PSec: Programming Secure Distributed Systems using Enclaves

Shivendra Kushwah
University of California, Berkeley
shivendrakushwah@berkeley.edu

Ankush Desai
Amazon Inc
ankushpd@amazon.com

Pramod Subramanyan
Indian Institute of Technology, Kanpur
spramod@cse.iitk.ac.in

Sanjit A. Seshia
University of California, Berkeley
sseshia@berkeley.edu

ABSTRACT

We introduce PSec, a domain-specific language for programming secure distributed systems. PSec is a state-machine based programming language with information flow control capabilities that leverages Intel SGX enclaves to provide security guarantees at runtime. Combining state machines and information flow control with hardware enclaves enables programmers to build complex distributed systems without inadvertently leaking sensitive information to adversaries. We formally prove the security properties of PSec and evaluate our work by programming several real-world examples, including One Time Passcode and Secure Electronic Voting systems. We present performance results of PSec systems and show that there is an acceptable performance overhead of ~3x for long running systems with a possible minimum of ~1.2x, as compared to baseline systems that do not provide any security guarantees.

CCS CONCEPTS

• **Security and privacy → Distributed systems security**; *Information flow control.*

KEYWORDS

Secure Distributed Systems; Enclaves; Trusted Computing; Programming Languages; Computer Security

ACM Reference Format:
Shivendra Kushwah, Ankush Desai, Pramod Subramanyan, and Sanjit A. Seshia. 2021. PSec: Programming Secure Distributed Systems using Enclaves. In *2021 ACM Asia Conference on Computer and Communications Security (ASIA CCS '21), June 7–11, 2021, Hong Kong, Hong Kong*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3433210.3453113

1 INTRODUCTION

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 maintainable components. Since this design is powerful and used widely, 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 when using higher level languages, programmers generally have to understand the basics of cryptography to properly initialize and effectively use cryptographic code and libraries for sensitive operations. This can prove troublesome, as in the case of the Sony PS3 Private Signing Key leak [13] where programmers actually used the same random number for each ECDSA signature, enabling hackers to crack the private key and trick PS3s into running malicious code. In the distributed setting, using proper cryptography is especially critical to transmit secure messages. However, even if the message is sent securely, there are no guarantees that the receiving machine will not accidentally leak the underlying sensitive data to adversaries. Addressing this challenge requires combining cryptography with information flow analysis and validating certain trust assumptions about the platform on which the system is deployed.

1.1 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 [25], SIF [3], Swift [2], and Fabric [14] utilize language-based information flow control to enforce confidentiality and integrity policies on data passed through the system. A main assumption behind these approaches is the correctness of the trust designation system, which requires entities to specify which nodes they believe to be running compliant 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 assumptions on the hardware and operating system. 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 Trusted Execution Environments (TEEs) such as Intel SGX [5] to reduce trust assumptions on hardware and operating systems while enabling programmers to readily implement secure applications. IMP_e [8] is an information flow control calculus that uses Intel SGX enclaves to provide secure memory. However, IMP_e is not tailored to the distributed setting.

ASIA CCS '21, June 7–11, 2021, Hong Kong, Hong Kong.

ACM ISBN 978-1-4503-8287-8/21/06.
https://doi.org/10.1145/3433210.3453113

Verifiable Confidential Cloud Computing (VC3) [18] does enable secure distributed computation with enclaves, but is restricted to the MapReduce case and is not tailored for general applications. Ryoan [12] creates a secure distributed sandbox that leverages SGX to protect against malicious distributed hosts. The authors design various secure protocols for sensitive data transmission and implement a variety of applications, but it is important to note that this is not a language based approach. Overall, although enclave-based frameworks do exist, there are not many programming frameworks that readily enable easy programming of distributed enclave applications. There is a need for a programming language that provides programmers the right level of abstraction while enabling them to effectively write distributed secure code. Although EActors [16], a more recent example, does try to accomplish this by presenting an actor-based programming language that leverages Intel SGX for trusted communication, it neither provides sufficient programmer protections nor protects programs from inadvertently leaking data.

For our work, we seek to combine the aforementioned lines of research and create a high-level programming language that provides the right level of abstraction and enables programmers to easily write secure distributed applications. We leverage hardware enclaves for secure distributed computing and provide a high level language with information flow guarantees to prevent programmers from inadvertently leaking data. We build on top of the P programming language [6, 7], a state-machine based language for distributed systems, because of its easy to understand nature and ability to express complex applications. We extend various P constructs as well as add new language features to support secure distributed computing.

1.2 Contributions and Roadmap

To summarize, the core novel contributions of this work are:

(1) We present a state-machine based programming language for creating secure distributed systems, with an information flow control type system to prevent secure data from being leaked to untrusted systems or being maliciously corrupted.
(2) We present a runtime to enable the secure creation of state machines and the ability to securely send messages between them. We also create a trust designation system in order to establish trust between state machines. Finally, we provide formal proofs for the confidentiality and integrity properties provided by our programming framework.
(3) We present initial performance metrics on an implementation of our language and system (located at our GitHub repo[1]). We have implemented a variety of examples to demonstrate language expressiveness and measure our language overhead to be ~3x for longer running systems, with a possible minimum of ~1.2x.

The remainder of this paper is structured as follows: Section 2 provides an overview of PSec; Section 3 describes the PSec language design; Section 4 discusses the PSec Type Checker and provides formalisms; Section 5 describes PSec's implementation; Section 6 presents an evaluation of our system; and Section 7 and Section 8 conclude and discuss future work.

[1] https://github.com/ShivKushwah/PSec

2 OVERVIEW

In this section, we provide an overview of PSec, *a language that provides high level state-machine based abstractions for implementing secure distributed systems.* We build on top of P [6, 7], a state-machine based programming language for building safe asynchronous distributed systems. PSec allows programmers to create regular P state machines (untrusted state machines, or USMs) as well as state machines hosted within secure hardware (secure state machines, or SSMs). Programmers can mark certain variables as `secure` to prevent them from being inadvertently leaked to the untrusted world and can use PSec's `send` primitive to securely send messages. PSec leverages TEEs such as Intel SGX and combines them with secure machine creation and message exchange protocols to enable programmers to design distributed systems with security guarantees (guarantees detailed in Section 4.3).

2.1 Background

The P Programming Language: P is a language for asynchronous event-driven programming. It allows the programmer to specify the system as a collection of state machines that communicate with each other using events. P unifies modeling, programming, and verification, and generates executable C code to bridge the gap between a high-level model and the low-level implementation. It has been used to develop the USB 3.0 driver inside Windows 8.1 and is currently run on hundreds of millions of devices worldwide. More recently, P is being used inside Amazon Web Services to build reliable distributed services. We refer the readers to the modular P paper [7] for more details. Given its easy to use nature and ability to express complex distributed applications, P serves as an ideal base to build from and we found it to be readily adaptable to target the creation of secure distributed systems.

Intel SGX: Intel Software Guard Extensions (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 the OS) and is only decrypted on the fly by the CPU to run the commands. Intel also provides a way to attest these enclaves to verify their identity and that they have not been tampered with.

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 only has access to a restricted set of LibC++, which makes converting normal programs to "enclave" programs an arduous task. Further wrappers on top of the Intel SGX SDK do exist (such as Microsoft Open Enclave SDK [15]), but these also require extensive knowledge of basic enclave primitives. Although enclaves provide confidentiality and integrity properties, it is hard to leverage them without building up a non-trivial amount of domain expertise. In this paper, we will demonstrate how PSec leverages this technology and makes it easier for programmers to utilize enclaves to build complex applications.

2.2 Civitas Secure Voting

Throughout this work, we will take a secure voting system as a running example to illustrate key PSec features. Civitas [4] is a

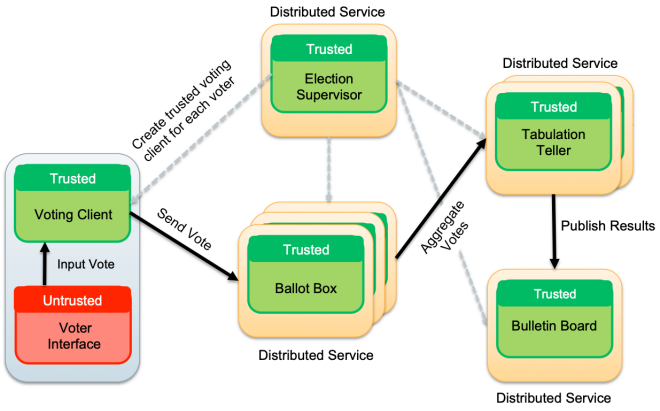

Figure 1: Civitas Architecture

remote electronic voting system designed at Cornell that formally provides coercion resistance (voters cannot prove how they voted) and voter verifiability (final tally is correct) properties. As depicted in Figure 1, voters send votes to Ballot Box machines in the voting phase. Then, in the vote counting phase, these votes are submitted to the Tabulation Tellers who eliminate duplicate votes and votes with incorrect credentials. The final votes are aggregated and sent to the Bulletin Board, which displays the results of the election. Throughout, the Election Supervisor performs administrative tasks and starts/stops the election. The Civitas design relies on various cryptography schemes to ensure election data is untampered with, and correctly deduplicated and counted.

