GlucOS: Security, correctness, and simplicity for automated insulin delivery

Hari Venugopalan
hvenugopalan@ucdavis.edu
UC Davis

Shreyas Madhav Ambattur Vijayanand
smvijay@ucdavis.edu
UC Davis

Caleb Stanford
cdstanford@ucdavis.edu
UC Davis

Stephanie Crossen
scrossen@ucdavis.edu
UC Davis

Samuel T. King
kingst@ucdavis.edu
UC Davis

Abstract—Type 1 Diabetes (T1D) is a metabolic disorder where an individual's pancreas stops producing insulin. To compensate, they inject synthetic insulin. Computer systems, called automated insulin delivery systems, exist that inject insulin automatically. However, insulin is a dangerous hormone, where too much insulin can kill people in a matter of hours and too little insulin can kill people in a matter of days.

In this paper, we take on the challenge of building a new trustworthy automated insulin delivery system, called GlucOS. In our design, we apply separation principles to keep our implementation simple, we use formal methods to prove correct the most critical parts of the system, and we design novel security mechanisms and policies to withstand malicious components and attacks on the system.

We report on real world use for one individual for 6 months using GlucOS. Our data shows that for this individual, our ML-based algorithm runs safely and manages their T1D effectively. We also run our system on 21 virtual humans using simulations and show that our security and safety mechanisms enable ML to improve their core T1D measures of metabolic health by 4.3% on average. Finally, we show that our security and safety mechanisms maintain recommended levels of control over T1D even in the face of active attacks that would have otherwise led to death.

GlucOS is open source and our code is available on GitHub.

1. Introduction

Type 1 diabetes (T1D) is an autoimmune disease in which an individual's pancreas stops producing insulin. People living with T1D must therefore take synthetic insulin and play the role of their pancreas. However, replicating the pancreas's native insulin release patterns is extremely difficult, and optimal T1D self-management often requires hourly adjustments in insulin dosing both in response to physiological changes and in prediction of upcoming carbohydrate intake and energy expenditure (e.g., exercise, stress) [51], [56]. Even when following this labor-intensive process of T1D self-management, affected individuals remain at risk for acute episodes of hypoglycemia (low blood sugar, which can cause loss of consciousness, seizure, or death) or ketoacidosis (which carries an associated risk for multi-organ injury), as well as for long term vascular complications (e.g., kidney damage, nerve damage, blindness, heart attack, stroke) that are a result of chronic hyperglycemia (high blood sugar) [46]. 8.4 million people are living with T1D worldwide [25].
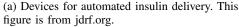
The last two decades have witnessed the development of therapeutic technology to assist T1D management, including continuous glucose monitoring (CGM) devices and insulin pumps. Most recently, the integration of CGM sensors and insulin pumps into "artificial pancreas" or automated insulin delivery systems has enabled a simultaneous improvement in health outcomes and reduction in the burden of care for T1D-affected individuals [38].
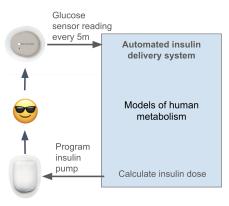
However, existing automated insulin delivery systems are complex, and there is no way to assure beyond any doubt that they are implemented correctly. For example, techniques from formal methods have not been applied to verify automated insulin delivery systems. With little reason to trust the systems and their core algorithms, users and developers are also reluctant to incorporate experimental algorithms into the prediction loop, such as advanced ML-driven insulin delivery. Researchers have shown that ML can improve automated insulin delivery systems [64], [33], [63], [59], [65], [58], yet the lack of security and safety for these ML predictions keeps people from using it. If ML is going to be part of medical decisions, we need the security community to lead the way on how to do this safely.

In this paper, we argue for applying traditional security and operating systems design principles to make automated insulin delivery systems more trustworthy. This includes going beyond just patient privacy or medical device security, to *systems security* for automated insulin delivery systems to ensure that they are implemented correctly and that the incorporation of ML does not jeopardize their safety.
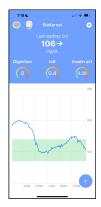
Automated insulin delivery systems are different from traditional applications that the security community focuses on, but they are important. Research on operating systems or web browsers can list scores of CVEs from current systems and show how their system improves security by eliminating

1

(a) Devices for automated insulin delivery. This figure is from jdrf.org.

(b) Closed-loop architecture.

(c) Our system for automated insulin delivery.

Figure 1: Overview of automated insulin delivery devices, closed-loop architecture, and apps.

the impact of some or all of these vulnerabilities by virtue of their design. Automated insulin delivery systems do not have a long list of CVEs that we can reference. However, our position is that we need to make sure that these systems *never* get CVEs, because if they do, security vulnerabilities can kill people. Thus, we focus on building a new system, using security first principles and formal methods, to avoid security vulnerabilities and mitigate their damage from the start.

Designing and implementing trustworthy automated insulin delivery systems brings with it three main challenges. (1) How can automated insulin delivery design decisions improve rather than worsen the security posture of such software for end users? (2) How do we apply formal methods to a complex, distributed system that includes a human? (3) How can we cope with ML mispredictions, including those from malicious ML?

In this paper, we take on these challenges and design and implement a new trustworthy automated insulin delivery system, called GlucOS. Our design for GlucOS strives to achieve the following four primary goals: to use sound security principles to separate security and safety concerns into critical and non-critical components; to keep our implementation as simple as possible; to prove correct the most critical components; and to enable any predictive algorithms, including deep neural network ML, to control the insulin pump securely and safely.

GlucOS improves on the current state-of-the-art in automated insulin delivery systems by, for the first time, enabling deep neural network ML to control the insulin pump securely and safely. Our key insight for security is that **over a long enough period of time, all correct algorithms will dose the same amount of insulin.** For example, on a given day, a person will need to dose a fixed amount of insulin to absorb all of the glucose from the food they eat. Since synthetic insulin action is slow, injecting insulin early by predicting future metabolic states is critical for long-term health. GlucOS's support for modern ML-based algorithms provides users with this predictive power to dose

insulin. For security, GlucOS pairs a predictive ML model with a conservative and safe model to provide the ML with enough flexibility to control the insulin pump proactively while staying within the bounds dictated by the safe model.

Our architecture and system are a stark contrast to current systems. Current systems are complex, their architecture provides little or no isolation boundaries, they are unable to withstand attacks on their system, and they are unable to use deep neural network ML securely and safely. Given that mistakes with insulin delivery can kill people, our goal is to have a fully-featured automated insulin delivery system with security and safety as the primary design considerations. Even if parts of the system are malicious, GlucOS continues to operate and ensure that the individual using the system can manage their T1D safely.

We report our experiences running GlucOS on one individual for six months to manage their T1D. Running our software on a real human forces us to design a practical and real system. Our evaluation using virtual humans in a simulator shows that our security and safety mechanisms generalize beyond the specific individual.

Our novel contributions include:

- The clean-slate design and implementation of a simple yet effective automated insulin delivery system.
- Security mechanisms and policies that protect individuals from malicious ML, vulnerable pump drivers, and drastic changes in the human's glycemic response.
- The application of formal methods to prove correct the most critical parts of GlucOS.
- A case study describing our experiences from a real-world deployment, and results from simulation to show that our techniques generalize.

We give top priority to the health, safety, and ethics of the human using GlucOS to manage their T1D. Section 11.1 summarizes our safety protocol and Appendix A provides a detailed description.

2. Background

Type 1 Diabetes. T1D is an autoimmune condition in which the immune system attacks and destroys the insulin-making cells in the pancreas. Insulin is a hormone that enables the absorption of glucose (sugar) from the bloodstream into the brain, the muscles, and other organs and tissues where it is used as an energy source. When people eat, the digestive system converts food into glucose, which makes its way into the bloodstream, but it cannot then be used by the body without the action of insulin.

Since people with T1D are unable to produce their own insulin, they must inject synthetic insulin to replace this key function, and the appropriate dosing of this insulin is both complicated and labor-intensive. For example, insulin doses must be closely correlated with food intake, but the absorption of exogenous insulin is slower than food digestion, so people with T1D must precisely predict their future food intake in order to inject the right amount of insulin prior to each meal. Once injected, typical insulin formulations will affect glucose metabolism for up to three hours, and during that time will interact with other factors like exercise, sleep, stress, caffeine, and alcohol. However, the exact effects of these factors on an individual's glucose levels are even harder to predict than those of food alone, resulting in the need for continuous adjustments by the individual in order to maintain a balanced glycemic state. One solution that has emerged to address the complications of daily T1D management is automated insulin delivery systems, which connect a CGM sensor and insulin pump to enable automatic, individualized adjustments in insulin delivery based on glucose trends [14], [18], [40], [48], [17], [7], [4].

The goal of T1D management is to maintain glucose levels that are in a safe, target range (70-180 mg/dl) for the majority of the time – an outcome that has been linked to positive long-term health outcomes. Glucose levels 54-69 mg/dl (level 1 hypoglycemia) or less than 54 mg/dl (level 2 hypoglycemia) put individuals at risk for acute complications like impaired consciousness and seizure, while levels 181-250 mg/dl (level 1 hyperglycemia) or greater than 250 mg/dl (level 2 hyperglycemia) increase risks for long-term vascular damage. The main physiological concept in this paper is a goal of maintaining glucose levels that are in the target range the majority of the time.

Automated insulin delivery systems. Automated insulin delivery systems connect a CGM device – which measures interstitial glucose concentration every 1-5 minutes – with an insulin pump, which delivers a continuous subcutaneous insulin infusion. Using a variety of algorithms, these systems vary insulin delivery in response to CGM data, reducing the amount of human input and cognitive load required to achieve glycemic stability (Figure 1). Several commercial [17], [7], [4] and open source [40], [48] automated insulin delivery systems exist today.
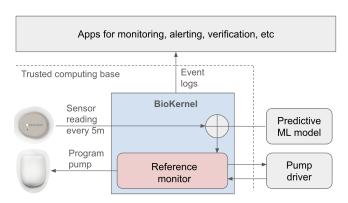

Figure 2: Overall architecture for GlucOS.

