PREPOSE: Security and Privacy for Gesture-Based Programming

MSR-TR-2014-146

Lucas Silva Figueiredo
Federal University of Pernambuco

Benjamin Livshits, David Molnar, and Margus Veanes
Microsoft Research

***Abstract*—With the rise of sensors such as the Microsoft Kinect, Leap Motion, and hand motion sensors in phones such as the Samsung Galaxy S5, *natural user interface* (NUI) has become practical. NUI raises two key challenges for the developer: first, developers must create new code to *recognize* new gestures, which is a time consuming process. Second, to recognize these gestures, applications must have access to depth and video of the user, raising *privacy problems.* We address both problems with PREPOSE, a novel domain-specific language (DSL) for easily building gesture recognizers, combined with a system architecture that protects user privacy against untrusted applications by running PREPOSE code in a trusted core, and only interacting with applications via gesture events.**

PREPOSE lowers the cost of developing new gesture recognizers by exposing a range of primitives to developers that can capture many different gestures. Further, PREPOSE is designed to enable static analysis using SMT solvers, allowing the system to check security and privacy properties *before* running a gesture recognizer. We demonstrate that PREPOSE is expressive by creating novel gesture recognizers for 28 gestures in three representative domains: physical therapy, tai-chi, and ballet. We further show that matching user motions against PREPOSE gestures is efficient, by measuring on traces obtained from Microsoft Kinect runs.

Because of the privacy-sensitive nature of always-on Kinect sensors, we have designed the PREPOSE language to be analyzable: we enable security and privacy assurance through precise static analysis. In PREPOSE, we employ a sound *static analysis* that uses an SMT solver (Z3), something that works well on PREPOSE but would be hardly possible for a general-purpose language. We demonstrate that static analysis of PREPOSE code is efficient, and investigate how analysis time scales with the complexity of gestures. Our Z3-based approach 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.

I. 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. In many ways, this is only the beginning: other companies such as Leap Motion and Prime Sense are bringing low-cost depth and gesture sensing to consumer electronics. The newest generation of smartphones such as Samsung Galaxy S5 supports rudimentary gestures as well.

Gesture store: User demand for these sensors is driven by exiting 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`.

There are also 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.

An App Store 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 1 shows our approach. Developers write gesture recognizers in a high-level domain-specific language we call 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."

Privacy of always-on third-party applications: For an always-on immersive application, access to the entire video stream is an obvious source of privacy concerns. These sensors are in people's bedrooms, living rooms, and offices, which makes the need to guard against malicious

