusted Create rule as follows:

$$\text{Trusted Create}\ \frac{\begin{array}{c} m_p, m_c \in \mathcal{M}_\mathcal{T} \quad n_p, n_c \in \mathbb{N} \\ tid_c = (m_c, n_c, 1) \quad uid_c = (m_c, n_c, 0) \\ S[m_p, n_p] = (s_p, \_) \quad (s_p, m_c, s'_p) \in New(m_p) \quad n_c = C[m_c] \end{array}}{\begin{array}{c} (S, B, C) \xrightarrow{(m_p,n_p),(m_c,n_c)} (S[(m_p, n_p) \mapsto (s'_p, tid_c),\ (m_c, n_c) \mapsto (s_0, \{tid_c, uid_c\})], \\ B[(m_c, n_c) \mapsto b_0],\ C[m_c \mapsto n_c + 1]) \end{array}}$$

In the Untrusted Create rule, we have a parent state machine (denoted with subscript p) creating a child state machine (denoted with subscript c). The parent machine will receive the untrusted handle of the child machine. This is a labeled transition because an outside observer can see which child machine is created as well as which machine sent the creation request. We denote the Untrusted Create rule as follows:

$$\text{Untrusted Create SSM}\ \frac{\begin{array}{c} m_p, m_c \in \mathcal{M} \quad (m_p \in \mathcal{M}_\mathcal{U} \wedge m_c \in \mathcal{M}_\mathcal{T}) \quad n_p, n_c \in \mathbb{N} \\ tid_c = (m_c, n_c, 1) \quad uid_c = (m_c, n_c, 0) \\ S[m_p, n_p] = (s_p, \_) \quad (s_p, m_c, s'_p) \in New(m_p) \quad n_c = C[m_c] \end{array}}{\begin{array}{c} (S, B, C) \xrightarrow{(m_p,n_p),(m_c,n_c)} (S[(m_p, n_p) \mapsto (s'_p, uid_c),\ (m_c, n_c) \mapsto (s_0, \{tid_c, uid_c\})], \\ B[(m_c, n_c) \mapsto b_0],\ C[m_c \mapsto n_c + 1]) \end{array}}$$

$$\text{Untrusted Create USM}\ \frac{\begin{array}{c} m_p, m_c \in \mathcal{M} \quad m_c \in \mathcal{M}_\mathcal{U} \quad n_p, n_c \in \mathbb{N} \\ uid_c = (m_c, n_c, 0) \\ S[m_p, n_p] = (s_p, \_) \quad (s_p, m_c, s'_p) \in New(m_p) \quad n_c = C[m_c] \end{array}}{\begin{array}{c} (S, B, C) \xrightarrow{(m_p,n_p),(m_c,n_c)} (S[(m_p, n_p) \mapsto (s'_p, uid_c),\ (m_c, n_c) \mapsto (s_0, \{uid_c\})], \\ B[(m_c, n_c) \mapsto b_0],\ C[m_c \mapsto n_c + 1]) \end{array}}$$

Sending Rules

In the Trusted Send rule, we have a sending SSM (denoted with subscript s) sending a Trusted Event to a receiving SSM (denoted with subscript r). The sending SSM must possess the trusted handle of the receiving SSM. After executing this command, the sending SSM transitions to its next state and the receiving SSM enqueues this trusted event in its input buffer. This is a labeled transition because an outside observer can infer the type of event that was sent (although they cannot infer the contents of the message payload itself) as well as which 2 parties the communication is happening between. We denote the Trusted Send rule as follows:

$$\text{Trusted Send}\ \frac{\begin{array}{c} m_s, m_r \in \mathcal{M}_\mathcal{T} \quad n_s, n_r \in \mathbb{N} \quad e \in \mathcal{E}_\mathcal{T} \quad v \in \mathcal{V}_\mathcal{T} \quad tid_r = (m_r, n_r, 1) \\ S[m_s, n_s] = (s_s, tid_r) \quad B[m_r, n_r] = b_r \quad (s_s, tid_r, e, v, s'_s) \in Enq(m_s) \end{array}}{(S, B, C) \xrightarrow{(m_s,n_s),(m_r,n_r),e} (S[(m_s, n_s) \mapsto (s'_s, tid_r)],\ B[(m_r, n_r) \mapsto b_r \bigodot (e, v)],\ C)}$$

In the Untrusted Send rule, we have a sending state machine (denoted with subscript s) sending an Untrusted Event to a receiving state machine (denoted with subscript r). The sending machine must possess the untrusted handle of the receiving machine. After executing this command, the sending machine transitions to its next state and the receiving machine enqueues this untrusted event in its input buffer. This is a labeled transition because an outside observer can corrupt the untrusted machines and view the message payload, event type, as well as which parties the message is being exchanged between. We denote the Untrusted Send rule as follows:

$$\text{Untrusted Send}\ \frac{\begin{array}{c} m_s, m_r \in \mathcal{M} \quad n_s, n_r \in \mathbb{N} \quad e \in \mathcal{E}_{\mathcal{U}} \quad v \in \mathcal{V}_{\mathcal{U}} \quad uid_r = (m_r, n_r, 0) \\ S[m_s, n_s] = (s_s, uid_r) \quad B[m_r, n_r] = b_r \quad (s_s, uid_r, e, v, s'_s) \in Enq(m_s) \end{array}}{(S, B, C) \xrightarrow{(m_s,n_s),(m_r,n_r),e,v} (S[(m_s, n_s) \mapsto (s'_s, uid_r)],\ B[(m_r, n_r) \mapsto b_r \bigodot (e, v)],\ C)}$$

3.2.3 Type Checker

Information Flow Analysis

One of the primary goals of PSec is to enable secure computation in a simple, high-level language. To achieve this goal, we also want to prevent programmers from accidentally leaking secret data. For example, a Secure Voting System application should not leak any identifying information of the voter to untrusted machines. Traditional approaches to accomplishing this involve using forms of encryption or access control frameworks. However, a fundamental problem with these approaches is although these techniques do prevent malicious machines from accessing data directly, they don't prevent machines with legitimate access from leaking this data to bad actors [32]. This problem can be combated by placing external monitors on the machines that check all output and block as necessary. However, it is hard for these detection systems to detect all forms of sensitive data (sensitive data that is slightly modified may not trigger these monitors).

A better approach is to implement Information Flow Control in PSec's type checker. This kind of static analysis enables us to enforce that no secret information is leaked. Information Flow Control works by augmenting the type system to include security labels for each type of variable [32]. In our system, we define two types of security labels (High/H and Low/L). These labels indicate that we want to maintain the confidentiality and integrity of certain values. Essentially, we want to prevent data with H labels from being leaked to adversaries and we also want to ensure that this data contains trusted information, rather than potentially malicious payloads. Programmers can mark variables with a `secure` type to indicate this and a High security label is automatically assigned to these variables. We need to propagate these labels and enforce certain rules to actually achieve these properties, and we describe this in the next few sections.

Our Adversary Model for the information flow system is that we are assuming that the adversary is allowed to view/infer the content of Low security variables and expressions at any time. They can also arbitrarily write to any Low security variables. However, the goal of

our type system is that, given the previous assumptions, we want to prevent the adversary from inferring or corrupting the values of any High security variables/expressions. More specifically, our goals are the following [6]:

1. Classify Expressions
2. Prevent Explicit Flows
3. Prevent Implicit Flows
4. Prevent Corruption

Goal 1 states that we need a way to figure out how to classify expressions as H and L. Goal 2 states that we should forbid H expressions from being assigned to L variables. This is because given our adversary model, the adversary will be able to view the L variable after the assignment statement is executed and infer information regarding the H variables used to construct the assigned expression. Goal 3 states that we need to prevent side channel attacks that will leak the value of H values. Finally, Goal 4 states that we want to protect the integrity of H values within our system.

Type Checker Rules

In this section, we describe the formal type checking rules of our system so that we can achieve the goals mentioned above. They are derived/adapted from [6], [23], and [32].

First of all, we define the security labels for P's native types. We demonstrate the rules for the `int` type, `machine_handle` type, and the `event` type as specific examples, but this pattern follows for all remaining types (such as `bool`, `StringType`, etc)

$$\Gamma(secure\_int) = H;$$
$$\Gamma(int) = L;$$
$$\Gamma(secure\_machine\_handle) = H;$$
$$\Gamma(machine\_handle) = L;$$
$$\Gamma(trusted\, event) = H;$$
$$\Gamma(event) = L;$$

Variables and Constants For our variable and constant rule, we have

$$\text{Constant}\frac{n \in \mathbb{N}}{\Gamma \vdash n : L} \qquad \text{Variable}\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

where τ represents the security type of the variable x. These rules are fairly straightforward.

Expressions Expression information flow need to propagate security labels correctly in order to maintain confidentiality as well as integrity properties.

For the remainder of this section, this

$$\Gamma \vdash e : \tau$$

means that e is a well typed expression with the τ security label.

Given the above, we define the expression rules to be the following:

$$\text{Unary Exp}\frac{\Gamma \vdash e : \tau}{\Gamma \vdash op\, e : \tau} \qquad \text{Binary Exp}\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ op\ e_2 : \tau}$$

where *op* in the unary case is `[!, -]` and in the binary case is `[+, -, *, /, <, <=, >, >=, ==, !=, &&, ||]`.

We additionally define a Declassify and Endorse function in order to allow the programmer to explicitly change security labels for practical uses cases.

$$\text{Declassify}\frac{\Gamma \vdash e : H \quad \Gamma \vdash \tau : L}{\Gamma \vdash (\textit{Declassify(e) as } \tau) : L}$$

$$\text{Endorse}\frac{\Gamma \vdash e : L \quad \Gamma \vdash \tau : H}{\Gamma \vdash (\textit{Endorse(e) as } \tau) : H}$$

Commands Command Information Flow utilizes subtyping in order to determine the security label of each command. This enables us to prevent implicit side channels. For commands, we have the following subtyping rule:

$$H\ com \leq L\ com$$

With this subtyping, we have the following subsumption rule:

$$\text{Subsumption Command}\frac{\tau_1\ com \leq \tau_2\ com \quad \Gamma \vdash c : \tau_1\ com}{\Gamma \vdash c : \tau_2\ com}$$

Based on the previous results, this

$$\Gamma \vdash c : \tau\ com$$

means that c is a well typed command that assigns to variables of type τ or higher

In terms of the command rules, we have the following:

$$\text{Assign}\frac{\Gamma \vdash e : \tau \quad \Gamma(x) = \tau}{\Gamma \vdash x := e : \tau\ com} \qquad \text{If Rule}\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash c_1 : \tau\ com \quad \Gamma \vdash c_2 : \tau\ com}{\Gamma \vdash if\ e\ then\ c_1\ else\ c_2 : \tau\ com}$$

$$\text{Sequence Rule}\frac{\Gamma \vdash c_1 : \tau\ com \quad \Gamma \vdash c_2 : \tau\ com}{\Gamma \vdash c_1;\ c_2 : \tau\ com} \qquad \text{While Rule}\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash c : \tau\ com}{\Gamma \vdash while\ e\ do\ c : \tau\ com}$$

Since sending messages to untrusted machines and printing output to the console are natural side channels, we denote the following commands with an L label. In the Untrusted Send rule, it is important to note that PSec enforces Untrusted Events are only allowed to contain non-sensitive payloads, and this is showcased in the L security label of the payload expression.

$$\text{UntrustedSend}\frac{\begin{array}{c} id \in \mathcal{H}_{\mathcal{U}} \quad ev \in \mathcal{E}_{\mathcal{U}} \quad v \in \mathcal{V}_{\mathcal{U}} \\ \Gamma \vdash id : L \quad \Gamma \vdash ev : L \end{array}}{\Gamma \vdash send\ id,\ ev,\ v : L\ com} \qquad \text{Print}\frac{\Gamma \vdash e : L}{\Gamma \vdash Print\ e : L\ com}$$

We regard sending messages to trusted SSMs as a trusted action (since we use the Trusted Message Sending protocol). As a result, we denote the following command with an H label. It is important to note that PSec enforces Trusted Events are only allowed to contain payloads with an H security label. This is important for the confidentiality proof in the later sections, and we note that this doesn't hinder programmers from sending L payloads with the Trusted Messaging Sending protocol. If necessary, the programmer can explicitly *Endorse* the payload on the sending SSM and use the $Declassify$ function on the receiving SSM to achieve this goal, which facilitates many practical use cases.

$$\text{TrustedSend}\frac{\begin{array}{c} id \in \mathcal{H}_{\mathcal{T}} \quad ev \in \mathcal{E}_{\mathcal{T}} \quad v \in \mathcal{V}_{\mathcal{T}} \\ \Gamma \vdash id : H \quad \Gamma \vdash ev : H \end{array}}{\Gamma \vdash send\ id,\ ev,\ v : H\ com}$$

Similarly, the Declassify and Endorse functions are denoted with an H label since they are trusted operations.

$$\text{Declassify}\frac{}{\Gamma \vdash Declassify :\ H\ com} \qquad \text{Endorse}\frac{}{\Gamma \vdash Endorse :\ H\ com}$$

One item to note is that PSec allows for external method calls implemented in C++ code. This approach is taken from P, where this functionality is provided for cases where the P language is not expressive enough to implement certain application features. We require the programmer to specify the input and output types (along with security labels) of this "foreign method". Since the protections offered by PSec can be subverted in the C++ code, we place the responsibility on the programmer to handle input safely and securely. "Foreign methods" are out of scope for the purposes of the thesis. PSec also provides for regular function definitions written directly in PSec, but since these function bodies can effectively be expanded in the state machine code, we do not provide formalisms for this feature in the body of this thesis.