3. Overview

This paper describes our design for GlucOS, a system for trustworthy automated insulin delivery. In this section, we first provide an overview of GlucOS's system architecture to describe the key components involved in automated insulin delivery. While we provide an overview of all components, we specifically focus on those components that are part of our trusted computing base. We then define our threat model in terms of our assumptions and goals with GlucOS.

3.1. Architecture

From a system design perspective, our primary goals are to keep our implementation simple and enable our use of formal methods to prove correct the most critical parts. For our software system architecture, we use separation principles from the OS and microkernel areas [10], [30] applied to the application layer to push complexity away from the most critical parts of the system. This architecture is similar to secure web browsers [26], [61], [57], [53], which also apply OS separation principles at the application layer. By decomposing the overall automated insulin delivery system, we strive to keep our trusted computing base simple. Also, we want flexibility so that people can run whatever closed-loop algorithms they want to and use whatever pump hardware they want – we focus on providing mechanisms, policies, and systems for security and safety.

Figure 2 shows our overall system architecture. At the core of our overall architecture is the BioKernel. The BioKernel is the component that interacts with the CGM and insulin pump hardware, executes the closed-loop dosing algorithm, runs security and safety checks, and produces event logs that other components use to learn the state of the system.

The BioKernel runs the closed-loop algorithm every five minutes as new CGM readings come in. Once the BioKernel receives new CGM readings, it collects the current metabolic state of the individual and sends it to a predictive ML model. The predictive ML model calculates the amount of insulin to inject based on its assessment of the individual's current

metabolic state and predictions of future metabolic states. The BioKernel then issues commands to the pump to deliver the calculated amount of insulin. A pump driver converts these commands into low-level I/O to program the pump and adjust insulin delivery.

To withstand attacks, the BioKernel serves as a *reference monitor* [12] to enforce security policies on the predictive ML model's pump commands and the pump driver's low-level I/O. This reference monitor architecture ensures that we can withstand attacks from malicious predictive ML as well as malicious pump drivers without having to know the internals of how they operate.

For predictive ML, our reference monitor enforces algorithmic security, where we bound the amount of insulin calculated by ML to stay within dynamic safety bounds computed by a *reactive safe physiological model*. For the pump driver, we interpose on the I/O path to ensure that only the pump commands issued by the BioKernel are programmed into the pump. These checks constitute our *pump invariants*, which, when violated switch the pump to operate in a safe mode.

Our use of formal methods focuses on the software related to delivering insulin, which is the most risky operation in any automated insulin delivery system. We use Hoare logic to prove key functions correct, define system states and transitions to handle runtime verification check failures safely, and we introduce the notion of a novel *biological invariant.* Our biological invariant is an end-to-end check of our theoretical calculations of how the individual's metabolism should behave compared to what we observe in practice, where violations of this invariant suggest a potentially catastrophic failure of the system.

3.2. Threat Model

We consider attacks from malicious predictive ML algorithms as well as malicious pump drivers. We assume that malicious ML can intentionally miscalculate the amount of insulin to inject and malicious pump drivers can issue arbitrary pump I/O operations to add, drop, or modify commands sent to the pump.

We also consider real-world safety concerns as part of our threat model. ML mispredictions are part of these safety concerns since even well-designed ML will sometimes make poor decisions. Safety concerns also include pump communication errors emerging from bugs in the pump driver, pump mechanical failures [55], and pump site skin inflammation or infection [36], all of which result in a mismatch between the amount of insulin calculated in closed loop and the actual amount of insulin active in the human.

Our goal with GlucOS is to not merely detect attacks and safety violations. Our goal is to build a secure system for automated insulin delivery that can withstand active attacks and real-world safety concerns.

We assume that CGM sensors are correct and operate within their error bounds. If the CGM sensors stop producing measurements, our system will be unable to handle it. We also assume that the Swift type system is correct and that the underlying operating system, the CGM driver and hardware, and the pump hardware we use are free of attacks and their device communications maintain integrity. Thus, we do not consider attacks emerging from tampered CGM sensors or pump hardware (such as adversarial attacks against predictive ML) in our threat model. These are important but complementary problems to our work, which focuses on the system architecture and algorithms for automated insulin delivery.

4. Algorithmic security: Withstanding malicious ML

None of the current automated insulin delivery systems use deep neural network ML. Although they do use ML in general, they use statistical methods, like linear regression or model predictive control [19], [60], [42], [52], [48], to learn physiological parameters. The reason for using this more traditional form of ML is sound: it provides determinism and has a physiological basis that people can use to vet its decisions. However, all of the recent advances in ML, like image classifiers and large language models, have come from deep neural networks, which researchers have shown also work well for predicting metabolic states for use in automated insulin delivery systems [64], [33], [63], [59], [65], [58]. Despite this promise, none of the current automated insulin delivery systems use more advanced deep neural networks due to the black-box nature of deep neural networks and the potentially dire consequences of mispredictions – an automated insulin delivery system's unchecked "hallucination" [1] could be lethal if it delivers an inappropriate insulin dose [20], [13].

Modern ML models make mistakes. High-profile examples include Google Photos classifying black people as gorillas [2], LinkedIn recruiting ML learning gender bias [5], Microsoft's chatbot, Tay, trained on Twitter data (what could go wrong) becoming racist [3], and ChatGPT large language model hallucinations [1]. In all of these examples, professional ML engineers with plenty of resources produced well-designed ML systems that hit a gray area in their training data or decision space and made bad decisions.

All automated insulin delivery systems that use ML need to have a mechanism to protect against malicious ML or benign ML that makes inevitable bad decisions. In this section, we discuss the design and implementation of GlucOS's algorithmic security and safety mechanisms.

4.1. Mechanism fundamentals

Our key insight that underpins our security mechanism design is that all correct insulin delivery algorithms will deliver the same amount of insulin over a long enough period of time. However, the timing of this insulin delivery matters for maintaining balanced glucose levels (Section 10). Thus, our algorithmic security mechanism pairs a predictive model that anticipates future metabolic states and doses insulin accordingly with a reactive safe model that measures the
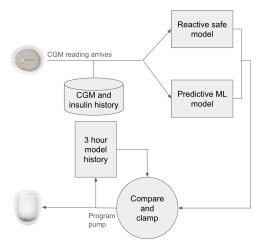
Figure 3: Algorithmic security mechanism for GlucOS.

current metabolic state and doses insulin based only on what it can measure currently.

Conceptually, the predictive model is the primary model in our system and should control the pump most of the time, and we use the reactive safe model for accounting so that we can track how far the predictive model deviates from a known-to-be-safe baseline. Then, we bound the size of this deviation. By combining these two models, we strive to get both the safety of a reactive safe model that operates only on facts (Section 4.2) and the tight control we expect from an ML-based predictive model (Section 4.3).

Figure 3 shows how our algorithmic security mechanism works. The whole process starts when a new CGM reading arrives. Once the BioKernel receives a new CGM reading, it collects a recent history of CGM readings and insulin dosing and sends these as inputs to both the reactive safe model and the predictive ML model. These models independently produce pump commands, called a temporary basal command. The temporary basal command sets the insulin delivery rate for the insulin pump for a 30-minute duration. The BioKernel will collect both temporary basal commands and look back at the differences between the reactive safe model and predictive ML model for the past three hours to see how much insulin delivery it should attribute to the predictive ML model. If the predictive ML model is still within its insulin delivery limits, it uses the predictive ML model's temporary basal command to program the pump. If it has exhausted its limits, we clamp it to fit within the insulin bounds we set relative to the reactive safe model. For more details about how this algorithm works, please see our implementation in GitHub (link omitted).

4.2. Reactive safe model

Since we use our reactive safe model to anchor our security policy, and it runs within our trusted computing base, our goals for this model are to have it be safe, easy to understand, and have a simple implementation.

To make this model safe, we base its calculations solely on facts observed automatically from the CGM and the insulin pump. Given these high-quality sources of data, and a simple insulin absorption physiological model, we can calculate how much insulin an individual needs and how much insulin they have injected but have not yet activated. From this calculation, we know how much insulin to inject or withhold.

These calculations are standard calculations that people who inject insulin manually use to determine dosing, making them easy for people who manage T1D or their medical care team to understand. The difference in our system is that we run these calculations automatically and every five minutes to adjust to the latest sensor readings. This basic formulation is a standard formulation used in most automated insulin delivery systems.

One wrinkle we add to this otherwise standard calculation is that we learn the physiological parameters we use in these calculations using a data-driven approach for safety. Our learning system consists of a weekly learning task that evaluates three weeks worth of data to infer broader physiological trends. These learned settings become the baseline for our parameters. Then, we use a PID controller from feedback control [22] to refine these values given recent observations. Being able to adapt is important for handling pump malfunction and pump insertion site safety violations (Section 3.2).

Calculating insulin is a simple calculation to implement, consisting of only 25 lines of formally verified code in the BioKernel (Section 8). However, despite this simplicity, it is effective (Sections 10 and 11). The main drawback of this model is that is it unable to detect underdosing insulin during food digestion quickly enough to avoid high glucose.

4.3. Predictive model

To predict insulin underdosing during digestion, we use a food prediction deep-learning model from the literature [45]. Our version of this model is a generalization where we model any source of glucose, not just food, but the details of it are beyond the scope of this paper. In our evaluation our model outperforms the reactive safe model both in simulation (Section 10) and for the individual that uses our system for their real-world insulin dosing (Section 11), showing why it is important for automated insulin delivery systems to have a practical way to adopt ML.