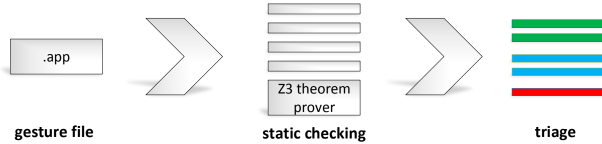


Fig. 1: Checking submissions to a gesture store. Submissions are marked as *safe* (green), *unsafe* (red), or *need human attention* (blue).

```
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. 2: Gesture example: `crossover-left-arm-stretch`.

or buggy third-party software critically important, without stifling developer productivity.

Gestures are an integral part of sensor-based always-on application[1]. As such, building new gestures is a fundamental part of software development. While, for instance, the Kinect SDK already includes a number of default gestures, developers typically need to add their own. Gesture development is a tricky process, which often depends on machine learning techniques requiring large volumes of training data [8]. These heavyweight methods are too expensive for many developers. Therefore, making gesture development easier would unlock the creativity of a larger class of developers.

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 2. PREPOSE lowers the cost of developing new gestures by exposing new primitives to developers that can express a wide range of natural gestures.

Monitoring applications: PREPOSE are particularly well-suited to what we call *monitoring applications*. 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. The chief purpose of these monitoring applications is to match against a set of pre-defined gestures, while running concurrently.

[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.)." [23]

Analyzable gestures: At the heart of PREPOSE is the idea of compiling gesture descriptions to formulas for an SMT solver; we use the Z3 solver [20]. These formulas 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 common gestures yet restrictive enough to ensure that key properties remain decidable. PREPOSE's is focus on the ability to statically and soundly analyze gesture-based programs:

1) PREPOSE validates that gestures have a basic measure of *safety*, i.e. they do not require the user to overextend herself physically in ways that may be dangerous;
2) PREPOSE ensures that gestures are internally valid, i.e. do not require the user to both keep her arms up *and* down;
3) PREPOSE tests whether a gesture conflicts with a reserved system-wide gesture such as the Kinect attention gesture;
4) PREPOSE finds potential conflicts within a set of gestures such as two gestures that would both be recognized from the same user movements.

PREPOSE has been designed with static analysis in mind, which allows checking of gestures when they are submitted to an app store, before being made available to users. Note that a an application that uses the Kinect SDK written in C++ or C# would generally require an extensive manual audit to ensure the lack of privacy leaks and security flaws.

Applications of 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 applications 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.