Sanity Checks Using these rules, we can perform a sanity check and construct the following proof trees for add operations with different security labels.

$$\text{Declassify Sanity Check}\frac{\Gamma \vdash int : L \quad \dfrac{\Gamma \vdash secure\_int : H \quad \Gamma \vdash int : L}{\Gamma \vdash (Declassify(secure\_int)\ as\ int) : L}}{\Gamma \vdash int + (Declassify(secure\_int)\ as\ int) : L}$$

$$\text{Endorse Sanity Check}\frac{\dfrac{\Gamma \vdash int : L \quad \Gamma \vdash secure\_int : H}{\Gamma \vdash (Endorse(int)\ as\ secure\_int) : H} \quad \Gamma \vdash secure\_int : H}{\Gamma \vdash (Endorse(int)\ as\ secure\_int) + secure\_int : H}$$

We can perform a sanity check on the if command as well. First of all, recall the if command has the following structure: `if e then c1 else c2`. Based on this, we construct the following cases and see whether our command rules lead to valid derivation trees.

1. Case 1: `if L then L com else L com`

   $$\frac{\Gamma \vdash e : L \quad \Gamma \vdash c_1 : L\,com \quad \Gamma \vdash c_2 : L\,com}{\Gamma \vdash if\ e\ then\ c_1\ else\ c_2 : Lcom}$$ → Succeeds. Overall Type is L.

2. Case 2: `if L then L com else H com`

   $$\frac{\Gamma \vdash e : L \quad \Gamma \vdash c_1 : L\,com \quad \dfrac{\Gamma \vdash c_2 : H\,com \quad H \leq L}{\Gamma \vdash c_2 : L\,com}}{\Gamma \vdash if\ e\ then\ c_1\ else\ c_2 : L\,com}$$ → Succeeds. Overall Type is L.

3. Case 3: `if L then H com else H com`

   $$\frac{\Gamma \vdash e : L \quad \dfrac{\Gamma \vdash c_1 : H\,com \quad H \leq L}{\Gamma \vdash c_1 : L\,com} \quad \dfrac{\Gamma \vdash c_2 : H\,com \quad H \leq L}{\Gamma \vdash c_2 : L\,com}}{\Gamma \vdash if\ e\ then\ c_1\ else\ c_2 : L\,com}$$

   → Succeeds. Overall Type is L.

4. Case 4: `if H then H com else H com`

   $$\frac{\Gamma \vdash e : H \quad \Gamma \vdash c_1 : H\,com \quad \Gamma \vdash c_2 : H\,com}{\Gamma \vdash if\ e\ then\ c_1\ else\ c_2 : Hcom}$$ → Succeeds. Overall Type is H.

5. Case 5: `if H then H com else L com` → Error! No Derivation Tree.

6. Case 6: `if H then L com else L com` → Error! No Derivation Tree.

If we look at this example, this only leaks information when e is equal to H and there is an L in either command $c1$ or $c2$. Intuitively this makes sense because since the adversary can view the value of L variables, they can see how the L variables have changed after execution and gain some knowledge about the H expression that is conditioned on. Our type checker prevents information from being leaked through these side channels.

We outline sample PSec code with comments in regards to information flow type checking in the Appendix, in Section A.3.

Observational Determinism Theorem Observational determinism [37, 7] is a property that, if satisfied, prevents adversaries from inferring sensitive information from the execution of the program and corrupting trusted values in our system. We state the following theorem:

Theorem 1 *If the PSec type checker terminates successfully for a given program P, then P satisfies the property of Observation Determinism when executed with the PSec Runtime.*

We provide a formal proof for this theorem as part of our evaluation in Section 3.4.3 of this thesis.

3.3 Language Implementation

As part of our language design, we assume that code running within Secure State Machines (SSMs) is trusted and unmodified and that adversaries cannot read sensitive data contained within these machines. By providing guarantees such as confidentiality of execution and integrity of code through remote attestation, Intel SGX can enable us to achieve these goals.

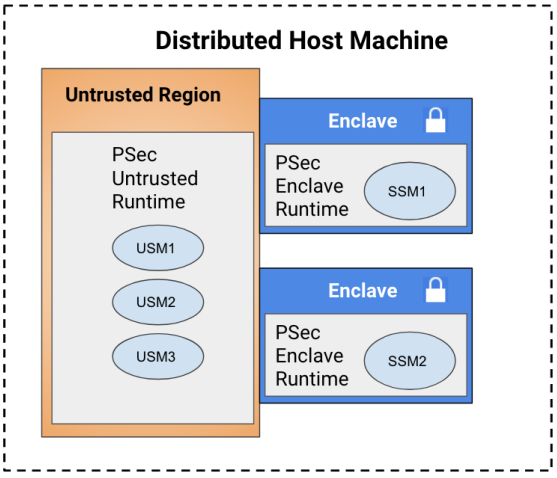

Figure 3.1: Distributed Host Architecture

We create 2 separate runtimes: the PSec Enclave Runtime, and the PSec Untrusted Runtime. After the programmer writes the PSec file, and it passes the type checking phase, the PSec Compiler is used to generate C code for the various state machines. The PSec Enclave Runtime as well as the generated code for all of the state machines is then compiled within each Intel SGX Enclave. The enclave, upon creation, spawns an instance of a PSec Process within secure memory that handles running the generated state machine code. Each enclave has the ability to create any state machine provided by the generated code, but the PSec Enclave Runtime enforces that each enclave only contains 1 SSM at any given time. Each distributed host machine also has an instance of the PSec Process running within untrusted memory that handles creating and managing USMs based on the previously generated state machine code. The PSec Untrusted Runtime within the distributed host is in charge of ferrying messages to the network for all of the SSMs and USMs located on this host machine. This overall architecture is shown in Figure 3.1.

3.3.1 Threat Model

Adversaries

We assume that there are different levels of adversaries and we provide different guarantees against each one.

1. Passive Network Observer
    - Can observe all network traffic and will attempt to extract relevant data from network requests
    - Cannot tamper or modify any network traffic
2. Active Man-in-the-Middle
    - Can observe all network traffic and will attempt to extract relevant data from network requests
    - Can tamper with network traffic and modify network requests in hopes of extracting relevant information
3. Privileged Attacker
    - Can observe all network traffic and will attempt to extract relevant data from network requests
    - Can tamper with network traffic and modify network requests in hopes of extracting relevant information
    - Can corrupt distributed host machines in our network and spin up its own malicious enclaves on these host machines as needed. Has privileged access in untrusted world and can read and tamper with all internal state for the distributed host machines, except state stored within enclaves

Guarantees against Adversaries

We design PSec in order to provide certain levels of guarantees against the aforementioned adversaries. We describe the guarantees provided in this section and discuss the implementation enabling these guarantees in later sections.

1. Passive Network Observer
    - PSec has necessary cryptography built-in to prevent network observers from determining message payloads for both Trusted and Untrusted Message Sending
    - Passive Network Observers will be able to determine which 2 parties are communicating as well as lengths of messages exchanged between the parties
    - State machines will execute their code as expected
2. Active Man-in-the-Middle
    - PSec has necessary cryptography built-in to enable state machines to detect if any message payloads have been tampered with
    - Active Man-in-the-Middle attackers will be able to determine which 2 parties are communicating as well as message lengths. They can also induce denial-of-service attacks by dropping network requests
    - State machines will execute their code as expected
3. Privileged Attacker
    - Privileged Attackers have control over the entire untrusted world (USMs, distributed host machines, and the network). They can additionally create SSMs by spinning up their own malicious enclaves. As a result, Privileged Attackers can send authenticated messages from compromised parties to other state machines in our system. They can also perform denial-of-service attacks by preventing messages from being sent and processed
    - PSec still provides guarantees against this more advanced adversary. PSec prevents SSMs from ever giving secrets to the untrusted world (such as to USMs) because these secrets are assumed to be easily leaked. PSec also prevents secrets from being sent to untrusted SSMs (SSMs that the current machine doesn't have the `capability` for)
    - SSMs created using the `Trusted Create` protocol are guaranteed to execute their code as expected. All other state machine are not guaranteed to execute their code correctly

It is important to note that denial-of-service attacks cannot be prevented in any case because this is a fundamental limitation of enclaves as they rely on their host machines

for network operations. Regardless, PSec guarantees that no secret data is ever leaked to adversaries.

Similar to other SGX-related work in this area (such as the EActors programming language [24]), we consider side-channel attacks (such as page-fault based attacks [36, 30], speculative execution attacks [2], cache-timing attacks [14, 1, 26], or general timing attacks) to be out of scope. These attacks have counter-measures [30, 29, 12, 15, 25, 20] that can be implemented on the side, and we leave that as future work.

3.3.2 System Architecture

In our system, we have multiple distributed host machines and a trusted Key Provisioning Server (KPS). The KPS stores all precomputed valid enclave measurements and serves as an intermediary to establish trust for various protocols. The public KPS key is baked into all enclaves so they can establish secure channels with the KPS. The programmer is able to take the PSec executable and spawn the KPS or different host machines by specifying different command-line arguments and running the executable on that physical machine. If spawning the KPS, the programmer additionally has to specify the network addresses for all the different distributed hosts in the system as well as the type of State Machines allowed to be hosted by those distributed hosts. This is important so that the KPS can include this information in its IP lookup table. Finally, the KPS is assumed to be deployed on a trusted server. This setup can be visualized in Figure 3.2.

In general, we assume that the first SSM in our system is created from a secure context. This means that when a USM creates the first SSM in order to kickstart the system, the USM assumes that the SSM is securely created even though the Untrusted Create protocol is used. This assumption is necessary to bootstrap trust in our system and can be made practical by having the initial SSM and USM be hosted by the same trusted distributed host. Once the first SSM is created and running successfully, it is able to go ahead and create additional trusted SSMs in our system (which can reside on untrusted distributed host machines) and perform trusted computations or create even more trusted SSMs during the execution of the program.

3.3.3 Definitions for Protocols

Identity

Every state machine has an associated unique `Identity` public/private key pair which establishes the identity of the machine. When we say that a state machine has authenticated with its `Identity`, this involves a signature over the private `Identity` key. When we refer to `Identity` in the rest of this paper, we mean the `Identity` public key. For example, a state machine sending its `Identity` to another state machine means that it is sending its `Identity` public key over. We will be explicit when we refer to the `Identity` private key.

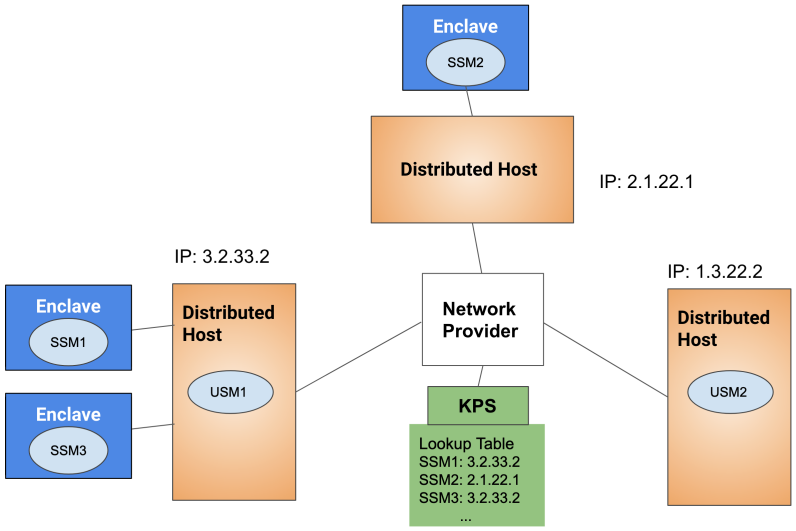

Figure 3.2: Network Architecture

Capability

Every SSM has a `Capability` associated with it. A `Capability` is essentially a public/private key pair that gives anyone who can prove they possess the private key the ability to send Trusted Events to the associated SSM. `Capabilities` must never leave into the untrusted world and must stay within enclaves since they are secret keys. When we refer to the `Capability` key, we mean the entire public/private key pair.

Secure Sigma* Channel

We use `Secure Sigma*` to establish secure channels between enclaves and the KPS. This channel is created using Intel's version of the Sigma Protocol (version of Authenticated Diffie-Hellman that addresses many of the weakness of Diffie-Hellman). Recall that the KPS Public Key is pre-baked into all enclaves. In this protocol, the enclave remote attests itself to the KPS and proves that it is legitimate and running PSec. The KPS authenticates using its private signing key while the enclave authenticates using Intel EPID.

3.3.4 Trusted Create Protocol

Recall that the `Trusted Create` Protocol is invoked when an SSM wants to create a new SSM. We denote the newly created SSM as the child SSM and the original SSM as the parent SSM. Upon successful completion, this protocol returns a `secure_machine_handle` typed object to the parent SSM that contains both the `Capability`, `Identity`, and network address information of the child SSM.

We outline the protocol in Figure 3.3. The protocol is divided into 2 parts as below:

Parent SSM - createMachineRequest()

When the `new` command is invoked on a SSM type, the Parent SSM actually calls the `createMachineRequest` method of the PSec Enclave Runtime. It passes in its `Identity` as well as the type of SSM it wishes to create. This method makes an `OCALL` to the PSec Untrusted Runtime to make a network request to the KPS to determine the IP address/port of a valid distributed host that can create the Child SSM. After receiving this information, the `createMachineRequest` method makes another `OCALL` to forward the machine creation request along with the Parent SSM's `Identity` to this valid distributed host. The expected response from this request is the `Identity` of the newly created Child SSM. After receiving this response, the parent enclave establishes a `Secure Sigma*` channel with the KPS. It then sends in (Parent SSM `Identity`, Child SSM `Identity`, Type of SSM Requested To Create) and receives the Child SSM `Capability`. The Parent Enclave then encapsulates the Child SSM's `Capability` key, `Identity`, and network address information in a `secure_machine_handle` object and returns it back to the programmer in the PSec code.