We implement a version of the Civitas example using the PSec language. Since in PSec we have confidentiality and integrity properties (Theorem 4.1), we can achieve close guarantees to Civitas with a streamlined implementation of the system (discussed in Section 6.1). In our system, the Election Supervisor provides a Secure Voting Client that the Voter uses to securely submit their votes. The votes are then securely sent to Ballot Box and ultimately end up at the Bulletin Board. We indicate this architecture in Figure 1 where the trusted components (in green) are implemented as PSec SSMs, the untrusted components are implemented as USMs, and the vote is marked as a secret to provide the guarantees described above.

2.3 PSec Language

We will use abridged code snippets (Figure 2) from our Civitas example to demonstrate key PSec language constructs. In this snippet, the Voter USM requests the Supervisor SSM to provision a trusted Secure Voting Client SSM on the same host as itself. Then, it validates this newly created SSM before entrusting it with its secret vote. After that, the Voting Client SSM securely submits this vote to the Ballot Box SSM. We discuss various PSec constructs below.

Events: In P, state machines communicate with each other by sending events asynchronously. PSec builds on this concept and introduces trusted events (contain non-malicious payloads originating from the trusted world) and untrusted events (may contain corrupt/incorrect payloads from untrusted world). Since trusted events contain sensitive/trusted data, we enforce that these events are only sent between trusted SSMs and their content never leaks to the untrusted world. In contrast, untrusted events can be sent between any two parties, so we prevent these events from containing sensitive payloads. In Figure 2, we declare the relevant untrusted events for our example on lines 1 - 4 and trusted events on lines 5 - 7. For example, we make eTrustedVote a trusted event because it contains a confidential vote as its payload.

Machine Declaration and Creation: We declare SSMs (such as line 8 for Election Supervisor) with a `secure_machine` annotation and USMs (line 50 for Voter) with a `machine` annotation. Upon creating a new state machine, a machine handle is returned to the creator machine. In lines 20 - 21 of Figure 2, the Supervisor SSM creates an instance of a Secure Voting Client SSM and receives a `secure_machine` handle in return. This is because a SSM to SSM creation gives us secure process creation guarantees (explained in Section 5.3); all other creation scenarios result in a `machine` handle being returned. PSec additionally introduces the `@` command to allow programmers to indicate where state machines should be created. By using the `@` command in this line, we receive the guarantee that the Secure Voting Client SSM will be created on the same physical host as the voterMachine (which in this case is a Voter USM).

Additionally, `secure_machine` handles are treated as sensitive and only reside in SSMs. We enforce secure send semantics by requiring trusted events to be sent by invoking the `send` command with a `secure_machine` handle as opposed to a regular `machine` handle. As shown in line 27, `secure_machine` handles can be converted to `machine` handles using the `Declassify` command, but not vice versa.

Each PSec state machine has programmer defined event handlers that execute when an event is received. In lines 16 - 28 of Figure 2, the Supervisor SSM has an event handler that executes when the eSecureVotingClientReq event is received. On receiving this event, the Supervisor SSM creates the Voting Client SSM and sends it a trusted event containing the `secure_machine` handle of the Ballot Box SSM (so that it knows where to securely send future votes to). Finally, the Supervisor SSM sends an untrusted event containing the `machine` handle of the Voting Client SSM to the requesting Voter USM so they can communicate later.

Local Authentication and Sealing: PSec provides a `localAuthenticate` construct that allows USMs to verify that any local SSMs are actually running valid code. This is important because it enables programmers to express applications in which clients may be running in the untrusted world, but need to use secure infrastructure for sensitive tasks. This construct is essential in our example (lines 57 - 62 in Figure 2) when the Voter USM wants to authenticate the Voting Client SSM provisioned on its system before trusting it with the confidential vote. Since the Voter USM can only send untrusted events, it needs to validate that the Voting Client SSM will `Endorse` its vote as a sensitive data type and forward it to the Ballot Box SSM as a eTrustedVote event (lines 81 - 86). We additionally provide a `seal` and `unseal` language construct to enable SSMs to securely store sensitive data in a encrypted form on untrusted memory (outlined in Section 3.3).

Information Flow Analysis: PSec utilizes information flow analysis to ensure that sensitive data is not leaked to the untrusted world and that trusted data is not corrupted. We elaborate more in Section 4.1, but the high level idea is that trusted events must only contain secure types while untrusted events should not contain any secure data types. This is shown on lines 81 - 86 in Figure 2,

```
1  event eSecureVotingClientReq: machine;
2  event eSecureVotingClientResp: machine;
3  event eSealedVote: sealed_data;
4  event eVote: (credential: string, vote: int);
5  trusted event eProvisionSecureClient: secure_machine;
6  trusted event eTrustedVote: (credential: secure_string,
7                               vote: secure_int);
8  secure_machine ElectionSupervisor {
9    var ballotBox: secure_machine;
10   start state Init {
11       entry {
12          ballotBox = new BallotBox(); goto WaitForClientReq;
13       }
14   }
15   state WaitForClientReq {
16     on eSecureVotingClientReq
17         do (voterMachine: machine) {
18         // Create SecureVotingClient on voterMachine host
19         var secureVotingClient: secure_machine;
20         secureVotingClient =
21                 new SecureVotingClient() @ voterMachine;
22         // Send the secure handle of the Ballot Box
23         send secureVotingClient,
24                 eProvisionSecureClient, ballotBox;
25         // Send the handle of the new SecureVotingClient
26         send voterMachine, eSecureVotingClientResp,
27               Declassify(secureVotingClient) as machine;
28     }
29     ...
30 }
31 secure_machine BallotBox {
32   var appendOnlyLog: secure_machine;
33   var memory: machine;
34   start state Init {
35       entry {
36           appendOnlyLog = new SecureTamperEvidentLog() @ this;
37           memory = new Memory() @ this; goto WaitForVotes;
38       }
39   }
40   state WaitForVotes {
41       on eTrustedVote do (payload:
42           (credential: secure_string, vote: secure_int)) {
43            var sealedVote : sealed_data;
44            sealedVote = seal(payload);
45            send memory, eSealedVote, sealedVote;
46         }
47     ...
48 }
```

```
50 machine Voter {
51     var credential: string;
52     var vote: int;
53     state Vote {
54         entry (secureVotingClient: machine) {
55         var machineTypeToValidate: string;
56         machineTypeToValidate = "SecureVotingClient";
57         if (localAuthenticate(secureVotingClient,
58                                     machineTypeToValidate)){
59             // Authenticated installed enclave
60             send secureVotingClient, eVote,
61                     (credential = credential, vote = vote);
62         }
63     }
64     ...
65 }
66
67 secure_machine SecureVotingClient {
68     var ballotBox: secure_machine;
69     state Provision {
70         on eProvisionSecureClient
71             do (payload: secure_machine){
72                 ballotBox = payload;
73                 goto SubmitVote;
74             }
75     }
76     state SubmitVote {
77     on eVote
78         do (payload: (credential: string, vote: int)) {
79           var secure_vote: secure_int;
80           var credential: secure_string;
81           secure_vote = Endorse(payload.vote) as secure_int;
82           credential = Endorse(payload.credential)
83                               as secure_string;
84           send ballotBox, eTrustedVote,
85                           (credential = credential,
86                            vote = secure_vote);
87         }
88     }
89     ...
90 }
```

Figure 2: PSec Code Snippets

where the non-secure payload needs to explicitly `Endorse`d before being placed in an eTrustedVote event, and also in the Supervisor SSM code snippet (lines 26 - 27), where the secureVotingClient (a `secure_machine` handle) needs to be `Declassify`ed to a `machine` handle for the untrusted eSecureVotingClientResp event.

2.4 PSec Adversary Model and Guarantees

Adversaries: We assume that there are different levels of adversaries and PSec provides different guarantees against each one.

(1) *Passive Network Observer:* Can observe all network traffic and extract relevant data from network requests.
(2) *Active Man-in-the-Middle:* Can additionally tamper with network traffic and extract relevant information.
(3) *Privileged Attacker:* Can additionally corrupt host machines and spin up malicious enclaves on these hosts as needed (containing valid PSec code or custom malicious code). Also, has privileged access in the untrusted world and can tamper with all internal state for the host machines, except state stored within enclaves.

