Protecting Chatbots from Toxic Content

Guillaume Baudart
IBM Research
USA

Julian Dolby
IBM Research
USA

Evelyn Duesterwald
IBM Research
USA

Martin Hirzel
IBM Research
USA

Avraham Shinnar
IBM Research
USA

Abstract

There is a paradigm shift in web-based services towards conversational user interfaces. Companies increasingly offer conversational interfaces, or chatbots, to let their customers or employees interact with their services in a more flexible and mobile manner. Unfortunately, this new paradigm faces a major problem, namely toxic content in user inputs. Toxic content in user inputs to chatbots may cause privacy concerns, may be adversarial or malicious, and can cause the chatbot provider substantial economic, reputational, or legal harm. We address this problem with an interdisciplinary approach, drawing upon programming languages, cloud computing, and other disciplines to build protections for chatbots. Our solution, called BotShield, is non-intrusive in that it does not require changes to existing chatbots or underlying conversational platforms. This paper introduces novel security mechanisms, articulates their security guarantees, and illustrates them via case studies.

CCS Concepts • **Security and privacy** → **Privacy protections**; • **Software and its engineering** → *Domain specific languages*;

Keywords Chatbot, Homomorphic Redaction, Context Digression

ACM Reference Format:
Guillaume Baudart, Julian Dolby, Evelyn Duesterwald, Martin Hirzel, and Avraham Shinnar. 2018. Protecting Chatbots from Toxic Content. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '18), November 7–8, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3276954.3276958

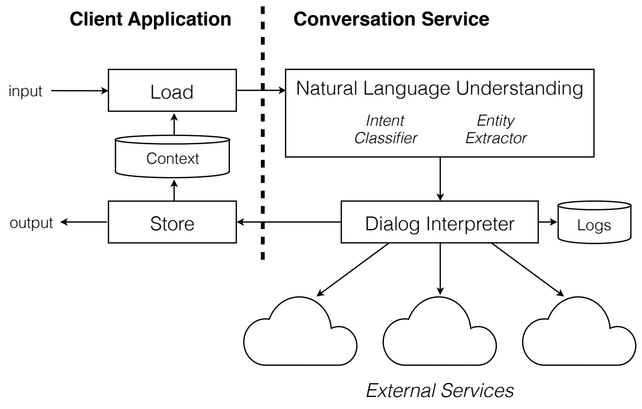

Figure 1. Architecture of a chatbot application. The chatbot client passes user input utterances and the context containing the information required to resume the conversation to the conversation service for processing. The natural language understanding (NLU) component processes the input utterance to extract user intent and entities. Those are used by the dialog interpreter to orchestrate conversational flow and application business logic. The business logic is served by external services.

1 Introduction

Web-hosted services are experiencing a paradigm shift towards conversational interfaces. Companies increasingly deploy virtual conversational agents (i.e., *chatbots*) as the interface between their services and their customers and employees [Vaziri et al. 2017]. Chatbots have the advantage of being delivered not just through web browsers, but also through text-based messaging software or even as skills on voice-enabled devices. Chatbots are built as cloud-hosted artificial intelligence (AI) applications. To make them easier to deploy, scale, and update, chatbots usually use a micro-services architecture, for instance, based on stateless cloud functions. A primary AI component of chatbots is their natural-language understanding (NLU) module, but additional AI components may be included in the response generation. Figure 1 shows the typical software architecture of a chatbot application built using commercially available chatbot platforms [Facebook 2011; IBM 2016b; Microsoft 2015]. AI components such as NLU are best-effort and data-hungry: they face accuracy limitations, which the provider continually addresses by collecting and mining production logs.

