PREPOSE: Privacy, Security, and Reliability for Gesture-Based Programming

Lucas Silva Figueiredo[*], David Molnar[†], Margus Veanes[†], and Benjamin Livshits[†]
Federal University of Pernambuco[*] Microsoft Research[†]

Abstract—With the rise of sensors such as the Microsoft Kinect, Leap Motion, and hand motion sensors in phones (i.e., Samsung Galaxy S6), gesture-based interfaces have become practical. Unfortunately, today, to recognize such gestures, applications must have access to depth and video of the user, exposing sensitive data about the user and her environment. Besides these privacy concerns, there are also security threats in sensor-based applications, such as multiple applications registering the same gesture, leading to a conflict (akin to Clickjacking on the web).

We address these security and privacy threats with PREPOSE, a novel domain-specific language (DSL) for easily building gesture recognizers, combined with a system architecture that protects privacy, security, and reliability with untrusted applications. We run PREPOSE code in a trusted core, and only return specific gesture events to applications. PREPOSE is specifically designed to enable precise and sound static analysis using SMT solvers, allowing the system to check security and reliability properties *before* running a gesture recognizer. We demonstrate that PREPOSE is expressive by creating a total of 28 gestures in three representative domains: *physical therapy*, *tai-chi*, and *ballet*. We further show that runtime gesture matching in PREPOSE is fast, creating no noticeable lag, as measured on traces from Microsoft Kinect runs.

To show that gesture checking at the time of submission to a *gesture store* is fast, we developed a total of four Z3-based static analyses to test for basic gesture safety and internal validity, to make sure the so-called protected gestures are not overridden, and to check inter-gesture conflicts. Our static analysis scales well in practice: safety checking is under 0.5 seconds per gesture; average validity checking time is only 188 ms; lastly, for 97% of the cases, the conflict detection time is below 5 seconds, with only one query taking longer than 15 seconds.

1 Introduction

Over 20 million Kinect sensors are in use today, bringing millions of people in contact with games and other applications that respond to voice and gestures. Other companies such as Leap Motion and Prime Sense are bringing low-cost depth and gesture sensing to consumer electronics. The newest generation of smart phones such as Samsung Galaxy S5 supports rudimentary gestures as well.

Context of prior work: The security and privacy community is starting to pay attention to concerns created by the emergence of these technologies. Specifically, we have seen several proposals on the intersection of augmented reality, privacy, and security. D'Antoni *et al.* [8] provides a high-level overview of the problem space. Darkly [16], like our work, puts a layer between the untrusted application and raw sensor data. Unlike us, Darkly lacks a formal semantics and does not allow precise reasoning about application properties. Jana *et al.* [15] introduces the notion of an OS abstraction called a recognizer which enables gesture detection. Yet their approach fails to provide a way to extend the system with new recognizers in a safe manner. SurroundWeb [34] demonstrates what a 3D web browser modified with new abstractions for input and output to protect privacy and security would look like. Yet it also lacks the capacity for precise automatic reasoning. We are also inspired by world-drive access control [30], which attempts to restrict applications from accessing sensitive objects in the environment. Lastly, Proton [19] is an example of defining a higher-level abstraction for gestures that enables precise reasoning.

1.1 Background

User demand for sensors such as Kinect is driven by exciting new applications, ranging from immersive Xbox games to purpose-built shopping solutions to healthcare applications for monitoring elders. Each of these sensors comes with an SDK which allows third-party developers to build new and compelling applications. Several devices such as Microsoft Kinect and Leap Motion use the *App Store* model to deliver software to the end-user. Examples of such stores include Leap Motion's Airspace `airspace.com`, Oculus Platform, and Google Glassware `http://glass-apps.org`.

These platforms will evolve to support multiple *untrusted applications* provided by third parties, running on top of a *trusted core* such as an operating system. Since such applications are likely to be distributed through centralized App stores, there is a chance for application analysis and enforcement of key safety properties. Below we describe some of the specific threats posed by applications to each other and to the user. We refer the reader to D'Antoni [8] for a more comprehensive discussion of threats. To address these threats, we introduce PREPOSE, a novel domain specific language and runtime for writing gesture recognizers. We designed this language with semantics in terms of SMT formulas. This allows us to use the state of the art SMT solver Z3 both for static analysis and for runtime matching of gestures to user movements.

1.2 A Case for Controlled Access to Skeletal Data

There is a natural trade-off between the platform functionality provided to potentially untrusted applications and possible threats to the end-user. We take a two-pronged approach to deliver a degree of security, privacy, and reliability. Privacy is achieved through the use of a domain-specific language PREPOSE, whereas security and reliability are both achieved through the use of sound static analysis. By combining system design and sound static analysis, PREPOSE improves the security, privacy, and reliability properties of gesture programming. We discuss privacy-related issues in this section and security and reliability in Section 1.3.

PREPOSE raises the privacy bar, keeping in mind that perfect privacy is elusive. The degree to which end-users are comfortable with privacy disclosure varies considerably as well. Therefore it is important to analyze different points in the design space for untrusted applications that use gesture recognition.

Figure 1 summarizes three different levels of functionality for untrusted applications that need gesture recognition. On the bottom, applications can be written in languages such as C++ and have access to raw video and depth. Access to the raw video stream is seen as highly privacy-sensitive [15, 34]. In the middle, applications are written in memory-safe languages such as C# or Java and have access only to the skeleton API provided by Kinect for Windows. What is less obvious is that at the middle level, the skeleton data also leads to potential loss of privacy. Specifically, the following attacks are possible

- The skeleton API reveals how many people are in the room. This may reveal whether the person is alone or not. If alone, perhaps she is a target for robbery; if she's found to be not alone, that may reveal that she's involved with someone illicitly.
- The skeleton API reveals the person's height (relative height of joints is exposed, and the Kinect API allows mapping from skeleton points to depth space so actual height as well). The application could distinguish people by "fingerprinting" skeletons.
- The skeleton API reveals fine grained position of the person's hands. The application can in principle learn something about what they write if they write on a whiteboard, for example.

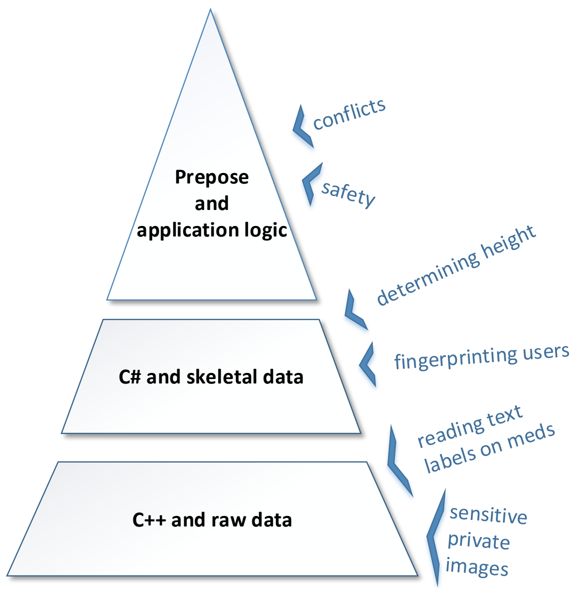


Fig. 1: Three different levels of data access for untrusted applications that perform gesture recognition. We call out threats to the user at each levels.

Category	Property	Description
Reliability	gesture safety	validates that gestures have a basic measure of physical safety, i.e. do not require the user to overextend herself physically in ways that may be dangerous.
Reliability	inner validity	checks for inner contradictions i.e. do not require the user to both keep her arms up *and* down.
Security	protected gestures	tests whether a gesture conflicts with a reserved system-wide gesture such as the Kinect attention gesture (`http://bit.ly/1JlXk79`).
Security	conflicts	finds potential conflicts within a set of gestures such as two gestures that would both be recognized from the same user movements.

Fig. 2: Properties statically checked by PREPOSE. The first two properties are reliability properties which aid gesture developers. The second two are security properties that prevent untrusted applications from conflicting with the OS or with other applications.

1.3 Static Analysis for Security & Reliability

At the heart of PREPOSE is the idea of compiling gesture descriptions to formulae for an SMT solver such as Z3 [26]. These formulae capture the semantics of the gestures, enabling precise analyses that boil down to satisfiability queries to the SMT solver. The PREPOSE language has been designed to be both expressive enough to support complex gestures, yet restrictive enough to ensure that key properties remain decidable. In this paper we focus on the four properties summarized in Figure 2 and detailed in Section 3.4. Note that a gesture-based application written in C++ or Java would generally require an extensive *manual audit* to ensure the lack of privacy leaks and security flaws.

1.4 Threat Model

PREPOSE, at the top of the pyramid in Figure 1, provides the next layer of privacy by mediating *direct* access to the skeleton API. While the threats emanating from raw video access and skeleton access are eliminated by design, in PREPOSE we worry about higher-level properties such as inter-gesture conflicts and gesture safety.

This is akin to how programming in a memory-safe language allows one to focus on enforcing semantic security properties without worrying about buffer overruns. As a matter of security and privacy in depth, PREPOSE is at the higher level within the pyramid, following the classic security principle of least privilege.

As is often the case with privacy mechanisms, there are some side channels that are harder to protect from. In

```
GESTURE crossover-left-arm-stretch:
   POSE relax-arms:
       point your left arm down,
       point your right arm down.

   POSE stretch:
       rotate your left arm 90 degrees counter
           clockwise on the frontal plane,
       touch your left elbow with your right hand.

   EXECUTION:
       relax-arms,
       slowly stretch and hold for 30 seconds.
```

Fig. 3: Gesture example: `crossover-left-arm-stretch`. A gesture is composed of a sequence of poses. The gesture is completed if the poses are matched in the sequence specified in the `EXECUTION` block.

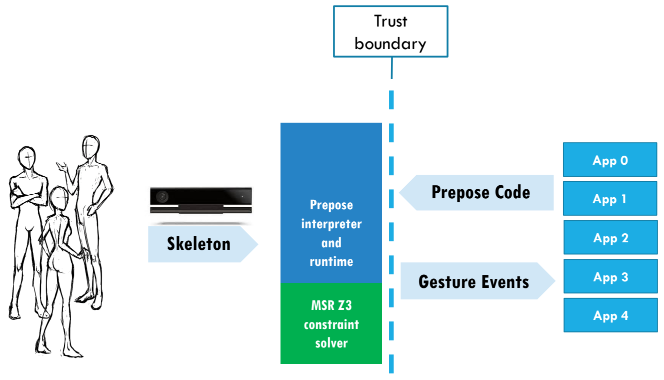


Fig. 4: Security architecture of Prepose.

our scenario, Prepose does not directly protect against tracking the user by learning which gestures they can perform (only some users are capable of certain gestures) or whether, for example, their house is big enough by testing if the user is able to perform gestures that require a greater freedom of movement.

While we do not to attempt to catalog all the possible attacks that may emerge [8], relying on Prepose gives us confidence that untrusted applications can do less harm than if they had additional capabilities (lower within the pyramid).

1.5 Prepose Architecture

The trusted core of Prepose enforces privacy by mediating between applications and the raw sensor data. Inter-application conflicts and unsafe gestures are avoided through static analysis powered by the Z3 SMT solver. Figure 4 shows our architecture and the security boundary we draw.

Gesture store: We are also inspired by App Stores for *developer components*, such as the Unity 3D Asset store which offers developers the ability to buy models, object, and other similar components (`https://www.assetstore.unity3d.com`). Today, when developers write their own gesture recognizers from scratch, they use machine learning methods, or libraries from github and sourceforge. Our focus in this paper is on *gesture recognizers*, which are integral components of AR applications responsible for detecting gestures performed by users.

As in the case of mobile apps, the App Store centralized distribution model provides a unique opportunity to ensure the security and privacy of gestures *before* they are unleashed on unsuspecting users. As such, our approach in Prepose is to check gestures when they are *submitted* to the gesture store.

Figure 5 summarizes our approach. Developers write gesture recognizers in a high-level domain-specific language, Prepose, then submit them to the gesture store. Because our domain-specific language has been carefully engineered, we can perform precise and sound static analyses for a range of security and privacy properties. The results of this analysis tell us whether the submitted gesture is "definitely OK," "definitely not OK," or, as may happen occasionally, "needs attention from a human auditor." In our experiments in Section 5, we encountered only one case of reasoning needing attention. A reasonable approach would be to reject submissions that do not qualify as "definitely OK."

Improving gesture authoring experience: In addition to addressing threats from untrusted applications, a language-based approach can improve gesture authoring. Gestures are an integral part of sensor-based always-on application, the equivalent of UI events like *left mouse click*, *double-click*, etc. in regular applications[1]. While, for instance, the Kinect SDK already includes a number of default gestures, developers typically need to add their own. Different applications often require different sets of gestures, and, as such, building new gestures is a fundamental part of software development.

Gesture development is a tricky process, which often depends on machine learning techniques requiring large volumes of training data [9]. These heavyweight methods are both expensive and time-consuming for many developers, resulting in mostly large game studios being able to afford gesture development. Therefore, making gesture development easier would unlock the creativity of a larger class of developers. Prepose aids this with sound static analyses for *reliability* properties of gestures, such as whether the gesture definition is self-contradictory.

Prepose language and runtime: This paper proposes Prepose, a language and a runtime for authoring and checking gesture-based applications. For illustration, a code snippet supported by our system in shown in Figure 3. This code is translated into logical formulas which are checked at runtime against the user's actual positions using an SMT solver.

Prepose is built as a library on top of the released Kinect SDK. Applications link against this library. The source code of Prepose is available on Github (URL omitted for anonymity). Prepose lowers the cost of developing

[1]To quote a blog entry: "After further experimenting with the Kinect SDK, it became obvious what needed to come next. If you were to create an application using the Kinect SDK, you will want to be able to control the application using gestures (i.e. waving, swiping, motions to access menus, etc.)." [32]

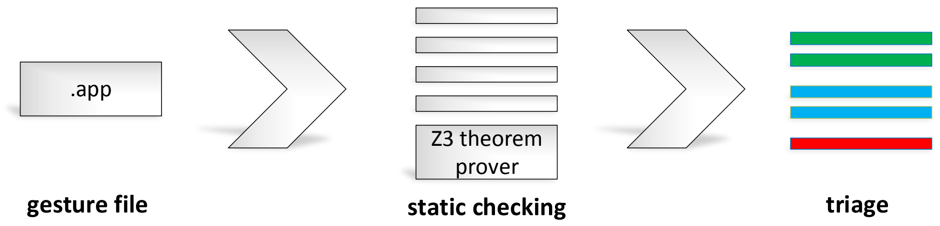


Fig. 5: Checking submissions to a gesture store. Submissions are marked as *safe* (green), *unsafe* (red), or *need human attention* (blue).

new gestures by exposing new primitives to developers that can express a wide range of natural gestures.

Application domains implemented in Prepose**:** To demonstrate the expressiveness of Prepose, we experiment with three domains that involve different styles of gestures: physical therapy, dance, and tai-chi. Given the natural syntax of Prepose and a flat learning curve, we believe that other domains can be added to the system quite easily. For each of these gestures, we then performed a series of analyses enabled by Prepose, including conflict detection, as well as safety, security, and privacy checks.