Guarantees: We provide an overview of the PSec security guarantees against the aforementioned adversaries and discuss the implementation enabling them in later sections.

(1) *Passive Network Observer:* PSec uses cryptography to prevent network observers from determining message payloads, but the adversary can determine which 2 parties are communicating as well as message types and lengths.
(2) *Active Man-in-the-Middle:* PSec enables state machines to detect if any message payloads have been tampered with, but the adversary can induce denial-of-service (DoS) attacks by dropping or corrupting messages.
(3) *Privileged Attackers:* Privileged attackers have control over the entire untrusted world (USMs, distributed hosts, network). As a result, privileged attackers can send authenticated messages from compromised parties to other state machines in our system. However, since PSec prevents SSMs from ever giving secrets to the untrusted world, sensitive data cannot be leaked this way. Privileged attackers can additionally create SSMs by spinning up their own malicious enclaves running custom code. However, SSMs additionally do not send secrets to untrusted SSMs (SSMs that the current machine does not have a `secure_machine` handle for). Generally, any data received from the parties mentioned above is regarded as untrusted.

It is important to note that DoS 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. Host machines can refuse to service requests from the enclave at any moment in time. Regardless, PSec guarantees that no secret data is ever leaked.

Additionally, PSec utilizes a trusted Key Provisioning Server (KPS) to bootstrap trust for our various protocols and serve as the root of trust for machines in our system. We assume that the KPS has a correct implementation and is hosted on a trusted server to

enable us to provide the aforementioned guarantees. We elaborate more on the design on the KPS and how it ties in with the security of the system in Section 5.1.

Finally, similar to other SGX-related work in this area (such as the EActors programming language [16]), we consider side-channel attacks (such as page-fault attacks [20] and cache-timing attacks [9]) to be out of scope. These attacks have counter-measures that can be implemented independently [10, 19] from our approach.

3 PSEC LANGUAGE DESIGN

While designing PSec, we wanted to create language constructs that enable programmers to implement real-world systems. Secure distributed systems, which consist of trusted entities, may have to interact with external untrusted entities 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. We accomplish this by enabling programmers to choose between 2 types of PSec state machines: secure state machines (SSMs) and untrusted state machines (USMs). SSMs are trusted to run sensitive code and correctly handle secret data while USMs run in the untrusted world. PSec is built as an extension to the P language, and we outline notable differences in the following sections.

3.1 Machine Creation

Rationale: Since PSec machines are created dynamically, we need a way to securely create state machines on the fly. In addition to this, we need a way to designate trust so that state machines are confident that they are communicating with trusted entities. We define *trust* in the following way: if A *trusts* B, A assumes that B will not leak any sensitive data nor send any malicious inputs.

Our initial approach had SSMs trust all other SSMs running valid PSec code for that particular application. This would prevent attacks in which privileged adversaries spin up SSMs containing malicious, custom code. However, this designation of trust is too broad because adversaries can spin up their own SSMs (running valid PSec code) and convince our SSMs to send them sensitive data. Although SSMs do not leak data to their hosts, the adversaries can more easily perform DoS attacks on local machines and our machines would be waiting on responses that will never come.

A more sensible designation relies on having trust chains centered around machine creation. In this scheme, the primary driver is that SSMs trust any SSMs they have created and these child SSMs trust their parent SSM. The secondary driver is through trust designation – child SSMs additionally trust SSMs deemed by a fellow trusted SSM to be trustworthy. 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 on a trusted host machine. If this trusted host machine is corrupted, the worst case scenario is again a denial of service. 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

Both of these utilize the PSec Runtime to send the state machine creation request to the correct host machine. In the *untrusted create* case, the creator machine receives the `machine` handle of the newly created machine. In the *trusted create* case, the creator SSM receives the handle as well as the `capability` to send trusted data to the newly created SSM. The type of the returned handle is `secure_machine` to indicate this `capability`. The creator SSM can share this `capability` by sending this secret `secure_machine` handle to other trusted SSMs.

Since PSec is a distributed programming language, we want to allow the programmer to be able to denote where they want to create state machines if necessary. We introduce an optional `@` extension to the `new` command that designates the new state machine to be created on the same physical host as the input handle. As an example from Civitas on lines 36-37 (Figure 2), each Ballot Box SSM creates a corresponding Append Only Log SSM to record sensitive votes as well as a Memory USM on the same physical host as itself. We depict the relevant code below:

```
1 var appendOnlyLog: secure_machine;
2 var memory: machine;
3 ...
4 entry {
5     appendOnlyLog = new SecureTamperEvidentLogMachine() @ this;
6     memory = new MemoryStateMachine() @ this;
7 }
```

3.2 Message Sending

Rationale: We need to define multiple message sending types in order to capture all interactions in a distributed system. First of all, SSMs should be able to exchange trusted messages with sensitive data with other SSMs they trust. In addition to this, they should also be able to exchange messages from USMs as well as other untrusted SSMs (one simple usecase being to accept input from user systems and send back computed output). SSMs should be able to differentiate between these 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 *trusted sending* (used to exchange messages with trusted content between SSMs) and *untrusted sending* (for all other usecases). We additionally define trusted events and untrusted events where programmers can use *trusted sending* to send messages encapsulated in trusted events and *untrusted sending* for messages in untrusted events. Programmers can specify different event handlers for state machines when they receive a trusted 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`). This type of sending can only occur between 2 SSMs, and this `capability` is indicated by the possession of the `secure_machine` handle. These messages are sent over a secure channel and the PSec Type Checker enforces that trusted events can only be sent using *trusted 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.

Untrusted Sending: *Untrusted sending* is inferred by the compiler through the `machine` handle type. 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 with encryption but since the messages themselves may originate from the untrusted world, their content is untrusted. The following code snippet shows the template for a send:

```
// Trusted Send
send targetSecureMachine, TrustedEvent, securePayload;
// Untrusted Send
send targetMachine, UntrustedEvent, payload;
```

3.3 Additional Language Features

Sealing: The `seal` and `unseal` language constructs enable programmers to encrypt data in SSMs such that only instances of that specific SSM are able to decrypt it. We define a new type for the encrypted data from the `seal` command (`sealed_data`), and the command takes in a PSec value and returns the sealed data. In Civitas, once the Ballot Box SSM receives a vote, it seals the vote and sends it to untrusted memory. We have included the relevant code snippet from lines 43-45 in Figure 2:

```
var vote: (secure_string, secure_int);
...
var sealedVote : sealed_data;
sealedVote = seal(vote);
send memory, eSealedVote, sealedVote;
// Can later retrieve using
// `unseal(sealedVote) as (secure_string, secure_int)`
```

Local Authenticate: The `localAuthenticate` language construct allows USMs to verify that any SSMs running on the same distributed host are actually running valid code. This is necessary to establish a one-way trust from a USM to a particular SSM since we already have mechanisms to establish SSM to SSM trust. This construct is essential in the Civitas example when the Voter USM wants to authenticate the Secure Voting Client SSM provisioned on its system before trusting it with the confidential vote. We include lines 57 - 62 from the code snippet in Figure 2:

```
if (localAuthenticate(votingSecureMachine, machineTypeToValidate)){
    // Authenticated installed enclave
    send votingSecureMachine, eVote,
            (credential = credential, vote = vote);
}
```

4 PSEC FORMALISMS AND GUARANTEES

4.1 PSec Type Checker

One of the primary goals of PSec is to enable secure computation in a simple, high-level programming language. As part of this, we want to prevent programmers from accidentally leaking secret data. A potential approach to achieve this involves using cryptography or access control frameworks. However, a fundamental problem with these approaches is that they do not prevent entities (with legitimate access) from accidentally leaking secret data to bad actors or provide any sort of warning to programmers. This problem can be combated by utilizing external monitors, but it is hard for these systems to detect potentially mutated forms of sensitive data.

PSec uses a different approach based on enforcing information flow control (IFC) policies through static type-checking. This kind of static analysis enables us to ensure that no secret information is leaked and it works by augmenting the type system to include security labels. In our system, we define two types of security labels (High/H and Low/L) similar to other work in this field [23]. These labels indicate that we want to maintain the confidentiality and integrity of secret values. Essentially, we want to prevent data with H labels from being leaked and we also want to ensure that this data contains trusted information, rather than potentially malicious payloads. Programmers can utilize the secure types of variables by prefixing the type with `secure_` to indicate this, and we propagate these labels across the program during analysis.