Figure 1 shows an example of when the predictive ML model correctly detects digestion and doses insulin to maintain balance. This figure shows a screenshot from the BioKernel's main user interface. The individual ate breakfast at 9 am and their glucose levels rose. The predictive ML model correctly detects that the individual is going to absorb additional glucose from digestion, resulting in a predicted glucose level rise over the next hour. Accordingly, the predictive ML model injects enough insulin to cover the rising glucose from the meal and keep the individual's glucose levels in range.
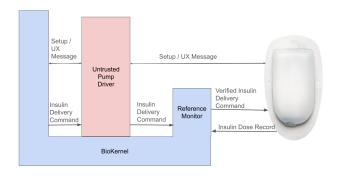
Figure 4: We interpose on communications between the pump driver and the pump to handle untrusted pump drivers to deliver insulin.

5. Driver security: Withstanding malicious pump drivers

Pump drivers provide abstractions for insulin delivery applications to communicate with insulin pumps. Much like traditional device drivers, they have complex implementations since they help accomplish a wide range of tasks including pump setup/teardown, managing pump UX, handling insulin delivery/suspension, managing pump errors, accurately tracking insulin doses etc. Different pump drivers use different implementations to carry out these tasks.

While CVEs do not currently exist for pump drivers, there is strong possibility for bugs or vulnerabilities in their implementation given their complexity and the fact that they not been formally verified. Since drivers control insulin pumps, these vulnerabilities can even lead to the death of individuals. For example, consider a scenario where the user decides to inject 3 U of insulin using the BioKernel. The BioKernel would relay this as a command to the pump through abstractions provided by the pump driver. If the pump driver inadvertently or intentionally issues this command twice, 6 U of insulin would be delivered to the user, which would put the user in danger by drastically reducing the amount of glucose in their body.

With GlucOS, our goal is to support their complexity by allowing users to employ pump drivers of their choice while upholding security. Our security goals are to not just protect users from inadvertent bugs in pump drivers, but to also protect them from malicious pump drivers which could intentionally add, modify or drop pump commands.

5.1. Reference monitor overview

We introduce a reference monitor as part of the BioKernel to handle untrusted pump drivers, which interposes on communications between the pump driver and the pump. We choose to vet these communications instead of implementing our own pump driver to give users the ability to use any pump driver of their choice. This also helps keep our trusted computing base small thereby easing formal verification.

Figure 4 provides an overview of our reference monitor. We only validate commands pertaining to insulin delivery since they pose the most dangerous threats to the user. From a high-level, the pump registers insulin commands with the reference monitor before issuing them to the pump driver. In the reference monitor, we check to ensure that the command issued by the pump driver is consistent with the command registered by the BioKernel. At the reference monitor, we also intercept all records of delivered insulin doses to ensure their integrity for closed loop calculations. On detecting manipulation from the pump driver, we transition insulin delivery to a default safe state (described in Section 7) where there is no immediate threat to the user. We do not validate communication pertaining to pump setup / UX since users can directly detect the respective manipulations. Figure 5 lists the different types of messages pertaining to insulin delivery exchanged between a pump driver and an insulin pump. We list other message types in Appendix B.

5.2. Design Principles

Avoid programming the pump. We avoid issuing insulin delivery commands to keep our implementation simple.

Ensure that the user always stays safe. Delay in delivering insulin does not pose any immediate threat to the user. However, delay in cancelling a wrongly issued insulin delivery command (such as one with a high insulin dose) can potentially kill the user. Thus, in the interest of keeping users safe, we violate the principle of not issuing commands to the pump in two specific cases of cancelling insulin delivery.

Disregard non-critical functionality in the interest of security. Some pump drivers incorporate an automatic retry mechanism when they're unable to communicate with the pump. Such retry mechanisms complicate validation, since it becomes difficult to distinguish between duplicate commands issued as part of a retry versus duplicate commands issued as the result of a bug or an attack. We, thus, disregard such functionality by treating all duplicate commands as bugs or attacks.

5.3. Reference monitor design and implementation

The BioKernel invokes methods exposed by the reference monitor to register commands to be sent to the pump before invoking the relevant methods in the pump driver. The BioKernel also registers two callbacks with the reference monitor, one to be invoked on successfully receiving a response from the pump and the other in case of error.

The pump driver then updates its internal states, encodes the command and its relevant parameters and sends it to the reference monitor. The reference monitor then checks that the pump driver has not added new commands or modified the parameters before encrypting and sending them to the pump. If the reference monitor detects added or modified commands, it transitions the pump to operate in a default safe state and rejects all further commands from the driver.

Message name	Message description
getStatus	Request insulin delivery status from the pump
statusResponse	Insulin dose delivery information from the pump
errorResponse	Insulin dose delivery error from the pump
setBasalSchedule	Set default rate of insulin delivery
setTempBasal	Set temporary rate of insulin delivery
bolus	Deliver instant insulin dose
cancelDelivery	Cancel insulin delivery

Figure 5: Types of insulin related messages exchanged between between a pump driver and an insulin pump.

We employ a bounded delay of one second to detect dropped commands. Concretely, the reference monitor detects dropped commands if it does not receive anything from the pump driver within one second of registering a command from the BioKernel. It treats dropped commands the same way as added or modified commands by transitioning the pump to a default safe state and rejecting all further commands. We transition the user out of this safe state and start accepting commands only after the user manually configures the BioKernel to trust the driver again.

The reference monitor operates differently on detecting dropped commands that deal with cancelling insulin delivery. Users will seek to cancel insulin delivery if they accidentally triggered an insulin delivery command or incorrectly set its parameters. Since excess insulin injection is dangerous, the reference monitor would automatically send these commands to the pump before transitioning the pump to the default state and rejecting all further commands. There are two commands that deal with cancelling insulin delivery: cancelDelivery and setting the suspend option on setTempBasal. cancelDelivery seeks to cancel a previous insulin delivery command and (if the delivery is still ongoing) and the suspend option on setTempBasal seeks to suspend all insulin delivery until the user resumes delivery.

We focus our implementation on Loop's OmniBLE [6] pump driver. OmniBLE allows iOS applications to communicate with OmniPod pumps over Bluetooth. We include OmniBLE's cryptography and Bluetooth communication modules as part of the BioKernel and incorporate the reference monitor just before encrypting messages sent to the pump/ decrypting messages received from the pump.

OmniBLE does not incorporate automatic retry in its implementation. However, OmniBLE occasionally sends additional commands as part of its operation. For example, when sending a setTempBasal command, if OmniBLE's internal states indicate that delivery is still in progress as part of a previous setTempBasal command, it first issues a cancelDelivery command to cancel that delivery. To remove ambiguity in terms of whether the cancelDelivery command was issued as part of setTempBasal, we enforce OmniBLE to always issue the cancelDelivery command. We evaluate GlucOS in terms of driver security in Section 10.4.

Overall, we employ straight-forward checks to detect manipulations from buggy/malicious pump drivers. However, the reference monitor's design is more nuanced to handle the full range on insulin commands, missing out on any of which could potentially kill the user.

6. End-to-end security: Biological invariant

A key novelty in our overall approach is our introduction of a *biological invariant* that provides end-to-end verification. End-to-end verification in this context means that we verify the BioKernel's glucose calculations against observations of the user's metabolic state. If there is a large mismatch between our calculations and what we observe, then it means that our calculations are wrong. These deviations can happen from drastic changes in the user's glucose absorption, like during exceptionally vigorous exercise or if the pump insertion site hits a vein and delivers insulin directly into the blood stream. Regardless of the cause, the biological invariant detects these conditions and prevents harm to the user.

The biological invariant closes the gap between verified software and human metabolism by incorporating a direct check against the user's physiological data into the formal verification process. The biological invariant provides a runtime mechanism to detect when GlucOS's calculations have become unsound. Specifically, the biological invariant compares the BioKernel's theoretical calculation of the user's blood glucose level against the actual observed blood glucose readings. Theoretical glucose levels provide an estimate of how the system thinks a given insulin dose will influence glucose levels. GlucOS uses its estimate of the human's insulin sensitivity when calculating these doses. A significant divergence between the theoretical and observed values indicate a potential catastrophic failure mode, where GlucOS's estimates of the user's insulin sensitivity and calculations have diverged from the user's real physiological state. The biological invariant allows the system to safely transition into a default safe state (Section 7). If the divergence between theoretical and observed glucose exceeds a threshold, indicating a loss of confidence in the model's calculations, GlucOS discontinues automated insulin delivery and enters the safe state.

The biological invariant primarily focuses on resolving the undesirable situation where GlucOS believes that the person is less sensitive to insulin than they really are. In this case, the delivered insulin will remove more glucose than required, leading to potentially fatal low glucose levels. Consider the example of microbolusing when the user is above recommended glucose levels. If the system assumes a state where it believes the user is less sensitive to insulin than they really are and the user is 2x more sensitive, it will inject 2U of insulin when only 1U is required. While the system believes this will restore the user to safe levels, the actual glucose suppression will now push the user towards dangerously low glucose levels. Due to the higher
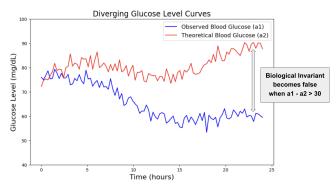
Figure 6: This figure shows an example of diverging differences between theoretical glucose and the observed glucose levels for a user. GlucOS will lose confidence in its calculationsand tranisition to a default safe state once it goes beyond a fixed difference threshold, where the biological invariant fails.
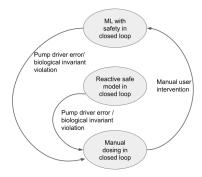

Figure 7: GlucOS's state machine to transition users to the safest manual dosing state when losing trust in insulin delivery (i.e., with a buggy or malicious pump driver) or when the biological invariant is not satisfied.

clinical risk, we emphasize triggering the invariant in the undercalculation of insulin sensitivity case to transition to the default safe state. We set a threshold of 30 mg/dL for the maximum allowable divergence as shown in Figure 6 based on clinical evaluations showing detectability of significant glycemic deviations above this level[37], [32].

We formally model and verify the biological invariant using the Dafny language. The core aspects we specify include:

(a) A ghost variable representing the ground-truth physiological blood glucose level of the user. This models the actual, unobserved glucose state inside the human body that the system is trying to estimate and control.