Monitoring applications in Prepose**:** We discovered that Prepose is particularly well-suited to what we call *monitoring applications* which can be implemented with Prepose gestures and a small amount of "bookkeeping" code. For example, Kinect Sports includes a tai-chi trainer, which instructs users to struck tai-chi poses and gives real-time feedback on how well they do, which is easily captured by Prepose and supported by the runtime we have built. For another example, Atlas5D is a startup that installs multiple sensors in the homes of seniors and monitors seniors for any signs of a fall or another emergency. Another example of such an application for physical therapy is shown in Figure 8a or can be seen in a video at `http://reflexionhealth.com`. These applications can run, concurrently, for weeks on end, with only minimal needs to report results (such as completing a certain level within the tai-chi application) to an external server.

1.6 Contributions

Our paper makes the following contributions:

- **Prepose.** Proposes a programming language and a runtime for a broad range of gesture-based immersive applications designed from the ground up with security and privacy in mind. Prepose follows the principle of *privacy by construction* to eliminate the majority of privacy attacks.
- **Static analysis.** We propose a set of static analysis algorithms designed to soundly find violations of important security and reliability properties. This analysis is designed to be run within a gesture App Store to prevent malicious third-party applications from affecting the end-user.
- **Expressiveness.** To show the expressiveness of Prepose, we encode 28 gestures for 3 useful application domains: *therapy*, *dance*, and *tai-chi*.
- **Performance evaluation.** Despite being written in a domain-specific language (DSL), Prepose-based gesture applications pay a minimal price for the extra security and privacy guarantees in runtime overhead; tasks like pose matching take milliseconds. Our static analysis scales well in practice: safety checking is under 0.5 seconds per gesture; average validity checking time is only 188 ms; lastly, for 97% of the cases, the conflict detection time is below 5 seconds, with only one query taking longer than 15 seconds.

We also to wish to point out that the Prepose project is open-sourced at `https://github.com/Microsoft/prepose`.

1.7 Paper Organization

The rest of the paper is organized as follows. Section 2 provides some background on gesture authoring. Section 3 gives an overview of Prepose concepts and provides some motivating examples. Section 4 describes our analysis for security and privacy in detail. Section 5 contains the details of our experimental evaluation. Sections 7 and 9 describe related work and conclude.

2 Background

2.1 Security and Privacy Threats in AR

Augmented reality (AR) is computing that overlays artificial objects on top of the human senses such that the artificial and the real seamlessly blend together. Today, shipping AR experiences come in form factors ranging from "magic windows" on phones and tablets all the way to high end headsets such as the Meta or the Microsoft HoloLens that add visual and audio objects to the user's world.

An example of a magic window is Pokemon Go, which became an overnight success by asking people to look through their phones to capture Pokemon, while moving around in the real world. For the headset, an example application is the Microsoft Galaxy Explorer, which lets the wearer to "fly through" the solar system and beyond, using eye gaze and gestures to pick out the next planet to visit.

The rise of fast phone processors, ever-cheaper MEMS gyroscopes, inertial sensing units, and advanced high-speed video processing for object registration means that AR capabilities, which used to cost hundreds of thousands of dollars, are now available on commodity phones. Even headsets have dropped to single thousands of dollars, making them within reach for enthusiasts and specialized commercial applications alike.

AR raises fundamental new challenges because, to work properly, applications must *continuously sense* the environment, and must overlay artificial objects on the real world. In most AR applications, interaction is accomplished through gestures or other visual recognition, which means that applications need some kind of access to video streams or they cannot work. How can we support untrusted applications, such as in a phone "app store" model or the Web model with untrusted pages? How can we prevent applications from maliciously "overwriting" real world objects or misleading the user?

At the same time, AR has familiar challenges as well. For example, applications may be written in game frameworks such as Unity, which has a "component store" allowing developers to buy new object recognition algorithms or specific 3D models, as they are needed. This store has the same tradeoffs as app stores on phones, game consoles, and tablets: how can we enable as many people as possible to sell components in the store, while still protecting the end-user from malicious code? What are the right abstractions and the right tradeoffs to strike? More generally, this is the problem of *safe extensions* for a core platform. For this paper, we focus on extensions that provide *gesture recognition*, described more fully below, because gestures are crucial for interacting with AR applications.

We now recap the existing single application model common for augmented reality applications on today's platforms. Next we discuss the challenges with extending this model to multiple untrusted applications in the same system. Then we describe our approach and give the necessary background on gesture programming to highlight how our approach works for this context.

2.2 Single Application Programming Model

Today, common AR applications for phones, for the Xbox game console, or for the Kinect sensor assume that a *single application* has control of the machine at one time. The application then typically includes a library that talks to the hardware, runs object recognition, and then exposes events to the application for processing. For example, a Kinect for Windows application includes a library that talks to the Kinect sensor, runs a machine learning model to extract the locations of people in the Kinect's field of view, and finally sends an event with detected skeleton positions to the application. A phone application may use a toolkit such as Vuforia to recognize markers in the world, or simply use location services to display relevant content, as in Pokemon Go. The key aspect of this model is that the application has complete control of the device.

2.3 Multiple Applications

Multiple applications sharing the same AR raise a host of issues, as summarized by D'Antoni *et al.* [8]. Here, we focus on the problem of *safe extensions* for gesture recognition. When there are multiple programs, it is not possible to give each one *exclusive* access to sensor data, such as a raw video or depth stream. Additionally, with untrusted programs, such as those found in app stores or on the Web, giving access to the raw sensor stream would reveal private information about the user. Previous work has addressed this by restricting access to only *recognizers*, special operating system abstractions that encapsulate object detection code [15]. The key downside of this approach is that it requires a fixed set of recognizers that cannot be changed by applications. For example, the Xbox Kinect game "NBA Baller Beats" included code to recognize a basketball bouncing up and down in time to music, which was never originally envisioned by the Kinect developers. For another example, the Windows 10 application platform for HoloLens as of August 2016 provides only a limited set of gestures for applications and no way for developers to register new gesture recognizers.

2.4 Programming with Gestures

As AR systems grow, so too does the field of security and privacy for these systems. We do not claim to solve every problem in this paper. Rather, our approach is to use a domain specific language that enables language-level sandboxing and precise analyses. The time to explore such approaches is now, while there is not yet a dominant AR platform. Because gesture recognition is a key component of these systems, we start there.

Today, developers of immersive, sensor-based applications pursue two major approaches to creating new gesture recognizers. First, developers write code that explicitly encodes the gesture's movements in terms of the Kinect Skeleton or other similar abstraction exposed by the platform. Second, developers use machine learning approaches to synthesize gesture recognition code from labeled examples. We discuss the pros and cons of each approach each in turn.

Manually written: In this approach, the developer first thinks carefully about the gesture movements in terms of an abstraction exposed by the platform. For example, the Kinect for Windows platform exposes a "skeleton" that encodes a user's joint positions. The developer then writes custom code in a general-purpose programming language such as C++ or C# that checks properties of the user's position and then sets a flag if the user moves in a way to perform the gesture. For example, the Kinect for Windows white paper on gesture development [20] contains code for a simple *punch* gesture, shown in Figure 6.

```
// Punch Gesture
if ( vHandPos.z-vShoulderPos.z>fThreshold1 &&
     fVelocityOfHand > fThreshold2 ||
     fVelocityOfElbow > fThreshold3 &&
     DotProduct(vUpperArm, vLowerArm) > fThreshold4)
{
     bDetect = TRUE;
}
```

Fig. 6: A simple punch gesture.

The code checks that the user's hand is "far enough" away from the shoulder, that the hand is moving "fast enough," that the elbow is also moving "fast enough," and that the angle between the upper and lower arm is greater than a threshold. If all these checks pass, the code signals that a *punch gesture* has been detected.

Manually-written poses require no special tools, data collection, or training, which makes them easy to start with. Unfortunately, they also have significant drawbacks.

- First, the code is hard to understand because it typically reasons about user movements at a low level. For example, the code uses a dot-product to check the angle between the lower and upper arm instead of an abstraction that directly returns the angle.

- Second, building these gestures requires a trained programmer and maintaining code requires manually tweaking threshold values, which may or may not work well for a wider range of users. Third, it is difficult to statically analyze this code because it is written in a general purpose programming language, so gesture conflicts or unsafe gestures must be detected at runtime.

- Finally, the manually coded gesture approach requires the application to have access to sensor data for the purpose of recognizing gestures. This raises privacy problems, as we have discussed: a malicious developer may directly embed some code to capture video stream or skeleton data to send it to `http://evil.com`.

Machine learning: The leading alternative to manually-coded gesture recognizers is to use *machine learning* approaches. In machine learning approaches, the developer first creates a *training set* consisting of videos of people performing the gesture. The developer then *labels* the videos with which frames and which portions of the depth or RGB data in the frame correspond to the gesture's movements.

Finally, the developer runs an existing machine learning algorithm, such as AdaBoost, to synthesize gesture recognition code that can be included in a program. Figure 7 shows the overall workflow for the Visual Gesture Builder, a machine learning gesture tool that ships with the Kinect for Windows SDK.

The developer takes recordings of many different people performing the same gesture, then tags the recordings to provide labeled data. From the labeled data, the developer synthesizes a classifier for the gesture. The classifier runs as a library in the application.

Machine learning approaches have important benefits compared to manually-written poses. If the training set contains a diverse group of users, such as users of different sizes and ages, the machine learning algorithm can "automatically" discover how to detect the gesture for different users without manual tweaking. In addition, improving the gesture recognition becomes a problem of data acquisition and labeling, instead of requiring manual tweaking by a trained programmer. As a result, many Kinect developers today use machine learning approaches.

On the other hand, machine learning has drawbacks as well. Gathering the data and labeling it can be expensive, especially if the developer wants a wide range of people in the training set. Training itself requires setting multiple parameters, where proper settings require familiarity with the machine learning approach used. The resulting code created by machine learning may be difficult to interpret or manually "tweak" to create new gestures. Finally, just as with manually written gestures, the resulting code is even more difficult to analyze automatically and requires access to sensor data to work properly.

3 Overview

We first show a motivating example in Section 3.1. Next, we discuss the architecture of Prepose and how it provides security and privacy benefits (3.2). We then introduce basic concepts of the Prepose language and discuss its runtime execution (3.3). Finally, we discuss the security and privacy issues raised by an App Store for gestures, and show how static analysis can address them (3.4).

3.1 Motivating Example

Existing application on Kinect: Figure 8a shows a screen shot from the Reflexion Health physical therapy product. The reader is strongly encouraged to watch the video at `http://reflexionhealth.com` for more context. Here, a Kinect for Windows is pointed at the user. An on-screen animation demonstrates a target gesture for the user. Along the top of the screen, the application gives an English description of the gesture. Also on screen is an outline that tracks the user's actual position, enabling the user to compare against the model. Along the top, the program also gives feedback in English about what movements the user must make to properly perform the therapy gesture.

Reflexion is an example of a broader class of *trainer applications* that continually monitor a user and give feedback on the user's progress toward gestures. The key point is that trainer applications all need to continuously monitor the user's position to judge how well the user performs a gesture. This monitoring is explicit in Reflexion Health, but in other settings, such as Atlas5D's eldercare, the monitoring may be implicit and multiple gestures may be tracked at once.

Encoding existing poses: We now drill down into an example to show how applications can encode gesture recognizers using the Prepose approach. Figure 8b shows a common ballet pose, taken from an instructional book on ballet. The illustration is accompanied by text describing the pose. The text states in words that ankles should be crossed, that arms should be bent at a certain angle, and so on.

Gestures in Prepose: Figure 8 shows the Prepose code which captures the ballet pose. Because of the way we have designed the Prepose language, this code is close to

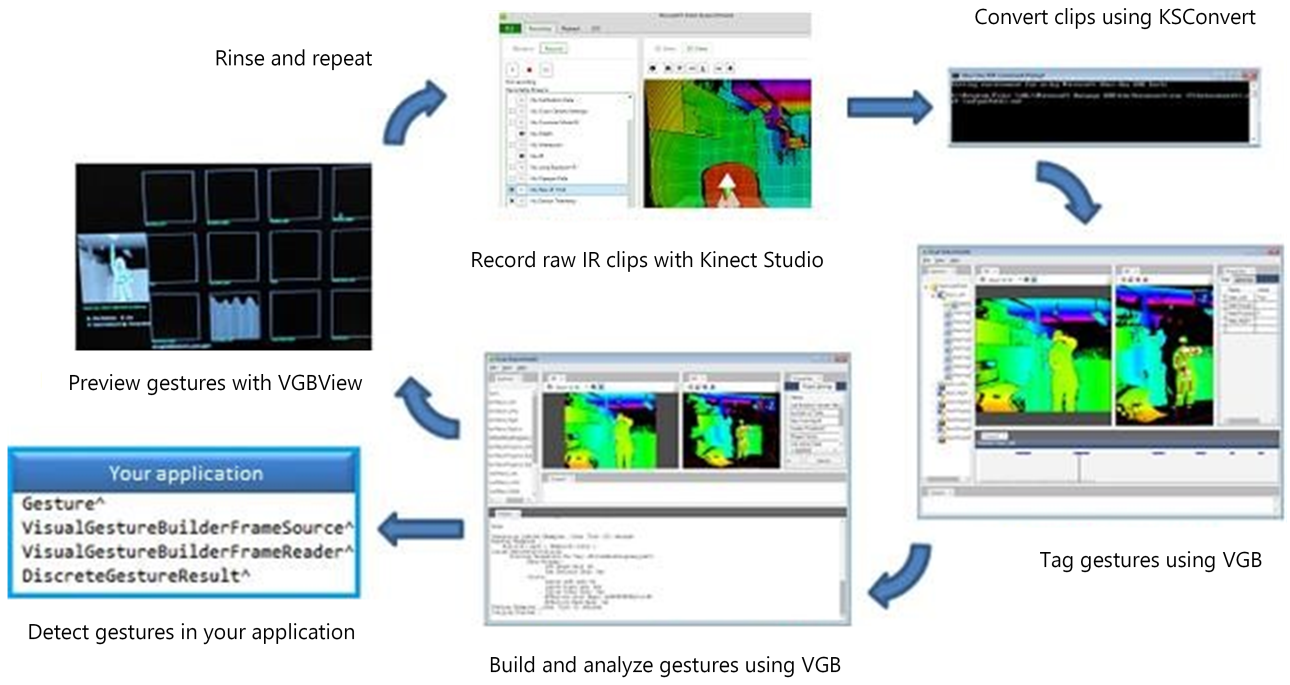

Rinse and repeat
Convert clips using KSConvert
Record raw IR clips with Kinect Studio
Preview gestures with VGBView
Your application
Gesture^
VisualGestureBuilderFrameSource^
VisualGestureBuilderFrameReader^
DiscreteGestureResult^
Detect gestures in your application
Build and analyze gestures using VGB
Tag gestures using VGB

Fig. 7: Workflow for machine-learning based gesture recognition creation in the Kinect Visual Gesture Builder [20].

the English description of the ballet pose. A ballet trainer application would include this code, which is then sent to the PREPOSE runtime for interpretation.

3.2 Architectural Goals

Figure 4 shows the architecture of PREPOSE. Multiple applications run concurrently. Each application has one or more gestures written in the PREPOSE language. These applications are not trusted and do not have access to raw sensor data. Instead, applications register their gesture code with a trusted PREPOSE runtime. This runtime is responsible for interpreting the gestures given access to raw depth, video, or other data about the user's position. When a gesture is recognized, the runtime calls back to the application which registered the gesture.

We draw a security boundary between the trusted component and untrusted applications. Only PREPOSE code crosses this boundary from untrusted applications to trusted components. In our implementation, the trusted component is written in managed C#, which makes it difficult for an untrusted application to cause a memory safety error. Our design therefore provides assurance that untrusted applications will not be able to access private sensor data directly, while still being able to define new gesture recognizers.