With IFC, the PSec type checker guarantees that sensitive data (and any mutations) remains within trusted machines and never leaks to the untrusted world. For example, we enforce that untrusted events only contain L data while trusted events only contain H labeled data (this also is necessary to guarantee the integrity of values in the system). However, we provide `Declassify` and `Endorse` commands for flexibility to implement real world usecases (such as lines 81 - 86 in Figure 2 where we need to `Endorse` the vote before forwarding it to ensure it will be treated securely, even though the vote input was from a USM). One additional usage of `Declassify` is to take a `secure_machine` and remove its `capability`, outputting a non-sensitive `machine` handle. We require the programmers to manually verify the usage of `Declassify` and `Endorse` operations, but by making the usage of these commands explicit, we prevent many classes of bugs since this is a much smaller surface to check.

In our type checker, we additionally provide a foreign function interface where programmers can define implementations of PSec functions in C++, provided they create a typed PSec function interface. Our information flow analysis holds provided the interface contract is met. Our experience is that this is a much smaller surface for programmers to check for potential issues, but gives PSec the power and flexibility to implement functions used in real world examples in a native low-level language (e.g. the hashing function mentioned in the OTP example in Section 6.1) .

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

4.2 Formalism

Definitions. We present the definitions required to describe the formal guarantees of PSec.

(1) Let $\mathcal{Z}$ be the set of all machine identifiers. A machine identifier is a pair $\mathcal{M} \times \mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers and $\mathcal{M}$ is the set of names of all state machines.
(2) Let $s \in \mathcal{S}$ be the state of a PSec state machine where $\mathcal{S}$ is the set of all possible local states for any given state machine.
(3) Let $\mathcal{E}_{\mathcal{T}}$ represent the set of names of all the trusted events and $\mathcal{E}_{\mathcal{U}}$ represent the untrusted events. Let $\mathcal{E}$ be $\mathcal{E}_{\mathcal{T}} \cup \mathcal{E}_{\mathcal{U}}$.
(4) Let $\mathcal{B}$ represent the set of all possible values for any given input event buffer (state machines enqueue PSec events into the target machine's input buffer). This buffer is a sequence of $(e, v) \in \mathcal{E} \times \mathcal{V}$ pairs, where $\mathcal{V}$ is the set of all possible payloads for PSec events.
(5) The configuration of our system is a tuple: $G = (S, B, C)$ where S is a partial map from $\mathcal{Z}$ to $\mathcal{S}$, B is a partial map from $\mathcal{Z}$ to $\mathcal{B}$, and C is a partial map from $\mathcal{M}$ to $\mathbb{N}$. Essentially, $S[m, n]$ represents the state of the nth instance of machine m, $B[m, n]$

represents its input buffer, and $C[m]$ represents the number of instances of machine type m that currently exist. A trace of our program (similar to a trace in P [7]) is a finite sequence $G^0 \xrightarrow{a_0} \ldots \xrightarrow{a_{n-1}} G^n$ for $n \in \mathbb{N}$ such that $G^i \xrightarrow{a_i} G^{i+1}$ for each $i \in [0, n)$ where a represents the information that can be inferred from the transition.

Observational Equivalence: Two values are observationally equivalent ($\mathcal{L}$) if they are indistinguishable to a low-level observer [24]. For example, an observer should not be able to differentiate between different H values, but can do so for two L values. The observational equivalence rule for primitive values with type τ in typing context Γ is that if τ is L, then the values must be exactly equivalent:

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

Generalizing this rule, we have the following:
For states S_1, S_2 (where *uvals* returns a map of non-sensitive variables to their values in state s), all corresponding L values must be equivalent:

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

For buffers B_1, B_2, all untrusted events/payloads must be equivalent:

$$B_1 \approx_{\mathcal{L}} B_2 \implies$$

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

Machine instance maps C_1, C_2 must be equivalent:

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

For configurations G_1, G_2:

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

For traces π_1, π_2:

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

Observation Function: 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. We provide the formal operational semantics of the PSec language in Figure 7 in the Appendix.

4.3 PSec System Guarantees

The PSec Runtime and type checker allow our system to satisfy the property of observational determinism.

Observational Determinism: Observational determinism [24] 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.

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. The dual of this is to prove that adversaries cannot corrupt trusted values in our system (we call this Trusted Equivalence and we detail this, $\approx_{\mathcal{H}}$, and $Obs_{\mathcal{H}}$ in the Appendix, in Section C.2.). 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:

$$\forall \pi_1, \pi_2 \in \mathcal{P}, \ n \in \mathbb{N}$$

$$\pi_1 = G_1^0 \xrightarrow{a_0} \ldots \xrightarrow{a_{n-1}} G_1^n, \ \pi_2 = G_2^0 \xrightarrow{a_0} \ldots \xrightarrow{a_{n-1}} G_2^n$$

For confidentiality,

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

For integrity,

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

Theorem 4.1. *If a* PSec *program type checks, then the program satisfies the property of observation determinism when executed using the* PSec *Runtime.*

We include proofs in the Appendix, in Section C.

5 IMPLEMENTATION

Our PSec design relies on two assumptions: (1) the code running within SSMs is trusted (i.e. does not leak sensitive information and is certified) and (2) the adversaries cannot read sensitive information contained within SSMs. The first assumption is partly achieved through the language since SSMs are implemented using PSec and hence satisfy the desired IFC properties statically. To ensure that these static guarantees hold at runtime and to satisfy both assumptions, we provide a runtime that implements various constructs described in Section 3 and leverages Intel SGX to ensure confidentiality of execution and integrity of code through attestation. In this section, we briefly describe our runtime implementation.

5.1 System Architecture

Our runtime consists of two parts: the PSec Secure Runtime, and the PSec Untrusted Runtime. The PSec Secure Runtime is responsible for executing a SSM inside an enclave and implements all the necessary functionalities for our constructs. The PSec Untrusted Runtime is in charge of USMs and additionally ferries messages to the network for all of the state machines.

In our system, we have multiple distributed host machines and a trusted Key Provisioning Server (KPS). The overall architecture is shown in Figure 3. Each distributed machine can host multiple SSMs (one per enclave) and USMs. The KPS stores all precomputed enclave measurements and serves as an intermediary to establish trust. Additionally, the public KPS key is baked into all enclaves. All network connections between any machines in the system use openSSL TLS connections, and the KPS also serves as a certificate authority for all valid host machines. We assume the KPS has a correct implementation and is deployed on a trusted server. As we will demonstrate, the KPS serves as the root of trust for all the SSMs in our system. It is important to note that the KPS is always queried through our PSec runtime and is only strictly utilized during machine creation time. For SSMs, this means that any interactions to the KPS are handled by the PSec Secure Runtime which cannot be maliciously tampered, a fact that ensures the security of the overall system. Note that in our current implementation there is a single KPS server and hence a single point of failure - this can easily be

addressed by making the KPS as a fault-tolerant distributed service that maintains a database of all the information required to authenticate new enclaves.

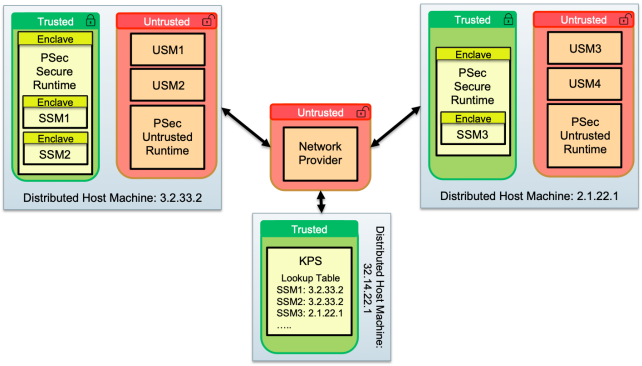

Figure 3: PSec System Architecture

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, it can create additional trusted SSMs (which can reside on untrusted hosts) and bootstrap the trusted distributed system.

5.2 Definitions for Protocols

We provide necessary definitions that we will use in the next section to describe the implementations of our various protocols.

Identity: Every state machine has an associated unique `Identity`, an asymmetric key pair. When we say that a state machine has authenticated using its `Identity`, this involves a signature over the private `Identity` key. When we refer to `Identity` (without explicitly stating private key), we mean the `Identity` public key.

Capability: Every SSM has a `Capability` (an asymmetric key pair) associated with it. Anyone who possesses the `Capability` of an SSM gains the ability to send trusted events to that SSM. `Capabilities` must never leak into the untrusted world and must stay within enclaves since they are secrets. When we refer to the `Capability`, 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 SGX's version of the Sigma Protocol (version of authenticated Diffie-Hellman provided by SGX SDK). In this protocol, the enclave remote attests itself to the KPS. It proves that it is legitimate and running PSec, and then bootstraps a secure channel. The KPS authenticates using its private signing key while the enclave authenticates using Intel EPID.