Unfortunately, the new paradigm of conversational interfaces suffers from a major problem, namely *toxic content* in user input. Toxic content includes personally identifiable information (PII) that should be kept out of logs and untrusted services for privacy reasons [Krishnamurthy and Willis 2009; McCallister et al. 2010]. Toxic content also includes malicious user utterances aimed at exploiting the chatbot which can cause data confidentiality breaches or theft of intellectual property [Fredrikson et al. 2014; Tramèr et al. 2016]. More generally, toxic content includes anything that harms the service provider, including abusive customers, legal and regulatory issues, or reputation damage [Wikipedia 2016]. Mitigating these risks is a broadly-recognized but as-yet unsolved issue. For instance, the 2004 VoiceXML standard (a W3C standard chatbot programming model) declares "A future version of VoiceXML may specify criteria by which a VoiceXML Processor safeguards the privacy of personal data" [McGlashan et al. 2004]. But a 2010 special report by the National Institute of Standards and Technology [McCallister et al. 2010] still lists "unwanted record of private information" as a threat vector for personal assistants with chatbot interfaces. Indeed, even in 2018, chatbot platforms lack reusable PII solutions; instead, public chatbot deployments either ignore the issue or, at best, address it one chatbot at a time.

This paper presents our interdisciplinary research into solving the problem of toxic content for conversational interfaces to web-hosted services. Our research draws upon ideas from programming languages, cloud computing, artificial intelligence, and privacy and security research. In contrast to traditional security risks, the toxic content problem arises at the conversation level, that is, only when examining the semantic contents of user utterances. Security solutions that operate at lower levels, e.g., network-level denial-of-service defenses [Zargar et al. 2013] or system-level buffer overflow defenses [Cowan et al. 2000], offer little help. We designed a solution called *BotShield*, representing a first step towards conversation-level security. At this point, BotShield includes the following defense mechanisms to protect chatbots from toxic content:

- Homomorphic redaction to address private content, and
- Context digression to address adversarial and malicious content.

Homomorphic redaction is a mechanism for safe handling of PII in user utterances, such as the credit card number in Line 12 of Figure 2. Conventional redaction replaces the private content with a redacted string: `XXXX-XXXX-XXXX-7807`. But that may prevent the chatbot itself from working correctly (understanding that a credit card number was entered) and the external services from performing their function (processing the credit card number). Therefore, we introduce homomorphic redaction: privacy of the original utterances

```
1   C: Hello!
2      I can help you with the Computer Store.
3      What would you like to do?
4   H: Order a personal apple laptop with a
5      13 inches screen.
6   C: I have a apple laptop with a 13 inch
7      screen for personal use.
8      Would you like me to place the order?
9   H: Yes
10  C: What credit card number should we use
11     to place this order?
12  H: Use my VISA 4556-6412-5875-7807
13  C: I placed your order using your VISA.
14     Your order number is 4556-011a9-7807.
15  H: Please list my past orders.
16  C: You previously ordered:
17     - personal hp 15in
18     - apple 13in for work
```

Figure 2. Example dialog between a human customer (H) and a computer store chatbot (C). In this example, the credit card information is processed without special handling, and may thus end up plainly visible in logs or leak to untrusted external services.

is preserved while still enabling limited computation over the redacted version. While there is previous work using a lattice of homomorphic encryption schemes for web-based services [Dong et al. 2016; Tetali et al. 2013], this paper is the first to offer homomorphic redaction for chatbots, enabling NLU over redacted data. Previous work on redaction for chatbots was not homomorphic, e.g. [Salesforce 2016].

Context digression is an intervention mechanism for temporarily diverting the conversational flow to a subdialog in response to a trigger. The trigger consists of detected toxic content or other security needs, and the subdialog implements proper handling of these security needs. When the digression completes, the original conversation resumes where it was interrupted. Context digression serves as a modular conversational escalation mechanism. We show how context digressions can be used to implement two different security applications, namely on-the-fly authentication and conversation escalation, for instance, when the user becomes frustrated. While there exist earlier chatbot programming models offering subdialogs [Lucas 2000], this paper is the first to make them non-intrusive and to isolate their environments. Previous work on context isolation did not focus on chatbots, e.g. [Berger et al. 2009; Siefers et al. 2010].