Child SSM - createMachineAPI()

When a distributed host receives a network request, the PSec Untrusted Runtime processes the request. In the case of a SSM creation request, it receives the Parent SSM `Identity` and the type of SSM that needs to be created. It goes ahead and creates a new enclave on the host machine and then calls the `createMachineAPI` PSec Enclave Runtime method with the appropriate parameters. The `createMachineAPI` method checks to make sure that there are no existing SSMs running in the current enclave and that the child SSM type is a valid type. It then initializes a new PSec Process inside the enclave and creates the Child SSM by calling PrtMkMachine within the existing PSec process. It also generates an `Identity` for the child SSM. After this initial setup, the enclave needs to establish a `Capability` for this SSM so that it can receive Trusted Events. The Child Enclave does this by establishing a `Secure Sigma*` channel with the KPS and sending it (Parent SSM `Identity`, Child SSM `Identity`, Type of SSM Requested To Create). The KPS generates a new `Capability`, stores (Parent SSM `Identity`, Child SSM `Identity`, Type of SSM Requested To Create) `->` Child SSM `Capability`, and sends the `Capability` back to the Child SSM. Once the Child Enclave receives the `Capability`, it stores it and makes an `OCALL` to the PSec Untrusted Runtime in order to send the Child SSM `Identity` back to the Parent Enclave.

Guarantees

The `Trusted Create` Protocol gives us many guarantees against even our privileged attacker. First of all, if the Parent SSM successfully retrieves the `Capability` of the Child SSM from the KPS, it receives a guarantee that the Child SSM was securely created in a valid enclave. This guarantee is ensured by the fact that the KPS validates the Child Enclave through the `Secure Sigma*` protocol before generating a `Capability` for it. The Child SSM receives the guarantee that only its Parent SSM receives its `Capability` and that the Parent SSM

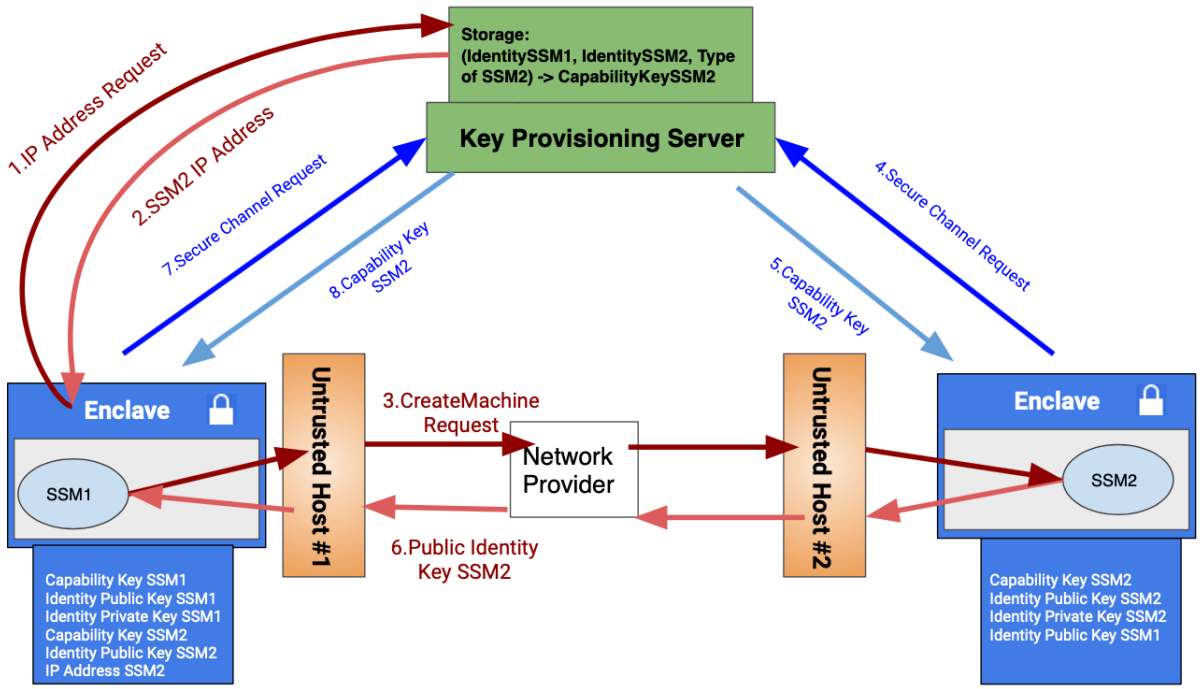

Figure 3.3: Trusted Create Protocol

is running inside a valid enclave. Once again, this guarantee is ensured since the Parent Enclave is validated by the KPS through the `Secure Sigma*` protocol before it can retrieve the `Capability`.

This protocol has limitations, but we show that the limitation are the same limitations that are inherent to enclaves. This protocol is vulnerable to a Denial of Service attack by any of the distributed host machines since the host machines can refuse to send messages across the network. Another limitation is that the distributed hosts can modify the Child SSM `Identity` that is sent back across the network to the Parent Enclave or modify the requested type of SSM to create. However, in this case, the Parent Enclave will not be able to retrieve the `Capability` from the KPS and will realize that the `Trusted Create` call has failed. The KPS's final message to the child enclave containing the `Capability` of the child machine can be blocked (DoS), preventing the child SSM from ever receiving Trusted Events. The distributed host machines can also give the wrong IP address of the Child SSM from the KPS so that the Parent Enclave is unable to contact the correct machine (another form of DoS). It is important to note that in all of these cases, this protocol doesn't leak any secret data and the enclaves are generally aware of when failures/malicious behavior occur.

3.3.5 Untrusted Create Protocol

The `Untrusted Create` Protocol is invoked during all other machine creation cases. We denote the newly created machine as the child machine and the original machine as the parent machine. Upon completion, this protocol returns a `machine_handle` typed object to the parent machine that contains the `Identity` as well as network address information of the newly created child.

Similar to `Trusted Create`, the protocol is divided into 2 parts as below:

Parent Machine - createUntrustedMachineRequest()

When the `new` command is invoked on an USM type (or an SSM type being created from a parent USM machine), the parent machine actually calls the `createUntrustedMachineRequest` method of the PSec Runtime. This method makes a network request to the KPS to determine the IP address/port of a valid distributed host that can create the child machine SSM. After receiving this information, the `createUntrustedMachineRequest` method forwards the machine creation request along with the Parent SSM's `Identity` to this valid distributed host. The expected response from this request is the `Identity` of the newly created Child SSM, and the PSec Runtime encapsulates this `Identity` as well as the network address information into a `machine_handle` and returns it to the parent machine.

Child Machine - createUntrustedMachineAPI()

When a distributed host receives a network request, the PSec Untrusted Runtime processes the request. If the request is to create a new SSM (since USMs can request to create SSMs), a new enclave is created and the request is forwarded to the PSec Enclave runtime (the process proceeds in a manner similar to `Trusted Create`). However, in this case, the `Capability` of the newly created SSM is not shared with any other parties. If the request is to create a new USM, the PSec Untrusted Runtime creates a new child USM within the existing PSec process by calling PrtMkMachine and also generates an `Identity` for the child machine. In both cases, the newly created `Identity` is sent back across the network to the parent machine.

Guarantees

The `Untrusted Create` Protocol doesn't provide many guarantees and is a best-effort protocol. Since these creation requests don't contain any sensitive information, they are sent across the network in plaintext. However, as a result, adversaries can conduct a wide variety of attacks. Adversaries can modify the requests to prevent the correct machines from being created (effective denial of service). They can also modify the `Identity` that is returned by the child machine and intercept all future untrusted messages sent from the parent machine. This protocol ensures correct and secure creation in the face of our first adversary, a network snooper. In order to protect against more advanced Man-In-The-Middle network

adversaries, we can potentially implement Public Key Infrastructure additionally at the host machine level (instead of just the state machine level). In this design, messages sent from one physical host machine to another in the system would have an additional layer of encryption and would be processed and decrypted by the PSec Runtime in the receiving distributed host machine. We have not implemented this for now, and outline a possible design for this additional feature as part of our future work in Section 4.1.

3.3.6 Trusted Send Protocol

The `Trusted Send` Protocol is used when an SSM wants to send a Trusted Event containing potentially sensitive data to another SSM. We denote the first SSM as the Sending SSM and the second SSM as the Receiving SSM. The Sending SSM must have the `Capability` of the Receiving SSM as well as its `Identity` handle.

TrustedSend()

When the Sending SSM invokes the `send` command with a `secure_machine_handle`, it actually calls the `TrustedSend` method in the PSec Enclave Runtime. As before, this connection goes through the Sending Enclave's distributed host, the network provider, and the Receiving Enclave's distributed host before it terminates in the Receiving Enclave. The Sending Enclave authenticates itself by using the `Capability` and the receiving Enclave authenticates itself by using its `Identity`. After a secure channel is created, the Trusted Event is sent over and passed to the Receiving SSM by the PSec Enclave Runtime. In terms of the specifics, we outline the protocol below. We have SSM A wanting to send a trusted message to SSM B. We are assuming that SSM A already has obtained `PublicIdentityKeyB`, `PublicCapabilityKeyB`, `PrivateCapabilityKeyB`, and `IPAddressAndPortMachineB` and SSM B has `PublicIdentityKeyB`, `PrivateIdentityKeyB`, `PublicCapabilityKeyB`, and `PrivateCapabilityKeyB`.

1. SSM A randomly generates a new `sessionKey` using `sgx_read_rand`
2. SSM A constructs the following message M: M = `sessionKey`
3. SSM A constructs EncryptedMessage: EncryptedMessage = E(M) where `PublicIdentityKeyB` is used for encrypting
4. SSM A sends "InitComm:SSM-A-Identity:SSM-B-Identity:EncryptedMessage" to `IPAddressAndPortMachineB`
5. SSM B receives the message and decrypts it using `PrivateIdentityKeyB` and saves the `sessionKey`
6. SSM B sends back a "Success!" response in plaintext

7. SSM A generates a random IV
8. SSM A constructs M: M = `PublicIdentityKeyB` ‖ nonce ‖ Trusted Event Payload
9. SSM A constructs E(M ‖ Sign(M)) = (EncryptedMessage, MAC) where encryption occurs using `sessionKey` and IV and signing occurs over `PrivateCapabilityKeyB`
10. SSM A sends over "TrustedSend:SSM-B-Identity:IV:MAC:EncryptedMessage"
11. SSM B receives this message, decrypts using the `sessionKey`, verifies the signature over `PublicCapabilityKeyB`, and checks the nonce to prevent against replay attacks. SSM B extracts the TrustedEventPayload and handles this event.
12. SSM B updates the nonce and generates a new IV. It then sends back the following confirmation message: IV:MAC:E("Success" ‖ nonce) where the `sessionKey` is used for encryption
13. SSM A receives this message and verifies that the trusted message was successfully received

Guarantees

This protocol guarantees confidentiality and integrity of the data being sent against even privileged attackers. We guarantee that Trusted Events are never leaked to the untrusted world through this sending process and that they are sent by a trusted SSM to the intended SSM. As before, this protocol is vulnerable to DoS attacks because the distributed host machines can choose to drop the messages at any point of time.

3.3.7 Untrusted Send Protocol

Invoking the `send` command with a `machine_handle` actually calls the `UntrustedSend` method in the PSec Runtime. This method establishes a secure channel with the receiving machine by using the receiving machine's `Identity` as a means of authentication. It is important to note that this authentication isn't important toward providing any additional guarantees and is easily spoofable by Man-in-the-Middle attackers that can construct messages that seem to be coming from a particular machine. However, the confidentiality and integrity of existing message is still protected against these type of attackers. After a secure channel is created, the Untrusted Event is sent over and passed to the receiving machine by the PSec Runtime. In terms of the specifics, we outline the protocol below. We have Machine A wanting to send a Untrusted Event to Machine B. We are assuming that Machine A already has obtained `PublicIdentityKeyB` and `IPAddressAndPortMachineB`, and Machine B has `PublicIdentityKeyB` and `PrivateIdentityKeyB`.

1. Machine A randomly generates a new `sessionKey`

2. Machine A constructs the following message M: M = `sessionKey`
3. Machine A constructs EncryptedMessage: EncryptedMessage = E(M) where `PublicIdentityKeyB` is used for encrypting
4. Machine A sends "InitComm:Machine-A-Identity:Machine-B-Identity:EncryptedMessage" to `IPAddressAndPortMachineB`
5. Machine B receives the message and decrypts it using `PrivateIdentityKeyB` and saves the `sessionKey`.
6. Machine B sends back a "Success!" response in plaintext
7. Machine A generates a random IV
8. Machine A constructs M: M = PublicIdentityKeyB ‖ nonce ‖ UntrustedEventPayload
9. Machine A constructs E(M ‖ Sign(M)) = (EncryptedMessage, MAC) where encryption occurs using `sessionKey` and IV and signing occurs over `PrivateIdentityKeyA`
10. Machine A sends over "UntrustedSend:Machine-B-Identity:IV:MAC:EncryptedMessage"
11. Machine B receives this message, decrypts using the `sessionKey`, verifies the signature over `PublicIdentityKeyA`, and checks the nonce to prevent against replay attacks. Machine B extracts the UntrustedEventPayload and handles this event.
12. Machine B updates the nonce and generates a new IV. It then sends back the following confirmation message: IV:MAC:E("Success" ‖ nonce) where the `sessionKey` is used for encryption
13. Machine A receives this message and verifies that the message was successfully received

Guarantees

Since these messages are potentially originating from the untrusted world, this protocol doesn't validate whether machines are running valid PSec code. As a result, privileged adversaries may be able to corrupt one, or both, parties involved and steal their keys to forge or decrypt these Untrusted messages. Assuming that such privileged adversaries have not compromised the sending or receiving party, this protocol guarantees confidentiality and integrity of the data being sent. In particular, this protocol prevents Man-in-the-Middle network attackers from learning anything useful about the data. However, Man-in-the-Middle attackers are able to spoof incoming messages with a custom payload to a target state machine that appear to be coming from a state machine of their choice. Since PSec state machines don't differentiate where events are coming from, this has the same result as the attackers creating a new legitimate connection to a target state machine and sending

that custom payload with the Untrusted Send protocol, and therefore doesn't change the guarantees we provide. Once again, network adversaries can do a denial-of-service attack that cannot be prevented in any case.