5.3 Trusted Create Protocol

Recall that the *trusted create* protocol is invoked when a 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` typed handle to the parent SSM. We outline the protocol in Figure 4. The protocol is divided into 2 parts:

Parent SSM - createMachineRequest(): When the `new` command is invoked on a SSM type, the parent SSM calls the `createMachineRequest` method of the PSec Secure Runtime. It passes in its `Identity` as well as the type of SSM it wishes to create. If the `@` command is not used, this method first makes an `OCALL` to the PSec Untrusted Runtime to make a network request (using TLS) to the KPS to determine the IP address/port of a valid distributed host for 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 (again using TLS). 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 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 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 Secure Runtime method. This initializes a new PSec process inside the enclave, creates the child SSM, and generates an `Identity` for it. 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 To Create). The KPS stores (parent SSM `Identity`, child SSM `Identity`, Type of SSM To Create) -> child SSM `Capability`, and sends the newly generated `Capability` back to the child SSM. Once the child enclave receives the `Capability`, it 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 guarantees against even our most privileged attacker from Section 2.4. First of all, if the parent SSM successfully retrieves the `Capability` from the KPS, it receives a guarantee that the child SSM was securely created since the KPS validates the child enclave through the `Secure Sigma*` protocol before generating this `Capability`. The child SSM also receives the guarantee that only its parent SSM receives its `Capability` due to KPS validation.

This protocol has limitations, but they are the same limitations that are inherent to enclaves. This protocol is vulnerable to a DoS attack by any of the distributed host machines since they can drop messages, modify the requested type of SSM to create, or modify the child SSM `Identity` that is returned. 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. It is important to note that in

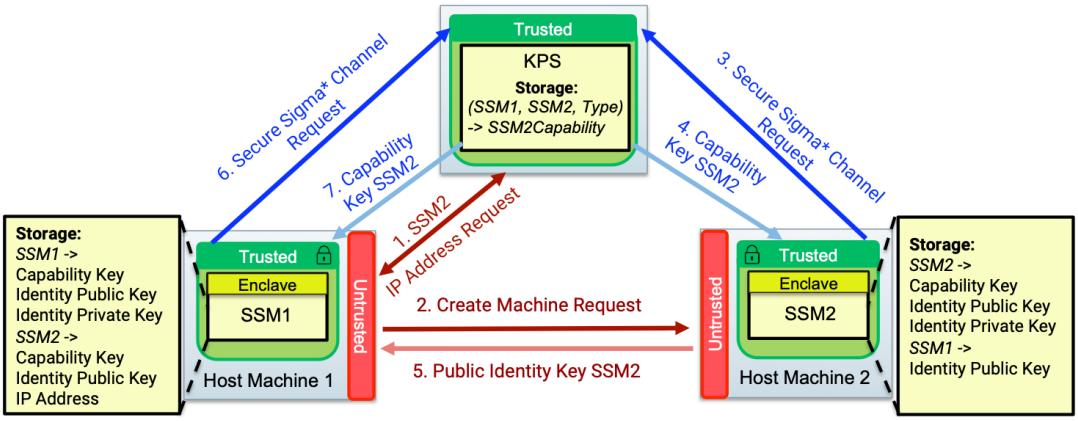

Figure 4: Trusted Create Protocol

all of these cases, this protocol does not leak any secret data and the enclaves are generally aware of when failures occur.

5.4 Untrusted Create Protocol

The *untrusted create* protocol is invoked during all other machine creation cases. Upon completion, this protocol returns a `machine` handle that contains the `Identity` as well as network address information of the newly created child. This protocol follows a similar flow to the *trusted create* protocol, but is a "best-effort" protocol that does not involve `Capability` generation nor KPS attestation.

Guarantees: The *untrusted create* protocol gives us fewer guarantees as compared to the *trusted create* protocol. Since this protocol leverages TLS, it is resistant to the first 2 levels of adversaries (passive network observers and man-in-the-middle attackers), but can be subverted much more easily by privileged adversaries. Any corruption or subversion in the receiving distributed host machine may cause the wrong machine to be spawned (or no machine at all!), and there is no way to verify this information.

5.5 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` and `Identity` of the receiving SSM (all encapsulated in the `secure_machine` handle). When the sending SSM invokes the `send` command with a `secure_machine` handle, the PSec Secure Runtime generates a session key, parses the relevant information from the `secure_machine` handle, and sends a TLS network request to the receiving SSM with the session key encrypted by the receiving SSM's `Identity`. As a quick overview, it then essentially serializes the PSec Event payload, signs over this with the receiving SSM's private `Capability` key, appends these two parts together, and sends this entire chunk encrypted with the session key over. The session key is reused on subsequent connections and for brevity, we are not outlining our use of nonces, IVs, and MACs. On a high level, the sending SSM authenticates using the `Capability` while the receiving SSM authenticates using its `Identity`. We have this additional layer of protection in addition to TLS so that messages are only readable by the receiving SSM and not by the host. One item to note is that we could have implemented `Capabilities` as shared symmetric keys and provided the same authentication properties by utilizing MACs. Although this does not provide additional security guarantees, it could provide potential performance improvements and we leave this as a future implementation improvement.

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. We do note that adversaries may be able to determine the event type or message length, but the payload will always remains a secret. As before, this protocol is vulnerable to DoS attacks because messages can be dropped.

5.6 Untrusted Send Protocol

Invoking the `send` command with a `machine` handle uses our *untrusted send* protocol. This is very similar to the *trusted send* protocol, except this method establishes a secure channel by using the `Identity` of both machines as a means of authentication. After a secure channel is created, the untrusted event is sent over and passed to the receiving machine by the PSec Untrusted Runtime. Once again, the connection between the two distributed host machines uses TLS as an additional layer of protection.

Guarantees: Since these messages are potentially originating from the untrusted world, this protocol does not 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 either party, this protocol guarantees confidentiality and integrity of the message payload being sent. In particular, this protocol prevents man-in-the-middle network attackers from learning anything useful about the data, and once again, is vulnerable to DoS attacks.

5.7 Seal and Local Authenticate

The implementations of the `seal/unseal` and the `localAuthenticate` commands are relatively straightforward. For `seal/unseal`, we leverage Intel SGX's sealing capability provided by the native SDK. For

`localAuthenticate`, we obtain an on-demand measurement of the target enclave and compare it to an expected measurement.

5.8 Limitations

Currently, Intel SGX requires enclaves to be registered with the Intel IAS Server beforehand so that one of the signatures during the remote attestation exchange can be verified. 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. This makes it not possible to pre-register our enclaves with the IAS and as a result, we skip this registration and signature verification process and leave automating it as future work. We additionally use Intel's provided sample code for 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 hosts can only handle one network request at a time. We have not implemented multi-threading 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. We would also like to implement more robust error handling, and leave these implementation improvements as future work.

Finally, we acknowledge that since we provide a foreign function interface (similar to the Rust language) where programmers can define implementations of PSec functions in C++ for added flexibility, bugs that exist in this surface can cause possible problems. One possible intermediate solution might be automate the verification of the foreign function code. We leave this for future work.

6 EVALUATION

We evaluate the PSec framework along two dimensions:
Expressiveness. We implement real-world applications using PSec to demonstrate its expressiveness and ease of programming.
Performance. We compare the performance of systems built using PSec against the insecure baseline systems programmed in P.

6.1 Examples

We evaluate the PSec framework by implementing the following examples taken from related work [4, 12, 21].

One Time Passcode : One Time Passcode (OTP) services are often used in 2-Factor Authentication schemes. Many secure implementations rely on having users supply their password along with an OTP code computed by a tamper resistant hardware token (usually a function of a shared secret and current time) to authenticate. Although these implementations provide strong security guarantees, distributing these hardware tokens is often inconvenient in practice. Recent schemes such as the one proposed by Hoekstra et al. [11] replace these physical tokens with Intel SGX (already present in modern computers with Intel CPUs). Intel SGX provides similar security guarantees because after an enclave is provisioned, attackers cannot easily conduct remote attacks and need access to the same physical machine as the user to generate valid OTP codes.

