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[†]
lsf@cin.ufpe.br {dmolnar,margus,livshits}@microsoft.com

Abstract—With the rise of sensors such as the Microsoft Kinect, 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 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.

Index Terms—H.5.1.b [Multimedia Information Systems] Artificial, augmented, and virtual realities; I.5.4.d [Applications] Face and gesture recognition; D.4.6 [Operating Systems] Security and Privacy Protection; K.4.1.d [Public Policy Issues] Human safety

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.

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, and some also 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.

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

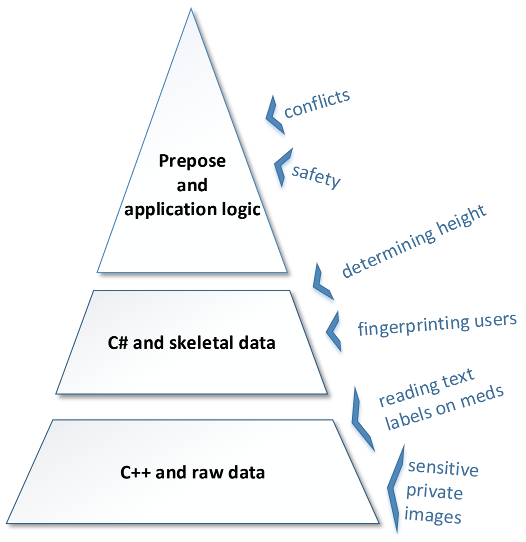


Fig. 1: Three different levels of data access for untrusted applications that perform gesture recognition. We call out threats to the user at each levels.

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 Loris D'Antoni [2] 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.

1.1 A Case for Controlled Access to Skeletal Data

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 [5, 14]. 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.

1.2 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 [10]. 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 regarding Reliability and Security which are summarized as follows:

1) Prepose 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;
2) Prepose checks for inner contradictions 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.

1.3 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. Additionally, our language syntax is compatible with natural language, which increases its readability and semantics for user guidance and errors handling. This way, a gesture inner validity or a safety fault is easier to be read and corrected.
- **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.

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

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.

2.2 Single Application Programming Model

Today, common AR applications 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.* [2]. 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 [5]. The key downside of this approach is that it requires a fixed set of recognizers that cannot be changed by applications.

2.4 Programming with Gestures

Today, developers of immersive, sensor-based applications pursue two major approaches to creating new gesture recognizers.

- The developer may write 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. This approach is typically reasons about user movements at a low level, requires the tuning of threshold values, and is not amenable to automatic reasoning.
- The leading alternative to manually-coded gesture recognizers is to use *machine learning* approaches. In order to create the recognizer the developer takes recordings of many different people performing the same gesture, then tags the recordings to provide labeled data. However, gathering the data and labeling it can be expensive. 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. 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 Techniques

PREPOSE applications can be composed out of gestures, and gestures composed out of poses and execution steps as shown by the following written example:

```
GESTURE right_shoulder_abduction:

  POSE relax_arm:
  point your right arm down.

  POSE perform_abduction:
  rotate your right arm 90 degrees to your right.

  EXECUTION:
    relax_arm,
    perform_abduction.
```

Each pose in PREPOSE is built as a one or more actions (point, rotate, put, align or touch). By its turn each action requires specific parameters, for example, the point action written on the code above requires a body part ("right arm") and a direction ("down"). For researchers who wish to extend PREPOSE, we have uploaded an a ANTLR version of the PREPOSE grammar to `http://binpaste.com/fdsdf`.

Interacting with the solver: PREPOSE compiles programs written in the PREPOSE language to formulae in Z3, a state-of-the-art SMT solver. In order to illustrate interaction with the Z3 solver consider the statement "put your left elbow behind your neck". 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 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. 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.

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.

3.1 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.