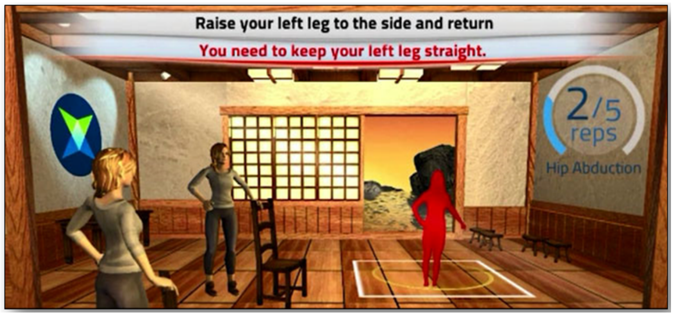
PREPOSE has been designed for analyzability. Developers submit code written in the PREPOSE language to a gesture App Store. During submission, we can afford to spend significant time (say, an hour or two) on performing static analyses. We now describe the specific security and privacy properties we support, along with the analyses needed to check them.

3.3 Basic Concepts in Prepose

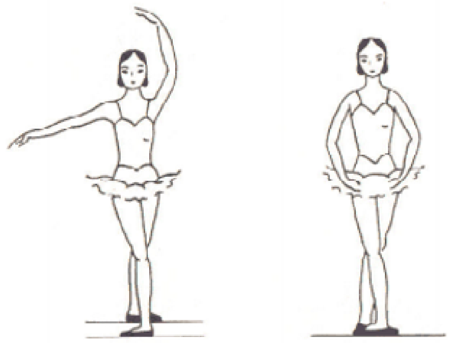
In contrast to the approaches above, PREPOSE defines a domain specific language for writing gesture recognizers. The basic unit of the PREPOSE language is the *pose*. A pose may contain *transformations* that specify the target position of the user explicitly, or it may contain *restrictions* that specify a range of allowed positions. A pose composes these transformations and restrictions to specify a function that takes a body position and decides if the position *matches* the pose. At runtime, PREPOSE applies this function to determine if the user's current body position matches the pose. For poses that consist solely of transformations, PREPOSE also at runtime synthesizes a *target position* for the user, enabling PREPOSE to measure how close the user is to matching the pose and provide real time feedback to the user on how to match the pose.

A *gesture* specifies a sequence of poses. The user must match each pose in the sequence provided. The gesture is said to match when the last pose in the sequence matches. At runtime, PREPOSE checks the user's body position to see if it matches the current pose.

In our current implementation, PREPOSE poses and gestures are written in terms of the *Kinect skeleton*. The Kinect skeleton is a collection of *body joints*, which are distinguished points in a three-dimensional coordinate space that correspond to the physical location of the user's head, left and right arms, and other body parts. Our approach, however, could be generalized to other methods of sensing gestures. For example, the Leap Motion hand sensor exposes a "hand skeleton" to developers and we could adapt the PREPOSE runtime to work with Leap Motion or other hand sensors.


Raise your left leg to the side and return
You need to keep your left leg straight.

(a) A physical therapy application. On the right, the application displays the user's current position. Along the top, the application describes the gesture the user must perform.



(b) Ballet poses.

```
GESTURE fourth-position-en-avant:
    POSE cross-legs-one-behind-the-other:
       put your left ankle behind your right ankle,
       put your left ankle to the right
                     of your right ankle.
       // do not connect your ankles.

    POSE high-arc-arms-to-right:
       point your arms down,
       rotate your right arm 70 degrees up,
       rotate your left elbow 20 degrees to your left,
       rotate your left wrist 25 degrees to your right.

    EXECUTION:
       // fourth-position-en-avant-composed
       stand-straight,
       point-feet-out,
       stretch-legs,
       cross-legs-one-behind-the-other,
       high-arc-arms-to-right.
```

(c) A sample ballet gesture written in PREPOSE. The gesture defines two *poses*, which are specifications of a body position. Then, the gesture *execution* specifies the sequence of poses that must be matched to perform the gesture.

Fig. 8: Motivating example.

Poses: A pose contains either *transformations* or *restrictions*. A transformation is a function that takes as input a Kinect skeleton and returns a Kinect skeleton. Transformations in PREPOSE include "rotate" and "point", as in this example PREPOSE code:

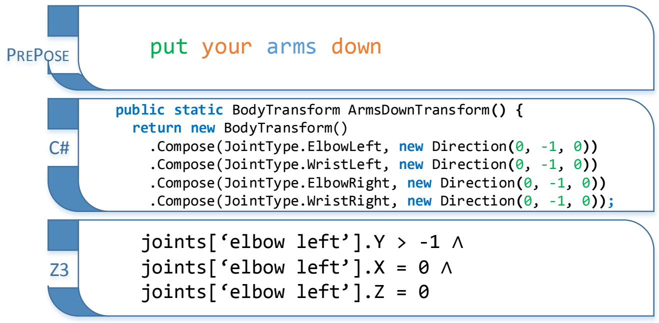
```
rotate your left wrist 30 degrees to the front
rotate your right wrist 30 degrees to the front
point your right hand up
```

In the first line, the transformation "rotate" takes as arguments the name of the user skeleton joint "left wrist," the amount of rotation "30 degrees," and the direction of rotation. The second line is similar. The third line is a transformation "point" that takes as arguments the name of a user skeleton joint and a direction "up." When applied to a skeleton position, the effect of all three transformations is to come up with a single new target skeleton for the user.

A restriction is a function that takes as input a Kinect skeleton, checks if the skeleton falls within a range of allowed positions, and then returns true or false. An example restriction in PREPOSE looks like this:

```
put your right hand on your head
```

The intuition here is that "on your head" is a restriction because it does not explicitly specify a single position. Instead, a range of allowed positions, namely those there the hand is within a threshold distance from the head, is denoted by this function. Here, the function "put" takes as arguments two joints, the "right hand" and the "head." The function returns true if the right hand is less than a threshold distance from the head and false otherwise. Poses can incorporate multiple transformations and multiple restrictions. The pose matches if all restrictions are true and the user's body position is also closer than a threshold to the target position.



```
put your arms down
```

```
public static BodyTransform ArmsDownTransform() {
  return new BodyTransform()
    .Compose(JointType.ElbowLeft, new Direction(0, -1, 0))
    .Compose(JointType.WristLeft, new Direction(0, -1, 0))
    .Compose(JointType.ElbowRight, new Direction(0, -1, 0))
    .Compose(JointType.WristRight, new Direction(0, -1, 0));
```

```
joints['elbow left'].Y > -1 ∧
joints['elbow left'].X = 0 ∧
joints['elbow left'].Z = 0
```

Fig. 9: Runtime correspondence: PREPOSE, C#, and Z3.

Gestures: Gestures consist of zero or more pose declarations, followed by an *execution sequence*. For example, a gesture for doing "the wave" might contain the following:

```
EXECUTION:
    point-hands-up,
    point-hands-forward,
    point-hands-down.
```

That is, to do "the wave," the user needs to put her hands up, then move her hands from there to pointing forward, and then finally point her hands downward. The gesture *matches* when the user successfully reaches the end of the execution sequence.

Our PREPOSE runtime allows multiple gestures to be loaded at a time. The execution sequence of a gesture can use any pose defined by any loaded gesture, which allows developers to build libraries of poses that can be shared by different gestures.

Runtime execution: Figure 9 shows the stages of runtime processing in PREPOSE. A high-level PREPOSE statement is compiled into C# code, which in turn defines an SMT formula. The formula is used both for runtime matching and static analysis.

3.4 Gesture Security and Reliability

At gesture submission time, we apply static analysis to the submitted PREPOSE program. This analysis can be performed within the App store before the user is allowed to download a new application that contains gestures. Conflict checking may also be done as information about which applications are installed is already available to the App store. Conceivably, the analysis may be done on the client as well. The results of this analysis tell us whether the submitted gesture is "definitely OK," "definitely not OK," or, as may happen occasionally, "needs attention from a human auditor." This kind of triage is fairly typical in the App store context.

We currently perform the four analyses summarized in Figure 2. As we explain below, this analysis amounts to queries resolved by the underlying SMT solver, Z3.

Gesture safety: The first analysis is for *gesture safety*. Just because it's possible to ask someone to make a gesture does not mean it is a good idea. A gesture may ask people to overextend their limbs, make an obscene motion, or otherwise potentially harm the user. To prevent an unsafe gesture from being present in the store, we first define *safety restrictions*. Safety restrictions are sets of body positions that are not acceptable. Safety restrictions are encoded as SMT formulas that specify disallowed positions for Kinect skeleton joints.

Internal validity: It is possibly in PREPOSE to write a gestures that can never be matched. For example, a gesture that requires the user to keep their arms both up *and* down contains an internal contradiction. We analyze PREPOSE gestures to ensure they lack internal contradictions.

Reserved gestures: A special case of conflict detection is detecting overlap with *reserved gestures*. For example, the Xbox Kinect has a particular *attention gesture* that opens the Xbox OS menu even if another game or program is running. Checking conflicts with reserved gestures is important because applications should not be able to "shadow" the system's attention gesture with its own gestures.

Conflict detection: We say that a pair of gestures *conflicts* if the user's movements match both gestures simultaneously. Gesture conflicts can happen accidentally, because gestures are written independently by different application developers. Alternatively, a malicious application can intentionally register a gesture that conflicts with another application. In PREPOSE, because all gestures have semantics in terms of SMT formulas, we can ask a solver if there exists a sequence of body positions that matches both gestures. If the solver completes, then either it certifies that there is no such sequence or gives an example.

```
Declarations
  app                  ::=  APP id : (gesture . ) + EOF
  gesture              ::=  GESTURE id : pose + execution
  pose                 ::=  POSE id :
                            statement ( , statement ) *  .
  statement            ::=  transform | restriction
  execution            ::=  EXECUTION :
                            (repeat the following steps number
                            executionStep( , executionStep) *
                             | executionStep( , executionStep) * )
  executionStep        ::=  motionConstraint ?
                             id ( and holdConstraint) ?
Transforms
  transform            ::=  pointTo
                             | rotatePlane
                             | rotateDirection
  pointTo              ::=  point your ?
                            bodyPart(( , your ? bodyPart) *
                             and your ? bodyPart) ?
                            ( to | to your ) ? direction
  rotatePlane          ::=  rotate your
                            bodyPart(( , your ? bodyPart) *
                             and your ? bodyPart) ?
                             number degrees
                            angularDirection on the ?
                            referencePlane
  rotateDirection      ::=  rotate your bodyPart
                            (( , your ? bodyPart) *
                             and your ? bodyPart) ?
                             number degrees
                            ( to | to your ) ?
                            direction
Restrictions
  restriction          ::=  dont ? touchRestriction
                             | dont ? putRestriction
                             | dont ? alignRestriction
  touchRestriction     ::=  touch your ?
                            bodyPart with your ?
                            side hand
  putRestriction       ::=  put your ?
                            bodyPart(( , your ? bodyPart) *
                             and your ? bodyPart) ?
                            relativeDirection bodyPart
  alignRestriction     ::=  align your ?
                            bodyPart(( , your ? bodyPart) *
                             and your ? bodyPart) ?
Skeleton
  bodyPart             ::=  joint | side arm | side leg | spine
                             | back | arms | legs | shoulders
                             | wrists | elbows | hands
                             | hands tips | thumbs | hips
                             | knees | ankles | feet | you
  joint                ::=  centerJoint | side sidedJoint
  centerJoint          ::=  neck | head | spine m id |
                            spine base | spine shoulder
  side                 ::=  left | right
  sidedJoint           ::=  shoulder | elbow | wrist | hand |
                            hand tip | thumb | hip | knee |
                            ankle | foot
  direction            ::=  up | down | front | back | side
  angularDirection     ::=  clockwise | counter clockwise
  referencePlane       ::=  frontal plane | sagittal plane |
                            horizontal plane
  relativeDirection    ::=  in front of your | behind your |
                            (( on top of ) |
                            above ) your | below your |
                            to the side of your
  motionConstraint     ::=  slowly | rapidly
  holdConstraint       ::=  hold for number seconds
  repeat               ::=  repeat number times
```

Fig. 10: BNF for PREPOSE. The start symbol is *app*.

4 Techniques

Figure 10 shows a BNF for PREPOSE which we currently support. This captures how PREPOSE applications can be composed out of gestures, gestures composed out of poses and execution blocks, execution blocks can be composed

$$\textsc{Rotate-Frontal+}\quad \frac{\texttt{Rotate-Frontal}(j, a, \texttt{Clockwise})}{\begin{array}{c} j.Y = cos(a) \cdot j.Y + sin(a) \cdot j.Z \\ j.Z = -sin(a) \cdot j.Y + cos(a) \cdot j.Z \end{array}}$$

$$\textsc{Rotate-Frontal-}\quad \frac{\texttt{Rotate-Frontal}(j, a, \texttt{CounterClockwise})}{\begin{array}{c} j.Y = cos(a) \cdot j.Y - sin(a) \cdot j.Z \\ j.Z = sin(a) \cdot j.Y + cos(a) \cdot j.Z \end{array}}$$

$$\textsc{Rotate-Sagittal+}\quad \frac{\texttt{Rotate-Sagittal}(j, a, \texttt{Clockwise})}{\begin{array}{c} j.X = cos(a) \cdot j.X + sin(a) \cdot j.Y \\ j.Y = -sin(a) \cdot j.X + cos(a) \cdot j.Y \end{array}}$$

$$\textsc{Rotate-Sagittal-}\quad \frac{\texttt{Rotate-Sagittal}(j, a, \texttt{CounterClockwise})}{\begin{array}{c} j.X = cos(a) \cdot j.X - sin(a) \cdot j.Y \\ j.Y = sin(a) \cdot j.X + cos(a) \cdot j.Y \end{array}}$$

$$\textsc{Rotate-Horizontal+}\quad \frac{\texttt{Rotate-Horizontal}(j, a, \texttt{Clockwise})}{\begin{array}{c} j.X = cos(a) \cdot j.X + sin(a) \cdot j.Z \\ j.Z = -sin(a) \cdot j.X + cos(a) \cdot j.Z \end{array}}$$

$$\textsc{Rotate-Horizontal-}\quad \frac{\texttt{Rotate-Horizontal}(j, a, \texttt{CounterClockwise})}{\begin{array}{c} j.X = cos(a) \cdot j.X - sin(a) \cdot j.Z \\ j.Z = sin(a) \cdot j.X + cos(a) \cdot j.Z \end{array}}$$

Fig. 11: Transformations translated into Z3 terms. j is the joint position (with X, Y, and Z components); a is the rotational angle.

out of execution steps, etc[2].

The grammar is fairly extensible: if one wishes to support other kinds of transforms or restrictions, one needs to extend the PREPOSE grammar, regenerate the parser, and provide runtime support for the added transform or restriction. Note also that the PREPOSE grammar lends itself naturally to the creation of developer tools such as context-sensitive auto-complete in an IDE or text editor.

4.1 Prepose to SMT Formulas

PREPOSE compiles programs written in the PREPOSE language to formulae in Z3, a state-of-the-art SMT solver.

Translating basic transforms: Figure 11 captures the principles of translating PREPOSE transforms into Z3 terms; the figure shows the different variants of how *rotatePlane* from Figure 10 is translated by way of illustration. These are update rules that define the $\langle X, Y, Z \rangle$ coordinates of the joint j to which the transformation is applied. Note that *rotatePlane* transformations take the plane p and direction d as parameters. Depending on the type of rotation, namely, the rotation plane, one of these rules is picked. These coordinate updates generally require a trigonometric computation[3].