(b) The theoretical calculation of the blood glucose level according to the system's model. This captures the glucose prediction made by the system based on its assumptions about insulin dynamics, metabolism etc.

(c) The observed value of the blood glucose level from glucose readings from the user.

(d) The invariant property itself, which asserts that the absolute difference between the theoretical model glucose and the ground-truth physiological glucose cannot exceed a threshold of 30 mg/dL.

Our primary goal of verification in this segment is to make sure that there are no execution sequences that can violate the invariant without reaching a state where automated insulin delivery is disabled. This verification also ensures the biological invariant correctly captures our intention - if the system's theoretical model deviates too far from the real physiological state, automated dosing is stopped to prevent potential over/under delivery of insulin. We also verify framing properties, showing the invariant's state updates are precise and do not inadvertently modify other system variables. The verified Dafny specifications are then translated to Swift to integrate the biological invariant into the runtime system implementation for execution, with high assurance from the Dafny verification that they correctly implement the

safe-guarding behavior in correspondence with the proven formal model.

Incorporating the biological invariant enables fundamental safety assurances not possible with traditional software verification alone. First, it constrains the set of possible erroneous system behaviors by restricting manual divergence between the theoretical model and actual physiological state. Section 10.5 shows that the biological invariant improves the system's time-in-range glycemic control significantly compared to baseline when the system's model and knowledge of user parameters becomes unsound. Second, it provides higher confidence of overall system reliability by directly incorporating physiological cross-checks into the verification process. We consider the biological invariant an important conceptual contribution that can extend the formal verification process to dynamically check and respond to deviations at runtime forming the complete bridge between the software and human physiology.

Traditional software verification techniques like model checking [35] and deductive verification [24] have been explored in the past for hybrid system verification [23] that interact with the human body, but they do not completely account for physiological variability between individuals or model inaccuracies during the insulin delivery process.

7. Runtime failure management

In this section, we describe GlucOS's *state machine* that combines its security components, i.e., algorithmic security (Section 4), driver security (Section 5) and end-to-end security (Section 6) together to protect users in light of their corresponding failures.

GlucOS's state machine consists of three operating states: (1) ML with safety operating in closed loop, (2) reactive safe model operating in closed loop and (3) manual dosing in closed loop. ML provides the tightest level of control in managing glucose but poses the largest immediate threat from excess insulin since it preemptively injects insulin by anticipating rise in glucose. The reactive safe model

provides slightly lesser control, but poses milder immediate threat since it only injects insulin to account for measured excess glucose. Manual dosing offers the least amount of control but poses no threat since it only injects the minimum insulin needed for sustenance.

GlucOS predominantly operates in the state running ML (with safety) in closed loop. Certain real-world failures (such as missing CGM readings) transition GlucOS to operate in the state running the reactive safe model in closed loop. However, such failures are simple and we discuss ways to handle them in Appendix C. In this section, we focus on transitions pertaining to the manual dosing state.

We transition to the safest manual dosing state when we do not trust insulin delivery (i.e., with a buggy or malicious pump driver). We also transition to the safest manual operation when the biological invariant is not satisfied to ensure that we do not risk the possibility of removing too much glucose from our insulin doses. If we end up in the manual dosing state, we enforce that only manual user intervention can transition back to other states. We visualize these state transitions in Figure 7.

8. Formal verification

A single error or vulnerability in an automated insulin delivery system can have life-threatening consequences for the user. This criticality motivates our use of formal verification in our development and testing process. Through formal verification, we aim to confirm two critical aspects of our insulin delivery system. First, we validate the goals and claims behind our critical insulin delivery functions. Second, we verify that GlucOS applies the correct state transitions in precarious scenarios to protect the user when the system loses confidence in one or more of its components due to miscalculation or malicious behavior.

Systematically identifying the invariants and postconditions in alignment with our claims that we verify for a critical function in our system is important. Our first step is to clearly define what the function is supposed to do. This helps list all the inputs that the function takes, including their types and ranges. Then, we define the expected outputs, including their types and valid ranges. Determining safety and domain-specific constraints for each function plays a major role in ensuring that the function never produces an unexpected output that puts a user at risk. For example, a general constraint would be that a basal rate should never be negative as it is invalid. A domain-specific constraint is defining a maximum basal rate threshold that is in accordance with medical guidelines. For a full list of the invariants that we define and prove, see Appendix D.

These steps help define the preconditions, postconditions and invariants that help prove the properties of our verified functions. We run formal verification in Dafny to verify our claims on how a function works. After we verify a function, we port the code to Swift to make it part of the BioKernel.

We verify invariants and postconditions for the system's critical functions in the aforementioned approach. For instance, we verify the microbolusing function to ensure a

minimum time interval between consecutive micro-boluses to prevent frequent dosing in short intervals (within 4 minutes), confirm that the user glucose level is significantly above the target(atleast 20 mg/dL) before administering a bolus and that the bolus amount is within safe and predefined limits. These are claims about our function that we formally verify to assure that they are always correct.

In another example, we prove correct our insulin delivery calculations to ensure that we maintain accurate accounting of insulin delivery. For this verification task, we verify our insulin delivery over a duration of time function to guarantee non-negative results, correctly handle zero duration or non-overlapping time segments, making sure that the dosage does not exceed recommended limits, and accurately calculate insulin delivery based on time segment overlap.

We perform this type of verification on all the critical functions involved in the insulin delivery process. This high-level of assurance helps us trust the the decisions that GlucOS makes.

We also formally verify the state machine we use for defining system transitions under undesirable scenarios (Section 7). We discover several potential issues in the state transition logic and function design during verification. For example, there were cases where the system could transition to an unsafe state under certain conditions, potentially leading to excessive insulin delivery. We also identified bugs related to violating valid bounds and domain-specific constraints. A more detailed discussion of the bugs have been included as part of subsection 10.6 in our evaluation.

9. Implementation

In our implementation, the BioKernel is an iOS app on iPhone or iPad hardware. It communicates with a Dexcom G7 CGM and Omnipod DASH insulin pump via Bluetooth low energy and runs the core closed-loop algorithm. For the BioKernel, we use the Swift programming language, the Actor abstraction for data-race-free concurrency, and JSON files stored on disk to capture persistent state.

Although the BioKernel is a fully featured and stand alone automated insulin delivery system, GlucOS uses an event logging system that enables separate apps to implement complementary features while remaining isolated from the BioKernel. We implement our event logging system using a Google Cloud App Engine app and the BioKernel produces logs that other apps consume. The other apps include an app for alerting when the individual is likely to experience hypoglycemia or hyperglycemia, an app to replay closed-loop runs to verify the results, and an app for running simple ML on data to calculate therapeutic settings.

To keep predictive ML and the pump driver isolated, we use the Swift type system and object boundaries to maintain isolation. Our architecture also supports the use of process boundaries for even strong isolation and to implement OS-level sandboxing. iOS lacks support for proper process APIs (fork, exec etc), so we use the Swift type system instead.

For CGM and insulin pump drivers, we use LoopKit [41] and isolate untrusted portions using the Swift type system.

9

Patient type	TIR with ML model	TIR with reactive safe model
Adults	93.49%	88.25%
Adolescents	85.23%	81.98%
Children	86.6%	77.31%

Figure 8: Average proportion of time spent in range by 21 virtual patients in identical scenarios with a our predictive ML model and our reactive safe model.

10. Evaluation

Given the risks associated with running ML for automated insulin delivery, we answer most questions on 21 virtual humans from an FDA-approved simulator [9]. The simulator generates CGM readings across different age groups including children, adolescents and adults. We also evaluate security mechanisms on virtual patients to see if they can generalize to individuals with different physiologies. We present experiences of running GlucOS with a real individual in the next section.

10.1. Does ML lead to better outcomes for managing T1D?

We confirm findings from prior research that ML leads to better outcomes [47], [66], [34] by comparing the average proportion of time spent by virtual patients in range when using our reactive safe model (Section 4.2) against our predictive ML model (Section 4.3). As mentioned in Section 2, the American Diabetes Association (ADA) recommends that an individual with T1D should spend more than 70% of a day (also called TIR) [8], [21] in order to remain healthy.

We train separate predictive ML models (feed-forward neural networks trained to minimize MSE loss) for each virtual patient and similarly tune the physiological parameters of the reactive safe model to suit each patient. We ensure that we run the reactive safe model and the predictive ML model on identical conditions within the simulator.

We show the average proportion of time spent in range by virtual patients from each age group (adults, adolescents and children) in Figure 8. We can see that ML helps virtual patients across all age groups spend a larger amount of time in range, thereby showing that ML leads to better outcomes for managing T1D.

10.2. What are the risks associated with untrusted ML algorithms for managing T1D?

We assess the impact of predictions made by intentionally malicious ML algorithms on managing T1D. Since GlucOS is designed to be flexible with any predictive algorithm, it could be subject to attacks from malicious ML algorithms. We intentionally introduce an order of magnitude difference to the amount of insulin predicted by the predictive ML model in Section 10.1. In one experiment, we dose one-tenth of the amount of insulin and in another experiment, we dose
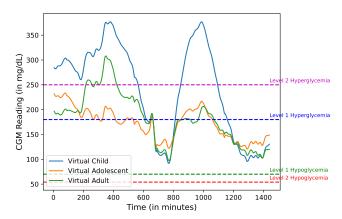

Figure 9: Variation in CGM readings for 3 virtual patients when running a predictive ML algorithm that intentionally doses one-tenth the amount of the required insulin. The CGM readings staying mostly above the Level 1 hyperglycemic range would have drastic effects on well-being.
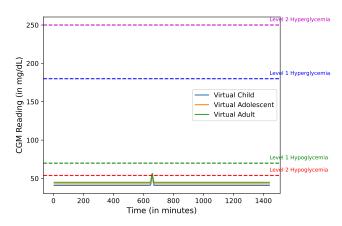

Figure 10: Variation in CGM readings for 3 virtual patients when running a predictive ML algorithm that intentionally doses ten times the amount of the required insulin. The CGM readings staying below the Level 2 hypoglycemic range would have likely been fatal to all 3 patients.

ten times the amount of insulin. We run these malicious ML algorithms within the simulator on each virtual patient on the same scenarios as the previous experiment.