One option for implementing defense mechanisms would have been to modify the client-side code and the chatbot script by hand, perhaps augmenting the programming model to facilitate that. However, the by-hand approach would complicate the chatbot logic and burden its designer with considering all corner-cases. To avoid these issues, BotShield's security mechanisms are designed to be *non-intrusive*. The chatbot designer does not need to manually change the chatbot script to use BotShield. Instead, the chatbot designer

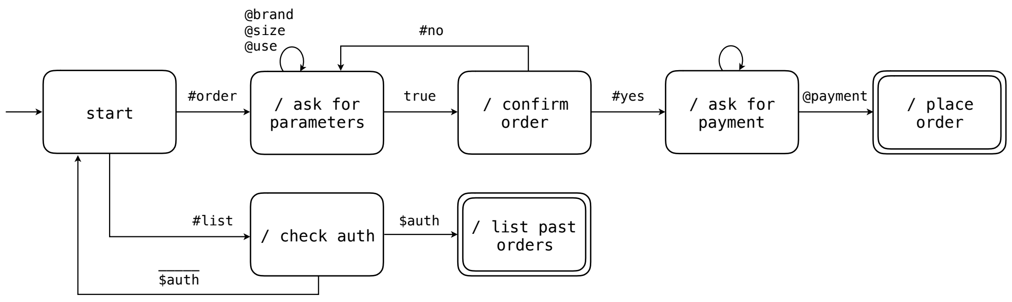

Figure 3. State machine of an example chatbot program with user intents indicated with a # prefix, entities with a @ prefix, and context variable with a $ prefix.

provides a few lines of configuration code to configure their homomorphic redaction and/or context digression defenses. Based on these configurations, BotShield instantiates a set of pre- and post-processors that implement the configured defense. In the case of homomorphic redaction, BotShield also transparently inserts secured entities into the chatbot script to enable the chatbot's NLU component to operate on redacted private information. In both cases (homomorphic redaction and context digression), BotShield offers separation of concerns [Kiczales et al. 2001; Tarr et al. 1999]: the chatbot designer can focus on the core chatbot logic, and the security concern is kept in separate configuration files. AI specialists should not have to become security experts, and conversely, security experts should not have to understand the logic of a particular chatbot to protect it.

We have developed a BotShield prototype for chatbots built using the IBM Watson Assistant [IBM 2016b], but the same concepts could be applied to other commercial chatbot technology offerings [Facebook 2011; Microsoft 2015; Patil et al. 2017].

In summary, this paper contributes the following:

- A homomorphic redaction scheme that protects PII while enabling interaction with a chatbot.
- A generic non-invasive context digression implementation for chatbots.
- Application of context digressions to implement security mechanisms.
- A novel architecture based on serverless computing.
- An instantiation of this architecture to protect a retail chatbot.

BotShield follows two overall design principles. First, on the security side, BotShield follows the principle of minimizing trust. Rather than trying to anticipate all possible attack vectors, BotShield reveals information to system components only on a need-to-know basis. Second, on the implementation side, BotShield follows the principle of separation of concerns. It handles chatbot security as a separate concern that it transparently embeds into conversations. With this separation, we can free chatbot designers from worrying about security risks and allow them to focus their efforts on the target conversational application aspects. We hope this paper will inspire more wide-spread adoption of these design principles in chatbots.

2 Chatbot Programming Model

A chatbot interacts with a user via a conversation, that is, in a series of text messages (utterances) between the two. Such systems are typically structured as a client-server application as illustrated in Figure 1. The application takes the user input and sends it to the server-side chatbot. The chatbot first processes the utterance with the natural language understanding (NLU) module, which derives intents (i.e. desired user actions) and entities (i.e. to what the user is referring). The chatbot then feeds these intents and entities into the dialog interpreter, which executes the dialog script authored by the chatbot designer. The dialog interpreter may call external services and generates the output, which is sent back to the user via the client.

To make this a dialog rather than a disconnected series of utterances, state can be kept in the conversation *context*; the dialog interpreter can read values from and write values to this state to keep track of information to be carried through the dialog. This context implements what linguists call common ground [Clark and Brennan 1991]. Cloud-hosted applications use stateless micro-services for scaling, fault tolerance, and security [Newman 2015], which is why Figure 1 keeps the state in the client.