3.3.8 Implementation Limitations

Currently, in order to correctly implement remote attestation, Intel SGX requires enclaves to be registered with the Intel IAS Server beforehand. During the remote attestation verification protocol, one of the signatures during the exchange needs to be verified by this server. In our current implementation, PSec code written by the programmer is compiled as part of the enclave which results in a different enclave measurement for different PSec programs. As a result, there is no way to pre-register our enclaves with the IAS server. As a result, we skip this registration and signature verification process for now, and leave automating the registration process as future work. We additionally use Intel's provided sample code for implementing remote attestation and it is important to note that Intel states that this code is not a production level implementation.

In our current implementation, our distributed hosts and the KPS can only handle one network request at a time. Additionally, we have not implemented multi-threading in our initial implementation and PSec programs proceed in an overall sequential fashion where state machines execute in a predefined order and requests are sent after previous requests complete. There is one effective thread executing in the overall system at any given moment in time.

Finally, in order to have optimal compatibility with the Intel's internal SGX encryption/decryption libraries, we used the Intel's `sample_libcrypto` for cryptographic operations outside the enclave. This is the default library provided in Intel's sample applications, and Intel notes that this library is not at production level. We have not substituted this library for a different one yet.

3.4 Evaluation

We evaluate PSec in multiple dimensions. First, we give performance metrics on an initial implementation of our language and system (located at `https://github.com/ShivKushwah/PSec`). Then, we evaluate our system on 2 key examples to demonstrate the expressivity of our language. Finally, we give formal proofs of confidentiality and integrity properties provided by our system.

3.4.1 Performance

We present our performance results on an initial implementation of our system. We run our experiments by deploying the distributed hosts on Azure Confidential Compute Instances running on 3.7GHz Intel Xeon E-2176G Processors with SGX technology on Ubuntu 18.04.

We chose to rent 2 Standard DC2s_v2 instances (2 vcpus, 8 GiB memory) in the UK South region and connected the instances to the same virtual network. All SGX-code is running in `HARDWARE` mode rather than `SIMULATION` mode.

We present our initial results for different operations below. It is important to note that all measurements take place with the sender and receiver machine on different Azure hosts. For example, any send or create operation was measured by deploying different PSec machines on our 2 Azure instance and measuring the operation across the machines. In the Send cases, we sent the same payload across the different tests for consistency (in our case, this was the tuple `(1, "hello-world")`). Additionally, the KPS is hosted on one of the 2 Azure instances.

We outline our performance results in Table 3.1 and Table 3.2, and display them pictorially in Figure 3.4 and Figure 3.5. We took 5 samples for each configuration on May 25, 2020 and averaged the results. Trusted Create (SSM) involved an SSM creating another SSM. Untrusted Create (SSM) involved a USM creating an SSM (since the new SSM contacts the KPS to receive its own `Capability`, this takes approximately the same time as Trusted Create). Untrusted Create (USM) involved a USM creating another USM. Untrusted Send was measured from a USM to an SSM while Trusted Send was measured from an SSM to another SSM. Finally, since sending messages over a previously established connection saves overhead, we also measured subsequent message sending (denoted with an "again").

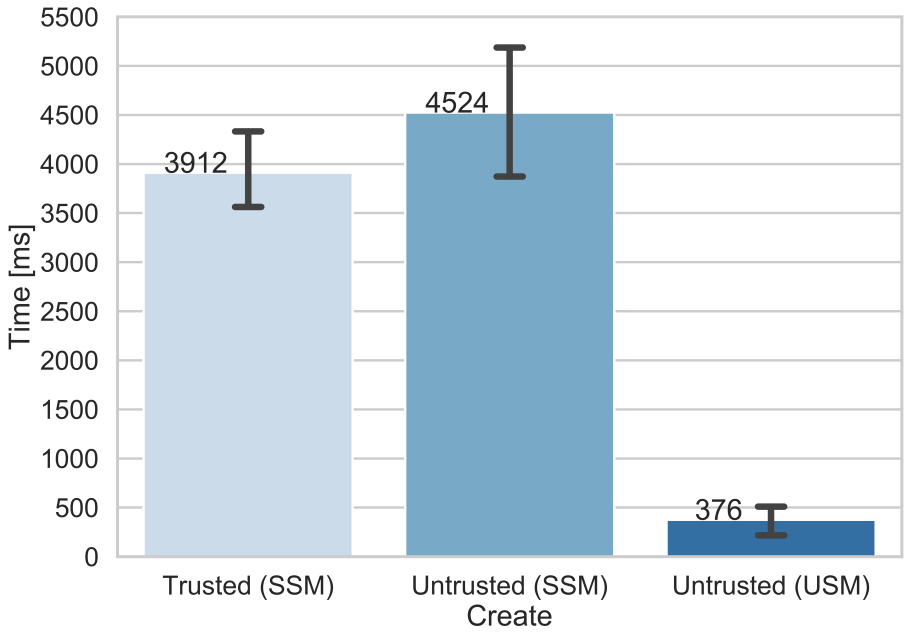

Figure 3.4: Create Performance Graph

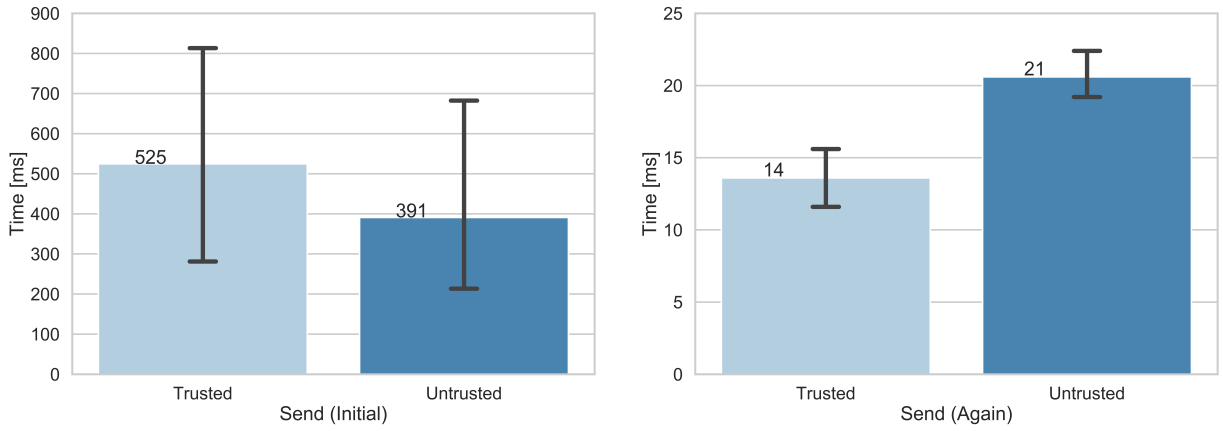

Figure 3.5: Send Performance Graph

	Create		
	Trusted (SSM)	Untrusted (SSM)	Untrusted (USM)
Average (in milliseconds)	3912	4524	376
Standard Deviation	482.9	898.8	193.5

Table 3.1: Create Performance Results

	Send (Initial)		**Send (Again)**	
	Trusted	Untrusted	Trusted	Untrusted
Average (in milliseconds)	525	391	14	21
Standard Deviation	349.7	328.9	2.61	2.07

Table 3.2: Send Performance Results

3.4.2 Expressivity

We showcase the expressivity of PSec by programming 2 key examples.

One Time Passcode Service Example

One Time Passcode (OTP) Services are often used in 2-Factor Authentication schemes where "something you know" such as a password is combined with "something you have", such as a cell phone. Not all implementations of 2-Factor Authentication are secure in practice (for example, attacks on schemes reliant on phone number verification have been demonstrated in recent years). Many secure implementations rely on using tamper resistant hardware tokens (such as the Feitian c100 OATH OTP Token) that establish a pre-shared secret with the Authenticating server. Upon authentication request, the user supplies their regular credentials as well as the OTP code computed by the hardware token (the code is a function of the pre-shared secret and time duration since the secret was provisioned on the token). Although these implementations provide stronger security guarantees, distributing these hardware tokens is often inconvenient in practice. Recent schemes such as the one proposed by Hoekstra et al. [16] replace these physical hardware tokens with Intel SGX (which is already present in modern computers with Intel CPUs). Intel SGX provides similar security guarantees because after an Intel SGX enclave is provisioned, attackers cannot easily conduct remote attacks and actually need access to the same physical machine as the user to generate valid OTP codes. We implement a version of an SGX-OTP service using the PSec language.

In order to test PSec, we implement a version of an SGX-OTP service, particularly a Bank 2-Factor Authentication example. In our system, we have a Bank Enclave SSM, a Client Web Browser USM, and a Client Enclave SSM. The Bank Enclave SSM handles authentication as well as creating and provisioning the Client Enclave SSM. The Client Web Browser USM is used by the user to perform the authentication request, and the Client Enclave SSM is used to generate the OTP codes. We deploy our system with the setup in Figure 3.6 where we have the Bank Enclave SSM on one distributed host and the Client Web Browser USM and Client Enclave SSM on another host.

We implement the system in PSec as in Figure 3.8 and Figure 3.9. The diagram's key is in Figure 3.7. There are 2 phases: the setup phase and the sign in phase. In the setup phase, the Client Web Browser USM authenticates with the Bank Enclave SSM and requests to enable 2-Factor. Upon successful authentication, the Bank Enclave SSM creates a Client Enclave SSM on the same distributed host as the Client Web Browser USM. It then provisions the newly created Client Enclave SSM with a master secret, thus ending the setup phase. In the sign in phase, the Client Web Browser USM wants to authenticate with the Bank Enclave SSM. In order to do so, it first sends a request to the Client Enclave SSM (located on the same distributed host) with its credentials and receives an OTP code in return. It is important to note that the Client Enclave SSM computes this OTP code as a function of the input credential, so an OTP code is generated regardless of the correctness of the input credential. After receiving this OTP code, the Client Web Browser USM forwards this OTP code along with its credential to the Bank Enclave SSM. The Bank Enclave SSM sends back either Auth Failure or Auth Success and the client is either successfully logged in or redirected to attempt to login again.

Although we were able to write up this example in PSec and execute it successfully, we

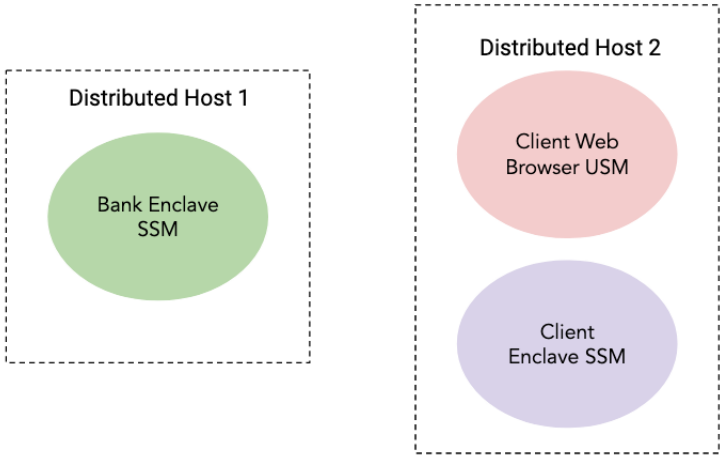

Figure 3.6: OTP System Diagram

outline some shortcomings below. First of all, although PSec enables us to create this secure distributed system, it requires us to know all of the machines in the system beforehand. There is no way to dynamically add new host machines to the system once it has started executing (such as new Client host machines for potentially new customers). In addition to this, in order to make this example more secure, the Bank Enclave SSM should only take local messages from the Client Web Browser USM and block all messages from any other parties. Since PSec state machines don't distinguish where incoming Untrusted Events are coming from, it is not possible to provide this protection in this version of our implementation. We leave implementing these features as part of our future work in Section 4.1.