Figure 9 and Figure 10 show the CGM readings on 3 randomly chosen virtual patients on a particular scenario for the experiments where we dose less and dose more respectively. When dosing less, we see that all patients spend most of their time above Level 1 hyperglycemia which would have long-term effects on their well-being. When dosing more, all patients fully go below Level 2 hypoglycemia, which would have killed them. Across all virtual patients, we see an average drop in TIR from 88.44% with correct dosing to 23.32% when dosing less and an average drop in TIR from 88.44% to 0% when dosing
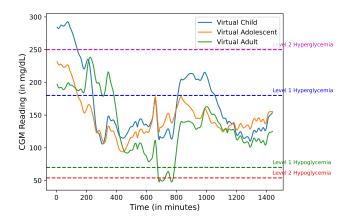
10

Figure 11: Variation in CGM readings for 3 virtual patients when clamping an algorithm that intentionally tries to dose one-tenth the amount of the required insulin. On average, the CGM readings stay in range over 70% of the time.
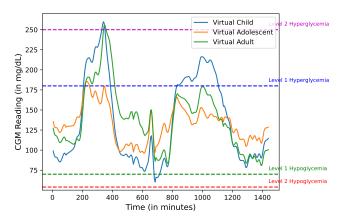

Figure 12: Variation in CGM readings for 3 virtual patients when clamping an algorithm that intentionally tries to dose ten times the amount of the required insulin. On average, CGM readings stay in range over 70% of the time.

more. Correspondingly, when dosing less, we observe an average increase in the time in hyperglycemia from 3.77% with correct dosing to 65.05%. Similarly, when dosing more, we observe an average increase in the time in hypoglycemia from 0% to 100%.

These results further motivate the need for security mechanisms when providing individuals with the flexibility to use any algorithm to manage T1D.

10.3. What is the impact of GlucOS's algorithmic security mechanism on managing T1D?

GlucOS's algorithmic security mechanisms should keep the blood glucose of individuals in range even in presence of malicious ML algorithms. The mechanisms should also ensure that they do not significantly dampen the level of control provided by an efficient algorithm.

Impact on malicious algorithms. We repeat the experiments from the previous section with the same scenarios. We run the same malicious algorithms, but also run the reactive safe model to set bounds on the amount of insulin the malicious algorithms can inject. Figure 11 and Figure 12 show CGM readings on the same 3 random virtual patients on the same scenarios from the experiment in the previous section. When running either malicious model, we see that all patients spend over 70% of their time in range. These results show that GlucOS protects individuals even in presence of intentionally malicious algorithms. On average, across all virtual patients, we still see a drop in TIR from 88.44% with correct dosing to 73.89% when employing safety while dosing less and a drop in TIR from 88.44% to 78.22% when dosing more. While the times spent in range is less than correct dosing, the impact of incorrect dosing is significantly dampened with patients spending the recommended amount of time in range.

Impact on effective benign algorithms. We evaluate the impact of GlucOS's algorithmic security mechanisms

Patient type	TIR with clamped Predictive ML model
Adults	92.88%
Adolescents	82.38%
Children	85.42%

Figure 13: Average proportion of time spent in range by 21 virtual patients in the same scenarios in Section 10.1 when clamping the predictive ML with the reactive safe model.

on benign algorithms such as the predictive ML model used in Section 10.1. We repeat the experiment of running the predictive ML model on all virtual patients in the same random scenarios while maintaining the same parameters for the clamps and reactive model used for the malicious algorithms evaluation.

Comparing Figure 13 against Figure 8 we see that there is only a slight drop in the average proportion of time spent in range by virtual patients across all age groups when employing GlucOS's algorithmic security mechanisms. The average proportion of time spent in range being higher than the reactive safe model shows that GlucOS's security does not come at the cost of the benefits from ML.

10.4. Does GlucOS ensure pump operation while overcoming buggy/malicious pump drivers?

We first evaluate GlucOS's driver security mechanism in terms of its impact on benign pump drivers that accurately relay pump communications. We then evaluate the impact on detecting additional, modified or dropped communication from a malicious pump driver. Our mechanism should ensure that all communications go through with a benign pump driver and should catch all manipulated communication with a malicious pump driver. We do not use the simulator for this evaluation since it does not incorporate a pump driver.

Figure 14: We attached an Omnipod DASH insulin pump to a stuffed animal and modified the OmniBLE pump driver to intentionally add, drop and modify insulin commands. Our driver security mechanism detected all manipulations.

Patient type	TIR without biological invariant	TIR with biological invariant
Adults	49.93%	88.92%
Adolescents	55.83%	75.26%
Children	67.41%	77.45%

Figure 15: Average proportion of time spent in range by 21 virtual patients when fluctuating insulin sensitivities. We compare cases with and without our end-to-end security mechanism using the biological invariant.

Impact on malicious pump drivers. As shown in Figure 14, we attached an Omnipod DASH insulin pump to a stuffed animal. We then modified the source code of the OmniBLE pump driver to intentionally add, drop and modify insulin commands. We detected all manipulated pump commands showing that our driver security mechanism has 0 false negatives in detecting manipulated commands.

Impact on benign pump drivers. For one week, we logged messages from the BioKernel to unmodified OmniBLE as well as messages sent from OmniBLE to the pump on an individual running GlucOS (See Section 11). We report 100% consistency between the commands sent to OmniBLE and commands issued from OmniBLE showing that GlucOS's driver security mechanisms do not incur any false positives.

These results show that GlucOS can ensure pump operation while withstanding attacks from malicious drivers.

10.5. Is the biological invariant effective?

We run experiments on the simulator to evaluate if GlucOS's use of the biological invariant to transition to manual insulin dosing leads to better T1D management. Since the simulator does not provide support to fluctuate insulin sensitivity, we intentionally scale up or scale down values for insulin sensitivity when calculating insulin doses in closed loop. We trigger cases where we scale down the

insulin sensitivity to simulate scenarios where the system assumes a lower sensitivity value than the user's actual sensitivity more frequently since this are more relevant to evaluating the biological invariant.

Across all virtual patients, we record an average TIR of 57.56% when fluctuating insulin sensitivity without incorporating the biological invariant. In contrast, we record an average TIR of 80.54% when incorporating the biological invariant to transition to manual dosing. Figure 15 shows the average TIR across age groups. We clarify that we use the correct insulin sensitivity values to compute bolus doses when dosing manually, since users would know their true insulin sensitivity when taking in manual doses.

10.6. Did formal verification help with correctness?

During the formal verification process of our secure insulin delivery system, we uncovered various bugs and inconsistencies that could have compromised the safety and reliability of the system in niche scenarios. We identified a total of 9 bugs across different categories. 3 of these bugs were related to invalid ranges for variables such as negative value returns, domain-specific out of range values, potential division by zero errors etc. in the insulin dosing calculations which were resolved by adding appropriate checks and error handling mechanisms to prevent such operations and ensure graceful handling of these cases. Additionally, we found 4 instances where the implemented functions violated their intended postconditions along the insulin delivery path, which were addressed by thoroughly reviewing and modifying the implementations to correctly satisfy the postconditions, preserving the intended behavior and meeting our claims for critical funcrions. Moreover, our analysis uncovered 2 situations where certain functions lacked proper error handling mechanisms, potentially leading to undefined behavior or system failures. By addressing these bugs and incorporating formal verification throughout the development lifecycle, we significantly enhanced the robustness, reliability, and safety of our secure insulin delivery system.

11. Using GlucOS in the real world

In this section, we report on our experiences working with one individual, Bob, since November 2023. We show our results for using GlucOS with our reactive safe model closed-loop algorithm as well as with our predictive ML model closed-loop algorithm paired with our security mechanism. We also measure system performance, report on errors, and quantify our implementation simplification efforts.

Overall, our study is a correlation study, where we evaluate one individual using our system and compare segments of time with each other.

11.1. Ethical considerations

In this section, we outline the steps we take to ensure that we uphold high ethical standards and ensure that the human using our system, who we will call Bob, is safe.

Glucose range (mg/dl)	Range label	Reactive 67 days	Reactive last wk.	ML 1 wk
<54	L2 Hypo	0.08%	**0%**	**0%**
54-69	L1 Hypo	0.59%	0.48%	**0%**
70-180	In range	96.35%	91.90%	**97.22%**
181-250	L1 Hyper	2.79%	7.62%	**2.78%**
>250	L2 Hyper	0.18%	**0%**	**0%**

Figure 16: The percent of time spent in different glucose ranges for the reactive safe model and the predictive ML model. Bold values represent the best results for each range.

First, one of the co-authors of this paper is a board-certified Endocrinologist, who specializes in T1D. They are both a clinician with an active practice seeing patients living with T1D and a researcher who focuses on research on T1D. They designed the safety protocol that we detail in Appendix A, which defines the testing we conduct to ensure that the individual using our system is safe, and the specific criteria we use to stop the study, if needed.

Second, we submitted our study to our university's IRB, which determined that our "self evaluation" is exempt from IRB (case number omitted, but available).

Third, Bob increased the frequency of his visits to his Endocrinologist, which includes lab work to confirm that he remained in good overall metabolic health.

Overall, GlucOS system had a large positive impact on the individual who used it. Bob's lab-based A1C test (a blood test that measures average glucose levels over time), which includes his time using GlucOS, was 5.8%. That is nearly non-diabetic levels of control – healthy individuals will have an A1C of 5.6% or lower. This result is a profound outcome for Bob, and GlucOS had a massive positive impact on Bob's health and quality of life.