A. 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.
- **Static analysis.** We propose a set of static analysis algorithms designed to soundly find violations of important security and privacy properties. This analysis is designed to be run within an gesture App Store to prevent malicious third-party applications from affecting the end-user.

- **Expressiveness.** To show the expressiveness of PREPOSE, we demonstrate how to encode 28 gestures for 3 useful applications: *therapy*, *dance*, and *tai-chi*.
- **Performance evaluation.** Despite being written in a domain-specific language (DSL), PREPOSE-based gesture applications barely pay a price for the extra security and privacy guarantees.
- **Ingestion-time checking.** Through comprehensive experiments, we demonstrate that static analysis of broad range of gestures is well within the time limits imposed by App Store review processes. Our Z3-based approach has more than acceptable performance. Pose matching in PREPOSE averages 4 ms. Synthesizing target poses 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 90% of the cases, the conflict detection time is below 0.17 seconds.

B. Paper Organization

The rest of the paper is organized as follows. Section II provides some background on gesture authoring. Section III gives an overview of PREPOSE concepts and provides some motivating examples. Section IV describes our analysis for security and privacy in detail. Section V contains the details of our experimental evaluation. Sections VI and VII describe related work and conclude.

II. BACKGROUND

Today, developers of NUI 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 [7] shows the following code for a simple *punch* gesture:

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

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 3 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

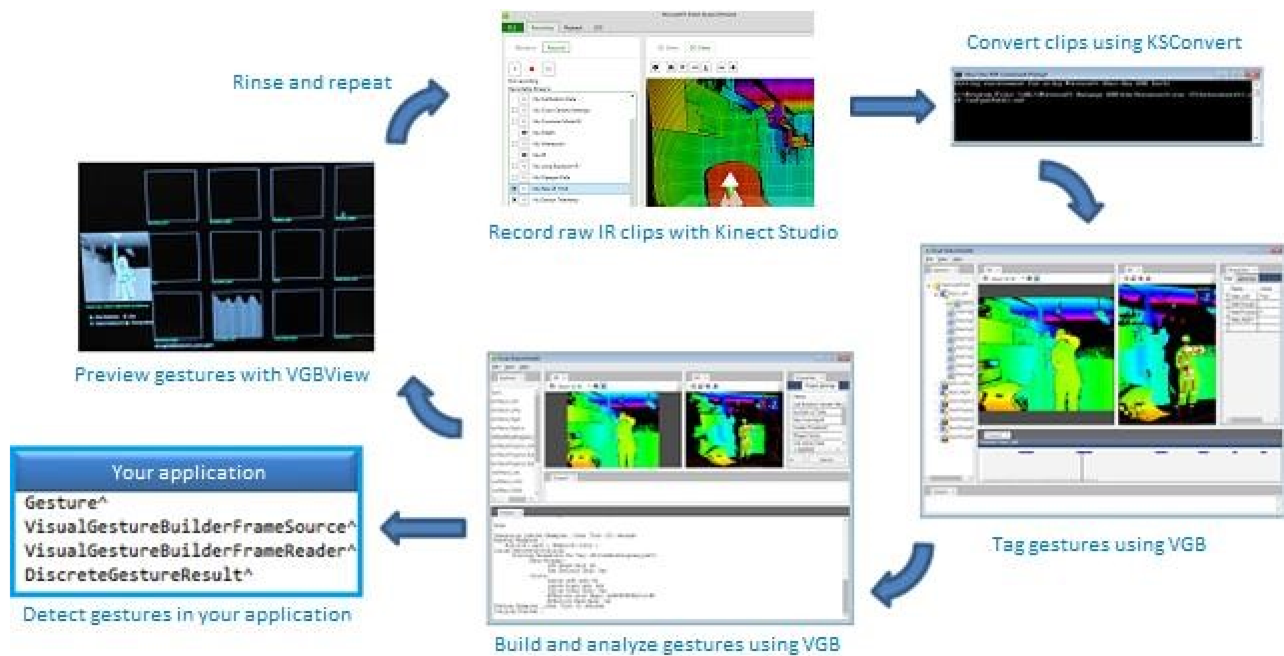


Fig. 3: Workflow for machine-learning based gesture recognition creation in the Kinect Visual Gesture Builder [7].

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.

III. Overview

We first show a motivating example in Section III-A. Next, we discuss the architecture of PREPOSE and how it provides security and privacy benefits (III-B). We then introduce basic concepts of the PREPOSE language and discuss its runtime execution (III-C). Finally, we discuss the security and privacy issues raised by an App Store for gestures, and show how static analysis can address them (III-D).

A. Motivating Example

Existing application on Kinect: Figure 4a shows a screen shot from the Reflexion Health physical therapy product. 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.

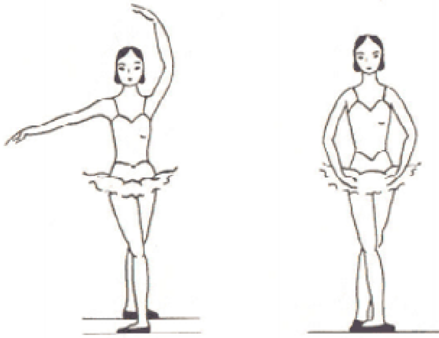
Encoding existing poses: We now drill down into an example to show how applications can encode gesture recognizers using the PREPOSE approach. Figure 4b shows a common ballet pose, taken from an instructional book on ballet. The illustration is accompanied by text describing the pose. The text states in words that ankles should be crossed, that arms should be bent at a certain angle, and so on.