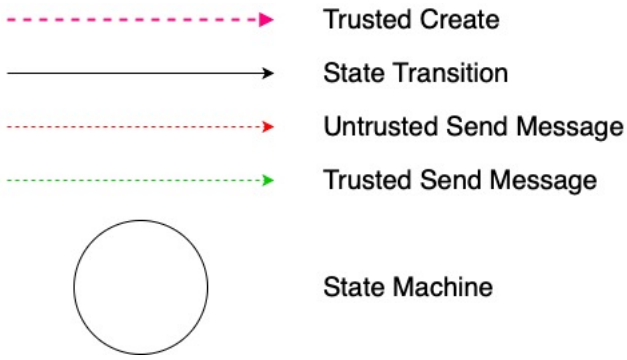

Figure 3.7: Diagram Key

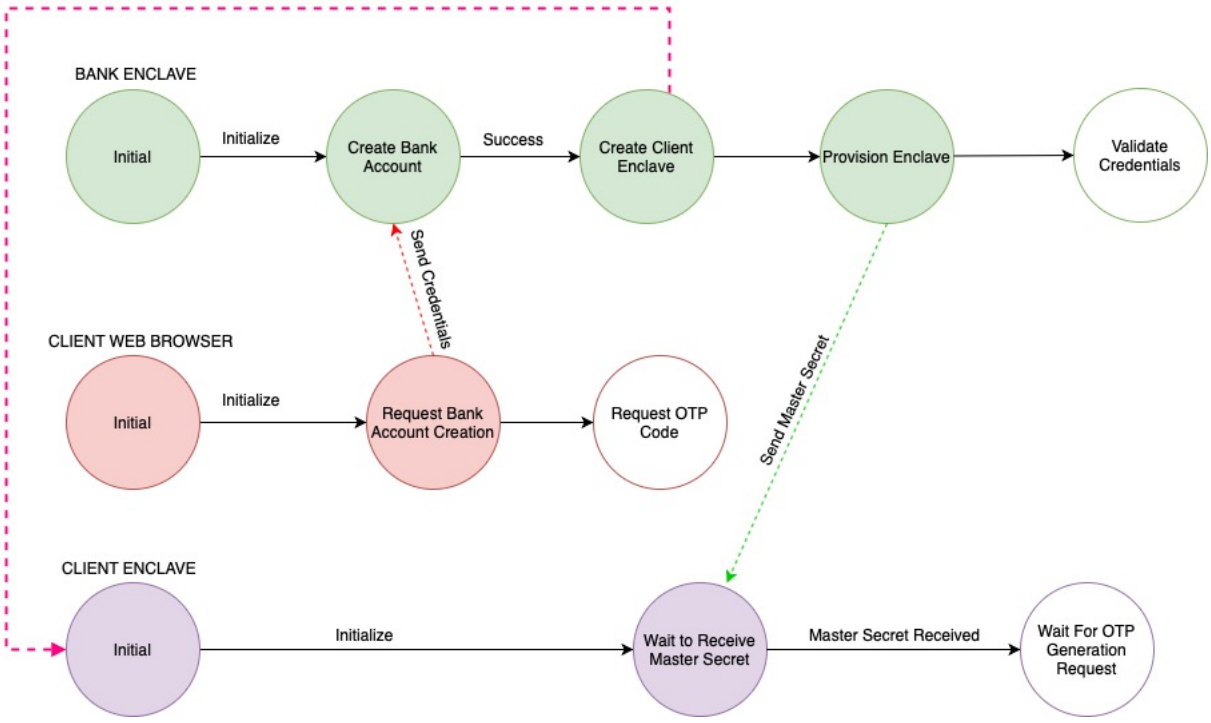

Figure 3.8: OTP PSec Setup Phase

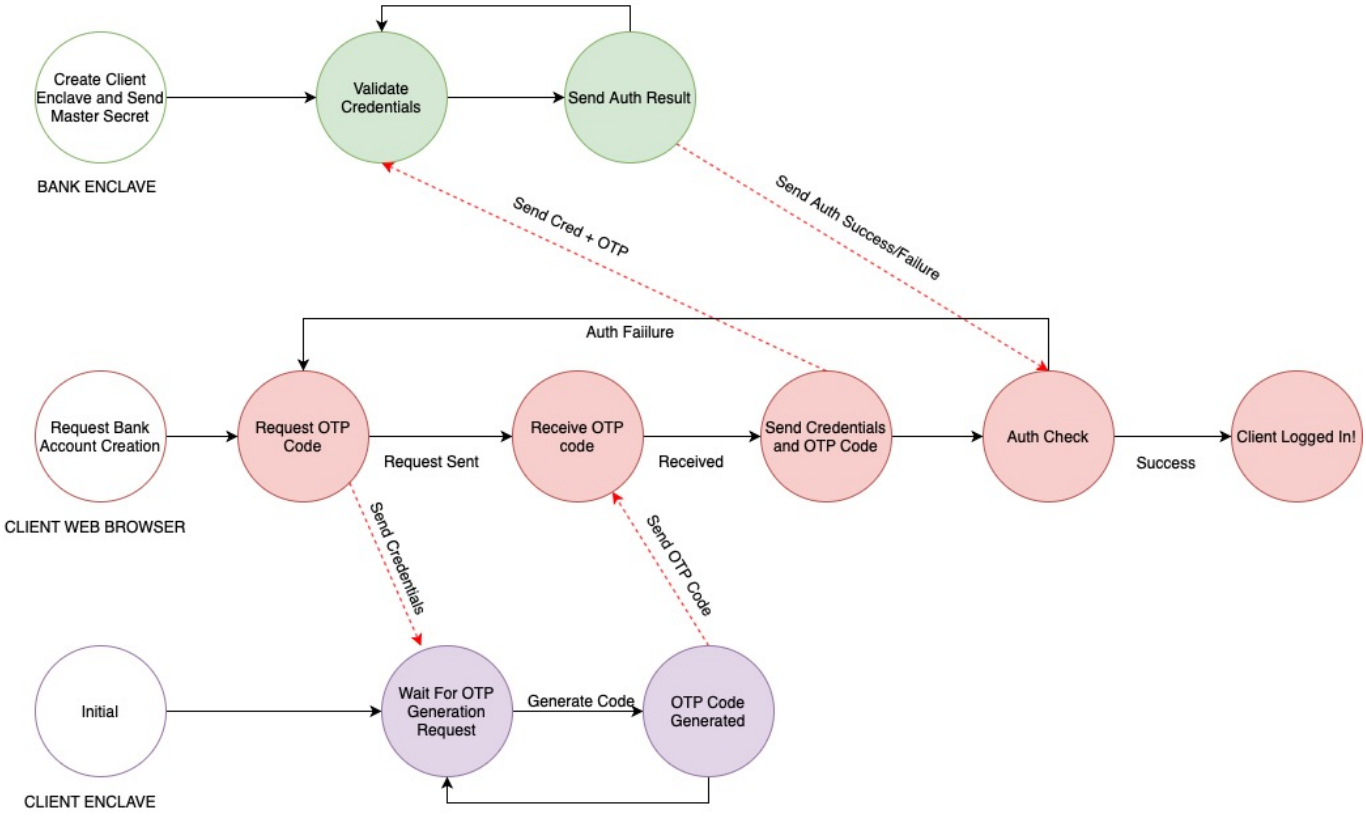

Figure 3.9: OTP PSec Sign In Phase

Civitas Secure Voting Example

The Civitas Secure Voting System is an electronic voting system designed by Cornell University that claims to be the first voting system to provide coercion resistance, voter verifiability, and be suitable for use in remote voting [5]. This system consists of an Election Supervisor and Bulletin Board as well as Ballot Boxes, Registration Tellers, and Tabulation Tellers. The Election Supervisor starts and stops the election, as well as performs other administrative tasks. The Ballot Boxes collect votes by storing them in tamper-evident logs and submit them to Tabulation Tellers during the Vote Counting Phase. The Tabulation Tellers eliminate duplicate votes and votes submitted with incorrect credentials, and the final vote list is sent to the Bulletin Board, which displays the results of the election. The Civitas design relies on utilizing various cryptography schemes to ensure that the various machines in the system do not tamper with election data and correctly perform operations such as vote duplication removal and vote counting.

We choose to implement a version of Civitas using PSec. Since in PSec we have the notion of Secure State Machines that run on trusted hardware, we can achieve similar guarantees with a streamlined implementation of the system. In our system, we have a Voter USM, Secure Voting Enclave SSM, Secure Supervisor SSM, Ballot Box SSM, Append Only Log SSM, Bulletin Board SSM, and Tabulation Teller SSM. We deploy the Voter USM and Secure Voting Enclave SSM on the same distributed host, and the rest of the machines on a different distributed host, as in Figure 3.10. We implement both the Voting phase and the Vote Counting phase and choose to not implement the Voter Registration phase and assume registration is done out-of-band (through a physical registration teller), as recommended by the Civitas paper. We depict these two phases in Figure 3.11 and Figure 3.12. The key is once again located at Figure 3.7.

Initially, the Secure Supervisor SSM creates all of the required SSMs to record and count votes (the Ballot Box SSM, the Bulletin Board SSM, and the Tabulation Teller SSM). The Voter USM sends a request to the Secure Supervisor SSM so that it can create and provision a Secure Voting Enclave SSM on the same distributed host as the Voter USM. In the Voting Phase, the Voter USM submits a (Vote, Credential) tuple to the Secure Voting Enclave SSM. The Secure Voting Enclave SSM sends this vote and credential to the Ballot Box SSM, which records this tuple by sending it to its Append Only Log SSM. In order to provide protections against voter coercion, the submitted credential may be a fake credential and the vote will be recorded for now, but discarded during the vote counting phase. The Vote Counting phase is initiated by the Secure Supervisor SSM by notifying the Ballot Box SSM that the voting period has ended. The Ballot Box SSM retrieves the vote log from the Append Only Log SSM and forwards it to the Tabulation Teller SSM. The Tabulation Teller SSM removes duplicate votes and eliminates votes with fake credentials, and forwards these results to the Bulletin Board SSM. Here, the votes are tallied and the election winner as well as a list of credentials that have been used to vote are stored. The Secure Voting Enclave SSM then queries this Bulletin Board SSM to verify that its vote was counted and sends the election winner to the Voter USM.

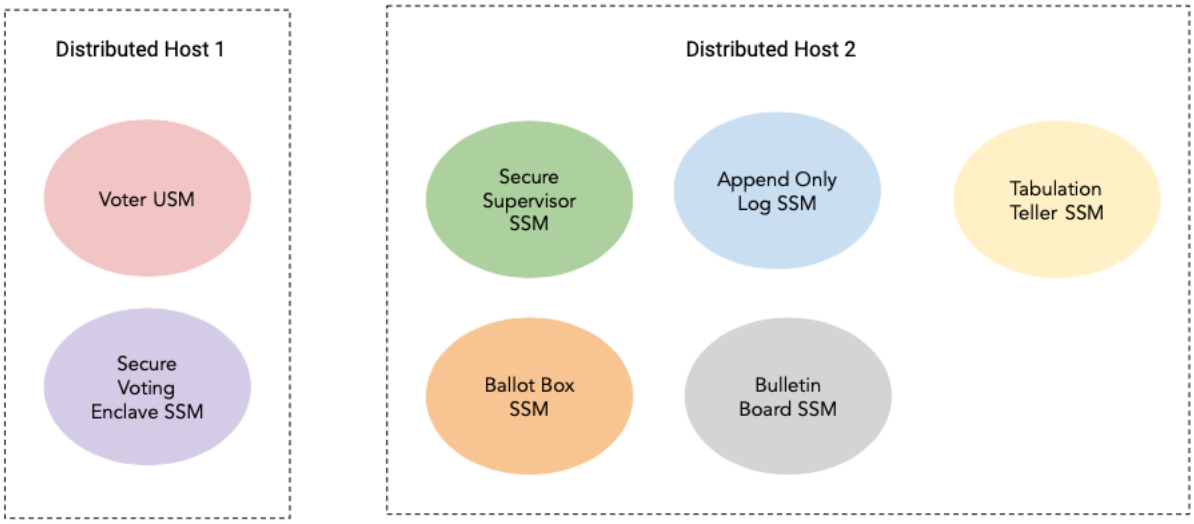

Figure 3.10: Civitas System Diagram

Our implementation provides many of the same guarantees as the original Civitas paper. However, we provide a weaker coercion resistance property. The original Civitas paper assumes the existence of anonymous channels (channels where the adversary is unable to snoop and cannot tell if any particular host machine has sent any data across). Since we aren't able to create these anonymous channels in PSec (we provide secure channels, not necessarily anonymous channels which can be implemented through mesh networks like TOR), we would not be able to provide these strong guarantees in any case. We guarantee a weaker form of coercion resistance where the voter can supply a fake credential to a coercer and still submit a vote with their real credentials. In this case, the coercer will submit a vote with incorrect credentials and will not be able to determine any wrongdoing until election results are finally released, at which point they will realize their vote was not counted. At this point, it will be too late for the coercer to change the outcome of the election, but we would ideally like to prevent the coercer from learning this information at any point of time. We leave this extension of our Civitas implementation as future work. As stated before, PSec doesn't allow for dynamically adding new host machines to the system (such as new Voter host machines), and we also leave this for future work.