11.2. Is an ML-based closed-loop system safe and effective in practice?

Bob has been using GlucOS in a closed loop since November 23rd, 2023. From November 23 to January 28th, 2024 we used our reactive safe model to make control decisions. Starting on January 28th Bob started to use our predictive ML model for periods of time to get used to the new system, and then starting on January 31st, 2024 starting using our predictive ML model exclusively. We report on one week's worth of data from the predictive ML model starting on January 31st ending on February 6th, 2024.

Figure 16 shows the amount of time Bob spent in different glucose level ranges over the duration of our study. Data from the reactive safe model covers 67 days, followed by one week of data from our predictive ML model. We also show data for the last week of using the reactive safe model to show trends over time. Overall, Bob had the best results when using ML.

Next, we measure how often the predictive ML model was responsible for programming the pump vs our reactive safe model. For the one week when Bob was using the pre-

dictive ML model, 33.0% of the time both the predictive ML model and reactive safe model issued the same commands, 39.9% of the time the BioKernel used the predictive ML model's commands, and 27.1% of the time the BioKernel used the reactive safe model's commands. We believe that this distribution represents a suitable balance for providing the ML with enough flexibility to improve outcomes, while having tight enough bounds to limit potential damage from bad or malicious ML predictions.

11.3. How does the BioKernel perform?

We look at two aspects of system performance when evaluating our real-world use of GlucOS. First, we measure how long our closed-loop algorithm takes to run and second, we count how often the closed-loop algorithm runs successfully end-to-end. We use Bob's iPhone 14 to collect all of the data we show here and these numbers are averages across the entire six month period.

On average, the BioKernel spends 4.3s running the closed loop algorithm, where almost half of that time is dominated by CGM and pump communications. Of the components within the closed-loop algorithm, the ML predictions take 2.2s on average and the safety logic takes 19ms (0.019s). Since the system runs in the background, we believe that iOS runs the device with reduced computational capacity to save power. But overall, 4.3s of wall-clock time for every five minutes our closed-loop algorithm runs with only a tiny fraction of that coming from our safety logic represents practical system performance.

For successful closed-loop end-to-end execution, over the evaluation period the closed-loop algorithm runs successfully 96.5% of the time, with 2.1% of the runs failing due to pump communication failures, which is our largest class of failure.

11.4. How simple is our BioKernel?

To evaluate the complexity of our BioKernel, we count the lines of code in our implementation and compare against Loop, another open-source automated insulin delivery system.

Using the "cloc" utility the BioKernel app has 4.0k lines of code compared to 39.8k lines of code in the Loop app, an order of magnitude reduction. We omit the lines of code coming from LoopKit, which both the BioKernel and Loop use, because our pump security measures remove most of it from our trusted computing base. However, LoopKit and the drivers have a substantial amount of code, weighing in at 70.1k lines of code, showing the importance of our security measures.

To explain why we have such a large reduction in source code, we outline the differences between the two systems. First, Loop has several features that we move outside of the BioKernel. These features include a remote interface for insulin dosing, a Watch app, Siri command interfaces, third party libraries, tutorials, complex meal announcements, and

physiological simulations. Second, we simplify our implementation of features that are shared between the BioKernel and Loop. These shared features include a simplified local storage implementation, simplified concurrency support, minimal therapeutic settings, and a simplified closed-loop algorithm. As our evaluation shows, we can still provide tight control over T1D with this simplified core system and still support most or all these additional features architecturally.

12. Related work

Previous research has looked at the security of implanted medical devices in general [29], [16], [54], in addition to looking at insulin pumps in particular [39], [50], with more recent work looking at providing improved security [44], [11]. Also, recent work has looked at applying formal methods to insulin pumps for high assurance [49]. These works focus on the device and their communication channel. In contrast, with GlucOS we assume that these devices are correct and secure and focus our efforts on the software we use to run the automated insulin delivery system.

Our work on coping with malicious pump drivers builds on previous work on malicious device drivers in operating systems [62], [15], [27], [28], [43], [31]. Our work applies these principles to a new domain: automated insulin delivery systems.

13. Conclusion

Our audacious long-term goal is to turn a T1D diagnosis from a death sentence into an indicator of longevity, where people living with T1D will be expected to live longer than their healthy peers. This longevity will come by virtue of the tight control that they maintain over their metabolism through advanced computer systems. Our first step towards this goal is to ensure that people can use trustworthy computer systems to manage their glucose.

References

[1] Chatbots may 'hallucinate' more often than many realize. https://www.nytimes.com/2023/11/06/technology/chatbots-hallucination-rates.html.

[2] Google mistakenly tags black people as 'gorillas,' showing limits of algorithms. https://www.wsj.com/articles/BL-DGB-42522.

[3] In 2016, microsoft's racist chatbot revealed the dangers of online conversation. https://spectrum.ieee.org/in-2016-microsofts-racist-chatbot-revealed-the-dangers-of-online-conversation.

[4] Insulet omnipod 5. https://www.omnipod.com/.

[5] Linkedin's job-matching ai was biased. the company's solution? more ai. https://www.technologyreview.com/2021/06/23/1026825/linkedin-ai-bias-ziprecruiter-monster-artificial-intelligence/.

[6] Omnipod bluetooth pumpmanager for loop. https://github.com/LoopKit/OmniBLE/tree/dev.

[7] Tandem control iq. https://www.tandemdiabetes.com/products/automated-insulin-delivery/control-iq.

[8] Time-in-range and diabetes, 2022. https://www.endocrine.org/patient-engagement/endocrine-library/time-in-range-and-diabetes.

[9] simglucose, 2024. https://github.com/jxx123/simglucose.

[10] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for unix development. 1986.

[11] Usman Ahmad, Hong Song, Awais Bilal, Shahzad Saleem, and Asad Ullah. Securing insulin pump system using deep learning and gesture recognition. In *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pages 1716–1719. IEEE, 2018.

[12] James P Anderson et al. Computer security technology planning study. Technical report, ESD-TR-73-51, 1972.

[13] Stephen R Benoit, Yan Zhang, Linda S Geiss, Edward W Gregg, and Ann Albright. Trends in diabetic ketoacidosis hospitalizations and in-hospital mortality—united states, 2000–2014. *Morbidity and Mortality Weekly Report*, 67(12):362, 2018.

[14] B Wayne Bequette. A critical assessment of algorithms and challenges in the development of a closed-loop artificial pancreas. *Diabetes technology & therapeutics*, 7(1):28–47, 2005.

[15] Silas Boyd-Wickizer and Nickolai Zeldovich. Tolerating malicious device drivers in linux. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*, 2010.

[16] Wayne Burleson, Shane S Clark, Benjamin Ransford, and Kevin Fu. Design challenges for secure implantable medical devices. In *Proceedings of the 49th annual design automation conference*, pages 12–17, 2012.

[17] Luz E. Castellanos, Courtney A. Balliro, Jordan S. Sherwood, Rabab Jafri, Mallory A. Hillard, Evelyn Greaux, Rajendranath Selagamsetty, Hui Zheng, Firas H. El-Khatib, Edward R. Damiano, and Steven J. Russell. Performance of the Insulin-Only iLet Bionic Pancreas and the Bihormonal iLet Using Dasiglucagon in Adults With Type 1 Diabetes in a Home-Use Setting. *Diabetes Care*, 44(6):e118–e120, 06 2021.

[18] Claudio Cobelli, Eric Renard, and Boris Kovatchev. Artificial pancreas: past, present, future. *Diabetes*, 60(11):2672–2682, 2011.

[19] Claudio Cobelli, Eric Renard, and Boris Kovatchev. Artificial Pancreas: Past, Present, Future. *Diabetes*, 60(11):2672–2682, 10 2011.

[20] Philip E Cryer. Severe hypoglycemia predicts mortality in diabetes. *Diabetes care*, 35(9):1814–1816, 2012.

[21] Nuha A. ElSayed, Grazia Aleppo, Vanita R. Aroda, Raveendhara R. Bannuru, Florence M. Brown, Dennis Bruemmer, Billy S. Collins, Marisa E. Hilliard, Diana Isaacs, Eric L. Johnson, Scott Kahan, Kamlesh Khunti, Jose Leon, Sarah K. Lyons, Mary Lou Perry, Priya Prahalad, Richard E. Pratley, Jane Jeffrie Seley, Robert C. Stanton, and on behalf of the American Diabetes Association Gabbay, Robert A. 6. glycemic targets: Standards of care in diabetes—2023. *Diabetes Care*, 46:S97–S110, 2022.

[22] Gene F Franklin, J David Powell, Abbas Emami-Naeini, and J David Powell. *Feedback control of dynamic systems*, volume 4. Prentice hall Upper Saddle River, 2002.

[23] Martin Fränzle and Christian Herde. Hysat: An efficient proof engine for bounded model checking of hybrid systems. *Formal Methods in System Design*, 30:179–198, 2007.

[24] Nathan Fulton, Stefan Mitsch, Jan-David Quesel, Marcus Völp, and André Platzer. Keymaera x: An axiomatic tactical theorem prover for hybrid systems. In *Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25*, pages 527–538. Springer, 2015.

[25] Gabriel A Gregory, Thomas I G Robinson, Sarah E Linklater, Fei Wang, Stephen Colagiuri, Carine de Beaufort, Kim C Donaghue, Jessica L Harding, Pandora L Wander, Xinge Zhang, Xia Li, Suvi Karuranga, Hongzhi Chen, Hong Sun, Yuting Xie, Richard Oram, Dianna J Magliano, Zhiguang Zhou, Alicia J Jenkins, Ronald CW Ma, Dianna J Magliano, Jayanthi Maniam, Trevor J Orchard, Priyanka Rai, and Graham D Ogle. Global incidence, prevalence, and mortality of type 1 diabetes in 2021 with projection to 2040: a modelling study. *The Lancet Diabetes & Endocrinology*, 10(10):741–760, 2022.

[26] Chris Grier, Shuo Tang, and Samuel T King. Secure web browsing with the op web browser. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 402–416. IEEE, 2008.

[27] Chris Grier, Shuo Tang, and Samuel T. King. Secure web browsing with the op web browser. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 402–416, 2008.