The dialog interpreter from Figure 1 is typically scripted in some programming model [McTear 2002]. While there are high-level dialog programming models, e.g., frames [Bobrow et al. 1977], most of them can use finite state machines as a low-level intermediate representation. Figure 3 gives an example finite state machine for a simple shopping chatbot. Boxes are states, and transitions are predicated upon intents (indicated with a # prefix) or entities (indicated with a @ prefix) or context variable (indicated with a $ prefix). Execution initially starts in the `start` node. If the initial intent is `#order`, then the system goes to the next state, `ask for`

```
request = {
  "utterance": "Use my VISA 4556-6412-5875-7807",
  "context": {
    "bot_id": "computer store",
    "node": "ask for payment",
    "auth": false,
    "brand": "apple",
    "size": 13,
    "use": "personal" } }
```

Figure 4. Payment request.

```
response = {
  "utterance": Use my VISA 4556-6412-5875-7807",
  "output": "I placed your order using your VISA. Your
     order number is 4556-011a9-7807.",
  "context": {
    "bot_id": "computer store",
    "node": "place order",
    "auth": false,
    "brand": "apple",
    "size": 13,
    "use": "personal",
    "payment": "VISA 4556-6412-5875-7807",
    "order_number": 4556-011a9-7807 } }
```

Figure 5. Payment response.

parameters. This state looks for @brand, @size, and @use entities to specify what to order. After getting those entities, the system moves to the next state to confirm the order, and then finally place it.

More concretely, at each turn the client application sends a request to the conversational service comprising two components: the user utterance and the context. The context contains all the information required to resume the conversation, such as the current node in the state machine or all the information gathered so far. For instance, consider the payment request and response in Figure 2 Lines 12-14. Figure 4 shows the request sent to realize a payment in our computer store. The conversational agent then executes one step of the dialog logic. In our example, the NLU identifies the entity @payment in the utterance and proceeds to the place order node. The dialog interpreter then calls an external service to place the order which returns an order number. Finally, the response contains the output text and an updated context. For instance, Figure 5 shows the response of the computer store to the previous request.

3 Homomorphic Redaction

This section deals with the need for redaction: some parts of the user utterance may contain private information that the conversation service and unauthorized external services should not be able to access. Additionally, we would like to return a suitably redacted string to the application user interface to display to the user, giving them an easy way to understand what private information was redacted. We will

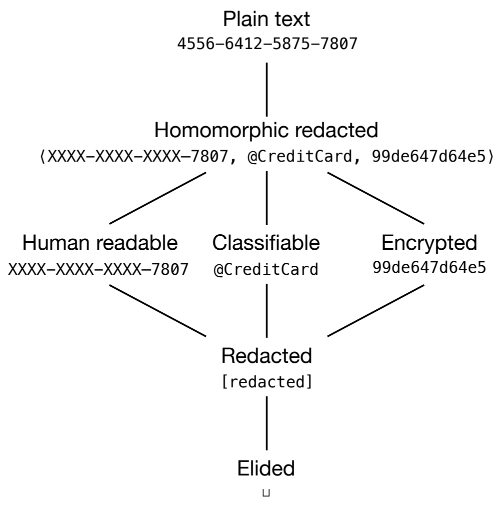

Figure 6. Redaction lattice.

arrange for the response from the conversation service to contain an optionally redacted version of the given input that can be used for this purpose.

This section explores various design possibilities, leading up to the final design, a non-intrusive redaction solution that still enables the NLU and authorized external services to function as intended.

3.1 Simple Redaction

To redact an utterance, we must first intercept it. This is done by interposing an appropriate redaction processor in between the application and the conversation service. This is mostly transparent, as the interposed processor can mimic the conversation service and pass on the redacted message as needed.