Translating basic restrictions: Figure 12 shows how PREPOSE restrictions are translated to Z3 constraints. Auxiliary functions *Angle* and *Distance* that are further compiled down into Z3 terms are used as part of compilation. Additionally, thresholds th_{angle} and $th_{distance}$

[2]For researchers who wish to extend PREPOSE, we have uploaded an a ANTLR version of the PREPOSE grammar to `http://binpaste.com/fdsdf`

[3]Because of the lack of support for these functions in Z3, we have implemented *sin* and *cos* applied to a using lookup tables for commonly used values.

$$\textsc{Align}\quad \frac{\texttt{Align}(j_1, j_2)}{\Gamma \vdash Angle(j_1, j_2) < th_{align}}$$

$$\textsc{LowerThan}\quad \frac{\texttt{LowerThan}(j)}{\Gamma \vdash j.Y < sin(th_{angle})}$$

$$\textsc{Put-Front}\quad \frac{\texttt{Put-Front}(j_1, j_2, \texttt{InFrontOfYour})}{\Gamma \vdash j_1.Z > j_2.Z + th_{distance}}$$

$$\textsc{Put-Behind}\quad \frac{\texttt{Put-Behind}(j_1, j_2, \texttt{BehindYour})}{\Gamma \vdash j_1.Z < j_2.Z - th_{distance}}$$

$$\textsc{Put-Right}\quad \frac{\texttt{Put-Right}(j_1, j_2, \texttt{ToTheRightOfYour})}{\Gamma \vdash j_1.X > j_2.X + th_{distance}}$$

$$\textsc{Put-Left}\quad \frac{\texttt{Put-Left}(j_1, j_2, \texttt{ToTheLeftOfYour})}{\Gamma \vdash j_1.X < j_2.X - th_{distance}}$$

$$\textsc{Put-Top}\quad \frac{\texttt{Put-Top}(j_1, j_2, \texttt{OnTopOfYour})}{\Gamma \vdash j_1.Y > j_2.Y + th_{distance}}$$

$$\textsc{Put-Below}\quad \frac{\texttt{Put-Below}(j_1, j_2, \texttt{BelowYour})}{\Gamma \vdash j_1.Y < j_2.Y - th_{distance}}$$

$$\textsc{Touch}\quad \frac{\texttt{Touch}(j_1, j_2)}{\Gamma \vdash Distance(j_1 < j_2) < th_{distance}}$$

$$\textsc{KeepAngle}\quad \frac{\texttt{KeepAngle}(j_1, j_2)}{\Gamma \vdash Angle(j_1 < j_2) < th_{angle}}$$

Fig. 12: Restrictions translated into Z3 terms. Note that $th_{distance}$ and th_{angle} are *static* thresholds: they define what it means to perform a specific pose. For instance, *touching* a surface does not mean literally touching it; being very close to it is sufficient. As in Figure 11, j is the joint position (with X, Y, and Z components).

```
APP simple:
    GESTURE generic-left-punch :
        POSE prepare-punch :
            put your left elbow behind your neck.

        POSE execute-left-punch :
            put your left elbow in front of your neck.

        EXECUTION :
            prepare-punch,
            rapidly execute-left-punch.
```

Fig. 13: Left punch gesture used in our example.

are static thresholds that are part of pose definition, as opposed to runtime thresholds used for matching.

Interacting with the solver: In order to illustrate interaction with the Z3 solver we use the concrete sample gesture shown in Figure 13 consisting of two poses as a running example. The sample illustrates the execution of a left punch. The analysis we focus on here is that of checking that executing the gesture does not violate the default safety restrictions.

The default safety restrictions are stated in terms of arithmetic constraints on joint coordinates of the *body* that is represented by a dictionary from joints to real-numbered (3D) coordinates (x, y, z), as well as a dictionary from joints to norms (or reference coordinates). There are a total of 25 joints, such as `ElbowLeft`, `KneeRight`, etc., corresponding

```
(let ((a!1 (ite (>= (- |Head X| 0.0) 0.0)
                (- |Head X| 0.0)
                (- 0.0 (- |Head X| 0.0))))
      (a!2 (ite (>= (- |Head Y| 1.0) 0.0)
                (- |Head Y| 1.0)
                (- 0.0 (- |Head Y| 1.0))))
      (a!3 (ite (>= (- |Head Z| 0.0) 0.0)
                (- |Head Z| 0.0)
                (- 0.0 (- |Head Z| 0.0))))
      (a!5 (ite (>= (- |Neck X| 0.0) 0.0)
                (- |Neck X| 0.0)
                (- 0.0 (- |Neck X| 0.0))))
      (a!6 (ite (>= (- |Neck Y| 1.0) 0.0)
                (- |Neck Y| 1.0)
                (- 0.0 (- |Neck Y| 1.0))))
      (a!7 (ite (>= (- |Neck Z| 0.0) 0.0)
                (- |Neck Z| 0.0)
                (- 0.0 (- |Neck Z| 0.0))))
      (a!9 (ite (>= (- | pineMid X| 0.0) 0.0)
                (- | pineMid X| 0.0)
                (- 0.0 (- | pineMid X| 0.0))))
      (a!10 (ite (>= (- | pineMid Y| 1.0) 0.0)
                 (- | pineMid Y| 1.0)
                 (- 0.0 (- | pineMid Y| 1.0))))
      (a!11 (ite (>= (- | pineMid Z| 0.0) 0.0)
                 (- | pineMid Z| 0.0)
                 (- 0.0 (- | pineMid Z| 0.0))))
      (a!13 (ite (>= (- | pine houlder X| 0.0) 0.0)
                 (- | pine houlder X| 0.0)
                 (- 0.0 (- | pine houlder X| 0.0))))
      (a!14 (ite (>= (- | pine houlder Y| 1.0) 0.0)
                 (- | pine houlder Y| 1.0)
                 (- 0.0 (- | pine houlder Y| 1.0))))
      (a!15 (ite (>= (- | pine houlder Z| 0.0) 0.0)
                 (- | pine houlder Z| 0.0)
                 (- 0.0 (- | pine houlder Z| 0.0))))
      (a!17 (ite (>= (- | pineMid X| | pine houlder X|) 0.0)
                 (- | pineMid X| | pine houlder X|)
                 (- 0.0 (- | pineMid X| | pine houlder X|))))
      (a!18 (ite (>= (- | pineMid Y| | pine houlder Y|) 0.0)
                 (- | pineMid Y| | pine houlder Y|)
                 (- 0.0 (- | pineMid Y| | pine houlder Y|))))
      (a!19 (ite (>= (- | pineMid Z| | pine houlder Z|) 0.0)
                 (- | pineMid Z| | pine houlder Z|)
                 (- 0.0 (- | pineMid Z| | pine houlder Z|))))
      (a!21 (- (+ (- 0.0 | houlderLeft X|) | houlderRight X|)
               (+ (- 0.0 |HipLeft X|) |HipRight X|)))
      (a!22 (- (+ (- 0.0 | houlderLeft Y|) | houlderRight Y|)
               (+ (- 0.0 |HipLeft Y|) |HipRight Y|)))
      (a!24 (- (+ (- 0.0 | houlderLeft Z|) | houlderRight Z|)
               (+ (- 0.0 |HipLeft Z|) |HipRight Z|)))
      (a!27 (< (+ 0.0
                  (* |ElbowLeft Z| |ElbowLeft Norm|)
                  (* | houlderLeft Z| | houlderLeft Norm|)
                  (* | pine houlder Z| | pine houlder Norm|)
                  (* | pineMid Z| | pineMid Norm|)
                  (* |ElbowLeft Z| |ElbowLeft Norm|))
               (- (+ 0.0
                     (* |Neck Z| |Neck Norm|)
                     (* | pine houlder Z| | pine houlder Norm|)
                     (* | pineMid Z| | pineMid Norm|)
                     (* |Neck Z| |Neck Norm|))
                  (/ 1.0 10.0)))))
(let ((a!4 (ite (>= (ite (>= a!1 a!2) a!1 a!2) a!3)
                (ite (>= a!1 a!2) a!1 a!2)
                a!3))
      (a!8 (ite (>= (ite (>= a!5 a!6) a!5 a!6) a!7)
                (ite (>= a!5 a!6) a!5 a!6)
                a!7))
      (a!12 (ite (>= (ite (>= a!9 a!10) a!9 a!10) a!11)
                 (ite (>= a!9 a!10) a!9 a!10)
                 a!11))
      (a!16 (ite (>= (ite (>= a!13 a!14) a!13 a!14) a!15)
                 (ite (>= a!13 a!14) a!13 a!14)
                 a!15))
      (a!20 (ite (>= (ite (>= a!17 a!18) a!17 a!18) a!19)
                 (ite (>= a!17 a!18) a!17 a!18)
                 a!19))
      (a!23 (ite (>= (ite (>= a!21 0.0) a!21 (- 0.0 a!21))
                     (ite (>= a!22 0.0) a!22 (- 0.0 a!22)))
                 (ite (>= a!21 0.0) a!21 (- 0.0 a!21))
                 (ite (>= a!22 0.0) a!22 (- 0.0 a!22)))))
(let ((a!25 (ite (>= a!23 (ite (>= a!24 0.0) a!24 (- 0.0 a!24)))
                 a!23
                 (ite (>= a!24 0.0) a!24 (- 0.0 a!24)))))
(let ((a!26 (and true
                 (< a!4 (/ 153.0 200.0))
                 (< a!8 (/ 153.0 200.0))
                 (< a!12 (/ 153.0 200.0))
                 (< a!16 (/ 153.0 200.0))
                 (< a!20 (/ 153.0 200.0))
                 (not (< a!20 (/ 433.0 250.0)))
                 (not (and (< |ElbowLeft Z| 0.0) (> |ElbowLeft X| 0.0)))
                 (not (and (< |ElbowLeft Z| 0.0) (> |ElbowLeft Y| 0.0)))
                 (not (and (< |ElbowRight Z| 0.0) (< |ElbowRight X| 0.0)))
                 (not (and (< |ElbowRight Z| 0.0) (> |ElbowRight Y| 0.0)))
                 (< a!25 (/ 153.0 200.0))
                 (< |KneeLeft Y| 0.0)
                 (< |KneeRight Y| 0.0))))
  (and a!26 true a!27 a!26)))))
```

Fig. 14: Z3 terms for the internal validity constraints for **`prepare-punch`**.

to the different joint types in the Kinect API in `Microsoft.Kinect.dll`. The interested reader is referred to `https://github.com/Microsoft/prepose/Z3Experiments/Z3Experiments/Gestures/Analysis/Safety.cs`, method `DefaultSafetyRestriction`, for full details. Real numbers are modeled by rational numbers in this setting. The clear benefit of this is that satisfiability checking of the linear arithmetic constraints that arise as a result of the analysis is decidable.

First, each pose is checked for internal validity. This means that the current body constraint (that is a quantifier-free predicate over the joint constraints) is transformed according to the pose and the resulting body predicate (that is also a quantifier-free constraint over the joint constraints since the transformation does not introduce quantifiers) is evaluated for satisfiability in conjunction with a default *safety* condition. The default safety condition includes checks such as: neck and hips are not inclinated beyond a given threshold, hips are aligned with the shoulders or at lest within a safe range, elbows are not behind the back and not on the top/back sub-space, the inclination of wrists towards the back is not higher than the inclination of the elbows unless elbows are up or wrists are directed to torso, etc. If the transformed body constraint is unsatisfiable, this means that there exists no instance of the coordinates that would correspond to a *concrete* safe body position, and so the pose is deemed internally invalid.

In this gesture the first pose is in fact internally invalid, as the analysis correctly discovers that putting your left elbow behind your neck is not feasible for a typical human being. The Z3 terms for this gesture are shown in Figure 14; as the reader can see, there is a great deal of expansion that happens when translating PREPOSE gestures to Z3 terms.

If each pose is internally valid the poses are composed sequentially. Such sequential composition corresponds to constructing a predicate that describes all the possible body positions from the given initial predicate. Again, the resulting predicate has only positive occurrences of existential quantifiers, i.e., it is essentially quantifier-free, because a positive occurrence of an existential quantifier corresponds to an uninterpreted constant. Note however, that *negating* such a constraint would, in general, no longer be quantifier-free, if the quantifiers are treated as existential. The current analysis does not require operations that would introduce satisfiability checking of formulas involving universal quantifiers.

Runtime Execution: After a PREPOSE script is translated to Z3 constraints, we use the Z3 solver to match a user's movements to the gesture. The trusted core of PREPOSE registers with the Kinect skeleton tracker to receive updated skeleton positions of the user.

For each new position, the runtime uses the Z3 term evaluation mechanism to automatically apply gestures to the previous user's position to obtain the target (in a sense, ideal) position for each potential gesture. This target position is in turn compared to the current user's joints' position to see if there is a match and to notify the application.

Note that this is an approximate comparison where the level of precision can be specified by the application (see, for instance, Figure 15 with a slider for specifying the accuracy of the match). Note that this is a very lightweight use of the theorem prover, as we only evaluate terms without doing satisfiability checking. One could also have a custom runtime matching mechanism instead. Upon receiving a notification, the application may then give feedback to the user, such as encouragement, badges for completing a gesture, or movement to a more difficult gesture.

4.2 Security and Reliability

By design, PREPOSE is amenable to sound static reasoning by translating queries into Z3 formulae. Below we show how to convert key security and reliability properties into Z3 queries. The underlying theory we use is that of *reals*. We also use non-recursive data types (tuples) within Z3.

Please remember that these are static analyses that typically take place *before* gestures are deployed to the end-user — there is no runtime checking overhead. The properties below are also briefly summarized in Figure 2.

Unlike approximate runtime matching described above, static analysis is about *precise*, ideal matching. We do not have a theory of approximate equality that is supported by the theorem prover. We treat gestures such as $G : B \to B$, in other words, as functions that transform bodies in set B to new bodies.

Basic gesture safety: The goal of these restrictions is to make sure we "don't break any bones" by allowing the user to follow this gesture. We define a collection of safety restrictions pertaining to the head, spine, shoulders, elbows, hips, and legs. We denote by R_S the *compiled restriction*, the set of all states that are allowed under our safety restrictions. The compiled restriction R_S is used to test whether for a given gesture G

$$\exists b \in B : \neg R_S(G(b))$$

in other words, does there exist a body which fails to satisfy the conditions of R_S after applying G. R_S restricts the relative positions of the head, spine, shoulders, elbows, hips, and legs. The restriction for the head is shown below to give the reader a sense of what is involved:

```
var head = new SimpleBodyRestriction(body => {
    Z3Point3D up = new Z3Point3D(0, 1, 0);

    return Z3.Context.MkAnd(
        body.Joints[JointType.Head]
            .IsAngleBetweenLessThan(up, 45),
        body.Joints[JointType.Neck]
            .IsAngleBetweenLessThan(up, 45));
});
```

Inner validity: We also want to ensure that our gesture are not inherently contradictory, in other words, is it the case that all sequences of body positions will fail to match the gesture. An example of a gesture that has an inner contradiction, consider

```
    put your arms up;
    put your arms down;
```

Obviously *both* of these requirements cannot be satisfied at once. In the Z3 translation, this will give rise to a contradiction: $\texttt{joint}[\text{"}\texttt{rightelbow}\text{"}].\texttt{Y} = 1 \wedge \texttt{joint}[\text{"}\texttt{rightelbow}\text{"}].\texttt{Y} = -1$. To find possible contradictions in gesture definitions, we use the following query:

$$\neg\exists b \in B : G(b).$$

Protected gestures: Several immersive sensor-based systems include so-called "system attention positions" that users invoke to get privileged access to the system. These are the AR equivalent of Ctrl-Alt-Delete on a Windows system. For example, the Kinect on Xbox has a Kinect Guide gesture that brings up the home screen no matter which game is currently being played. The Kinect "Return to Home" gesture is easily encoded in PREPOSE and the reader can see this gesture here: `http://bit.ly/1JlXk79`. For Google Glass, a similar utterance is "Okay Glass." On Google Now on a Motorola X phone, the utterance is "Okay Google."