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: A set of restrictions are applied to ensure the input gesture is safe. 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

```
    point your arms up,
    point 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:

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

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.

4 Experimental Evaluation

We built a visual gesture development and debugging environment, which we call PREPOSE Explorer. Figure 2 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.

4.1 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 3 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

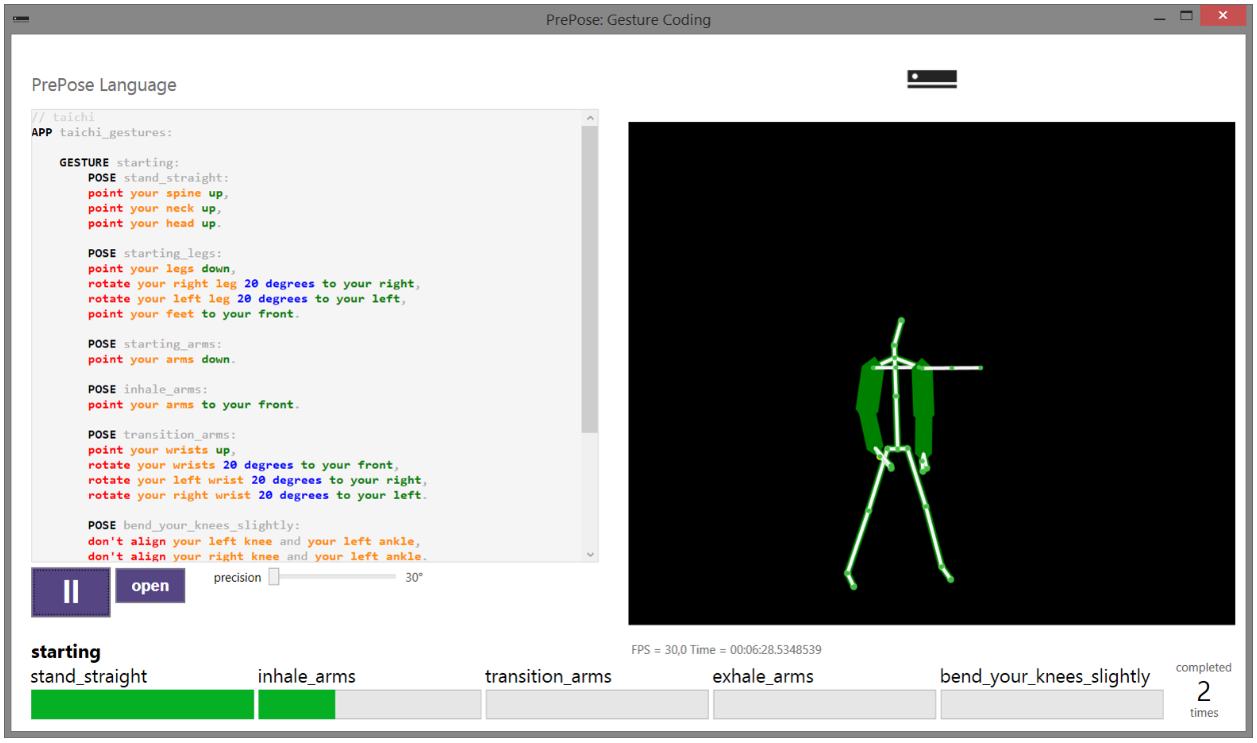
- We used this tool to measure the *expressiveness* of PREPOSE by creating 28 gestures in three different domains.
- We then ran some benchmarks to measure runtime performance and static analysis performance of PREPOSE. First, we report runtime performance, including the amount of time required to match 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.

4.2 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 3 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 3.

Prepose runtime gives feedback not only when the gesture is completed but also in which part (percentage) of the gesture the user is, in real-time. This way, it is possible to save a full log of the session, storing knowledge about each execution, even in cases that the user did not went all the way to the last pose


PrePose: Gesture Coding

PrePose Language