Gestures in PREPOSE: Figure 4 shows the PREPOSE code which captures the ballet pose. Because of the way we have designed the PREPOSE language, this code is close to 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.

B. Architectural Goals

Figure 5 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

(a) A physical therapy application from Reflexion Health. The application senses the user's position using a Kinect sensor. On the right, the application displays a visualization of the user's current position. Along the top, the application describes the gesture the user must perform in English. The application also provides feedback to the user on how to better perform the gesture.



(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. The execution includes poses that have been previously defined.

Fig. 4: Motivating example.

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

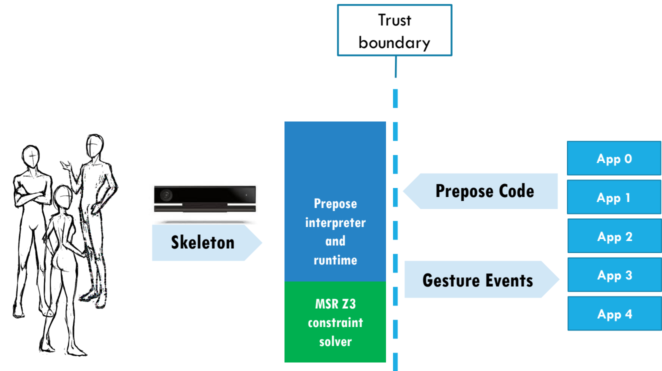


Fig. 5: Security architecture of Prepose.

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.

C. *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.

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.

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.

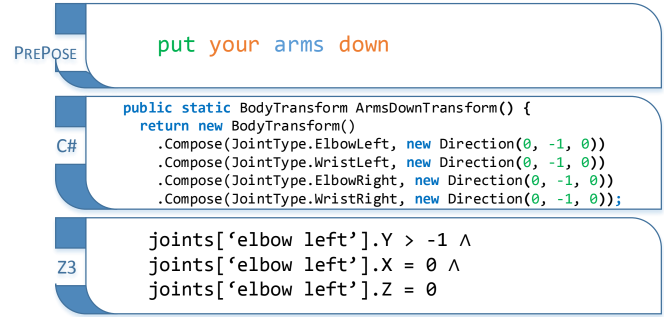


Fig. 6: Runtime correspondence: PREPOSE, C#, and Z3.

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 6 shows the stages of runtime processing in PREPOSE. A high-level PREPOSE statement is compiled into C# for the purposes of matching and to a Z3 formula for the purposes of static analysis.

D. *Gesture Safety and Privacy*

At gesture submission time, we apply static analysis to the submitted PREPOSE program. 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.

IV. TECHNIQUES

Figure 7 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 out of execution steps, etc[2].

The grammar is fairly extensible: if one wish 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.

A. *Compiling* PREPOSE *to SMT Formulas*

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

Basic transforms: Figure 8 captures the principles of translating PREPOSE transforms into Z3 terms. These are update rules that define the $\langle X, Y, Z\rangle$ coordinates of the joint to which the transformation is applied. Note that transformations take the plane p and direction d as parameters. These coordinate updates generally require a trigonometric computation. Because of the lack of support for these functions in Z3, we have implemented *sin* and *cos* using lookup tables for commonly used values.

Compound transforms: To build complex transforms that involve multiple joints, one can *compose* a list individual joint transforms, as illustrated below in the way one can encode the `arms-down` transform:

```
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))
    ;
}
```

Our runtime natively supports a number of compound gestures such as this one, as requires by the possible transforms in Figure 7.

Basic constraints: Figure 9 shows how PREPOSE restrictions are translated to Z3 constraints. Auxiliary functions

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

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. 7: BNF for PREPOSE. The start symbol is *app*.

Angle and *Distance* that are further compiled down into Z3 terms are used as part of compilation. Additionally, thresholds th_{angle} and $th_{distance}$ are used to define how closely the current positions match. These thresholds can be tuned in the runtime, as shown in PREPOSE Explorer

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

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

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

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

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

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

Fig. 8: Transformations translated into Z3 terms.

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

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

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

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

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

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

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

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

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

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

Fig. 9: Restrictions translated into Z3 terms.

in Figure 10.

B. Security and Privacy

By design, PREPOSE is amenable to sound static reasoning by translating queries into Z3 formulae. Below we show how to convert key security and privacy properties into Z3 queries.

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.

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: $\texttt{joint["rightelbow"].Y} = 1 \ \wedge$ $\texttt{joint["rightelbow"].Y} = -1$. To find possible contradictions in gesture definitions, we use the following query:

$$\forall b \in B : (G(b) \text{ is unsatisfiable}).$$

Protected gestures: Several NUI 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 positions that brings up a special system menu no matter which game is currently being played. 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 than 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

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. 11: 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.

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.

V. EXPERIMENTAL EVALUATION

We built a visual gesture development and debugging environment, which we call PREPOSE Explorer. Figure 10 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. We used this tool to measure the *expressiveness* of PREPOSE by creating 28 gestures in three different domains.

We then ran 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.

A. 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 11 shows the breakdown of how many gestures we created in each area, for 28 in total.

For example, Figure 12 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.

B. 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.