We want to make sure that PREPOSE gesture do not attempt to redefine system attention positions.

$$\exists b \in B, s \in S : G(b) = s.$$

where $S \subset B$ is the set of pre-defined system attention positions.

Conflict detection: Conflict detection, in contrast, involves two possibly interacting gestures G_1 and G_2.

$$\exists b \in B : G_1(b) = G_2(b).$$

Optionally, one could also attempt to test whether *compositions* of gestures can yield the same outcome. For example, is it possible that $G_1 \circ G_2 = G_3 \circ G_4$. This can also be operated as a query on sequences of bodies in B.

5 Experimental Evaluation

We built a visual gesture development and debugging environment, which we call PREPOSE Explorer. Figure 15 shows a screen shot of our tool. On the left, a text entry box allows a developer to write PREPOSE code with proper syntax highlighting. On the right, the tool shows the user's current position in green and the target position in white. On the bottom, the tool gives feedback about the current pose being matched and how close the user's position is to the target.

5.1 Dimensions of Evaluation

Given that PREPOSE provides guarantees about security and privacy by construction, we focused on making sure that we are able to program a wide range of applications that involve gestures, as summarized in Figure 16 and also partially shown in the Appendix. Beyond that we want to ensure that the PREPOSE-based gesture matching scales well to support interactive games, etc. To summarize

- We used this tool to measure the *expressiveness* of PREPOSE by creating 28 gestures in three different domains.

- We then ran some benchmarks to measure runtime performance and static analysis performance of Prepose. First, we report runtime performance, including the amount of time required to match a pose and the time to synthesize a new target position. Then, we discuss the results of benchmarks for static analysis.

Prior work has used surveys to evaluate whether the information revealed by various abstractions is acceptable to a sample population of users in terms of its privacy. Here, we are giving the application the least amount of information required to do its jobs, so these surveys are not necessary.

5.2 Expressiveness

Because the Prepose language is not Turing-complete, it has limitations on the gestures it can express. To determine if our choices in building the language are sufficient to handle useful gestures, we built gestures using the Prepose Explorer. We picked three distinct areas: therapy, tai-chi, and ballet, which together cover a wide range of gestures. Figure 16 shows the breakdown of how many gestures we created in each area, for 28 in total. These are complex gestures: the reviewers are encouraged to examine the code linked to from Figure 16.

For example, Figure 17 shows some of the poses from tai-chi captured by Prepose code. We chose tai-chi because it is already present in Kinect for Xbox games such as Your Shape: Fitness Evolved. In addition, tai-chi poses require complicated alignment and non-alignment between different body parts.

5.3 Pose Matching Performance

We used the Kinect Studio tool that ships with the Kinect for Windows SDK to record depth and video traces of one of the authors. We recorded a trace of performing two representative gestures. Each trace was about 20 seconds in length and consisted of about 20,000 frames, occupying about 750 MB on disk. We picked these to be two representative tai-chi gestures.

Our measurements were performed on an HP Z820 Pentium Xion E52640 Sandy bridge with 6 cores and 32 GB of memory running Windows 8.1.

For each trace, we measured the *matching time*: the time required to evaluate whether the current user position matches the current target position. When a match occurred, we also measured the *pose transition time*: the time required to synthesize a new target pose, if applicable.

Our results are encouraging. On the first frame, we observed matching times between 78 ms and 155 ms, but for all subsequent frames matching time dropped substantially. For these frames, the median matching time was 4 ms. with a standard deviation of 1.08 ms. This is fast enough for real time tracking at 60 FPS (frames per second).

For pose transition time, we observed a median time of 89 ms, with a standard deviation of 36.5 ms. While this leads to a "skipped" frame each time we needed to create a new pose, this is still fast enough to avoid interrupting the user's movements.

While we have made a design decision to use a theorem prover for runtime matching, one can replace that machinery with a custom runtime matcher that is likely to run even faster. When deploying Prepose-based applications on a less powerful platform such as the Xbox, this design change may be justified.

5.4 Static Analysis Performance

Safety checking: Figure 18 shows a near-linear dependency between the number of steps in a gesture and time to check against safety restrictions. Exploring the results further, we performed a linear regression to see the influence of other parameters such as the number of negative restrictions. The R^2 value of the fit is about 0.9550, and the coefficients are shown in the table to the right. The median checking time is only 2 ms. We see that safety checking is practical and, given how fast it is, could easily be integrated into an IDE to give developers quick feedback about invalid gestures.

Intercept	-4.44
NumTransforms	0.73
NumRestrictions	-2.42
NumNegatedRestrictions	-6.23
NumSteps	29.48

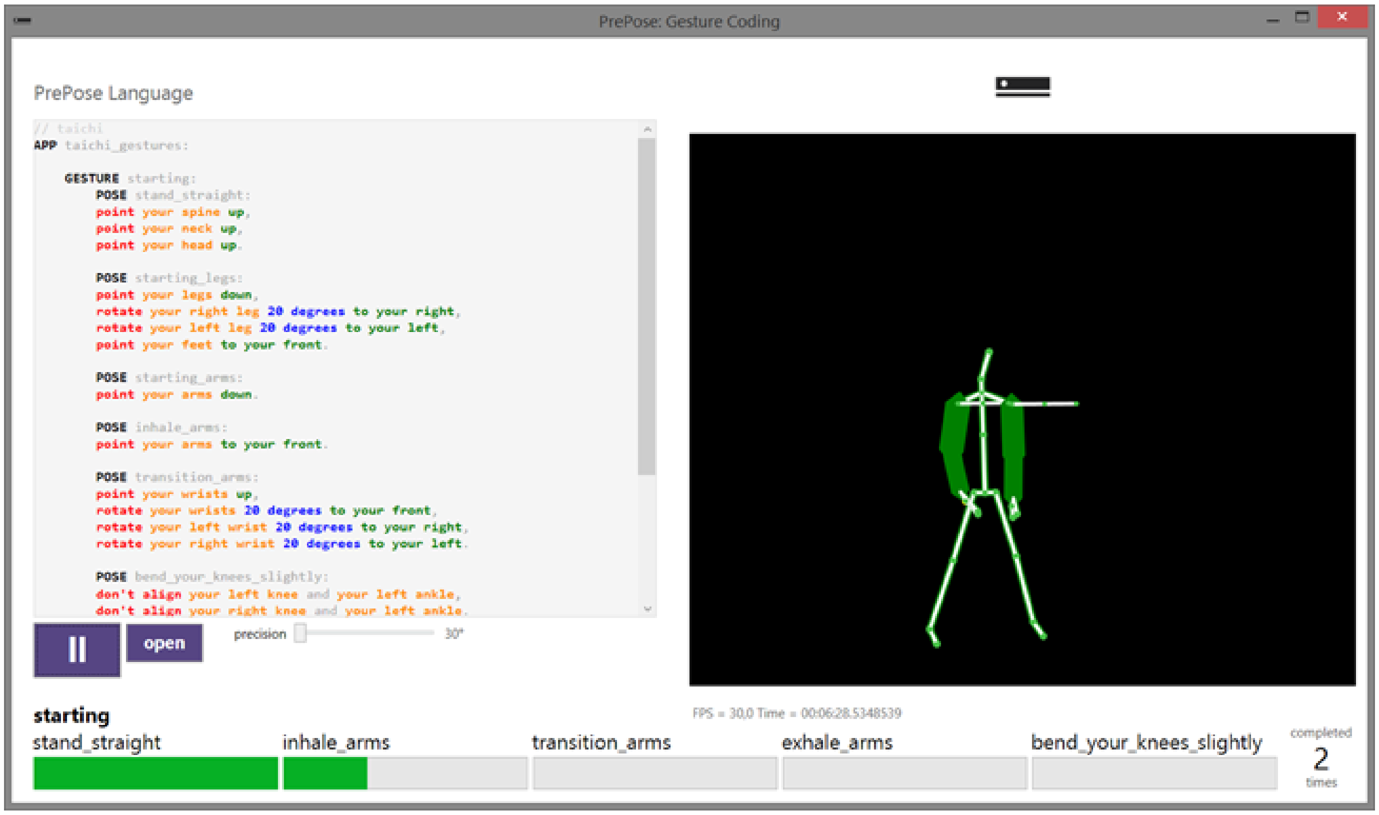
Validity checking: Figure 19 shows another near-linear dependency between the number of steps in a gesture and the time to check if the gesture is internally valid. The average checking time is 188.63 ms. We see that checking for internal validity of gestures is practical and, given how fast it is, could easily be integrated into an IDE to give developers quick feedback about invalid gestures.

Conflict checking: We performed pairwise conflict checking between 111 pairs of gestures from our domains. Figure 20 shows the CDF of conflict checking times, with the x axis in log scale. For 90% of the cases, the checking time is below 0.170 seconds, while 97% of the cases took less than 5 seconds and 99% less than 15 seconds. Only one query out of the 111 took longer than 15 seconds. As a result, with a timeout of 15 seconds, only one query would need attention from a human auditor.

6 Limitations

This work is the first step in defining a programmable way to limit the potential for privacy leaks in gesture-based programming. We are not claiming that we have solved all the potential privacy issues. In fact, we believe strongly that the attack model will evolve as this space rapidly changes.

A major challenge is to define a precise and easy to reason about attack model in this space. Our key contribution lies in going beyond the model that gives application direct access to hardware and providing an abstraction layer above that. It is exceedingly difficult to argue that that abstraction layer cannot be abused by a clever attacker. By way of analogy, consider an operating system



```
// taichi
APP taichi_gestures:

    GESTURE starting:
        POSE stand_straight:
        point your spine up,
        point your neck up,
        point your head up.

        POSE starting_legs:
        point your legs down,
        rotate your right leg 20 degrees to your right,
        rotate your left leg 20 degrees to your left,
        point your feet to your front.

        POSE starting_arms:
        point your arms down.

        POSE inhale_arms:
        point your arms to your front.

        POSE transition_arms:
        point your wrists up,
        rotate your wrists 20 degrees to your front,
        rotate your left wrist 20 degrees to your right,
        rotate your right wrist 20 degrees to your left.

        POSE bend_your_knees_slightly:
        don't align your left knee and your left ankle,
        don't align your right knee and your left ankle.
```

starting

stand_straight	inhale_arms	transition_arms	exhale_arms	bend_your_knees_slightly
🟩	🟩			

Fig. 15: Screenshot of PrePose Explorer in action.

Application	Gestures	Poses	LOC	URL
Therapy	12	28	225	`http://pastebin.com/ARndNHdu`
Ballet	11	16	156	`http://pastebin.com/c9nz6NP8`
Tai-chi	5	32	314	`http://pastebin.com/VwTcTYrW`

Fig. 16: We have encoded 28 gestures in PrePose, across three different applications. The table shows the number of total poses and lines of PrePose code for each application. Each pose may be used in more than one gesture. The Appendix has one of the PrePose applications, Ballet, listed as well.

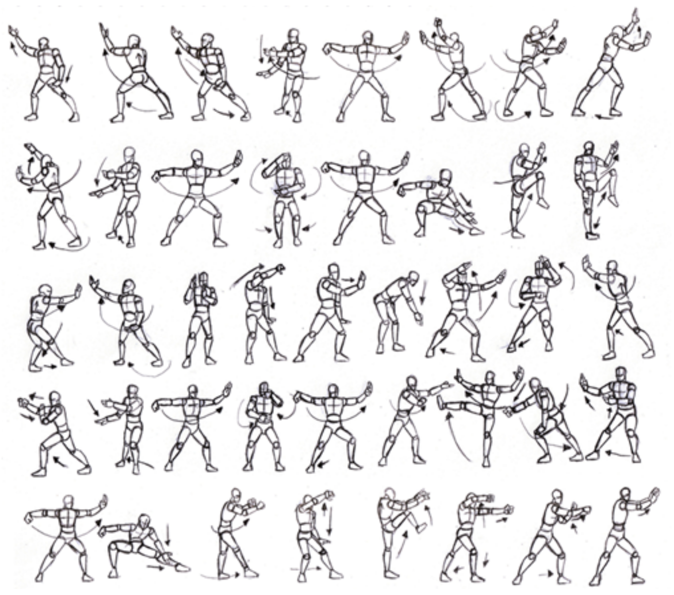


Fig. 17: The tai-chi gestures we have encoded using PrePose (`http : //pastebin.com/VwTcTYrW`) all come from this illustration.

mechanism that allows applications to register keystrokes (or key chords) such as `Ctrl` + `Alt` + `P`. While this makes it considerably more difficult to develop a keylogger, it is difficult to claim that one cannot determine whether the user is left-handed or possibly to fingerprint different users based on the frequency of their shortcut use. Similarly, in the context of PrePose, a clever attacker may define a "network" of really fine-grained gestures to collect statistics about the user.

A key advantage of PrePose is that when new attacks are discovered, they can be encoded as satisfiability queries, which gives one way to tackle these attacks as well. We see the following areas as extensions of our current work:

- We do not explicitly reason about the notion of time; there could be a pose that is safe for brief periods of time but is less safe when held for, say, a minute.
- Our current approach reasons about conflicts at the level of entire gestures. This does not preclude conflicts at the intermediate, sub-gesture level. A possible way to alleviate this situation is to automatically compile the current set of gesture into intermediate, *atomic* gestures, which could be validated for lack of

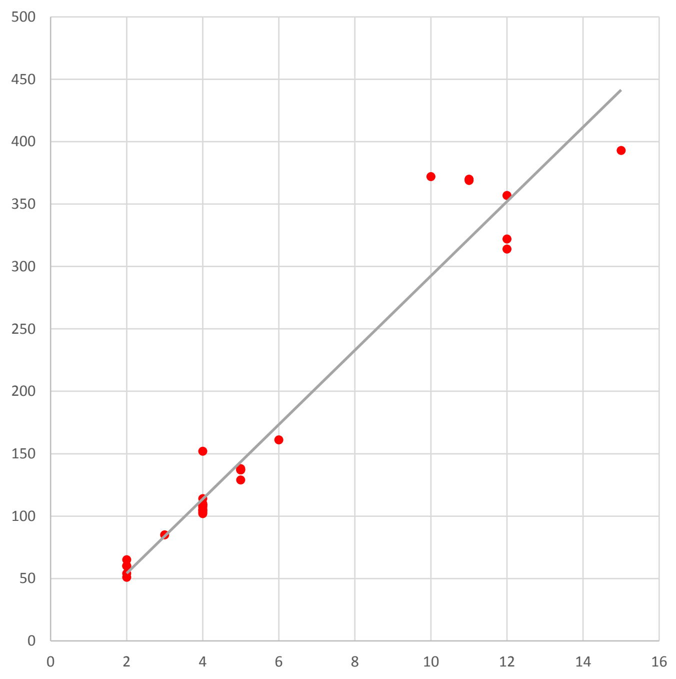


Fig. 18: Time to check for safety, in ms, as a function of the number of steps in the underlying gesture.