```
// 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.
```

precision 30°

FPS = 30,0 Time = 00:06:28.5348539

starting

stand_straight | inhale_arms | transition_arms | exhale_arms | bend_your_knees_slightly

completed 2 times

Fig. 2: 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. 3: 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.

of the gesture. Therefore, it is possible for example, to report to a physiotherapist a detailed execution of the treatment including the patient performance even if the gesture was not finished.

Additionally, on the particular case of therapy, there are exercises which require the user to move her body segments (e.g.: arms) within the limits of specific biomechanical body planes (Frontal, Horizontal and Sagittal). Although body planes are not yet directly supported by Prepose these gestures can be written as a tight sequence of motions on the same plane to satisfy these cases. That said, most of the therapy gestures written on our experiments (such as the 'crossover left arm stretch') do not require the definitions of body planes.

4.3 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.

4.4 Static Analysis Performance

Safety checking: On the left of figure 4 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: The middle part of figure 4 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. The right of figure 4 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.

5 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.

5.1 Gesture Building Tools

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

Proton [8] and Proton++ [7] 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.

Zhao et al. [15] 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.

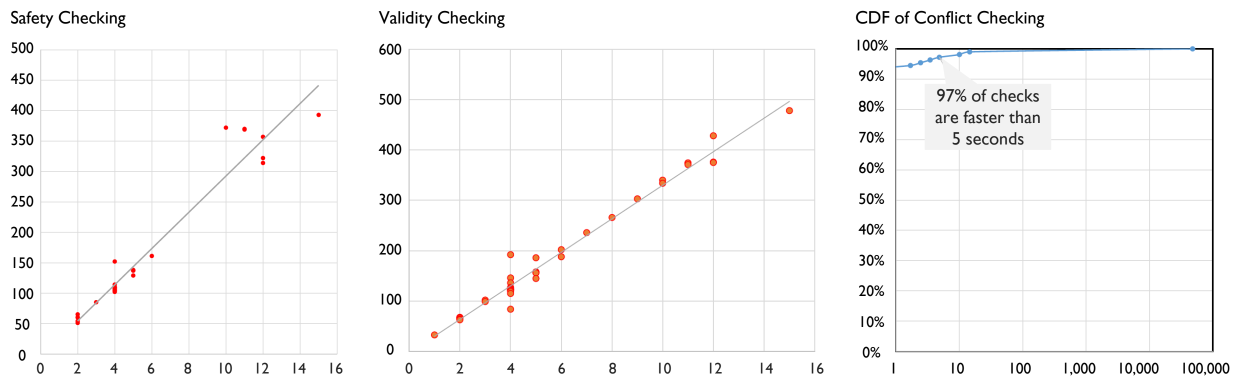


Fig. 4: On the right part the time to check for safety, in ms, as a function of the number of steps in the underlying gesture. On the middle the time to check internal validity, in ms, as a function on the number of steps in the underlying gesture. On the left the time to check for conflicts for a pair of gestures presented as a CDF. The x axis is seconds plotted on a log scale.

GDL [3] 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). GDL uses a near general programming syntax as shown on the following sample code: RULE RightElbow.x[0] $>$ Torso.x[0] & abs(RightShoulder.y[0] - RightElbow.y[0]) $<$ 50 THEN RightArmGestureName. 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.

Hoste and Signer [4] 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.

5.2 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 [14] 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 [6] 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 [2] and AR Apps with Recognizers [5] 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.

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 [12].

6 Summary and Looking Forward

This paper provides a foundation for programming and reasoning about gesture safety, security, and privacy. While this paper assumes that the developer will author the gesture code, we envision numerous possibilities related to automatically inferring PREPOSE programs *by demonstration* [1, 9, 11]. 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 [13]. 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.

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.

Authors

Lucas Silva Figueiredo

Lucas Silva Figueiredo is a research leader at the Voxar Labs on the Federal University of Pernambuco (UFPE) in Brazil. He is currently finishing his Ph.D. in Computer Science on the Informatics Center/UFPE. Obtained his MSc and BSc in Computer Science on the same Center. During his Ph.D., in 2014, Lucas spent three months as a research intern on Microsoft Research - Redmond, conducting research related to real time gesture recognition for user interaction and postural analysis. His research interests include Augmented and Virtual Reality, Gesture Recognition, Computer Vision and Natural Interaction. Lucas works in R&D&I projects since 2008. Currently he is conducting a research project for real-time in-air gestures recognition inside Voxar Labs.

Margus Veanes

Margus is a senior researcher at Microsoft Research Redmond doing research in symbolic automata theory with applications to program analysis and verification. His research interest range from formal language theory to program verification and optimization techniques powered by modern logical inference engines. Margus received his PhD in Computer Science from Uppsala University and is a professional member of IEEE and ACM.

David Molnar

I lead ”Project Springfield”, which packages pioneering technology and best practices from Microsoft into a cloud service everyone can use. Project Springfield builds on the ”whitebox fuzzing” technology invented by Patrice Godefroid and colleagues at Microsoft, putting it into a cloud service run by William Blum and our engineering team. Learn more about Project Springfield and sign up for a preview at https://www.microsoft.com/springfield (and our team site, coming soon!)

Prior to Project Springfield, I spent several years in the Security and Privacy Group at the Microsoft Research Redmond lab. Before MSR, I spent several years at the University of California Berkeley, where I finished a PhD with David Wagner. My area of focus is software security : software is eating the world, so if there's a problem with software, then the software might accidentally eat us. How can we manage this risk from security critical software errors?

Benjamin Livshits

Ben Livshits is a research scientist at Microsoft Research in Redmond, WA and an affiliate professor at the University of Washington. Originally from St. Petersburg, Russia, he received a bachelor's degree in Computer Science and Math from Cornell University in 1999, and his M.S. and Ph.D. in Computer Science from Stanford University in 2002 and 2006, respectively. Dr. Livshits' research interests include application of sophisticated static and dynamic analysis techniques to finding errors in programs.

Ben has published papers at PLDI, POPL, Oakland Security, Usenix Security, CCS, SOSP, ICSE, FSE, and many other venues. He is known for his work in software reliability and especially tools to improve software security, with a primary focus on approaches to finding buffer overruns in C programs and a variety of security vulnerabilities (cross-site scripting, SQL injections, etc.) in Web-based applications. He is the author of over 100 academic papers; Ben has also received dozens of patents and multiple tech transfer awards for bringing research in practice. Lately, he has been focusing on topics ranging from security and privacy to crowdsourcing an augmented reality. Ben generally does not speak of himself in the third person.

References

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

[2] 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.

[3] 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.

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

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

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

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.