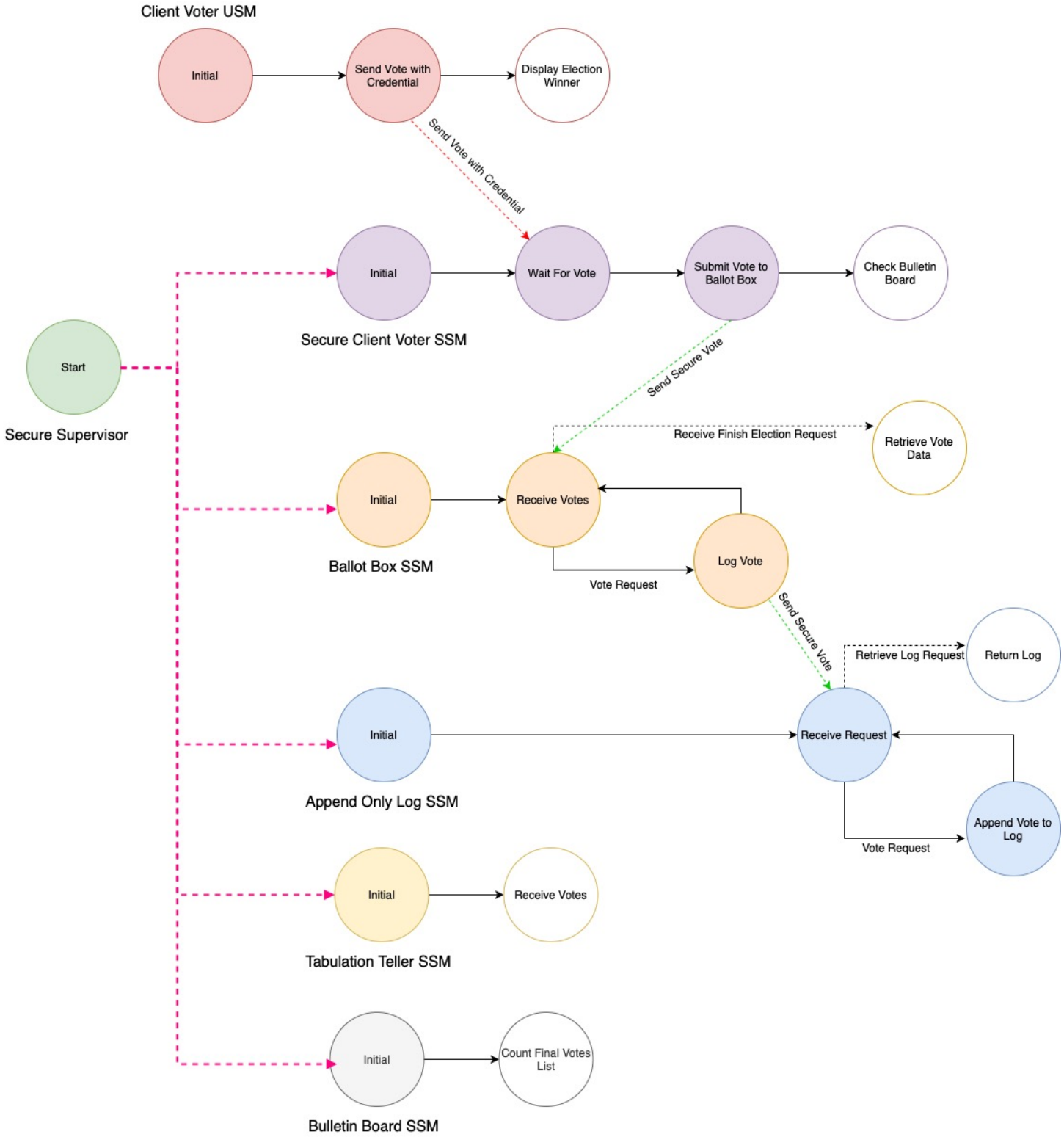

Figure 3.11: Civitas PSec Voting Phase

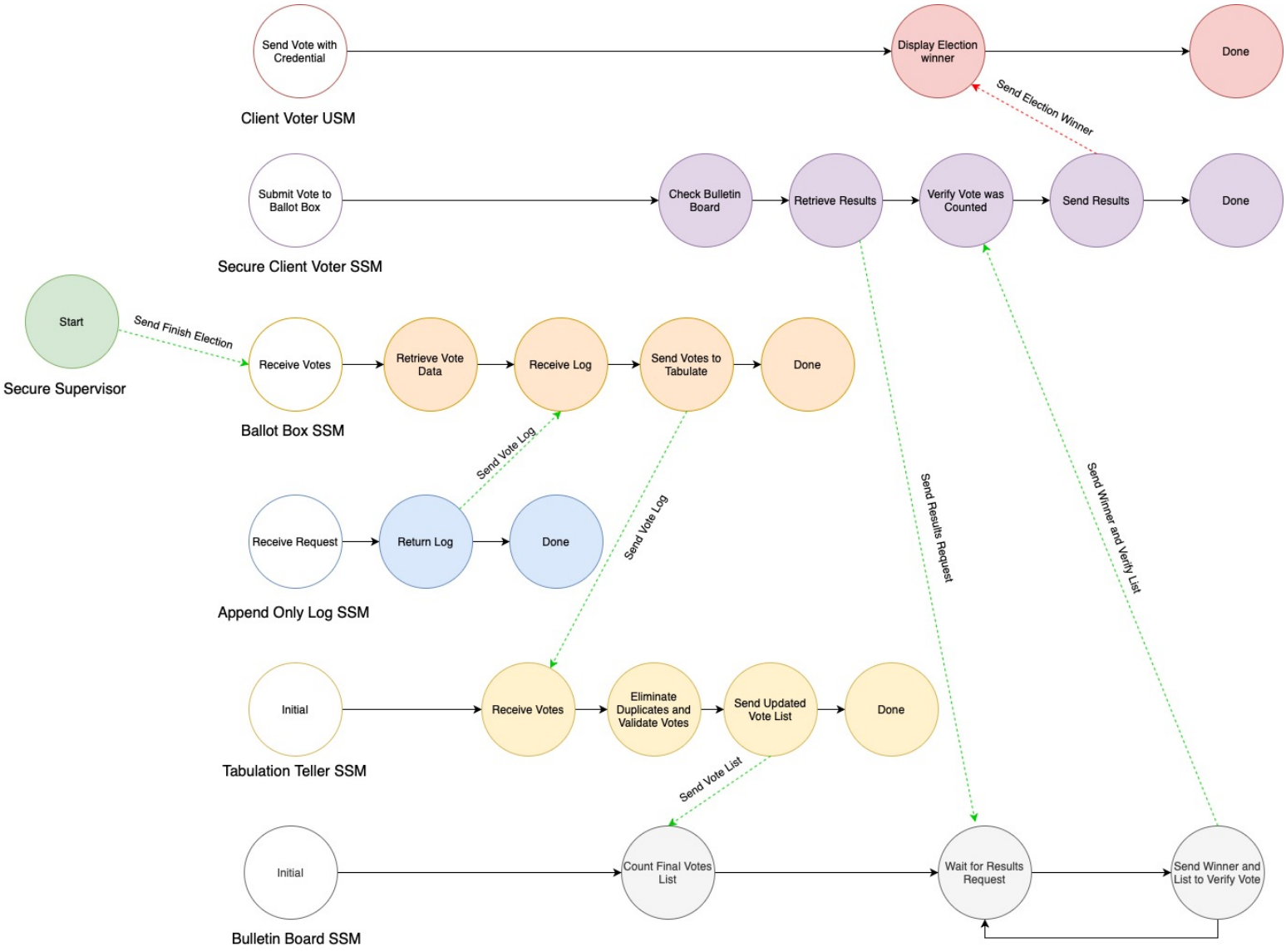

Figure 3.12: Civitas PSec Vote Counting Phase

3.4.3 Observational Determinism Security Proofs

In this section, we want to formally prove Theorem 1. Recall that observational determinism [37, 7] is a property that, if satisfied, prevents adversaries from inferring sensitive information from the execution of the program and corrupting trusted values in our system. The proof can be divided into independent confidentiality and integrity components and using the formalisms we have shown thus far, we can prove that PSec gives us these formal guarantees.

Note that this section contains necessary definitions, setup, and proof sketches. The full confidentiality proof is included in the Appendix, in Section A.1, and the full integrity proof is in Section A.2.

Configuration and Traces Definition

We will refer to the configuration of our system as G^k, which is equivalent to the (S^k, B^k, C^k) mentioned earlier. A trace of our program (similar to a trace in P [9]) is a finite sequence $G^0 \xrightarrow{a_0} ... \xrightarrow{a_{n-1}} G^n$ for $n \in \mathbb{N}$ such that $G^i \xrightarrow{a_i} G^{i+1}$ for each $i \in [0, n)$.

Proof of Confidentiality

Observational Equivalence Definition Observational ($\mathcal{L}$) Equivalence is used to denote that 2 values appear the same to a low-level observer/adversary [37, 7]. For example, in the case of 2 different H values, the observer will not be able to distinguish them (i.e. they are observationally equivalent). However, the observer will be able to tell 2 different L values apart (i.e. these values are not observationally equivalent). We define the observational equivalence rule for primitive values below:

$$v_1 \approx_{\mathcal{L}} v_2 \;\Leftrightarrow\; \Gamma \vdash v_i : \tau \;\wedge\; (\tau = L \;\Rightarrow\; v_1 = v_2)$$

Generalizing this rule, we have the following,
For states:

$$S_1 \approx_{\mathcal{L}} S_2 \;\Leftrightarrow\; uvals(S_1) = uvals(S_2)$$

For buffers:

$$B_1 \approx_{\mathcal{L}} B_2 \;\Leftrightarrow\; filter(B_1,\; \lambda(e_1, v_1).\; e_1 \in \mathcal{E}_{\mathcal{U}}) \;= filter(B_2,\; \lambda(e_2, v_2).\; e_2 \in \mathcal{E}_{\mathcal{U}})$$

For machine instance maps:

$$C_1 \approx_{\mathcal{L}} C_2 \;\Leftrightarrow\; \forall m \in \mathcal{M}.\; C_1[m] = C_2[m]$$

For configurations:

$$G_1 \approx_{\mathcal{L}} G_2 \;\Leftrightarrow\; (S_1 \approx_{\mathcal{L}} S_2) \;\wedge\; (B_1 \approx_{\mathcal{L}} B_2) \;\wedge\; (C_1 \approx_{\mathcal{L}} C_2)$$

For traces:

$$\pi_1 \approx_{\mathcal{L}} \pi_2 \;\Leftrightarrow\; \forall i \in [0, n).\; G_1^i \approx_{\mathcal{L}} G_2^i$$

Observation Function Definition We define an Observation function ($Obs_{\mathcal{L}}$) that maps transitions in our system to labels that indicate which information is leaked to our adversary. For example, in an Untrusted Send transition, $Obs_{\mathcal{L}}(G^t, G^{t+1}) = < (m_s, n_s), (m_r, n_r), e, v >$ since the sending machine identity, receiving machine identity, and message payload are assumed to be inferable by the adversary during this transition. We specify all of the labels associated with transitions from Section 3.2.2 to Section 3.2.2.

Confidentiality Proof Sketch The purpose of this proof is show that PSec prevents adversaries from inferring sensitive information from the execution of the program. A program satisfies the confidentiality property of observational determinism if for every pair of traces of the system, given that the initial configurations are observationally equivalent and the untrusted operations performed at every step are identical, then the traces are both observationally equivalent. In our case, we will be using our Observation function as a way to measure the operations performed at each step. Assume that $\mathcal{P}$ is the set of all possible PSec traces that are derived from programs that have passed our type checker. Stated formally, we need to prove the following for our system:

$$\forall \pi_1, \pi_2 \in \mathcal{P},\ n \in \mathbb{N},\ \pi_1 = G_1^0 \xrightarrow{a_0} ... \xrightarrow{a_{n-1}} G_1^n,\ \pi_2 = G_2^0 \xrightarrow{a_0} ... \xrightarrow{a_{n-1}} G_2^n.$$

$$(G_1^0 \approx_{\mathcal{L}} G_2^0)\ \wedge\ (Obs_{\mathcal{L}}(\pi_1) = Obs_{\mathcal{L}}(\pi_2)) \Rightarrow\ \pi_1 \approx_{\mathcal{L}} \pi_2$$

We outline the entirety of this proof in the Appendix, in Section A.1. In terms of a brief proof sketch, we start with a proof by induction.

Base Case $G_1^0 \approx_{\mathcal{L}} G_2^0$

Inductive Case Assume that there exists a k such that $G_1^k \approx_{\mathcal{L}} G_2^k$

Inductive Step We need to prove that if $Obs_{\mathcal{L}}(G_1^k, G^{k+1}{}_1) = Obs_{\mathcal{L}}(G_2^k, G^{k+1}{}_2)$, then $G^{k+1}{}_1 \approx_{\mathcal{L}} G^{k+1}{}_2$. There are multiple places where this condition holds, and we can continue via a proof by cases. We outline the remainder of this proof in Section A.1.

Proof of Integrity

Trusted Equivalence Definition In direct contrast to Observational Equivalence, Trusted ($\mathcal{H}$) Equivalence is concerned with the equivalence of trusted values in our system. We define the trusted equivalence rule for primitive values below:

$$v_1 \approx_{\mathcal{H}} v_2\ \Leftrightarrow\ \Gamma \vdash v_i : \tau\ \wedge\ (\tau = H\ \Rightarrow\ v_1 = v_2)$$

Generalizing this rule, we have the following,

For states:

$$S_1 \approx_{\mathcal{H}} S_2 \;\Leftrightarrow\; tvals(S_1) = tvals(S_2)$$

For buffers:

$$B_1 \approx_{\mathcal{H}} B_2 \;\Leftrightarrow\; filter(B_1,\ \lambda(e_1, v_1).\ e_1 \in \mathcal{E}_{\mathcal{T}}) \;= filter(B_2,\ \lambda(e_2, v_2).\ e_2 \in \mathcal{E}_{\mathcal{T}})$$

For machine instance maps:

$$C_1 \approx_{\mathcal{H}} C_2 \;\Leftrightarrow\; \forall m \in \mathcal{M}.\ C_1[m] = C_2[m]$$

For configurations:

$$G_1 \approx_{\mathcal{H}} G_2 \;\Leftrightarrow\; (S_1 \approx_{\mathcal{H}} S_2) \;\wedge\; (B_1 \approx_{\mathcal{H}} B_2) \;\wedge\; (C_1 \approx_{\mathcal{H}} C_2)$$

For traces:

$$\pi_1 \approx_{\mathcal{H}} \pi_2 \;\Leftrightarrow\; \forall i \in [0, n).\ G_1^i \approx_{\mathcal{H}} G_2^i$$

Trusted Observation Function Definition We define a Trusted Observation function ($Obs_{\mathcal{H}}$) that can be invoked by trusted parties to map trusted transitions in our system to labels that reveal changes in trusted state. This function is explicitly only defined for trusted transitions since are the only transitions that can modify trusted state. We outline the mappings below:

1. Trusted Create: $Obs_{\mathcal{H}}(G^t, G^{t+1}) \;= < (m_p, n_p), (m_c, n_c) >$
2. Trusted Send: $Obs_{\mathcal{H}}(G^t, G^{t+1}) \;= < (m_s, n_s), (m_r, n_r), e, v >$
3. Local Transition: $Obs_{\mathcal{H}}(G^t, G^{t+1}) \;= < tvals(s') >$
4. Dequeue Event: $Obs_{\mathcal{H}}(G^t, G^{t+1}) \;= < tvals(s'), e,\ (vje \in \mathcal{E}_{\mathcal{T}}) >$

Integrity Proof Sketch The purpose of this proof is to show that PSec preserves the integrity of trusted values. In essence, we need to show that adversaries are not able to corrupt trusted values with malicious payloads in our system. Assume that $\mathcal{P}$ is the set of all possible PSec traces that are derived from programs that have passed our type checker. Stated formally, we need to prove the following for our system:

$$\forall \pi_1, \pi_2 \in \mathcal{P},\ n \in \mathbb{N},\ \pi_1 = G_1^0 \xrightarrow{a_0} ... \xrightarrow{a_{n-1}} G_1^n,\ \pi_2 = G_2^0 \xrightarrow{a_0} ... \xrightarrow{a_{n-1}} G_2^n.$$

$$(G_1^0 \approx_{\mathcal{H}} G_2^0) \;\wedge\; (Obs_{\mathcal{H}}(\pi_1) = Obs_{\mathcal{H}}(\pi_2)) \Rightarrow\; \pi_1 \approx_{\mathcal{H}} \pi_2$$

We outline the entirety of this proof in the Appendix, in Section A.2. In terms of a brief proof sketch, we start with a proof by induction.

Base Case $G_1^0 \approx_{\mathcal{H}} G_2^0$

Inductive Case Assume that there exists a k such that $G_1^k \approx_{\mathcal{H}} G_2^k$

Inductive Step We need to prove that if $Obs_{\mathcal{H}}(G_1^k, G^{k+1}{}_1) = Obs_{\mathcal{H}}(G_2^k, G^{k+1}{}_2)$, then $G^{k+1}{}_1 \approx_{\mathcal{H}} G^{k+1}{}_2$. There are multiple places where this condition holds, and we can continue via a proof by cases. We outline the remainder of this proof in Section A.2.