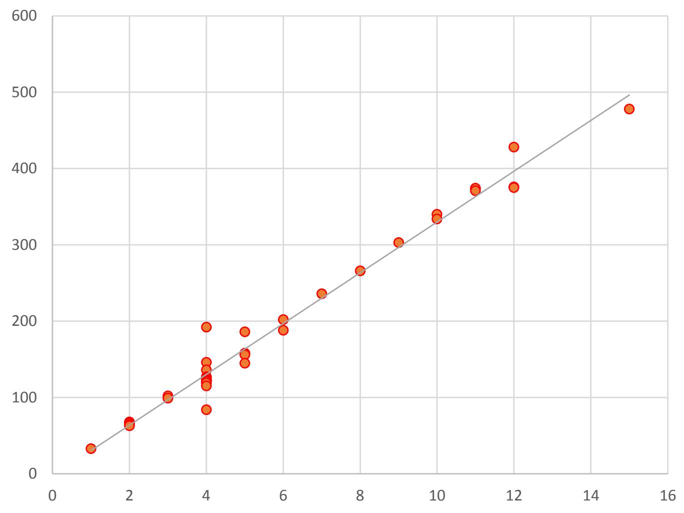


Fig. 19: Time to check internal validity, in ms, as a function on the number of steps in the underlying gesture.

conflicts.

- PREPOSE requires the developer to manually write gestures. A natural next step is to automatically synthesize gestures *by demonstration*.

7 Related Work

Below we first describe some gesture-building approaches, mostly from the HCI community, and then we talk about privacy in sensing-based applications.

7.1 Gesture Building Tools

Below, we list some of the key projects that focus on gesture creation. PREPOSE's approach is unique in that

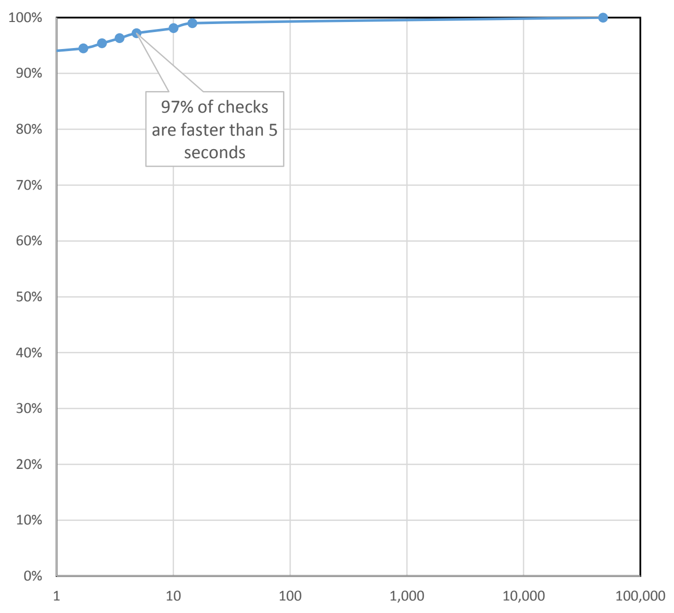


Fig. 20: Time to check for conflicts for a pair of gestures presented as a CDF. The x axis is seconds plotted on a log scale.

it focuses on capturing gestures using English-like commands. This allows gesture definitions to be modified more easily. PREPOSE differs from the tools below in that it focuses on security and privacy at the level of system design.

CrowdLearner [1] provides a crowd-sourcing way to collect data from mobile devices usage in order to create recognizers for tasks specified by the developers. This way the sampling time during the application development is shorter and the collected data should represent a better coverage of real use scenarios in relation to the usual in-lab sampling procedures. Moreover, it abstracts for developers the classifier construction and population, requiring no specific recognition expertise.

Gesture Script [23] provides a unistroke touch gesture recognizer which combines training from samples with explicit description of the gesture structure. By using the tool, the developer is enabled to divide the input gestures in core parts, being able to train them separately and specify by a script language how the core parts are performed by the user. This way, it requires less samples for compound gestures because the combinations of the core parts are performed by the classifier. The division in core parts also eases the recovery of attributes (e.g. number of repetitions, line length, etc.) which can be specified by the developer during the creation of the gestures.

Proton [19] and Proton++ [18] present a tool directed to multitouch gestures description and recognition. The gestures are modeled as regular expressions and their alphabet consists of the main actions (Down, Move and Up), and related attributes e.g.: direction of the move action; place or object in which the action was taken; counter which represents a relative ID; among others. It is shown that by describing gestures with regular expressions and a concise alphabet it is possible to easily identify ambiguity between two gestures previously to the test phase.

CoGesT [10] presents a scheme to represent hand and arms gestures. It uses a grammar which generates the possible descriptions, the descriptions are based on common textual descriptions and related to the coordinate system generated by the body aligned planes (sagittal, frontal and horizontal). The transcription is mainly related to relative positions and trajectories between them, relying on the form and not on functional classification of the gesture. Moreover it does not specify the detailed position but more broad relations between body parts. This way the specified gestures are not strongly precise. On the other hand, it enables users to produce equivalent gestures by interpreting the description and using their knowledge about gesture production.

BAP [7] approaches the task of coding body movements with focus on the study of emotion expression. Actors trained the system by performing specific emotion representations and these recorded frames were coded into pose descriptions. The coding was divided into anatomic (explicating which part of the body was relevant in the gesture) and form (describing how the body parts were moving). The movement direction was described adopting the orthogonal body axis (sagittal, vertical and transverse). Examples of coding: Left arm action to the right; Up-down head shake; Right hand at waist; etc.

Annotation of Human Gesture [28] proposes an approach for transcribing gestural movements by overlaying a 3D body skeleton on the recorded actors' gestures. This way, once the skeleton data is aligned with the recorded data, the annotation can be created automatically.

RATA [29] presents a tool to create recognizers for touch and stylus gestures. The focus is on the ease and rapidity of the gesture recognition developing task. The authors claim that within 20 minutes (and by adding only two lines of code) developers and interaction designers can add new gestures to their application.

EventHurdle [17] presents a tool for explorative prototyping of gesture use on the application. The tool is proposed as an abstraction of the gathered sensor data, which can be visualized as a 2D graphic input. The designer also can specify the gesture in a provided graphical interface. The main concept is that unistroke touch gestures can be described as a sequence of trespassed hurdles.

GestureCoder [24] presents a tool for multi-touch gesture creation from performed examples. The recognition is performed by creating a state machine for the performed gestures with different names. The change of states is activated by some pre-coded actions: finger landing; lifting; moving; and timeout. The ambiguity of recorded gestures is solved by analyzing the motion between the gestures using a decision tree.

GestureLab [5] presents a tool for building domain-specific gesture recognizers. It focuses on pen unistroke gestures by considering trajectory but also additional attributes such as timing and pressure.

MAGIC [2] and MAGIC 2.0 [21] are tools to help developers, which are not experts in pattern recognition, to create gesture interfaces. Focuses on motion gesture (using data gathered from motion sensors, targeted to mobile scenario). MAGIC 2.0 focuses on false-positive prediction for these types of gestures. MAGIC comes with an "Everyday Gesture Library" (EGL), which contains videos of people performing gestures. MAGIC uses the EGL to perform *dynamic* testing for gesture conflicts, which is complementary to our language-based *static* approach.

Zhao et al. [35] proposes a rule-based gesture recognizer for the physiotherapy domain. It exposes rules as an XML considering joints positions and the corresponding bones orientations. The 'hip abduction' gesture is demonstrated in 48 lines of XML code, which allows later edition and refinement of the gesture.

GDL [11] presents a DSL for gesture description. Rules are written using the language, and are defined by cause (specific body conditions) and effect (resulting Boolean value). The rules can be either logical or numerical, allowing the developer to set thresholds for specific conditions. Rules can be connected by using the effect of previous rules as input. It is also allowed to use rules that relate between previous and current poses, and in addition to specify a required waiting time for poses transition in order to consider the rule as valid.

Leech and Kostek [22] present a rule-based recognizer and a corresponding DSL for dynamic hand gestures recognition. Once dynamic gestures require motion, the used inputs are the direction and velocity of the performed hand movements. Rules are composed of boolean expressions using operators like "AND" and "NOT", while dealing with dynamic gestures, in order to specify that some constraints should be considered on the following moment the "THEN" command is applied.

Bedregal et al. [3] also propose a rule-based recognizer and DSL for hand gestures, using as input joints and separations between fingers tracked by a data glove. The DSL describes each pose of the hand on time. Terms for finger joints are STRAIGHT, CURVED or BENT referring to the angles for each joint, and the terms for finger separations are CROSSED, CLOSED, SEMI-OPEN and OPEN also referring to lateral angles. Despite these angle descriptions being represented at a high level discrete class, by using Fuzzy logic each joint or separation can partially belong to more than one class.

Hoste and Signer [13] analyze several gesture programming languages including Proton and GDL and propose 30 criteria to classify these solutions as well as enhance the discussion about their limitations and future possibilities. The criteria include topics like readability, reliability, customization and scalability in terms of performance.

7.2 Sensing and Privacy

The majority of work below focuses on privacy concerns in sensing applications. In PREPOSE, we add some *security* concerns into the mix, as well.

SURROUNDWEB [34] presents an immersive browser which tackles privacy issues by reducing the required privileges. The concept is based on a context sensing technology which can render different web contents on different parts of the room. In order to prevent the web pages to access the raw video stream of the room, SurroundWeb is proposed as a rendering platform through the Room Skeleton abstraction (which consists on a list of possible room "screens"). Moreover the SurroundWeb introduces a Detection Sandbox as a mediator between web pages and object detection code (never telling the web pages if objects were detected or not) and natural user inputs (mapping the inputs into mouse events to the web page).

Darkly [16] proposes a privacy protection system to prevent access of raw video data from sensors to untrusted applications. The protection is performed by controlling mechanisms over the acquired data. In some cases the privacy enforcement (transformations on the input frames) may reduce application functionality.

OS Support for AR Apps [8] and AR Apps with Recognizers [15] discusses the access the AR applications usually have to raw sensors and proposes OS extension to control the sent data by performing the recognizer tasks itself. This way the recognizer module is responsible to gather the sensed data and to process it locally, giving only the least needed privileges to AR applications.

MockDroid [4] proposes an OS modification for smart phones in which applications always ask the user to access the needed resources. This way users are aware of which information are being sent to the application whenever they run it, and then can decide between the trade-off of giving access or using the application functionality.

AppFence [12] proposes a tool for privacy control on mobile devices, which can block or shadow sent data to applications in order to keep the application up and running, but prevent exfiltration of on-device data. What You See is What You Get [14] proposes a widget which alerts users of which sensor is being requested by which application.

Recent work on world-driven access control restricts sensor input to applications in response to the environment, e.g. it can be used to disable access to the camera when in a bathroom [30]. Mazurek *et al.* surveyed 33 users to determine how they think about controlling access to data provided by a variety of devices, and discovered that many user's mental models of access control are incorrect [25]. Vaniea *et al.* performed an experiment to determine how users notice and fix access-control permission errors depending on where the access-control policy is spatially located on a web site [33].

8 Future Work

While this paper assumes that the developer will author the gesture code, we envision numerous possibilities related to automatically inferring Prepose programs *by demonstration* [6, 24, 27]. This approach has been used in several other areas of programming, in interacting with users who are not necessarily technologically sophisticated. In our context, we can readily foresee useful training scenarios such as the two below:

- a personal trainer at a gym demonstrating a personalized workout program, which gets notated as Prepose gestures and given to the gym goes to use at home for exercises during the week;
- a doctor working with patients with limited mobility who works on adaptation of user interfaces [31]. The doctor can demonstrate a gesture that corresponds to a mouse double-click and have that recorded by Prepose, etc.

In both of these cases, a intermediary specialist is working with a Prepose-equipped Kinect sensor, whose goal is to learn Prepose gestures for later use by end-users.

9 Conclusions

This paper introduces the Prepose language and runtime. Prepose allows developers to write high-level gesture descriptions that have semantics in terms of SMT formulas. Our architecture protects the privacy of the user by preventing untrusted applications from directly accessing raw sensor data; instead, applications register Prepose code with a trusted runtime. Sound static analysis helps eliminate possible security and reliability issues.

To test the expressiveness of Prepose, we have created 28 gestures in Prepose across three important and representative immersive programming domains. We also showed that Prepose programs can be statically analyzed quickly to check for safety, pairwise conflicts, and conflicts with system gestures.

Runtime matching in Prepose as well as static conflict checking, both of which reduce to Z3 queries, are sufficiently fast (milliseconds to several seconds) to be deployed. By writing gesture recognizers in a DSL deliberately designed from the ground up to support privacy, security, and reliability, we obtain strong guarantees without sacrificing either performance or expressiveness. Our Z3-based approach has more than acceptable performance in practice. Pose matching in Prepose averages 4 ms. Synthesizing target pose time ranges between 78 and 108 ms. Safety checking is under 0.5 seconds per gesture. The average validity checking time is only 188.63 ms. Lastly, for 97% of the cases, the conflict detection time is below 5 seconds, with only one query taking longer than 15 seconds.

Acknowledgments

We would like to express our gratitude to our collaborators working on the Kinect platform at Microsoft. We also thank the anonymous reviewers, whose input has made this paper a great deal stronger.

References

[1] S. Amini and Y. Li. Crowdlearner: rapidly creating mobile recognizers using crowdsourcing. In *Proceedings of the Symposium on User Interface Software and Technology*, 2013.

[2] D. Ashbrook and T. Starner. Magic: a motion gesture design tool. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 2159–2168. ACM, 2010.

[3] B. C. Bedregal, A. C. Costa, and G. P. Dimuro. Fuzzy rule-based hand gesture recognition. In *Artificial Intelligence in Theory and Practice*, pages 285–294. Springer, 2006.

[4] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan. MockDroid: trading privacy for application functionality on smartphones. In *Proceedings of the Workshop on Mobile Computing Systems and Applications*, 2011.

[5] A. Bickerstaffe, A. Lane, B. Meyer, and K. Marriott. Developing domain-specific gesture recognizers for smart diagram environments. In *Graphics Recognition. Recent Advances and New Opportunities*. Springer, 2008.

[6] A. Billard, S. Calinon, R. Dillmann, and S. Schaal. Robot programming by demonstration. In *Springer handbook of robotics*, pages 1371–1394. Springer, 2008.

[7] N. Dael, M. Mortillaro, and K. R. Scherer. The body action and posture coding system (bap): Development and reliability. *Journal of Nonverbal Behavior*, 36(2), 2012.

[8] L. D'Antoni, A. Dunn, S. Jana, T. Kohno, B. Livshits, D. Molnar, A. Moshchuk, E. Ofek, F. Roesner, S. Saponas, et al. Operating system support for augmented reality applications. *Hot Topics in Operating Systems (HotOS)*, 2013.

[9] S. Fothergill, H. Mentis, P. Kohli, and S. Nowozin. Instructing people for training gestural interactive systems. In *Proceedings of the Conference on Human Factors in Computing Systems*, 2012.

[10] D. Gibbon, R. Thies, and J.-T. Milde. CoGesT: a formal transcription system for conversational gesture. In *In Proceedings of LREC 2004*, 2004.

[11] T. Hachaj and M. R. Ogiela. Rule-based approach to recognizing human body poses and gestures in real time. *Multimedia Systems*, 20(1):81–99, 2014.

[12] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proceedings of the Conference on Computer and Communications Security*, 2011.

[13] L. Hoste and B. Signer. Criteria, challenges and opportunities for gesture programming languages. *Proc. of EGMI*, pages 22–29, 2014.