For each trace, we then measured the *matching time*: the time required to evaluate whether the current user

Fig. 12: Examples tai-chi gestures we have encoded using PREPOSE.

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.

C. Static Analysis Performance

Safety checking: Figure 13 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

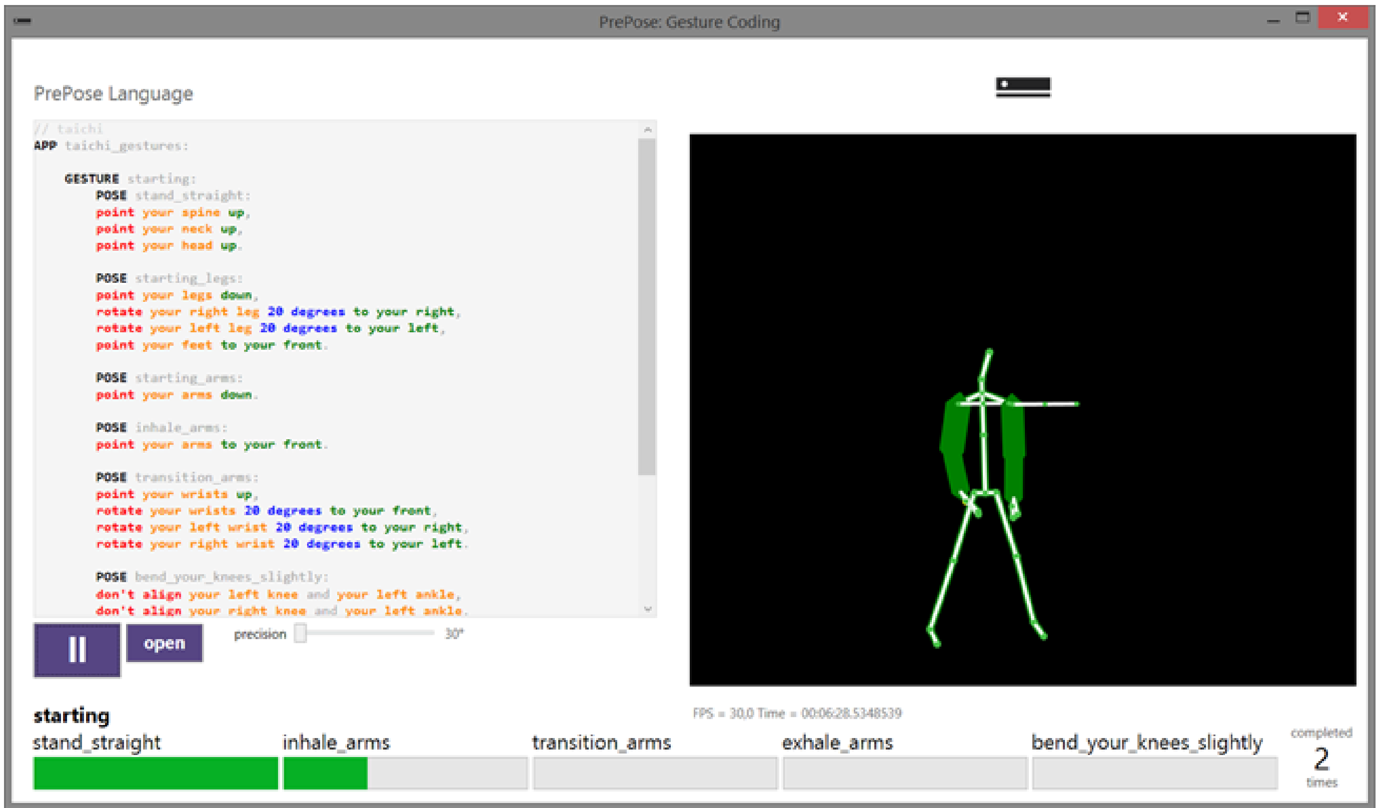


Fig. 10: Screenshot of PREPOSE Explorer in action.

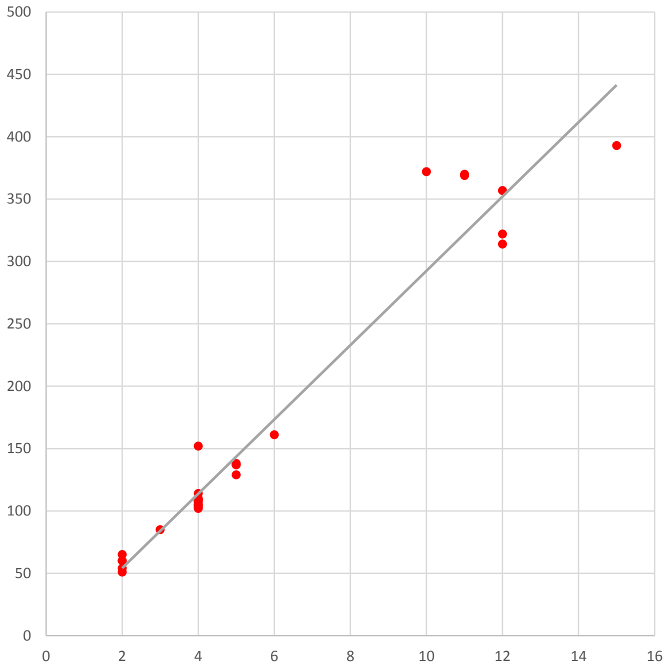


Fig. 13: Time to check for safety, in ms, as a function of the number of steps in the underlying gesture.

Validity checking: Figure 14 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.

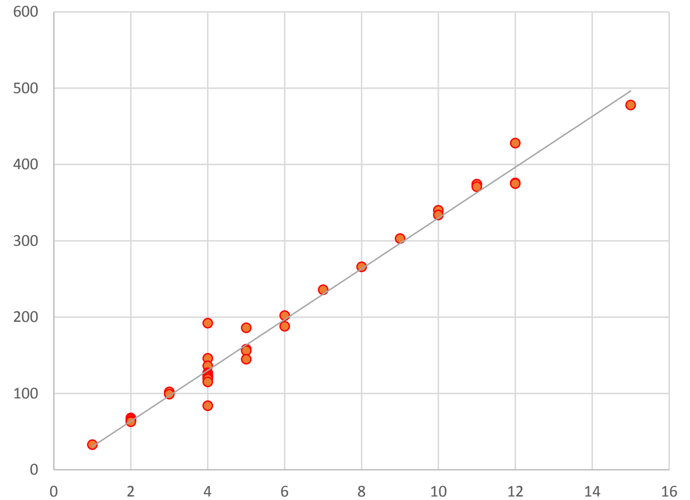


Fig. 14: Time to check internal validity, in ms, as a function on the number of steps in the underlying gesture.

Conflict checking: We performed pairwise conflict checking between 111 pairs of gestures from our domains.

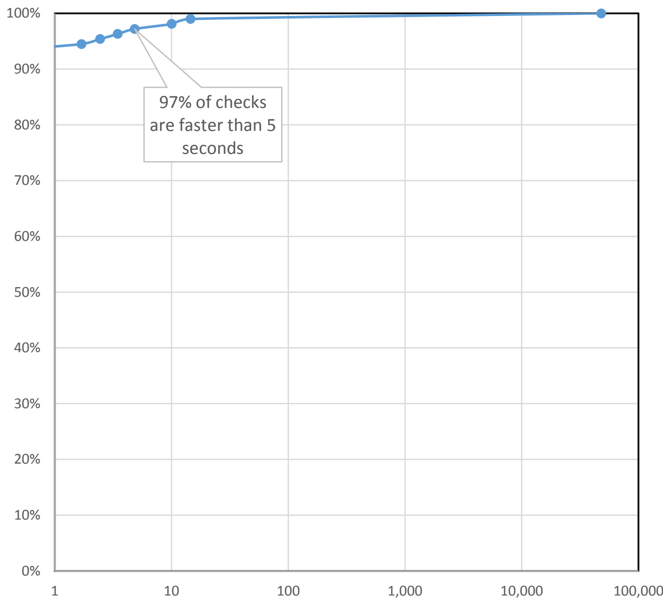


Fig. 15: Time to check conflicts for a pair of gestures presented as a CDF. The x axis is seconds plotted on a log scale.

Figure 15 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.

VI. Related Work

A. Gesture Building Tools

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 [18] 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 [16] and Proton++ [15] 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 [9] presents a scheme to represent hand and arms gestures on the space. 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 an equivalent gestures by interpreting the description and using their knowledge about gesture production.

BAP [5] approaches the task of coding body movements with focus on the study of emotion expression. The opposite way is performed, in which actors trained the system by performing specific emotion representations and these recorded frames were coded into poses 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 [21] proposes an approach for transcribing gestural movements by overlaying a 3D body skeleton on the recorded actorsâĂŹ gestures. This way, once the skeleton data is aligned with the recorded data, the annotation can be created automatically. It is limited to posing arms.

RATA [22] presents a tool to create recognizers for touch and stylus gestures. The focus is on the easiness and low-time consuming 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 [14] 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 [19] 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 [4] 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 [17] tool to help developers, which are not expert in pattern recognition, to create gesture interfaces. Focuses on motion gesture (using 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.

B. Sensing and Privacy

SurroundWeb [24] 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 data 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 webpages and object detection code (never telling the webpages if objects were detected or not) and natural user inputs (mapping the inputs into mouse events to the webpage).

Darkly [13] 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 [6] and AR Apps with Recognizers [12] 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 [3] proposes an OS modification for smartphones 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 [10] proposes a tool for privacy control on mobile devices which can block or shadow sent data to applications in order to maintain the application up and running but prevent sending on-device use only data.