Such a redaction processor first locates potentially private information (e.g. credit cards numbers) in the utterance. At that point, there are many options to take. As a running example, we will use the utterance "Use my VISA 4556-6412-5875-7807" from Figure 2, which contains a credit card number. The lattice in Figure 6 depicts the information revealed by the various design choices and their relationship.

The most radical choice is to completely elide the private information from the utterance that it passes on to the conversation service. This would result in the string "Use my VISA". It does not give the conversation service a way to know for sure if and where private information was present. However, depending on the surrounding context, one may still be able guess with high confidence. Furthermore, if the application user interface displays the string with the private information elided, that may also confuse the human. Instead, we could *redact* the private information, replacing it with a standard string such as "Use my VISA [redacted]".

```
{ "1": {
    "human": "XXXX-XXXX-XXXX-7807",
    "class": "CreditCard",
    "encrypt": "99de647d64e5" } }
```

Figure 7. JSON representation of combined homomorphic redaction in the conversation context object.

3.2 Homomorphic Redaction

The simple redaction schemes protect the private data, but do not allow it to be used for further processing. This section introduces more nuanced schemes that preserve the ability to process the private data. These schemes are not fully homomorphic but perform redaction in a way that preserves specific operations.

One form of redaction, which we call *human readable* redaction, preserves the ability of the user to reason about what was redacted. This is done by replacing the private data with a human readable representation of the type of the private data. For instance, our running example would be redacted to "`Use my VISA XXXX-XXXX-XXXX-7807`", replacing the credit card with a standard representation of the redacted data.

Instead of improving the human readability of the redaction, we can instead improve how the conversation service perceives it. *Classifiable redaction* preserves the ability of the conversation service to extract and classify redacted entities. For instance, it could redact the running example to "`Use my VISA @CreditCard`". The NLU unit can classify the *secured entity* "`@CreditCard`" correctly without seeing the private information.

All these examples irretrievably redact the private information. Instead, *encrypted redaction* encrypts the private information, enabling authorized external services to later decrypt and act on it, while not allowing the conversation service access to it. In our running example, this would look like "`Use my VISA ENC:99de647d64e5`".

3.3 Combining Homomorphic Redactions

Rather than choosing among these homomorphic redaction schemes, we can combine them, giving us a redaction scheme with all their benefits. This can be done by redacting the private information with an identifier (enabling multiple redactions), such as "`Use my VISA $1`". The three above forms of redactions can then be sent alongside, in the conversation service context. Figure 7 shows a possible representation in JSON format.

This representation is also extensible, allowing further information to be included. For example, if we wanted to enable external services to compare (but not decrypt) the private data, we could extend the JSON object to include an order-preserving version of the encrypted data as another field.

```
C: Hello!
   I can help you with the Computer Store.
   What would you like to do?
H: Order a personal apple laptop with a
   13 inches screen.
C: I have a apple laptop with a 13 inch
   screen for personal use.
   Would you like me to place the order?
H: Yes
C: What credit card number should we use
   to place this order?
H: Use my VISA XXXX-XXXX-XXXX-7807
C: I placed your order using your VISA.
   Your order number is 4556-011a9-7807.
H: Please list my past orders.
C: You previously ordered:
   - a personal hp 15in
   - a apple 13in for work
```

Figure 8. Redacted dialog example.

3.4 Non-Intrusive Combinations

The previously-mentioned combined scheme allows all the required information to be sent, but requires the conversation service, the user interface, and the authorized external services to be changed. We can instead modify the scheme to make it non-intrusive, in the sense that no manual modifications of either conversation service, user interface, or client application are required.

Going back to our example of classifiable redaction, "`Use my VISA @CreditCard`", we extend it slightly to encode the unique identifier, as "`Use my VISA @CreditCard[1]`", which can be ignored by the conversation service.

As in the combined scheme, we can pass the other redaction forms in the context, associated with the unique identifier. We then add additional processors in between the conversation service and the original application response, as well as the authorized external service. These each use the associated data to replace the redacted "`Use my VISA @CreditCard[1]`" with the appropriate redaction for that target: "`Use my VISA XXXX-XXXX-XXXX-7807`" for the application user interface, and "`Use my VISA 4556-6412-5875-7807`", the decrypted version of the utterance, for properly authorized external application services. Unauthorized external services will not have this replacement performed, so they will not be able to see the unencrypted private data. Section 5 presents an implementation of this final scheme.