In order to test PSec, we implement a version of a SGX-OTP service, specifically a Bank 2-Factor Authentication example (inspired from the Moat paper [21]). In our system, we have a Bank SSM, a Client Web Browser USM, and a Client SSM. In the setup phase, the Client Web Browser USM authenticates with the Bank SSM and requests to enable 2-Factor. The Bank SSM then creates a Client SSM on the same host as the Client Web Browser and provisions it with a master secret (marked with `secure`). In the sign-in phase, the Client Web Browser sends a request to the Client SSM with its credentials and receives an OTP code in return. The OTP code is computed by hashing the input credential with the master secret and is generated regardless of the correctness of the input credential (we implement this hash function in C++ using PSec's foreign function interface). After, the Client Web Browser forwards this code along with its credentials to the Bank SSM. The Bank SSM sends back either Auth Success or Auth Failure and the client is either successfully logged in or redirected to login again.

Secure Email Processing: Spam filtering services are increasingly important in the modern world due to a high proliferation of spam. We implement a version of a secure spam filtering service in PSec where the user can outsource email filtering and spam detection to a third party service while keeping the email text private and confidential. We create an Email User SSM, an Email User USM, and a Secure Spam Filter SSM. The Secure Spam Filter SSM provisions an Email User SSM on the same host machine as the requesting Email User USM. Then, the Email User USM sends the email text to the Email User SSM, which sends it to the Secure Spam Filter SSM for processing and returns SPAM/NOT SPAM. We mark the email text as a `secure` field to protect it from being leaked.

Health Analysis: Companies such as 23andMe provide health reports for users. However, medical data is inherently sensitive and we want to provide guarantees that this data will not be leaked. This is made harder since the company itself may be relying on third party cloud services (such as AWS). For our PSec implementation, we designate a User USM, a User SSM, a Secure Health Analyzer SSM (23andMe), and an AWS ML Host SSM. The User USM sends a request to the Secure Health Analyzer SSM and receives the handle of a User SSM newly provisioned on its system. It then sends the relevant medical data to the User SSM, which securely forwards it to the Secure Health Analyzer SSM. The Secure Health Analyzer then creates an instance of the AWS ML Host SSM, sends it the medical data, and returns the diagnosis (TRUE/FALSE). We mark the medical data as a `secure` field to protect it from being leaked. As part of our implementation, we do not code the health analysis logic since it is application-specific, but rather program the overall flow to show that PSec can express this type of distributed service.

Note that while the Secure Email Processing and Health Analysis examples were inspired from the Ryoan paper [12], our implementation provides slightly weaker guarantees. While the Ryoan secure sandbox is constructed to provide guarantees against a malicious application developer, we assume a trusted programmer writing potentially buggy code. This is a realistic assumption and enables us to be more flexible in our design while providing strong guarantees against malicious adversaries.

Secure Voting: We describe the Civitas Secure Voting System in Section 2.2. We implement both the voting phase and the vote counting phase and and assume voter registration is through a physical registration teller, as recommended by the Civitas paper. We mark votes as a `secure` field to protect them. Our implementation provides many of the same guarantees as the original Civitas

paper, but we provide a weaker coercion resistance property. Since the original Civitas paper assumes the existence of anonymous channels (not possible in PSec since we do not use 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 but still submit a vote with their real credentials. In this case, the coercer would submit an invalid vote, but would not realize any wrongdoing until election results are finally released. At this point, it is too late for the coercer to change the outcome. Ideally, we would like to prevent the coercer from learning this information.

6.2 Expressiveness

In Table 1, we describe the number of lines of code written for implementing all the examples using PSec.

Application	OTP	Email	Health	Civitas
PSec App [PSec]	217	109	120	426
PSec Foreign Fn [C/C++]	35	-	-	14
PSec Library Stubs [PSec]	21			
PSec Runtime [C/C++]	18,883			
PSec Type Checker [C#]	9,330			

Table 1: Lines of Code

The PSec App row represents the lines of code written in PSec to implement the high-level protocol logic of each application. This demonstrates the power of using our abstractions; programmers can implement secure distributed applications in a few hundred lines of PSec code without having to worry about the low-level details. The majority of the rows consist of one-time implementations (such as compiler extensions and supporting libraries) as a part of creating the PSec framework. All the low-level code that interacts with enclaves and implements cryptographic security primitives is shared across the applications and abstracted from the programmer.

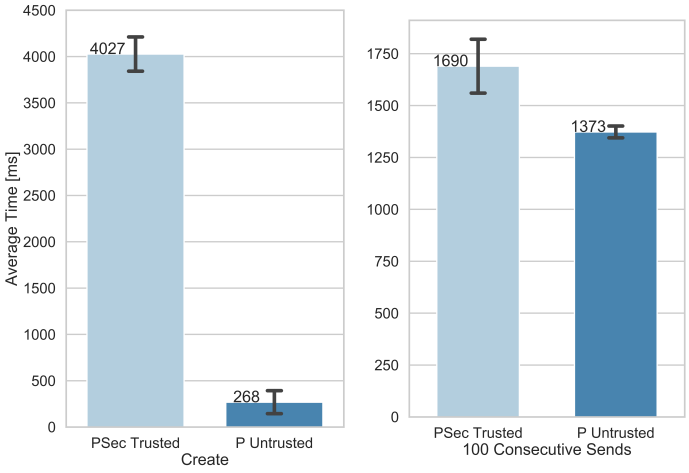

Figure 5: Granular Performance Metrics

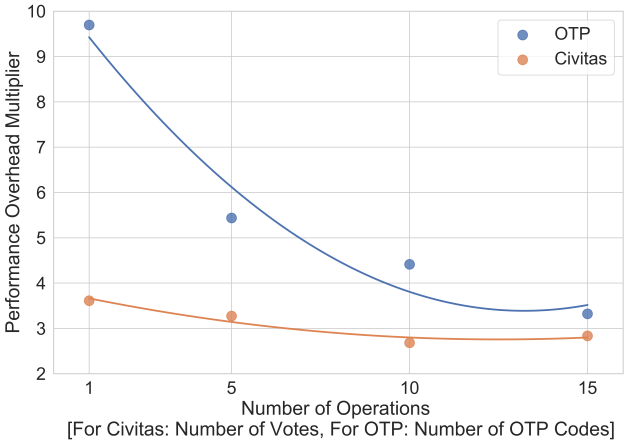

Figure 6: Performance Overhead Graph

6.3 Performance

We present our performance results on an initial implementation of our system. We deploy our code on Azure Confidential Compute instances running on 3.7GHz Intel Xeon E-2176G processors with SGX on Ubuntu 18.04. We chose to rent 2 Standard DC2s_v2s (2 vcpus, 8 GiB memory) in the US East region connected to the same virtual network. All SGX-code is running in HARDWARE mode.

Benchmark Overhead: We conducted experiments to benchmark the overhead of PSec versus a distributed version of P (using our *untrusted send* and *untrusted create* constructs). In Figure 5, we present the overall time for: (1) creating a SSM as compared to a P USM, and (2) performing 100 *trusted sends* as compared to 100 P *untrusted sends*. We conduct each test 100 times and indicate the standard deviation in the graph. Creating a SSM in PSec is ~15x slower than creating an USM in P, and this overhead is mainly because the SSM process creation involves several steps such as provisioning an enclave, communicating with the KPS, and the entire attestation procedure. The overhead for *trusted sends* through SSMs is a much smaller ~1.2x. Although the creation overhead may seem prohibitively high, we believe that in real-world systems, the usage pattern will tend to revolve more around send operations rather than machine creations.

To validate this claim, we compared overall application level timings of our PSec Civitas and OTP examples with their insecure P counterparts. For Civitas, we measure the time from when the client submits the vote to when the election's final results are retrieved. For OTP, we measure the time from when the client sends the initial request to the bank to successful authentication. We made each application perform 1 to 15 operations, average over several runs, and present the results in Figure 6. The overall application overhead is ~3x in just 15 operations, and is trending towards the theoretical minimum of ~1.2x (overhead of *trusted* over *untrusted sends*). Note that the OTP example has a bigger initial performance discrepancy since it is a smaller example and the high startup enclave provisioning overhead takes disproportionate time. We strongly believe that for most real-world applications that perform many operations over their lifetime, this performance overhead is encouraging for the security guarantees provided by using PSec.

Overall, given the strong security guarantees provided by PSec with the ease of programming at a high level of abstraction, we

believe this to be an acceptable performance overhead. Furthermore, this is partly implementation driven. For example, instead of dynamically creating and provisioning new enclaves, we can create a pool of validated enclaves beforehand or use switchless [22] calls to make enclave context changes less expensive. Changes like these would bring the performance overhead closer to that of native P.

7 CONCLUSION

In this paper, we presented PSec, a state-machine based programming framework for implementing secure distributed systems. To achieve this, we augmented the P language with an information flow control type system and added language constructs to enable programmers to write secure distributed systems with formal security guarantees, without having to worry about security protocols and their low-level implementations. To ensure that the security guarantees provided by the PSec programming language hold at runtime, we leveraged *enclaves* and constructed security protocols that take advantage of these trusted execution environments. We demonstrated the efficacy of the PSec framework by implementing real-world applications, and our results show that our language is sufficiently expressive and has an acceptable performance overhead for longer running systems (measured to be ~3x, but with a possible minimum of ~1.2x).