What You See is What You Get [11] proposes a widget which alerts users of which sensor is being requested by which application.

VII. Conclusions

We introduced the Prepose language, which allows developers to write high level gesture descriptions that have semantics in terms of SMT formulas. To test expressiveness, we created 28 gesture recognizers in Prepose across three important domains. Our architecture protects security and privacy by preventing untrusted applications from directly accessing raw sensor data; instead, they register Prepose code with a trusted runtime. We also showed that Prepose programs can be statically analyzed quickly to check for safety, pairwise conflict, and conflicts with system gestures. Both 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 language designed from the ground up to support security and privacy, we obtain strong guarantees without sacrificing either performance or expressiveness.

Our Z3-based approach has more than acceptable performance. Pose matching in Prepose averages 4 ms. Synthesizing target poses 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 90% of the cases, the conflict detection time is below 0.170 seconds, and 97% of cases less than 5 seconds, with only one query taking more than 15 seconds.

References

[1] S. Amini and Y. Li. Crowdlearner: rapidly creating mobile recognizers using crowdsourcing. In *Proceedings of the 26th annual ACM symposium on User Interface Software and Technology (UIST)*, 2013.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] K. 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.

[8] S. Fothergill, H. Mentis, P. Kohli, and S. Nowozin. Instructing people for training gestural interactive systems. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '12, 2012.

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

[10] 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.

[11] 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.

[12] 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 22nd USENIX Security Symposium*, 2013.

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

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

[15] K. Kin, B. Hartmann, T. DeRose, and M. Agrawala. Proton++: A customizable declarative multitouch framework. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology (UIST)*, 2012.

[16] K. Kin, B. Hartmann, T. DeRose, and M. Agrawala. Proton: Multitouch gestures as regular expressions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '12, pages 2885–2894, New York, NY, USA, 2012. ACM.

[17] 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.

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

[19] 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.

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

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

[22] 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.

[23] 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.

[24] J. Vilk, D. Molnar, E. Ofek, C. Rossbach, B. Livshits, A. Moshchuk, H. J. Wang, and R. Gal. SurroundWeb: Least Privilege for Immersive Web Rooms. Technical Report MSR-TR-2014-25, February 2014. http://research.microsoft.com/apps/pubs/default.aspx?id=209968.