[14] J. Howell and S. Schechter. What you see is what they get: Protecting users from unwanted use of microphones, camera, and other sensors. In *In Proceedings of Web 2.0 Security and Privacy Workshop*. Citeseer, 2010.

[15] S. Jana, D. Molnar, A. Moshchuk, A. Dunn, B. Livshits, H. J. Wang, and E. Ofek. Enabling fine-grained permissions for augmented reality applications with recognizers. In *Proceedings of the USENIX Security Symposium*, 2013.

[16] S. Jana, A. Narayanan, and V. Shmatikov. A Scanner Darkly: Protecting user privacy from perceptual applications. In *IEEE Symposium on Security and Privacy*, 2013.

[17] J.-W. Kim and T.-J. Nam. EventHurdle: supporting designers' exploratory interaction prototyping with gesture-based sensors. In *Proceedings of the Conference on Human Factors in Computing Systems*, 2013.

[18] K. Kin, B. Hartmann, T. DeRose, and M. Agrawala. Proton++: A customizable declarative multitouch framework. In *Proceedings of the Symposium on User Interface Software and Technology*, 2012.

[19] K. Kin, B. Hartmann, T. DeRose, and M. Agrawala. Proton: Multitouch gestures as regular expressions. In *Proceedings of the Conference on Human Factors in Computing Systems*, 2012.

[20] Kinect for Windows Team at Microsoft. Visual gesture builder: A data-driven solution to gesture detection, 2014. `https://onedrive.live.com/view.aspx?resid=1A0C78068E0550B5!77743&app=WordPdf`.

[21] D. Kohlsdorf, T. Starner, and D. Ashbrook. MAGIC 2.0: A web tool for false positive prediction and prevention for gesture recognition systems. In *Automatic Face & Gesture Recognition and Workshops*, 2011.

[22] M. Lech and B. Kostek. Hand gesture recognition supported by fuzzy rules and kalman filters. *International Journal of Intelligent Information and Database Systems*, 6(5):407–420, 2012.

[23] H. Lü, J. Fogarty, and Y. Li. Gesture script: Recognizing gestures and their structure using rendering scripts and interactively trained parts. 2014.

[24] H. Lü and Y. Li. Gesture coder: a tool for programming multi-touch gestures by demonstration. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI)*, 2012.

[25] M. L. Mazurek, J. P. Arsenault, J. Bresee, N. Gupta, I. Ion, C. Johns, D. Lee, Y. Liang, J. Olsen, B. Salmon, R. Shay, K. Vaniea, L. Bauer, L. F. Cranor, G. R. Ganger, and M. K. Reiter. Access control for home data sharing: Attitudes, needs and practices. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2010.

[26] L. D. Moura and N. Bjorner. Z3: An Efficient SMT Solver. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2008.

[27] C. G. Nevill-Manning. Programming by demonstration. *New Zealand Journal of Computing*, 4(2):15–24, 1993.

[28] Q. Nguyen and M. Kipp. Annotation of Human Gesture using 3D Skeleton Controls. In *LREC*. Citeseer, 2010.

[29] B. Plimmer, R. Blagojevic, S. H.-H. Chang, P. Schmieder, and J. S. Zhen. Rata: codeless generation of gesture recognizers. In *Proceedings of the 26th Annual BCS Interaction Specialist Group Conference on People and Computers*. British Computer Society, 2012.

[30] F. Roesner, D. Molnar, A. Moshchuk, T. Kohno, and H. J. Wang. World-driven access control. In *ACM Conference on Computer and Communications Security*, 2014.

[31] E. A. Suma, D. M. Krum, B. Lange, S. Koenig, A. Rizzo, and M. Bolas. Adapting user interfaces for gestural interaction with the flexible action and articulated skeleton toolkit. *Computers and Graphics*, pages 193–201, 2012.

[32] M. Tsikkos and J. Glading. Writing a gesture service with the Kinect for Windows SDK, 2011. `http://blogs.msdn.com/b/mcsuksoldev/archive/2011/08/08/ writing-a-gesture-service-with-the kinect-for-windows-sdk.aspx`.

[33] K. Vaniea, L. Bauer, L. F. Cranor, and M. K. Reiter. Out of sight, out of mind: Effects of displaying access-control information near the item it controls. In *IEEE Conference on Privacy, Security and Trust (PST)*, 2012.

[34] J. Vilk, D. Molnar, E. Ofek, C. Rossbach, B. Livshits, A. Moshchuk, H. J. Wang, and R. Gal. SurroundWeb: Mitigating Privacy Concerns in a 3D Web Browser . In *Proceedings of the Symposium on Security and Privacy*, 2015.