Chapter 4

Conclusion

In this thesis, we presented PSec, a programming language for secure distributed systems. We augmented the P programming language and added various language constructs to abstract various security protocols from the programmer and enable them to write high-level code for these systems. We designed our language to incorporate both Secure State Machines and Untrusted State Machines and created a trust designation mechanism. We finally augmented our language with an Information Flow Control type system to prevent sensitive data from being inadvertently leaked. We presented initial performance results of an implementation of our system and showcased the expressivity of our language by implementing a secure One Time Passcode Service and a version of the Civitas Secure Voting system. We also provided a formal proof formalizing the security guarantees of our system.

4.1 Future Work

We outlined the limitations of our language implementation in Section 3.3.8 and leave it for future work. We also would like to implement an additional layer of message integrity and confidentiality at the host machine level. As in the Untrusted Create case, these messages are sent across the network in plaintext and easy to modify by Man-In-The-Middle attackers. A potential idea to combat this problem would be to have IP address lookups at the KPS return the network address as well as the public key for that distributed host machine. In this case, network requests between host machines would have an additional layer of encryption and integrity (layering on top of encryption based on the receiving state machine's public `Identity` key). Another item mentioned previously is that PSec doesn't allow us to dynamically add new host machines to the system once it has started executing. Some changes we would need to make to enable this is to allow for the registration of new host machines in the KPS. We would also need to create PSec constructs that allow new host machines to query the KPS for the IP addresses/ports of all existing machines in order to automatically generate/retrieve PSec handles as well as constructs to provision a new PSec state machine on a distributed host of choice. In addition to this, there are some

potential use cases where accurately determining the origin of a particular PSec event for an Untrusted Send is important (outlined in the OTP example in Section 3.4.2). This will require adding new language constructs and modifying the Untrusted Send protocol. We have not implemented these features yet and leave this as potential future work.

Another future direction is to more directly verify security properties of PSec programs. This can be accomplished by translating these programs directly into a formal modeling and verification language such as UCLID5 [27] (located at `https://github.com/uclid-org/uclid`). If an automatic UCLID5 translation is achieved, we can more readily verify confidentiality and integrity properties of our PSec programs.

A more ambitious future direction for the PSec language would be to completely abstract the concept of Secure State Machines from the programmer. In this approach, programmers would simply designate certain variables as `secure` variables, and rest assured that the contents of these variables would not be stored in untrusted memory. The compiler would automatically infer which parts of the code are sensitive based on the aforementioned designation, and run those selective parts within enclave memory. If `secure` variables are sent from one state machine to another, they would actually be sent from the enclave from the first system to the enclave of the second system, ensuring they never leave trusted memory. Regular data would be stored outside the enclave on the host machines as before. This approach would enable an even higher of abstraction from the programmer and we leave this as a potential avenue of future work.

Appendix A

Additional

A.1 PSec Confidentiality Proof

We need to prove the following for our system:

$$\forall \pi_1, \pi_2 \in \mathcal{P},\ n \in \mathbb{N},\ \pi_1 = G_1^0 \xrightarrow{a_0} ... \xrightarrow{a_{n-1}} G_1^n,\ \pi_2 = G_2^0 \xrightarrow{a_0} ... \xrightarrow{a_{n-1}} G_2^n.$$
$$(G_1^0 \approx_{\mathcal{L}} G_2^0)\ \wedge\ (Obs_{\mathcal{L}}(\pi_1) = Obs_{\mathcal{L}}(\pi_2)) \Rightarrow\ \pi_1 \approx_{\mathcal{L}} \pi_2$$

We can prove this via induction and then a proof by cases.

Base Case $G_1^0 \approx_{\mathcal{L}} G_2^0$

Inductive Case Assume that there exists a k such that $G_1^k \approx_{\mathcal{L}} G_2^k$

Inductive Step We need to prove that if $Obs_{\mathcal{L}}(G_1^k, G^{k+1}{}_1) = Obs_{\mathcal{L}}(G_2^k, G^{k+1}{}_2)$, then $G^{k+1}{}_1 \approx_{\mathcal{L}} G^{k+1}{}_2$. There are multiple places where this condition holds, and we can continue via a proof by cases:

1. Create Transitions:
   - $Obs_{\mathcal{L}}(G_1^k, G^{k+1}{}_1) = Obs_{\mathcal{L}}(G_2^k, G^{k+1}{}_2) = < (m_p, n_p), (m_c, n_c) >$
   - For both Trusted Create and Untrusted Create transitions, the parent machine identity and receiving machine identity is assumed to be observable.
   - Therefore, for the Observation function for the two configurations to be equivalent, these must be identical in the corresponding transitions.
   - Since PSec has deterministic transitions, this would yield the same effect on both G_1^k and G_2^k (S^k and C^k change in the same way for both configurations while B^k remains the same).
   - Therefore, given that $G_1^k \approx_{\mathcal{L}} G_2^k$, then $G^{k+1}{}_1 \approx_{\mathcal{L}} G^{k+1}{}_2$ in this case.

2. Trusted Send Transition:
   - $Obs_{\mathcal{L}}(G_1^k, G^{k+1}{}_1) = Obs_{\mathcal{L}}(G_2^k, G^{k+1}{}_2) = < (m_s, n_s), (m_r, n_r), e >$
   - On a Trusted Send Transition, the sending machine identity, receiving machine identity, and event type is assumed to be observable.
   - Therefore, for the Observation function for the two configurations to be equivalent, these must be identical in the corresponding transitions.
   - The non-observable difference between the two transitions is the message payload in this case. Since the message payload must consist of secure, trusted data, the receiving machine adds the payload to its trusted local state, thus changing its $tvals$. After this transition takes place, $uvals(G^{k+1}{}_1) = uvals(G^{k+1}{}_2)$ regardless of the message payload sent in the 2 transitions. This means $S^{k+1}{}_1 \approx_{\mathcal{L}} S^{k+1}{}_2$.
   - Since the event that is enqueued in the buffer of the receiving machine is a trusted event, $B^{k+1}{}_1 \approx_{\mathcal{L}} B^{k+1}{}_2$. C^k remains unchanged for both configurations.
   - Therefore, given that $G_1^k \approx_{\mathcal{L}} G_2^k$, then $G^{k+1}{}_1 \approx_{\mathcal{L}} G^{k+1}{}_2$ in this case.
3. Untrusted Send Transition:
   - $Obs_{\mathcal{L}}(G_1^k, G^{k+1}{}_1) = Obs_{\mathcal{L}}(G_2^k, G^{k+1}{}_2) = < (m_s, n_s), (m_r, n_r), e, v >$
   - For a Untrusted Send Transition, the sending machine identity, receiving machine identity, event type, and message payload is assumed to be observable.
   - Therefore, for the Observation function for the two configurations to be equivalent, these must be identical in the corresponding transitions.
   - Since PSec has deterministic transitions, this would yield the same effect on both G_1^k and G_2^k (S^k and B^k change in the same way for both configurations while C^k remains unchanged).
   - Therefore, given that $G_1^k \approx_{\mathcal{L}} G_2^k$, then $G^{k+1}{}_1 \approx_{\mathcal{L}} G^{k+1}{}_2$ in this case.
4. Local Transition:
   - $Obs_{\mathcal{L}}(G_1^k, G^{k+1}{}_1) = Obs_{\mathcal{L}}(G_2^k, G^{k+1}{}_2) = < uvals(s') >$
   - For a Local Transition, the untrusted values stored in the local state of the machine are assumed to be observable (since they can be leaked through side channels).
   - Therefore, for the Observation function for the two configurations to be equivalent, these must be identical in the corresponding transitions.
   - For USMs, since the entirety of their state is untrusted, this would mean that the USMs must undergo the exact same transition in both configurations. For SSMs, this only applies if they only modify their untrusted state in this transition, which would additionally imply that the same transition must take place in both configurations. Since PSec has deterministic transitions, this would yield the same

effect on both G_1^k and G_2^k (S^k changes in the same way for both configurations while B^k and C^k remain unchanged).

- For SSMs, the non-observable difference between the two transitions would be changes in trusted state. This means that the SSM may modify its *tvals*. After this transition takes place, $uvals(G^{k+1}{}_1) = uvals(G^{k+1}{}_2)$ since these values are not modified, so $S^{k+1}{}_1 \approx_{\mathcal{L}} S^{k+1}{}_2$. In this case, B_k and C_k remain unchanged.
- Therefore, given that $G_1^k \approx_{\mathcal{L}} G_2^k$, then $G^{k+1}{}_1 \approx_{\mathcal{L}} G^{k+1}{}_2$ in this case.

5. Dequeue Transition:

- $Obs_{\mathcal{L}}(G_1^k, G^{k+1}{}_1) = Obs_{\mathcal{L}}(G_2^k, G^{k+1}{}_2) = < uvals(s'), e, \ (vje \in \mathcal{E}_{\mathcal{U}}) >$
- For a Dequeue Transition, the untrusted values stored in the local state as well as the newly dequeued value (if it is non-sensitive) of the machine are assumed to be observable.
- Therefore, for the Observation function for the two configurations to be equivalent, these must be identical in the corresponding transitions.
- For USMs, since the entirety of their state is untrusted and they can only receive and dequeue Untrusted Events, this would mean that the USMs must undergo the exact same transition in both configurations. For SSMs, this only applies if they are dequeuing an Untrusted Event, which would additionally mean that the transition must be the same across the 2 configurations. Since PSec has deterministic transitions, this would yield the same effect on both G_1^k and G_2^k (S^k and B^k change in the same way for both configurations while C^k remains unchanged).
- For SSMs, the non-observable difference between the two transitions would be the dequeuing of Trusted Events. The SSM may receive sensitive data through the Trusted Event, thus modifying its trusted state and appending values to its *tvals*. After this transition takes place, $uvals(G^{k+1}{}_1) = uvals(G^{k+1}{}_2)$ regardless of the Trusted Event type and payload sent in the 2 transitions, so $S^{k+1}{}_1 \approx_{\mathcal{L}} S^{k+1}{}_2$. Since the Untrusted Events in the buffer are not modified in both configurations, $B^{k+1}{}_1 \approx_{\mathcal{L}} B^{k+1}{}_2$. C_k in this case remains unchanged.
- Therefore, given that $G_1^k \approx_{\mathcal{L}} G_2^k$, then $G^{k+1}{}_1 \approx_{\mathcal{L}} G^{k+1}{}_2$ in this case.

Since we have proven this for all cases, we have proven our initial statement through induction and PSec satisfies the confidentiality property of observational determinism.

A.2 PSec Integrity Proof

We need to prove the following for our system:

$$\forall \pi_1, \pi_2 \in \mathcal{P},\ n \in \mathbb{N},\ \pi_1 = G_1^0 \xrightarrow{a_0} ... \xrightarrow{a_{n-1}} G_1^n,\ \pi_2 = G_2^0 \xrightarrow{a_0} ... \xrightarrow{a_{n-1}} G_2^n.$$
$$(G_1^0 \approx_{\mathcal{H}} G_2^0)\ \wedge\ (Obs_{\mathcal{H}}(\pi_1) = Obs_{\mathcal{H}}(\pi_2)) \Rightarrow\ \pi_1 \approx_{\mathcal{H}} \pi_2$$

We can prove this via induction and then a proof by cases.

Base Case $G_1^0 \approx_{\mathcal{H}} G_2^0$

Inductive Case Assume that there exists a k such that $G_1^k \approx_{\mathcal{H}} G_2^k$

Inductive Step We need to prove that if $Obs_{\mathcal{H}}(G_1^k, G^{k+1}{}_1) = Obs_{\mathcal{H}}(G_2^k, G^{k+1}{}_2)$, then $G^{k+1}{}_1 \approx_{\mathcal{H}} G^{k+1}{}_2$. We can continue via a proof by cases over the various transitions:

1. Trusted Create Transition:
   - $Obs_{\mathcal{H}}(G_1^k, G^{k+1}{}_1) = Obs_{\mathcal{H}}(G_2^k, G^{k+1}{}_2) = < (m_p, n_p), (m_c, n_c) >$
   - The parent machine identity and receiving machine identity must be identical in the corresponding transitions.
   - Since PSec has deterministic transitions, this would yield the same effect on both G_1^k and G_2^k (S^k and C^k change in the same way for both configurations while B^k remains the same).
   - Therefore, given that $G_1^k \approx_{\mathcal{H}} G_2^k$, then $G^{k+1}{}_1 \approx_{\mathcal{H}} G^{k+1}{}_2$ in this case.
2. Trusted Send Transition:
   - $Obs_{\mathcal{L}}(G_1^k, G^{k+1}{}_1) = Obs_{\mathcal{L}}(G_2^k, G^{k+1}{}_2) = < (m_s, n_s), (m_r, n_r), e, v >$
   - The sending machine identity, receiving machine identity, event type, and event payload must be identical in the corresponding transitions.
   - Since PSec has deterministic transitions, this would yield the same effect on both G_1^k and G_2^k (S^k and B^k change in the same way for both configurations while C^k remains unchanged).
   - Therefore, given that $G_1^k \approx_{\mathcal{H}} G_2^k$, then $G^{k+1}{}_1 \approx_{\mathcal{H}} G^{k+1}{}_2$ in this case.
3. Local Transition:
   - $Obs_{\mathcal{L}}(G_1^k, G^{k+1}{}_1) = Obs_{\mathcal{L}}(G_2^k, G^{k+1}{}_2) = < tvals(s') >$
   - In a Local Transition, the state machine may change either its trusted local state or its untrusted local state.