8 FUTURE WORK

We outlined the limitations of our language implementation in Section 5.8 and leave it for future work. Another potential addition to our PSec Runtime implementation would be to replace our current dependence on Intel IAS with attestation frameworks such as DCAP [17] or OPERA [1] which could potentially eliminate the need to pre-register our enclaves with Intel. However, DCAP requires using newer Intel chips with Flexible Launch Control support and OPERA is currently described as a prototype implementation. Porting over to these systems is non-trivial, but is a definite potential avenue of future work. A more ambitious future direction for the PSec language would be to automatically infer `secure_machine` annotations on state machines. In this approach, programmers would simply designate certain variables as `secure`. The compiler will ensure that these variables are not stored in untrusted memory by appropriately marking machines that handle this information as `secure_machine` and potentially creating intermediate secure environments as needed. This approach would enable an even lesser annotation overhead for the programmer since the compiler will learn the required annotations to ensure that secrets remain within trusted environments.

Acknowledgments

We dedicate this paper to the memory of Pramod Subramanyan (the third author) who tragically passed away while we were working on this project. Pramod was a brilliant researcher, a wonderful mentor and colleague, and a prime contributor to this paper; without him, it would not have been possible.

This work was supported in part by NSF grant CNS-1739816 and gifts from Microsoft and Intel. The second author started this work and made his main contributions while affiliated with UC Berkeley.

REFERENCES

[1] Guoxing Chen, Yinqian Zhang, and Ten-Hwang Lai. 2019. OPERA: Open Remote Attestation for Intel's Secure Enclaves. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*. 15 pages.

[2] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. 2007. Secure Web Applications via Automatic Partitioning. *SIGOPS Oper. Syst. Rev.* (Oct. 2007), 31–44.

[3] Stephen Chong, K. Vikram, and Andrew C. Myers. 2007. SIF: Enforcing Confidentiality and Integrity in Web Applications. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium (SS'07)*. Article 1.

[4] M. R. Clarkson, S. Chong, and A. C. Myers. 2008. Civitas: Toward a Secure Voting System. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*. 354–368.

[5] V. Costan and S. Devadas. 2016. Intel SGX Explained. *IACR Cryptol. ePrint Arch.* 2016 (2016), 86.

[6] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. 2013. P: Safe Asynchronous Event-Driven Programming. *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2013).

[7] Ankush Desai, Amar Phanishayee, Shaz Qadeer, and Sanjit Seshia. 2018. Compositional Programming and Testing of Dynamic Distributed Systems. In *ACM OOPSLA 2018*.

[8] Anitha Gollamudi and Stephen Chong. 2016. Automatic Enforcement of Expressive Security Policies Using Enclaves. *SIGPLAN Not.* 51, 10 (Oct. 2016), 494–513.

[9] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache Attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security* (Belgrade, Serbia) *(EuroSec'17)*.

[10] Daniel Gruss, Julian Lettner, Felix Schuster, Olga Ohrimenko, Istvan Haller, and Manuel Costa. 2017. Strong and Efficient Cache Side-Channel Protection Using Hardware Transactional Memory. In *Proceedings of the 26th USENIX Conference on Security Symposium (SEC'17)*. 217–233.

[11] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. 2013. 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)*. Article 11.

[12] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. 2016. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 533–549.

[13] Richard Leadbetter. 2020. *PS3 Security In Tatters.* Retrieved May 1, 2020 from https://openenclave.io/sdk/

[14] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. 2009. Fabric: A Platform for Secure Distributed Computation and Storage. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*. 321–334.

[15] Microsoft. 2020. *Microsoft Open Enclave SDK.* Retrieved January 1, 2020 from https://openenclave.io/sdk/

[16] Vasily A. Sartakov, Stefan Brenner, Sonia Ben Mokhtar, Sara Bouchenak, Gaël Thomas, and Rüdiger Kapitza. 2018. EActors: Fast and Flexible Trusted Computing Using SGX. In *Proceedings of the 19th International Middleware Conference (Middleware '18)*. 187–200.

[17] Vinnie Scarlata, Simon Johnson, James Beaney, and Piotr Zmijewski. 2018. *Supporting Third Party Attestation for Intel SGX with Intel Data Center Attestation Primitives.* Technical Report.

[18] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. 2015. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *2015 IEEE Symposium on Security and Privacy*. 38–54.

[19] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Network and Distributed System Security Symposium 2017 (NDSS'17)*.

[20] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. 2016. Preventing Page Faults from Telling Your Secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIA CCS '16)*.

[21] Rohit Sinha, Sriram Rajamani, Sanjit A. Seshia, and Kapil Vaswani. 2015. Moat: Verifying Confidentiality of Enclave Programs. In *The ACM Conference on Computer and Communications Security (CCS)*.

[22] Hongliang Tian, Qiong Zhang, Shoumeng Yan, Alex Rudnitsky, Liron Shacham, Ron Yariv, and Noam Milshten. 2018. Switchless Calls Made Practical in Intel SGX. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution (SysTEX '18)*.

[23] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. 2000. A Sound Type System For Secure Flow Analysis. *Journal of Computer Security* 4 (08 2000).

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

[25] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. 2002. Secure Program Partitioning. *ACM Trans. Comput. Syst.* (Aug. 2002).