[35] W. Zhao, R. Lun, D. D. Espy, and M. Reinthal. Rule based realtime motion assessment for rehabilitation exercises. In *Computational Intelligence in Healthcare and e-health (CICARE), 2014 IEEE Symposium on*, pages 133–140. IEEE, 2014.

```
///////////////////////////////////////////////////////
// Gestures described based on conditioning programs
// providade by American Acadmey of Orthipaedic Surgeons
// http://orthoinfo.aaos.org/topic.cfm?topic=A00672
///////////////////////////////////////////////////////

APP therapy:

    // for this gesture the user needs to have
    // a support table placing the other hand
    // over the table to help the balance
    GESTURE pendulum:
      POSE stand-straight:
        point your spine, neck and head up.

      POSE lean-forward:
        rotate your spine, neck and head
            30 degrees to your front,
        do not align your legs,
        align your back.

      POSE swing-forward:
        point your left arm down,
        rotate your left arm
            30 degrees to your front.

      POSE swing-back:
        point your left arm down,
        rotate your left arm
            30 degrees to your back.

      POSE swing-left:
        point your left arm down,
        rotate your left arm
            30 degrees to your left.

      POSE swing-right:
        point your left arm down,
        rotate your left arm
            30 degrees to your right.

      EXECUTION:
        stand-straight,
        lean-forward,
        slowly swing-forward,
        slowly swing-back,
        slowly swing-left,
        slowly swing-right,
        slowly swing-forward,
        slowly swing-left,
        slowly swing-back,
        slowly swing-right.

    GESTURE crossover-left-arm-stretch:
      POSE relax-arms:
        point your left arm down,
        point your right arm down.

      POSE stretch:
        rotate your left arm
            90 degrees
            counter clockwise
            on the frontal plane,
        touch your left elbow
            with your right hand.

      EXECUTION:
        relax-arms,
        slowly stretch and hold for 30 seconds.

    GESTURE crossover-right-arm-stretch:
      POSE stretch-reverse:
        rotate your right arm
            90 degrees counter
            clockwise
            on the frontal plane,
        touch your
            right elbow with your left hand.

      EXECUTION:
        relax-arms,
        slowly stretch-reverse and hold for 30 seconds.

    //GESTURE  = MIRROR crossover-left-arm-stretch;

    GESTURE passive-external-rotation:
```

```
  POSE left-wrist-to-the-left:
    point your elbows down,
    point your wrists to your front,
    rotate your left wrist
        45 degrees counter
        clockwise
        on the horizontal plane.

  POSE right-wrist-to-the-right:
    point your elbows down,
    point your wrists to your front,
    rotate your right wrist
        45 degrees
        counter clockwise
        on the horizontal plane.

  EXECUTION:
    left-wrist-to-the-left and hold for 30 seconds,
    right-wrist-to-the-right and hold for 30 seconds,
    left-wrist-to-the-left and hold for 30 seconds,
    right-wrist-to-the-right and hold for 30 seconds.

GESTURE elbows-flexion:
  POSE flex-left-elbow:
    rotate your left wrist
        180 degrees
        clockwise
        on the sagittal plane,
    point your left elbow down.

    POSE flex-right-elbow :
    rotate your
        right wrist
        180 degrees
        clockwise
        on the sagittal plane,
    point your right elbow down.

  EXECUTION:
    relax-arms,
    slowly flex-left-elbow and
        hold for 2 seconds,
    relax-arms,
    slowly flex-right-elbow and
        hold for 2 seconds.

GESTURE elbows-extension:
  POSE stretch-arm-up:
    point your left elbow up,
    point your right arm up,
    touch your right elbow
        with your left hand,
    align your spine.

  POSE flex-arm-back:
    point your elbows up,
    point your right wrist back,
    touch your right elbow
        with your left hand,
    align your spine.

  EXECUTION:
    stand-straight,
    flex-arm-back,
    slowly stretch-arm-up
        and hold for 2 seconds,
    slowly flex-arm-back.

GESTURE head-rolls:
  POSE head-to-left:
    point your head up,
    rotate your head
        20 degrees clockwise
        on the frontal plane.

  POSE head-to-right:
    point your head up,
    rotate your head 20 degrees
        counter clockwise
        on the frontal plane.

  POSE head-to-front:
    point your head up,
    rotate your head 20 degrees
        clockwise on the sagittal plane.
  POSE head-to-back :
    point your head up,
    rotate your head 20 degrees
        counter clockwise
        on the sagittal plane.

  EXECUTION :
    head-to-front,
    head-to-right and hold for 5 seconds,
    head-to-front,
    slowly head-to-left and hold for 5 seconds,
    slowly head-to-back,
    slowly head-to-right,
    slowly head-to-front,
    slowly head-to-left,
    slowly head-to-back,
    slowly head-to-right,
    slowly head-to-front,
    slowly head-to-left.

GESTURE standing-quadriceps-stretch:
  POSE lift-ankle:
    point your legs down,
    rotate your right ankle
        90 degrees to your back,
    align your spine.

  POSE catch-ankle:
    touch your right ankle with your right hand,
    align your spine.

  POSE bend-leg:
    touch your right ankle
        with your right hand,
    rotate your right ankle
        15 degrees to up,
    align your spine.

  EXECUTION:
    stand-straight,
    lift-ankle,
    catch-ankle,
    slowly bend-leg and
        hold for 30 seconds.

GESTURE half-squats:
  POSE up-squat:
    point your legs down,
    point your arms to your front,
    point your spine up.

  POSE down-squat:
    point your arms to your front,
    rotate your knees
        20 degrees to up,
    point your spine up.

  EXECUTION:
    up-squat,
    down-squat and
        hold for 5 seconds,
    up-squat,
    down-squat and
        hold for 5 seconds.

GESTURE shoulder-abduction:
  POSE arm-down:
    point your right arm down,
    rotate your right arm
        20 degrees to your right.

  // poses as pure transformations
  // from previous state
  POSE rot-up:
    rotate your right arm
        20 degrees up.

  POSE rot-down:
    rotate your right arm
        20 degrees down.

  EXECUTION:
    arm-down,
    //using repetitions to get further
    rot-up,
    rot-up,
    rot-up,
    rot-up,
    rot-up,
    rot-down,
    rot-down,
    rot-down,
    rot-down,
    rot-down.

  GESTURE shoulder-adduction:
    POSE arm-down-add:
      point your right arm down,
      rotate your right arm
          30 degrees to your front.

    EXECUTION:
      arm-down,
      //using repetitions to get further
      rot-up,
      rot-up,
      rot-up,
      rot-up,
      rot-up,
      rot-down,
      rot-down,
      rot-down,
      rot-down,
      rot-down.


//////////////////////////////////////////////////////
// Initial Ballet gestures of the Cecchetti Method
// Gestures described based on the book
// Technical Manual and Dictionary of Classical Ballet
// By Gail Grant
// From Dover Publications
// This particular set can be found
// in the following picture:
// http://mysylph.files.wordpress.com/2013/05/
// cecchetti-port-de-bra.jpg
/////////////////////////////////////////////////////

APP ballet:

  GESTURE first-position:
    POSE stand-straight:
      point your spine, neck and head up.

    POSE point-feet-out:
      point your right foot right,
      point your left foot left.

    POSE stretch-legs:
      align your left leg,
      align your right leg.

    POSE low-arc-arms:
      point your arms down,
      rotate your elbows 15 degrees up,
      rotate your left wrist
        5 degrees to your right,
      rotate your right wrist
        5 degrees to your left.

    EXECUTION:
      stand-straight,
      point-feet-out,
      stretch-legs,
      low-arc-arms.

  GESTURE second-position:
    POSE mid-arc-arms:
      point your arms down,
      rotate your elbows 30 degrees up,
      rotate your wrists 20 degrees up.

    POSE high-arc-arms:
      point your arms down,
      rotate your arms 70 degrees up.

    POSE open-legs-frontal-plane:
      point your legs down,
      rotate your right leg
        10 degrees to right,
      rotate your left leg
        10 degrees to left.
  EXECUTION:
    stand-straight,
    point-feet-out,
    stretch-legs,
    open-legs-frontal-plane,
    mid-arc-arms,
    high-arc-arms.

GESTURE third-position:

  POSE mid-arc-arms-to-right:
    point your arms down,
    rotate your right elbow
      30 degrees up,
    rotate your right wrist
      20 degrees up,
    rotate your left elbow
      10 degrees to your left,
    rotate your left wrist
      10 degrees to your right.

  EXECUTION:
    stand-straight,
    point-feet-out,
    stretch-legs,
    mid-arc-arms-to-right.

GESTURE fourth-position-en-avant:
  POSE cross-legs-one-behind-the-other:
    put your left ankle
      behind your right ankle,
    put your left ankle
      to the right of your right ankle.

  POSE high-arc-arms-to-right:
    point your arms down,
    rotate your right arm 70 degrees up,
    rotate your left elbow 20 degrees to your left,
    rotate your left wrist 25 degrees to your right.

  EXECUTION:
    stand-straight,
    point-feet-out,
    stretch-legs,
    cross-legs-one-behind-the-other,
    high-arc-arms-to-right.

GESTURE fourth-position-en-haunt:
  POSE high-arc-arms-to-right-and-up:
    point your right arm down,
    rotate your right arm 70 degrees up,
    point your left arm up,
    rotate your left elbow 15 degrees to your left,
    rotate your left wrist 5 degrees to your right.

  EXECUTION:
    stand-straight,
    point-feet-out,
    stretch-legs,
    cross-legs-one-behind-the-other,
    high-arc-arms-to-right-and-up.

GESTURE fifth-position-en-avant:
  POSE inner-arc-arms:
    point your arms down,
    rotate your right elbow 20 degrees to your right,
    rotate your right wrist 25 degrees to your left,
    rotate your left elbow 20 degrees to your left,
    rotate your left wrist 25 degrees to your right.

  EXECUTION:
    stand-straight,
    point-feet-out,
    stretch-legs,
    inner-arc-arms.

GESTURE fifth-position-en-haunt:
  POSE arc-arms-up:
    point your arms up,
    rotate your right elbow 15 degrees to your right,
    rotate your right wrist 5 degrees to your left,
    rotate your left elbow 15 degrees to your left,
    rotate your left wrist 5 degrees to your right.

  EXECUTION:
    stand-straight,
    point-feet-out,
    stretch-legs,
    arc-arms-up.

  GESTURE a-la-quatrieme-devant:
    POSE quatrieme-devant-legs:
      put your right leg in front of your left leg,
      point your left leg down,
      point your left foot left.

    EXECUTION:
      stand-straight,
      point-feet-out,
      quatrieme-devant-legs,
      high-arc-arms.

  GESTURE a-la-quatrieme-derriere:
    POSE quatrieme-derriere-legs:
      put your right leg behind your left leg,
      point your left leg down,
      point your left foot left.

    EXECUTION:
      stand-straight,
      point-feet-out,
      quatrieme-derriere-legs,
      high-arc-arms.

  GESTURE a-la-seconde:
    POSE seconde-legs:
      point your legs down,
      point your left foot left,
      rotate your right leg 20 degrees to your right.

    EXECUTION:
      stand-straight,
      point-feet-out,
      seconde-legs,
      high-arc-arms.

/////////////////////////////////////////////////////////
// Initial taichi gestures from the 24 form exercises
// Gestures described based on the book:
// Tai Chi for beginners and the 24 forms
// By Dr Paul Lam and Nancy Lake
// From limelightpress
////////////////////////////////////////////////////////

// taichi
APP taichi-gestures:

  GESTURE starting:
    POSE stand-straight:
      point your spine up,
      point your neck up,
      point your head up.

    POSE starting-legs:
      point your legs down,
      rotate your right leg 20 degrees to your right,
      rotate your left leg 20 degrees to your left,
      point your feet to your front.

    POSE starting-arms:
      point your arms down.

    POSE inhale-arms:
      point your arms to your front.

    POSE transition-arms:
      point your wrists up,
      rotate your wrists 20 degrees to your front,
      rotate your left wrist 20 degrees to your right,
      rotate your right wrist 20 degrees to your left.
      //put your left wrist near to your left shoulder,
      //put your right wrist near to your right shoulder.

    POSE bend-your-knees-slightly:
      do not align your left knee and your left ankle,
      do not align your right knee and your left ankle.

    POSE exhale-arms:
      point your elbows down,
      put your wrists below your elbows.

    EXECUTION:
```

```
    stand-straight,
    slowly bend-your-knees-slightly,
    slowly inhale-arms,
    slowly transition-arms,
    slowly exhale-arms.

GESTURE parting-the-wild-horses-mane:
  POSE on-right-foot-facing-front:
    put your right hand above your left hand,
    do not touch your right hand with your left hand,
    point your right leg down,
    //put your weight on the right foot,
    do not align your left leg.

  POSE on-right-foot-facing-left:
    do not align your right leg,
    align your left leg,
    point your left foot to your left.

  POSE left-arm-arc-up:
    point your left arm to your front,
    rotate your left arm 30 degrees to your left,
    rotate your left elbow 30 degrees down,
    rotate your left wrist 30 degrees up,
    do not align your left arm.

  POSE right-arm-face-down:
    point your right arm down,
    rotate your right wrist 15 degrees to your front,
    rotate your right wrist 15 degrees to your right,
    do not align your right arm.

  POSE left-bow-stance-legs:
    point your legs down,
    rotate your left knee 20 degrees to your front,
    rotate your left knee 20 degrees to your left,
    rotate your right leg 20 degrees to your right,
    do not align your left leg,
    do not align your right leg.

  POSE shift-weight-back:
    point your legs down,
    rotate your left leg 20 degrees up,
    rotate your left leg 20 degrees to your left,
    align your left leg,
    do not align your right leg.

  POSE right-arm-low-arc:
    point your right arm to your front,
    rotate your right elbow 30 degrees down.

  POSE final-horse-mane:
    //put your feet together,
    put your left hand above your right hand,
    do not touch your left hand with your right hand,
    do not align your legs.


  EXECUTION:
    stand-straight,
    //put-feet-under-shoulder,
    slowly inhale-arms,
    slowly exhale-arms,
    slowly bend-your-knees-slightly,
    on-right-foot-facing-front,
    slowly on-right-foot-facing-left,
    left-arm-arc-up,
    right-arm-face-down,
    left-bow-stance-legs,
    right-arm-low-arc,
    shift-weight-back,
    final-horse-mane.

GESTURE white-crane-spread-wings:
  POSE white-crane-start:
    put your left hand above your right hand,
    do not touch your left hand with your right hand,
    do not align your legs,
    put your right foot behind your left foot.

  POSE white-crane-mid:
    put your left hand to the right
        of your right hand,
    put your left hand above your right hand,
    put your left hand behind your right hand,
    do not align your legs,
    put your right foot behind your left foot.

  POSE white-crane-final:
    point your left arm down,
    rotate your left wrist 15 degrees to your front,
    rotate your left wrist 15 degrees to your right,
    do not align your left arm,
    point your right arm to your right,
    rotate your right elbow 20 degrees to your front,
    rotate your right elbow 15 degrees down,
    rotate your right wrist 90 degrees up,
    do not align your legs,
    put your right foot behind your left foot.

  EXECUTION:
    slowly white-crane-start,
    slowly white-crane-mid,
    slowly white-crane-final.

GESTURE brush-knee:
  POSE brush-knee-1:
    do not align your right leg,
    do not align your left leg,
    point your right arm to your front,
    rotate your right elbow 30 degrees down,
    point your left arm to your front,
    rotate your left arm 20 degrees to your left,
    rotate your left elbow 30 degrees down,
    rotate your left wrist 30 degrees up.

  POSE brush-knee-2:
    do not align your right leg,
    do not align your left leg,
    point your right arm to your front,
    rotate your right arm 20 degrees to your right,
    rotate your right elbow 20 degrees down,
    rotate your right wrist 20 degrees up,
    touch your right elbow with your left hand.

  POSE brush-knee-3:
    point your left leg down,
    rotate your left leg 15 degrees to your left,
    rotate your left leg 15 degrees up,
    align your left leg,
    do not align your right leg,
    point your right arm to your front,
    rotate your right arm 30 degrees to your right,
    rotate your right wrist 90 degrees up,
    point your left arm down,
    rotate your left arm 30 degrees up,
    rotate your left arm 15 degrees to your left,
    rotate your left wrist 5 degrees to your right,
    do not align your left arm.

  POSE brush-knee-4:
    align your right leg,
    do not align your left leg,
    point your left leg down,
    rotate your left knee 20 degrees to your front,
    rotate your left knee 20 degrees to your left,
    touch your left hip with your left hand,
    do not align your left arm,
    do not align your right arm,
    point your right arm to your front,
    rotate your right elbow 20 degrees down.

  POSE brush-knee-5:
    align your left leg,
    do not align your right leg,
    point your left arm to your front,
    rotate your left arm 20 degrees to your left,
    rotate your left elbow 20 degrees down,
    rotate your left wrist 20 degrees up,
    point your right arm to your front,
    rotate your right arm 20 degrees to your right,
    rotate your right elbow 40 degrees down,
    rotate your right wrist 40 degrees up.

  POSE brush-knee-6:
    do not align your right leg,
    do not align your left leg,
    point your left arm to your front,
    rotate your left arm 20 degrees to your left,
    rotate your left elbow 20 degrees down,
    rotate your left wrist 20 degrees up,
    touch your left elbow with your right hand.
  POSE brush-knee-7:
    point your right arm down,
    rotate your right arm 30 degrees up,
    align your right arm,
    point your left arm to your front,
    rotate your left arm 30 degrees to your left,
    rotate your left wrist 30 degrees up,
    put your right foot in front of your left foot,
    align your right leg,
    do not align your left leg.

  POSE brush-knee-8:
    point your right arm down,
    rotate your right wrist 5 degrees up,
    do not align your right arm,
    point your left arm to your front,
    rotate your left elbow 30 degrees down,
    put your left foot behind your right foot,
    align your left leg,
    do not align your right leg.

  POSE brush-knee-9:
    align your right leg,
    do not align your left leg,
    point your right arm to your front,
    rotate your right arm 60 degrees down,
    rotate your right arm 20 degrees to your right,
    rotate your right wrist 5 degrees up,
    do not align your right arm,
    point your left arm to your front,
    rotate your left elbow 30 degrees down,
    rotate your left wrist 75 degrees up.

  POSE brush-knee-10:
    do not align your right leg,
    do not align your left leg,
    point your right arm to your front,
    rotate your right arm 20 degrees to your right,
    rotate your right elbow 20 degrees down,
    rotate your right wrist 20 degrees up,
    touch your right elbow with your left hand.

  POSE brush-knee-11:
    point your left leg down,
    rotate your left leg 30 degrees to your left,
    rotate your left leg 30 degrees to your front,
    align your left leg,
    do not align your right leg,
    point your left arm down,
    rotate your left arm 30 degrees to your front,
    rotate your left wrist 5 degrees up,
    do not align your left arm,
    point your right arm to your right,
    rotate your right arm 20 degrees to your front,
    rotate your right wrist 90 degrees up.

  POSE brush-knee-12:
    align your right leg,
    do not align your left leg,
    point your left leg down,
    rotate your left knee 20 degrees to your front,
    rotate your left knee 20 degrees to your left,
    touch your left hip with your left hand,
    do not align your left arm,
    do not align your right arm,
    point your right arm to your front,
    rotate your right elbow 20 degrees down.

  EXECUTION:
    brush-knee-1,
    brush-knee-2,
    brush-knee-3,
    brush-knee-4,
    brush-knee-5,
    brush-knee-6,
    brush-knee-7,
    brush-knee-8,
    brush-knee-9,
    brush-knee-10,
    brush-knee-11,
    brush-knee-12.

GESTURE brush-knee-manual-interpolation-11:
  POSE brush-knee-1163:
    do not align your right leg,
    do not align your left leg,
    point your right arm to your front,
    rotate your right elbow 30 degrees down,
    point your left arm to your front,
    rotate your left arm 20 degrees to your left,
    rotate your left elbow 30 degrees down,
    rotate your left wrist 30 degrees up.
  EXECUTION:
    brush-knee-1,
    brush-knee-2,
    brush-knee-3,
    brush-knee-4,
    brush-knee-5,
    brush-knee-6,
    brush-knee-7,
    brush-knee-8,
    brush-knee-9,
    brush-knee-10,
    brush-knee-11.

GESTURE brush-knee-manual-interpolation-10:
  POSE brush-knee-111:
    do not align your right leg,
    do not align your left leg,
    point your right arm to your front,
    rotate your right elbow 30 degrees down,
    point your left arm to your front,
    rotate your left arm 20 degrees to your left,
    rotate your left elbow 30 degrees down,
    rotate your left wrist 30 degrees up.

  EXECUTION:
    brush-knee-1,
    brush-knee-2,
    brush-knee-3,
    brush-knee-4,
    brush-knee-5,
    brush-knee-6,
    brush-knee-7,
    brush-knee-8,
    brush-knee-9,
    brush-knee-10.

GESTURE brush-knee-manual-interpolation-9:
  POSE brush-knee-1111:
    do not align your right leg,
    do not align your left leg,
    point your right arm to your front,
    rotate your right elbow 30 degrees down,
    point your left arm to your front,
    rotate your left arm 20 degrees to your left,
    rotate your left elbow 30 degrees down,
    rotate your left wrist 30 degrees up.

  EXECUTION:
    brush-knee-1,
    brush-knee-2,
    brush-knee-3,
    brush-knee-4,
    brush-knee-5,
    brush-knee-6,
    brush-knee-7,
    brush-knee-8,
    brush-knee-9.

GESTURE brush-knee-manual-interpolation-8:
  POSE brush-knee-1245:
    do not align your right leg,
    do not align your left leg,
    point your right arm to your front,
    rotate your right elbow 30 degrees down,
    point your left arm to your front,
    rotate your left arm 20 degrees to your left,
    rotate your left elbow 30 degrees down,
    rotate your left wrist 30 degrees up.

  EXECUTION:
    brush-knee-1,
    brush-knee-2,
    brush-knee-3,
    brush-knee-4,
    brush-knee-5,
    brush-knee-6,
    brush-knee-7,
    brush-knee-8.
```

```
GESTURE brush-knee-manual-interpolation-7:
  POSE brush-knee-1211:
    do not align your right leg,
    do not align your left leg,
    point your right arm to your front,
    rotate your right elbow 30 degrees down,
    point your left arm to your front,
    rotate your left arm 20 degrees to your left,
    rotate your left elbow 30 degrees down,
    rotate your left wrist 30 degrees up.

  EXECUTION:
    brush-knee-1,
    brush-knee-2,
    brush-knee-3,
    brush-knee-4,
    brush-knee-5,
    brush-knee-6,
    brush-knee-7.

GESTURE brush-knee-manual-interpolation-6:
  POSE brush-knee-1231875:
    do not align your right leg,
    do not align your left leg,
    point your right arm to your front,
    rotate your right elbow 30 degrees down,
    point your left arm to your front,
    rotate your left arm 20 degrees to your left,
    rotate your left elbow 30 degrees down,
    rotate your left wrist 30 degrees up.

  EXECUTION:
    brush-knee-1,
    brush-knee-2,
    brush-knee-3,
    brush-knee-4,
    brush-knee-5,
    brush-knee-6.

GESTURE brush-knee-manual-interpolation-5:
  POSE brush-knee-1224:
    do not align your right leg,
    do not align your left leg,
    point your right arm to your front,
    rotate your right elbow 30 degrees down,
    point your left arm to your front,
    rotate your left arm 20 degrees to your left,
    rotate your left elbow 30 degrees down,
    rotate your left wrist 30 degrees up.
        EXECUTION:
    brush-knee-1,
    brush-knee-2,
    brush-knee-3,
    brush-knee-4,
    brush-knee-5.

GESTURE brush-knee-manual-interpolation-4:

  POSE brush-knee-12251:
    do not align your right leg,
    do not align your left leg,
    point your right arm to your front,
    rotate your right elbow 30 degrees down,
    point your left arm to your front,
    rotate your left arm 20 degrees to your left,
    rotate your left elbow 30 degrees down,
    rotate your left wrist 30 degrees up.
  EXECUTION:
    brush-knee-1,
    brush-knee-2,
    brush-knee-3,
    brush-knee-4.

GESTURE brush-knee-manual-interpolation-3:
  POSE brush-knee-1231:
    do not align your right leg,
    do not align your left leg,
    point your right arm to your front,
    rotate your right elbow 30 degrees down,
    point your left arm to your front,
    rotate your left arm 20 degrees to your left,
    rotate your left elbow 30 degrees down,
    rotate your left wrist 30 degrees up.

  EXECUTION:
    brush-knee-1,
    brush-knee-2,
    brush-knee-3.

GESTURE brush-knee-manual-interpolation-2:
  POSE brush-knee-12132:
    do not align your right leg,
    do not align your left leg,
    point your right arm to your front,
    rotate your right elbow 30 degrees down,
    point your left arm to your front,
    rotate your left arm 20 degrees to your left,
    rotate your left elbow 30 degrees down,
    rotate your left wrist 30 degrees up.
  EXECUTION:
    brush-knee-1,
    brush-knee-2.


GESTURE brush-knee-manual-interpolation-1:
  POSE brush-knee-123321:
    do not align your right leg,
    do not align your left leg,
    point your right arm to your front,
    rotate your right elbow 30 degrees down,
    point your left arm to your front,
    rotate your left arm 20 degrees to your left,
    rotate your left elbow 30 degrees down,
    rotate your left wrist 30 degrees up.

  EXECUTION:
    brush-knee-1.

GESTURE play-the-lute:
  POSE play-the-lute1:
    put your left foot in front of your right foot,
    do not align your left leg,
    do not align your right leg,
    point your right hand to your front,
    rotate your right elbow 15 degrees down,
    do not align your right arm,
    point your left arm down,
    rotate your left wrist 15 degrees to your front,
    do not align your left arm.

  POSE play-the-lute2:
    align your left leg,
    do not align your right leg,
    point your left arm to your front,
    rotate your left arm 20 degrees to your left,
    rotate your left elbow 20 degrees down,
    rotate your left wrist 20 degrees up,
    point your right arm to your front,
    rotate your right arm 20 degrees to your right,
    rotate your right elbow 40 degrees down,
    rotate your right wrist 40 degrees up.

  EXECUTION:
    play-the-lute1,
    play-the-lute2.
```