- If trusted local state is changed, according to the condition above, it must change in the same way for both configurations. This implies that the same transition must occur in both configurations. Since PSec has deterministic transitions, this would yield the same effect on both G_1^k and G_2^k (S^k changes in the same way for both configurations while B^k and C^k remain unchanged).
- If untrusted local state is changed, the state machines may differ in how this state is modified across the two configurations. This means that the state machines may modify their *uvals* differently from each other. However, after this transition takes place, $tvals(G^{k+1}{}_1) = tvals(G^{k+1}{}_2)$ since these values are not modified, so $S^{k+1}{}_1 \approx_{\mathcal{L}} S^{k+1}{}_2$. In this case, B_k and C_k remain unchanged.
- Therefore, given that $G_1^k \approx_{\mathcal{H}} G_2^k$, then $G^{k+1}{}_1 \approx_{\mathcal{H}} G^{k+1}{}_2$ in this case.

4. Dequeue Transition:

- $Obs_{\mathcal{L}}(G_1^k, G^{k+1}{}_1) = Obs_{\mathcal{L}}(G_2^k, G^{k+1}{}_2) \ = < tvals(s'), e, \ (vje \in \mathcal{E}_{\mathcal{T}}) >$
- In a Dequeue Transition, an Untrusted Event or a Trusted Event can be dequeued from the input buffer.
- If a Trusted Event is dequeued, according to the condition above, the following trusted states of the state machines must be identical across the 2 configurations. Since Trusted Events can only contain trusted payloads, the trusted payloads must be identical which additionally means that the transition must be the same across the 2 configurations. Since PSec has deterministic transitions, this would yield the same effect on both G_1^k and G_2^k (S^k and B^k change in the same way for both configurations while C^k remains unchanged).
- If an Untrusted Event is dequeued, the state machines may differ in their following untrusted states. Since Untrusted Events contain untrusted payloads, the state machines may receive additional untrusted state and append values to their *uvals*, which don't need to be identical across the 2 configurations. However, after this transition takes place, $tvals(G^{k+1}{}_1) = tvals(G^{k+1}{}_2)$ regardless of the Untrusted Event type and untrusted payload sent in the 2 transitions, so $S^{k+1}{}_1 \approx_{\mathcal{H}} S^{k+1}{}_2$. Since the Trusted Events in the buffer are not modified in both configurations, $B^{k+1}{}_1 \approx_{\mathcal{H}} B^{k+1}{}_2$. C_k in this case remains unchanged.
- Therefore, given that $G_1^k \approx_{\mathcal{H}} G_2^k$, then $G^{k+1}{}_1 \approx_{\mathcal{H}} G^{k+1}{}_2$ in this case.

Since we have proven this for all cases, we have proven our initial statement through induction and PSec satisfies the integrity property of observational determinism.

A.3 PSec Sample Annotated Code

The following PSec sample code contains comments in regards to information flow analysis type checking.

```
trusted event Ping;

secure_machine ExampleMachine {
    var ssn : secure_int;
    var insecure_int : int;
    var s_int : secure_int;
    var s_handle : secure_machine_handle;
    var handle : machine_handle;

    start state Initial {
        entry {
            ssn = Endorse(6123456) as secure_int; // Valid

            if (ssn == (Endorse(6123456) as secure_int)) { // Valid
                s_int = Endorse(7) as secure_int;
            } else {
                s_int = Endorse(8) as secure_int;
            }

            s_handle = this; // Valid
            handle = Declassify(this) as machine_handle;

            s_int = s_int + (Endorse(1) as secure_int); // Valid
            insecure_int = 10;
            s_int = Endorse(insecure_int) as secure_int;

            while (insecure_int < 15) { // Valid
                s_int = s_int + (Endorse(1) as secure_int);
                insecure_int = insecure_int + 1;
            }

            while (s_int < (Endorse(20) as secure_int)) { // Valid
                s_int = s_int + (Endorse(1) as secure_int);
                send s_handle, Ping;
            }

            // Invalid Code Below

            // Invalid because value of insecure_int after this if statement
            // will leak whether ssn is 6123456 or not
            // if (ssn == (Endorse(6123456) as secure_int)) {
            //     insecure_int = 7;
            // } else {
            //     insecure_int = 8;
            // }

            // Invalid because value of ssn will be leaked by assigning it to a
            // non-secret variable
            // insecure_int = ssn;

            // Invalid because value of s_int will be compromised by looking at value
            // of insecure_int after execution of while statement
            // while (s_int < (Endorse(15) as secure_int)) {
            //     s_int = s_int + (Endorse(1) as secure_int);
            //     insecure_int = insecure_int + 1;
            // }
        }
    }
}
```

Bibliography

[1] Ferdinand Brasser et al. "Software Grand Exposure: SGX Cache Attacks Are Practical". In: *Proceedings of the 11th USENIX Conference on Offensive Technologies*. WOOT'17. Vancouver, BC, Canada: USENIX Association, 2017, p. 11.

[2] Guoxing Chen et al. "SgxPectre Attacks: Leaking Enclave Secrets via Speculative Execution". In: (Feb. 2018).

[3] Stephen Chong, K. Vikram, and Andrew C. Myers. "SIF: Enforcing Confidentiality and Integrity in Web Applications". In: *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*. SS'07. Boston, MA: USENIX Association, 2007. ISBN: 1113335555779.

[4] Stephen Chong et al. "Secure Web Applications via Automatic Partitioning". In: *SIGOPS Oper. Syst. Rev.* 41.6 (Oct. 2007), pp. 31–44. ISSN: 0163-5980. DOI: 10.1145/1323293.1294265. URL: https://doi.org/10.1145/1323293.1294265.

[5] M. R. Clarkson, S. Chong, and A. C. Myers. "Civitas: Toward a Secure Voting System". In: *2008 IEEE Symposium on Security and Privacy (sp 2008)*. May 2008, pp. 354–368. DOI: 10.1109/SP.2008.32.

[6] Michael Clarkson. *Lecture notes in Information Flow Analysis*. 2016. URL: http://www.cs.cornell.edu/courses/cs5430/2016sp/l/25-vsi/lec.pdf.

[7] Michael R. Clarkson and Fred B. Schneider. "Hyperproperties". In: *J. Comput. Secur.* 18.6 (Sept. 2010), pp. 1157–1210. ISSN: 0926-227X.

[8] Ankush Desai. "Modular and Safe Event-Driven Programming". PhD thesis. EECS Department, University of California, Berkeley, Jan. 2020. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-3.html.

[9] Ankush Desai et al. "Compositional Programming and Testing of Dynamic Distributed Systems". In: *ACM OOPSLA 2018*. Nov. 2018. URL: https://www.microsoft.com/en-us/research/publication/compositional-programming-and-testing-of-dynamic-distributed-systems/.

[10] Ankush Desai et al. "P: Safe Asynchronous Event-Driven Programming". In: *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2013).

[11] Cédric Fournet and Jeremy Planu. "Compiling Information-Flow Security to Minimal Trusted Computing Bases". In: *To appear in the proceedings of Programming Languages and Systems, 20th European Symposium on Programming, ESOP 2011.* Springer-Verlag, Mar. 2011. URL: https://www.microsoft.com/en-us/research/publication/compiling-information-flow-security-minimal-trusted-computing-bases/.

[12] Yangchun Fu et al. "Sgx-Lapd: Thwarting Controlled Side Channel Attacks via Enclave Verifiable Page Faults". In: Oct. 2017, pp. 357–380. ISBN: 978-3-319-66331-9. DOI: 10.1007/978-3-319-66332-6_16.

[13] Anitha Gollamudi and Stephen Chong. "Automatic Enforcement of Expressive Security Policies Using Enclaves". In: *SIGPLAN Not.* 51.10 (Oct. 2016), pp. 494–513. ISSN: 0362-1340. DOI: 10.1145/3022671.2984002. URL: https://doi.org/10.1145/3022671.2984002.

[14] Johannes Götzfried et al. "Cache Attacks on Intel SGX". In: *Proceedings of the 10th European Workshop on Systems Security.* EuroSec'17. Belgrade, Serbia: Association for Computing Machinery, 2017. ISBN: 9781450349352. DOI: 10.1145/3065913.3065915. URL: https://doi.org/10.1145/3065913.3065915.

[15] Daniel Gruss et al. "Strong and Efficient Cache Side-Channel Protection Using Hardware Transactional Memory". In: *Proceedings of the 26th USENIX Conference on Security Symposium.* SEC'17. Vancouver, BC, Canada: USENIX Association, 2017, pp. 217–233. ISBN: 9781931971409.

[16] Matthew Hoekstra et al. "Using Innovative Instructions to Create Trustworthy Software Solutions". In: *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy.* HASP '13. Tel-Aviv, Israel: Association for Computing Machinery, 2013. ISBN: 9781450321181. DOI: 10.1145/2487726.2488370. URL: https://doi.org/10.1145/2487726.2488370.

[17] *Intel® Software Guard Extensions.* URL: https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions.html.

[18] Jed Liu et al. "Fabric: A Platform for Secure Distributed Computation and Storage". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles.* SOSP '09. Big Sky, Montana, USA: Association for Computing Machinery, 2009, pp. 321–334. ISBN: 9781605587523. DOI: 10.1145/1629575.1629606. URL: https://doi.org/10.1145/1629575.1629606.

[19] *Microsoft Open Enclave SDK.* July 2019. URL: https://openenclave.io/sdk/.

[20] Olga Ohrimenko et al. "Oblivious Multi-Party Machine Learning on Trusted Processors". In: *Proceedings of the 25th USENIX Conference on Security Symposium.* SEC'16. Austin, TX, USA: USENIX Association, 2016, pp. 619–636. ISBN: 9781931971324.

[21] Marco Patrignani et al. "Secure Compilation to Protected Module Architectures". In: *ACM Trans. Program. Lang. Syst.* 37.2 (Apr. 2015). ISSN: 0164-0925. DOI: 10.1145/2699503. URL: https://doi.org/10.1145/2699503.

[22] *PS3 Security In Tatters*. https://www.eurogamer.net/articles/digitalfoundry-ps3-security-in-tatters. Accessed: 2020-03-15.

[23] A. Sabelfeld and A.c. Myers. "Language-based information-flow security". In: *IEEE Journal on Selected Areas in Communications* 21.1 (2003), pp. 5–19. DOI: 10.1109/jsac.2002.806121.

[24] Vasily A. Sartakov et al. "EActors: Fast and Flexible Trusted Computing Using SGX". In: *Proceedings of the 19th International Middleware Conference*. Middleware '18. Rennes, France: Association for Computing Machinery, 2018, pp. 187–200. ISBN: 9781450357029. DOI: 10.1145/3274808.3274823. URL: https://doi.org/10.1145/3274808.3274823.

[25] Sajin Sasy, Sergey Gorbunov, and Christopher Fletcher. "ZeroTrace : Oblivious Memory Primitives from Intel SGX". In: Jan. 2018. DOI: 10.14722/ndss.2018.23243.

[26] Michael Schwarz et al. "Malware Guard Extension: abusing Intel SGX to conceal cache attacks". In: *Cybersecurity* 3 (Dec. 2020). DOI: 10.1186/s42400-019-0042-y.

[27] Sanjit A. Seshia and Pramod Subramanyan. "UCLID5: Integrating Modeling, Verification, Synthesis and Learning". In: *Proceedings of the 16th ACM-IEEE International Conference on Formal Methods and Models for System Design*. MEMOCODE '18. Beijing, China: IEEE Press, 2018, pp. 1–10. ISBN: 9781538661956.

[28] Sanjit Seshia et al. "A Design and Verification Methodology for Secure Isolated Regions". In: *Programming Languages Design and Implementation (PLDI)*. ACM - Association for Computing Machinery, June 2016. URL: https://www.microsoft.com/en-us/research/publication/a-design-and-verification-methodology-for-secure-isolated-regions/.

[29] Ming-Wei Shih et al. "T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs". In: *Network and Distributed System Security Symposium 2017 (NDSS'17)*. Internet Society, Feb. 2017. URL: https://www.microsoft.com/en-us/research/publication/t-sgx-eradicating-controlled-channel-attacks-enclave-programs/.

[30] Shweta Shinde et al. "Preventing Page Faults from Telling Your Secrets". In: *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ASIA CCS '16. Xiapos;an, China: Association for Computing Machinery, 2016, pp. 317–328. ISBN: 9781450342339. DOI: 10.1145/2897845.2897885. URL: https://doi.org/10.1145/2897845.2897885.

[31] Rohit Sinha et al. "Moat: Verifying Confidentiality of Enclave Programs". In: *The ACM Conference on Computer and Communications Security (CCS)*. ACM - Association for Computing Machinery, Oct. 2015. URL: https://www.microsoft.com/en-us/research/publication/moat-verifying-confidentiality-of-enclave-programs/.

[32] Geoffrey Smith. "Principles of Secure Information Flow Analysis". In: *Advances in Information Security Malware Detection* (), pp. 291–307. DOI: 10.1007/978-0-387-44599-1_13.

[33] Lily Sturmann. *Current Trusted Execution Environment Landscape*. Dec. 2019. URL: https://next.redhat.com/2019/12/02/current-trusted-execution-environment-landscape/.

[34] Pramod Subramanyan et al. "A Formal Foundation for Secure Remote Execution of Enclaves". In: Oct. 2017, pp. 2435–2450. DOI: 10.1145/3133956.3134098.

[35] *Trusted Platform Module - Microsoft 365 Security*. Sept. 2018. URL: https://docs.microsoft.com/en-us/windows/security/information-protection/tpm/trusted-platform-module-top-node.

[36] Wenhao Wang et al. "Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2421–2434. ISBN: 9781450349468. DOI: 10.1145/3133956.3134038. URL: https://doi.org/10.1145/3133956.3134038.

[37] S. Zdancewic and A. C. Myers. "Observational determinism for concurrent program security". In: *16th IEEE Computer Security Foundations Workshop, 2003. Proceedings.* 2003, pp. 29–43.

[38] Steve Zdancewic et al. "Secure Program Partitioning". In: *ACM Trans. Comput. Syst.* 20.3 (Aug. 2002), pp. 283–328. ISSN: 0734-2071. DOI: 10.1145/566340.566343. URL: https://doi.org/10.1145/566340.566343.