[28] Chris Grier, Shuo Tang, and Samuel T. King. Designing and implementing the op and op2 web browsers. *ACM Trans. Web*, 5(2), may 2011.

[29] Daniel Halperin, Thomas S Heydt-Benjamin, Kevin Fu, Tadayoshi Kohno, and William H Maisel. Security and privacy for implantable medical devices. *IEEE pervasive computing*, 7(1):30–39, 2008.

[30] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ-kernel-based systems. *ACM SIGOPS Operating Systems Review*, 31(5):66–77, 1997.

[31] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Fault isolation for device drivers. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 33–42, 2009.

[32] Irl B Hirsch, Dana Armstrong, Richard M Bergenstal, Bruce Buckingham, Belinda P Childs, William L Clarke, Anne Peters, and Howard Wolpert. Clinical application of emerging sensor technologies in diabetes management: consensus guidelines for continuous glucose monitoring (cgm). *Diabetes technology & therapeutics*, 10(4):232–246, 2008.

[33] Peter G. Jacobs, Pau Herrero, Andrea Facchinetti, Josep Vehi, Boris Kovatchev, Marc D. Breton, Ali Cinar, Konstantina S. Nikita, Francis J. Doyle, Jorge Bondia, Tadej Battelino, Jessica R. Castle, Konstantia Zarkogianni, Rahul Narayan, and Clara Mosquera-Lopez. Artificial intelligence and machine learning for improving glycemic control in diabetes: Best practices, pitfalls, and opportunities. *IEEE Reviews in Biomedical Engineering*, 17:19–41, 2024.

[34] Peter G. Jacobs, Pau Herrero, Andrea Facchinetti, Josep Vehi, Boris Kovatchev, Marc D. Breton, Ali Cinar, Konstantina S. Nikita, Francis J. Doyle, Jorge Bondia, Tadej Battelino, Jessica R. Castle, Konstantia Zarkogianni, Rahul Narayan, and Clara Mosquera-Lopez. Artificial intelligence and machine learning for improving glycemic control in diabetes: Best practices, pitfalls, and opportunities. *IEEE Reviews in Biomedical Engineering*, 17:19–41, 2024.

[35] Sumit K Jha, Edmund M Clarke, Christopher J Langmead, Axel Legay, André Platzer, and Paolo Zuliani. A bayesian approach to model checking biological systems. In *Computational Methods in Systems Biology: 7th International Conference, CMSB 2009, Bologna, Italy, August 31-September 1, 2009. Proceedings 7*, pages 218–234. Springer, 2009.

[36] Lauren G Kanapka, John W Lum, and Roy W Beck. Insulin pump infusion set failures associated with prolonged hyperglycemia: frequency and relationship to age and type of infusion set during 22,741 infusion set wears. *Diabetes technology & therapeutics*, 24(6):396–402, 2022.

[37] Boris P Kovatchev, Linda A Gonder-Frederick, Daniel J Cox, and William L Clarke. Evaluating the accuracy of continuous glucose-monitoring sensors: continuous glucose–error grid analysis illustrated by therasense freestyle navigator data. *Diabetes Care*, 27(8):1922–1928, 2004.

[38] Rama Lakshman, Charlotte Boughton, and Roman Hovorka. The changing landscape of automated insulin delivery in the management of type 1 diabetes. *Endocrine Connections*, 12(8), 2023.

[39] Chunxiao Li, Anand Raghunathan, and Niraj K Jha. Hijacking an insulin pump: Security attacks and defenses for a diabetes therapy system. In *2011 IEEE 13th international conference on e-health networking, applications and services*, pages 150–156. IEEE, 2011.

[40] Loop. An automated insulin delivery app for ios, built on loopkit. https://github.com/LoopKit/Loop.

[41] LoopKit. Tools for building automated insulin delivery systems on ios. https://github.com/LoopKit.

[42] Katrin Lunze, Tarunraj Singh, Marian Walter, Mathias D Brendel, and Steffen Leonhardt. Blood glucose control algorithms for type 1 diabetic patients: A methodological review. *Biomedical signal processing and control*, 8(2):107–119, 2013.

[43] Haohui Mai, Edgar Pek, Hui Xue, Samuel Talmadge King, and Parthasarathy Madhusudan. Verifying security invariants in expressos. *SIGPLAN Not.*, 48(4):293–304, mar 2013.

[44] Eduard Marin, Dave Singelée, Bohan Yang, Ingrid Verbauwhede, and Bart Preneel. On the feasibility of cryptography for a wireless insulin pump system. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pages 113–120, 2016.

[45] Clara Mosquera-Lopez, Leah M Wilson, Joseph El Youssef, Wade Hilts, Joseph Leitschuh, Deborah Branigan, Virginia Gabo, Jae H Eom, Jessica R Castle, and Peter G Jacobs. Enabling fully automated insulin delivery through meal detection and size estimation using artificial intelligence. *npj Digital Medicine*, 6(1):39, 2023.

[46] David M Nathan and DCCT/Edic Research Group. The diabetes control and complications trial/epidemiology of diabetes interventions and complications study at 30 years: overview. *Diabetes care*, 37(1):9–16, 2014.

[47] Giulia Noaro, Giacomo Cappon, Martina Vettoretti, Giovanni Sparacino, Simone Del Favero, and Andrea Facchinetti. Machine-learning based model to improve insulin bolus calculation in type 1 diabetes therapy. *IEEE Transactions on Biomedical Engineering*, 68(1):247–255, 2021.

[48] OpenAPS. The open artificial pancreas system project. https://github.com/openaps.

[49] Abhinandan Panda, Srinivas Pinisetty, and Partha Roop. A secure insulin infusion system using verification monitors. In *Proceedings of the 19th ACM-IEEE International Conference on Formal Methods and Models for System Design*, pages 56–65, 2021.

[50] Nathanael Paul, Tadayoshi Kohno, and David C Klonoff. A review of the security of insulin pump infusion systems. *Journal of diabetes science and technology*, 5(6):1557–1562, 2011.

[51] Stephen W Ponder and Kevin Lee McMahon. *Sugar Surfing: How to manage Type 1 diabetes in a modern world*. MediSelf Press Sausalito, CA, 2015.

[52] Griselda Quiroz. The evolution of control algorithms in artificial pancreas: A historical perspective. *Annual Reviews in Control*, 48:222–232, 2019.

[53] Charles Reis, Alexander Moshchuk, and Nasko Oskov. Site isolation: Process separation for web sites within the browser. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1661–1678, 2019.

[54] Michael Rushanan, Aviel D Rubin, Denis Foo Kune, and Colleen M Swanson. Sok: Security and privacy in implantable medical devices and body area networks. In *2014 IEEE symposium on security and privacy*, pages 524–539. IEEE, 2014.

[55] Banshi D Saboo and Praful A Talaviya. Continuous subcutaneous insulin infusion: practical issues. *Indian journal of endocrinology and metabolism*, 16(Suppl 2):S259, 2012.

[56] Gary Scheiner. *Think like a pancreas: A Practical guide to managing diabetes with insulin*. Hachette Go, 2020.

[57] Shuo Tang, Haohui Mai, and Samuel T King. Trust and protection in the illinois browser operating system. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[58] N. S. Tyler, C. M. Mosquera-Lopez, L. M. Wilson, and et al. An artificial intelligence decision support system for the management of type 1 diabetes. *Nat Metab*, 2:612–619, 2020.

[59] Josep Vehí, Iván Contreras, Silvia Oviedo, Lyvia Biagi, and Arthur Bertachi. Prediction and prevention of hypoglycaemic events in type-1 diabetic patients using machine learning. *Health Informatics Journal*, 26(1):703–718, 2020. PMID: 31195880.

[60] Roberto Visentin, Michele Schiavon, Rita Basu, Ananda Basu, Chiara Dalla Man, and Claudio Cobelli. Chapter 6 - physiological models for artificial pancreas development. In Ricardo S. Sánchez-Peña and Daniel R. Cherñavvsky, editors, *The Artificial Pancreas*, pages 123–152. Academic Press, 2019.

[61] Helen J Wang, Chris Grier, Alexander Moshchuk, Samuel T King, Piali Choudhury, and Herman Venter. The multi-principal os construction of the gazelle web browser. In *USENIX security symposium*, volume 28, 2009.

[62] Dan Williams, Patrick Reynolds, Kevin Walsh, Emin Gün Sirer, and Fred B Schneider. Device driver safety through a reference validation mechanism. In *OSDI*, volume 8, pages 241–254, 2008.

[63] Meng Zhang, Kevin B. Flores, and Hien T. Tran. Deep learning and regression approaches to forecasting blood glucose levels for type 1 diabetes. *Biomedical Signal Processing and Control*, 69:102923, 2021.

[64] T. Zhu, C. Uduku, K. Li, and et al. Enhancing self-management in type 1 diabetes with wearables and deep learning. *npj Digital Medicine*, 5:78, 2022.

[65] Taiyu Zhu, Kezhi Li, Pau Herrero, and Pantelis Georgiou. Basal glucose control in type 1 diabetes using deep reinforcement learning: An in silico validation. *IEEE Journal of Biomedical and Health Informatics*, 25(4):1223–1232, 2021.

[66] Taiyu Zhu, Chukwuma Uduku, Kezhi Li, Pau Herrero, Nick Oliver, and Pantelis Georgiou. Enhancing self-management in type 1 diabetes with wearables and deep learning. *npj Digital Medicine*, 5(1):78, 2022.