A PSEC INFORMATION FLOW TYPING EXAMPLE

```
trusted event tEvent : secure_int;
var ssn : secure_int;
var insecure_int : int;
var s_int : secure_int;
var s_handle : secure_machine;
var handle : machine;
...
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;

    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, tEvent, s_int;
    }

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

B OPERATIONAL SEMANTICS

We build on top of the operational semantics of P, presented in [7], and the notation specified in Section 4.2.

B.1 Additional Notation

(1) Let $\mathcal{M}$ represent the set of names of all the state machines, $\mathcal{M}_{\mathcal{T}}$ represents SSMs, and $\mathcal{M}_{\mathcal{U}}$ represents USMs
(2) Let $\mathcal{H}$ be the set of all machine handles (used to send events to state machines), which is $\mathcal{M} \times \mathbb{N} \times \mathbb{X}$, where $\mathbb{N}$ is the set of natural numbers and $\mathbb{X} \in \{0, 1\}$. Let $\mathcal{H}_{\mathcal{T}}$ represent the set of trusted machine handles (represented as $\mathcal{M} \times \mathbb{N} \times 1$) and $\mathcal{H}_{\mathcal{U}}$ represent the set of untrusted machine handles ($\mathcal{M} \times \mathbb{N} \times 0$)
(3) Let $\mathcal{V}_{\mathcal{T}}$ represent the set of all possible payloads that may be encapsulated in a PSec trusted event and $\mathcal{V}_{\mathcal{U}}$ represent those in a PSec untrusted event. Let $\mathcal{V}$ be $\mathcal{V}_{\mathcal{T}} \cup \mathcal{V}_{\mathcal{U}}$

We define various transition relations and functions below:

(1) $Local \subseteq \mathcal{S} \times \mathcal{H} \times \mathcal{S} \times \mathcal{H}$ 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) $Enq \subseteq \mathcal{S} \times \mathcal{H} \times \mathcal{E} \times \mathcal{V} \times \mathcal{S}$ 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 machine with handle id
(3) $Rem \subseteq \mathcal{S} \times \mathcal{B} \times \mathbb{N} \times \mathcal{S}$ 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) $New \subseteq \mathcal{S} \times \mathcal{M} \times \mathcal{S}$ 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 same for value v. We define $tids$ similarly for trusted machine handles. ids is a function that is $uids \cup tids$
(6) We define a function $uvals$ that returns a map of non-sensitive variables to their values in state s. We define $tvals$ similarly for secret variables. $vals$ maps all variables to their values in state s
(7) We define a helper function $containsAll(A, B)$ that returns true if map A contains all key value pairs 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 given IFC rules. 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" We need to formalize the concept that these machine handles cannot be created "out of thin air" [7] and must be present in local state 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 through an event (Rem). We formalize this as follows:
$\forall\ m \in \mathcal{M},\ id, id' \in \mathcal{H},\ s, s' \in \mathcal{S},\ e \in \mathcal{E},\ v \in \mathcal{V},\ n \in \mathbb{N},\ b \in \mathcal{B}$

(1) $(s, id, s', id') \in Local(m) \Rightarrow ids(s') \cup id' \subseteq ids(s) \cup \{id\}$
(2) $(s, b, n, s') \in Rem(m) \Rightarrow ids(s') \subseteq ids(s) \cup \{ids(v) \mid \exists e.b[n] = (e, v)\}$
(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 We need to make sure that local state flows correctly propagate sensitive state. We formalize the following with the assumption our information flow type system guarantees certain properties:
$\forall\ m \in \mathcal{M},\ id, id' \in \mathcal{H},\ s, s' \in \mathcal{S},\ e \in \mathcal{E},\ v \in \mathcal{V},\ n \in \mathbb{N},\ b \in \mathcal{B}$

(1) $(s, id, s', id') \in Local(m) \Rightarrow\ IFA(s, s')$
(2) $(s, b, n, s') \in Rem(m) \Rightarrow$
$containsAll(vals(s'),\ vals(s).put(\ (x, v) \mid \exists e.b[n] = (e, v)))$
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. As stated earlier in Section 4.2, the configuration of our system is the following tuple: (S, B, C).

B.2 Rules

We depict all of the rules in Figure 7. The labels on the transitions indicate the information observable by an adversary.

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. For the *dequeue event* transition, an observer is assumed to be able to infer the new untrusted state as well as the content of any dequeued untrusted event. Recall that dequeuing an event can add a handle to the current machine's local state if another machine has sent it.

Creation Rules We have a parent state machine (denoted with subscript p) creating a child state machine (subscript c) (both these state machines must be SSMs in the *trusted create* case). The parent machine will receive the handle of the child machine and an outside observer can see which child machine is created as well as which machine sent the creation request.

Sending Rules In the *trusted send* rule, we have a sending SSM (subscript s) sending a trusted event to a receiving SSM (subscript r) by using its trusted handle, and we have a similar action in the *untrusted send* rule (with an untrusted event and untrusted handle). After executing this command, the sending machine transitions to its next state and the receiving machine enqueues this event in its input buffer. For a *trusted send*, 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 parties the message is being exchanged between. In contrast, for an *untrusted send*, an observer can additionally view the message payload.

C OBSERVATIONAL DETERMINISM SECURITY PROOFS

In this section, we want to formally prove Theorem 4.1, which can be divided into confidentiality and integrity components.

C.1 PSec Confidentiality Proof

We need to prove the following for our system:

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

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 $Obs_{\mathcal{L}}(G_1^k, G_1^{k+1}) = Obs_{\mathcal{L}}(G_2^k, G_2^{k+1})$, then $G_1^{k+1} \approx_{\mathcal{L}} G_2^{k+1}$. We continue via a proof by cases:

(1) Create Transitions:
- $Obs_{\mathcal{L}}(G_1^k, G_1^{k+1}) = Obs_{\mathcal{L}}(G_2^k, G_2^{k+1}) = < (m_p, n_p), (m_c, n_c) >$
- The parent machine identity and receiving machine identity is assumed to be observable, which means they must be identical in corresponding transitions. Since PSec has deterministic transitions, S^k and C^k change in the same way for both configurations while B^k remains unchanged.

(2) Trusted Send Transition:
- $Obs_{\mathcal{L}}(G_1^k, G_1^{k+1}) = Obs_{\mathcal{L}}(G_2^k, G_2^{k+1}) = < (m_s, n_s), (m_r, n_r), e >$
- The sending machine identity, receiving machine identity, and event type must be identical across the corresponding transitions, but the non-observable difference between the two transitions is the message payload. The receiving machine adds the payload to its trusted local state, thus changing its *tvals*. However, $uvals(G_1^{k+1}) = uvals(G_2^{k+1})$ which means $S_1^{k+1} \approx_{\mathcal{L}} S_2^{k+1}$. Since the enqueued event is a trusted event, $B_1^{k+1} \approx_{\mathcal{L}} B_2^{k+1}$. C^k remains unchanged.

(3) Untrusted Send Transition:
- $Obs_{\mathcal{L}}(G_1^k, G_1^{k+1}) = Obs_{\mathcal{L}}(G_2^k, G_2^{k+1}) = < (m_s, n_s), (m_r, n_r), e, v >$
- The sending machine identity, receiving machine identity, event type, and message payload must be identical. Therefore, S^k and B^k change in the same way for both configurations while C^k remains unchanged.

(4) Local Transition:
- $Obs_{\mathcal{L}}(G_1^k, G_1^{k+1}) = Obs_{\mathcal{L}}(G_2^k, G_2^{k+1}) = < uvals(s') >$
- The untrusted values stored in the local state of the machine are assumed to be observable (since they can be leaked through side channels). This must mean that USMs undergo the exact same transition in both configurations. For SSMs, this only applies if they only modify their untrusted state in this transition, which again implies that the same transition must take place in both configurations. The non-observable difference between the two transitions for SSMs would be changes in trusted state (modifying *tvals*). After this transition takes place, $uvals(G_1^{k+1}) = uvals(G_2^{k+1})$ since these values are not modified, so $S_1^{k+1} \approx_{\mathcal{L}} S_2^{k+1}$ (B_k and C_k remain unchanged).

(5) Dequeue Transition:
- $Obs_{\mathcal{L}}(G_1^k, G_1^{k+1}) = Obs_{\mathcal{L}}(G_2^k, G_2^{k+1}) = < uvals(s'), e,\ (v | e \in \mathcal{E}_{\mathcal{U}}) >$
- The untrusted local state values as well as any newly dequeued values (if non-sensitive) of the machine are assumed to be observable. This means that USMs (since they only have untrusted state and can only receive untrusted events) or SSMs dequeuing untrusted events must undergo the exact same transition in both configurations. The non-observable difference between the two transitions would be the dequeuing of trusted events by SSMs. The SSM may receive sensitive data through the trusted event, thus modifying its *tvals*. However, $uvals(G_1^{k+1}) = uvals(G_2^{k+1})$ regardless of the trusted event payload, so $S_1^{k+1} \approx_{\mathcal{L}} S_2^{k+1}$. Since the untrusted events in the buffer are not modified in both configurations, $B_1^{k+1} \approx_{\mathcal{L}} B_2^{k+1}$. C_k in this case remains unchanged.

Since we have proven that $G_1^k \approx_{\mathcal{L}} G_2^k \implies G_1^{k+1} \approx_{\mathcal{L}} G_2^{k+1}$ for all cases, we have proven PSec satisfies the confidentiality property of observational determinism.

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

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

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

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

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

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

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

Figure 7: PSec **Operational Semantics Rules**

C.2 PSec Integrity Proof

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 have the following:

$$v_1 \approx_{\mathcal{H}} v_2 \ \Leftrightarrow\ \Gamma \vdash v_i : \tau \ \wedge\ (\tau = H \ \Rightarrow\ v_1 = v_2)$$
$$S_1 \approx_{\mathcal{H}} S_2 \ \Leftrightarrow\ tvals(S_1) = tvals(S_2)$$
$$B_1 \approx_{\mathcal{H}} B_2 \ \Leftrightarrow$$
$$filter(B_1,\ \lambda(e_1,v_1).\ e_1 \in \mathcal{E}_{\mathcal{T}}) = filter(B_2,\ \lambda(e_2,v_2).\ e_2 \in \mathcal{E}_{\mathcal{T}})$$
$$C_1 \approx_{\mathcal{H}} C_2 \ \Leftrightarrow\ \forall m \in \mathcal{M}.\ C_1[m] = C_2[m]$$
$$G_1 \approx_{\mathcal{H}} G_2 \ \Leftrightarrow\ (S_1 \approx_{\mathcal{H}} S_2) \ \wedge\ (B_1 \approx_{\mathcal{H}} B_2) \ \wedge\ (C_1 \approx_{\mathcal{H}} C_2)$$
$$\pi_1 \approx_{\mathcal{H}} \pi_2 \ \Leftrightarrow\ \forall i \in [0,n).\ G_1^i \approx_{\mathcal{H}} G_2^i$$

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.

(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*:
$Obs_{\mathcal{H}}(G^t, G^{t+1}) = < tvals(s'),e,\ (v|e \in \mathcal{E}_{\mathcal{T}}) >$

We need to prove the following for our system:

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

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 $Obs_{\mathcal{H}}(G_1^k, G_1^{k+1}) = Obs_{\mathcal{H}}(G_2^k, G_2^{k+1})$, then $G_1^{k+1} \approx_{\mathcal{H}} G_2^{k+1}$.

The proof proceeds in a very similar fashion to the confidentiality proof, and we are able to similarly show that $G_1^k \approx_{\mathcal{H}} G_2^k \implies G_1^{k+1} \approx_{\mathcal{H}} G_2^{k+1}$ for all cases, thus proving PSec satisfies the integrity property of observational determinism.