Appendix

Appendix A.
Protocol for real world use

One individual, who we will call Bob, used GlucOS for 2.5 months. We put in place a protocol where we would stop our study and ask Bob to cease using our system and revert back to his previous insulin regime. In our protocol, we have formal in-person meetings with Bob every week to check his data and make sure that he is not incurring risk as a result of using GlucOS. We would have stopped our study if any of the following conditions occurred:

- *Excessive hypoglycemia.* We define excessive hypoglycemia as having spent 5% or more of time during a week with CGM readings below 70 mg/dl or more than 1% below 54 mg/dl. These amounts of time spent in hypoglycemia would have represented a significant increase in time spent in hypoglycemia for Bob.
- *Insufficient CGM data.* If Bob would have had less than 70% of the time during a week with CGM

Message name	Message type	Message description
setupPod	Setup	Setup a new pump
assignAddress	Setup	Pair pump with phone
deactivatePod	Setup	Teardown pump
acknowledgeAlert	UX	Acknowledge alert from pump
configureAlerts	UX	Configure alerts from pump
beepConfig	UX	Configure pump beeps

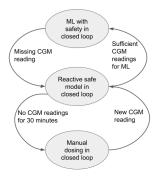Figure 17: Types of messages exchanged between a pump driver and pump pertaining to setup/UX.

Figure 18: GlucOS's CGM state machine to handle missing readings from CGMs.

readings, then the lack of CGM data would represent a decrease for Bob.
- *Insufficient closed-loop runs.* We designed GlucOS to run closed-loop algorithms every five minutes, or 288 times per day. If Bob had a day where our closed-loop algorithm ran successfully less than 200 times, it would represent a fundamental flaw in the system.
- *Serious complications due to diabetes.* Serious complications include any hospitalization for diabetes related issues, severe hypoglycemia where Bob was unable to recover from hypoglycemia himself and had to get help from someone else, diabetic ketoacidosis, or seizure.
- *New medical diagnosis requiring significant attention.* If Bob had received a new medical diagnosis during the study, the study would have had the potential to be a distraction for Bob.
- *New mental health diagnosis.* If Bob were struggling with mental health during the study, the study would have the potential to be a distraction.

Appendix B.
List of pump setup and UX communication messages

We list insulin message types pertaining to pump setup and UX in Figure 17.

Appendix C.
CGM state machine

Inability to retrieve readings from a CGM also count as runtime failures in addition to those discussed in Section 7. However, these failures are easier to handle. In this section, we describe the CGM state machine that handles such failures. We note that transitions from the state machine described in Section 7 supersede state transitions from the CGM state machine.

Since ML needs a stream of CGM readings as input, we transition from ML to the reactive safe model if we don't receive an input from the CGM. As long as the number of missing CGM readings is low (say, within 30 minutes), we can interpolate to account for the missing readings to operate the reactive safe model. Once we have too many missing CGM readings that make us lose confidence in interpolation, we transition to manual dosing. From manual dosing, we transition back to the reactive safe model on receiving a CGM reading. Lastly, from the reactive safe model, we transition back to ML once we have enough CGM readings that are required by the ML model. Figure 18 summarizes these transitions.

Appendix D.
Formal verification: invariants and bugs

In this section, we list the invariants used to formally verify critical functions of GlucOS. Figure 19 lists the bugs we found and fixed during formal verification.

D.1. Invariants

The MicroBolusing function ensures that micro boluses are always administered at safe time intervals while making sure that they are actually needed (the glucose level of the person is in fact high).

- Ensure no micro bolus in the last 4.2 minutes.
- Ensure glucose is at least 20 mg/dl above the target glucose level.
- Ensure predicted glucose is greater than the current glucose level minus 2.
- Ensure micro bolus amount is within the allowed range.
- Ensure bolus amount is rounded correctly if *pumpManager* is available.

We assume that the *BioKernel* is the only software issuing bolus commands.

The GuardRails function is responsible for ensuring a safe basal rate value, mindful of the user's glucose levels. It enforces several invariants:

- Ensure the result of rounding *newBasalRateRaw* to the supported basal rate is within the allowable range.
- Ensure the new basal rate does not exceed the maximum basal rate in the settings.

- Ensure the new basal rate is not negative.
- Ensure that if either the current glucose level or the predicted glucose level falls below or equals the shut-off glucose threshold, the new basal rate is set to 0.0.

The function operates under the assumption that all paths must go through the guard rails, with a specific evaluation comparing with the pump invariant. Additionally, it assumes the integrity of settings, trusting the underlying OS and file system and storage stack.

The insulinDeliveredForSegment function ensures that *intersectionStart* is not greater than *intersectionEnd* when computing the intersection between *self.startDate* and *self.endDate* with *startDate* and *endDate*.

- Verifies that *intersectionDuration* is non-negative and does not exceed *doseDuration*, ensuring it accurately reflects the overlapping time duration.
- Checks that the units of insulin (*units*) used for calculation are either *deliveredUnits* or *programmedUnits*, ensuring consistency in how the insulin amount is derived.
- Verifies that the computation (*units * intersectionDuration / doseDuration*) accurately represents the amount of insulin delivered during the intersection period.

The function assumes that the input parameters (*self.startDate*, *self.endDate*, *startDate*, *endDate*, *units*, and *doseDuration*) are valid and within the expected ranges. It is also assumed that the underlying data structures and calculations are accurate and consistent throughout the system.

The CreateBasalDose function requires a time gap greater than 1 second to proceed and return a valid *DoseEntry*.

- Ensures that the calculation of *basalRatePerSecond* from *basalRate* accurately represents the rate of insulin delivery per second.
- Ensures that the calculation of *unitsDelivered* accurately reflects the amount of insulin delivered over the time gap.
- The *DoseEntry* constructed should have consistent attributes and adhere to the specified type, units (*unitsPerHour*), and mutability (*false*).

The function assumes that the input parameters (*basalRate*, *startDate*, and *endDate*) are valid and within the expected ranges. It also assumes that the underlying time calculations and conversions between units are accurate and consistent.

The inferBasalDoses function ensures that *basalDoses* contains only *DoseEntry* instances where *type* is *.tempBasal*, *.resume*, or *.suspend*, sorted in ascending order by *startDate*.

- InsulinType should be determined correctly based on the last *.tempBasal* or *.bolus* type dose in *doses*, defaulting to *.humalog* if none are found.
- Each inferred basal dose added to *inferredBasalDoses* must be created using the *createBasalDose*

Bug Type	Description
Division by Zero Error	Potential division by zero error when calculating insulin dosage if the user entered the same start and end time for a basal rate, resulting in a duration of zero.
Negative Value Handling	The insulin dosing calculation for microbolusing and basal rate erroneously allowed negative values.
Out-of-Range Value Handling	The microbolusing function failed to handle cases where one of its calculation components was out of the expected range or resulted in a negative value.
Violation of Intended Postcondition	The function responsible for setting the insulin delivery duration violated its intended postcondition by allowing the end date to be earlier than the start date.
Violation of Intended Postcondition	The function responsible for calculating the total insulin dose violated its intended postcondition by returning an incorrect upper bound value under certain conditions.
Violation of Intended Postcondition	The function responsible for updating the user's insulin dosage violated its intended postcondition by failing to validate the input parameters correctly.
Violation of Intended Postcondition	The function responsible for calculation of 'unitsDelivered' needed additional postconditions accurately reflect the amount of insulin delivered over the time gap.
Lack of Error Handling	Additional error handling was added to insulin delivered over a time segment function to gracefully catch undesirable state.
Lack of Error Handling	Additional error handling was added to microbolusing function to gracefully catch undesirable states.

Figure 19: Report of bugs fixed by formally verifying GlucOS.

function with valid parameters and added in the correct chronological order.

- If the last dose in *basalDoses* is not of type *.suspend*, a valid basal dose must be inferred from its *endDate* to *at*.

The function assumes that the input parameters (*doses*, *basalDoses*, *at*, and *pumpRecordsBasalProfileStartEvents*) are valid and within the expected ranges. It also assumes the correctness of the underlying data structures and the *createBasalDose* function.

The insulinOnBoard function ensures that *doses* contains unique and valid *DoseEntry* instances filtered up to *at*.

- The total *iob* should accurately represent the cumulative insulin on board from all valid dose entries in *doses*.
- *basalDoses* should contain inferred basal doses that accurately reflect insulin on board contributions when *pumpRecordsBasalProfileStartEvents* is false.
- The final return value of *insulinOnBoard* should be the sum of *iob* and *basalIob*, representing the total insulin on board at *at*.

The function assumes that the input parameters (*doses*, *at*, and *pumpRecordsBasalProfileStartEvents*) are valid and within the expected ranges. It also assumes the correctness of the underlying data structures and the *inferBasalDoses* function.

The ActiveBolus function ensures that data is read consistently from disk before proceeding with any operations.

- Verifies that *DeduplicatedDoses(events: eventLog, at: at)* correctly filters out duplicates and retains only relevant dose entries up to *at*.
- Ensures that doses.filter accurately selects bolus entries that were active at the specified *at*.
- Verifies that *doses.last* correctly returns the last active bolus entry, ensuring it is non-null if a bolus exists within the specified criteria.

The function assumes that the input parameters (*eventLog* and *at*) are valid and within the expected ranges. It also assumes the correctness of the underlying data structures and the *DeduplicatedDoses* function, as well as the consistency and reliability of disk read operations.

The TempBasal (Safety Clamps) function ensures that the events array includes only events from safetyStates that fall within the time range *[start - duration, at)*.

- Verifies that the historicalMlInsulin value represents the total units of insulin delivered by the machine learning system over the specified time horizon.
- Ensures that the machine learning and safety temp basal rates are correctly converted to units of insulin based on the given duration.
- Ensures that the upper and lower bounds for insulin units are correctly adjusted based on historicalMlInsulin.

- Ensures that the difference between machine learning and safety temp basal units is correctly clamped within the calculated bounds.
- The clamped delta units should be converted back to a temp basal rate and added to the safety temp basal rate.
- The returned *SafetyTempBasal* object should correctly encapsulate the adjusted temp basal rate and historical machine learning insulin.

The function assumes that the input parameters (*safetyStates*, *start*, *duration*, *at*, *mlTempBasal*, *safetyTempBasal*, and *lastHistoricalMlInsulin*) are valid and within the expected ranges. It also assumes the correctness of the underlying data structures and the conversion functions between units, as well as the proper configuration and appropriateness of the safety clamp parameters and thresholds for the user.