This solution protects the private data from the conversation service and unauthorized external services and requires no manual changes to the conversation service. The application user interface receives a human-friendly redacted version of the utterance, suitable for display. Finally, authorized external services receive the unencrypted utterance, necessitating no modification.

Figure 8 displays what the final conversation from Figure 2 might look like, assuming that the application user

Table 1. Who sees what, focusing on the "4556-6412-5875-7807" part of our running example, "Use my VISA 4556-6412-5875-7807". For the combined and non-intrusive schemes, the conversation service receives the additional context shown in Figure 7. This does not give it a way to view the credit card number.

		External Services		
	Conversation Service	Trusted	Untrusted	Output
Elided	␣	␣	␣	␣
Redacted	`[redacted]`	`[redacted]`	`[redacted]`	`[redacted]`
Human readable	`xxxx-xxxx-xxxx-7807`	`xxxx-xxxx-xxxx-7807`	`xxxx-xxxx-xxxx-7807`	`xxxx-xxxx-xxxx-7807`
Classifiable	`@CreditCard`	`@CreditCard`	`@CreditCard`	`@CreditCard`
Encrypted	`99de647d64e5`	`99de647d64e5`	`99de647d64e5`	`99de647d64e5`
Combined	`$1` + [Fig 7]	`$1` + [Fig 7]	`$1` + [Fig 7]	`$1` + [Fig 7]
Non-intrusive	`@CreditCard[1]` + [Fig 7]	`4556-6412-5875-7807`	`@CreditCard[1]` + [Fig 7]	`xxxx-xxxx-xxxx-7807`

interface supports modifying the displayed conversation using the returned redaction string. While the credit card itself is redacted to be human readable, the conversation is able to proceed, indicating that the conversation service is able to classify the utterance correctly. Additionally, the displayed order number (for pedagogical purposes) leaks the first digits of the credit card number, demonstrating that the external service that places the order has access to the full credit card number. The conversation service, on the other hand, is not able to obtain the full credit card number.

Table 1 articulates the security guarantees by providing an overview of what happens to the credit card number in the example under the different schemes presented, as seen by the different parts of the architecture. For the combined and the final, non-intrusive combined scheme, it includes information about what is sent in the context to each part.

4 Context Digression

Context digressions trigger a new conversation in the middle of a dialog. This mechanism makes chatbots modular and reusable. This section presents a lightweight solution to implement context digressions on top of an off-the-shelf conversational service. We illustrate the expressiveness of this mechanism with two security examples: on-the-fly authentication and sentiment-based escalation.

4.1 Context and Continuation

As discussed in Section 2 the requests sent to the conversational service contain two pieces of information: the user utterance and the context that contains the identity of the chatbot including all the information required to resume the conversation between two turns.

To trigger a digression, we only need to substitute the context with a fresh one containing the identity of the chatbot that is responsible for the digression dialog. The only difficulty is to save the current context before the substitution to be able to resume the conversation when the digression ends. This could simply be done by the client application but would require the designer to adapt the application. Instead, we propose to store the current context as a special element of the fresh context: the *continuation*. Since the client application already manages the context between two conversation turns, it will transparently handle the continuation as part of the context. Then, when the digression ends, the only step required to resume the parent conversation is to substitute the current context with the continuation.

To implement a digression, we need to wrap the call to the conversational service with two processors. Preceding the call, a first processor `digress` monitors the digression condition and substitutes the context with a fresh one (with the current context as a continuation) when the condition is satisfied. Subsequent to the call, another processor `resume` substitutes the context of the response with the continuation when the digression ends. Assuming that the end of a dialog is signaled by the presence of a special element `resume` in the context, in JavaScript we write:

```javascript
function digress(request) {
  if (condition(request)) {
    request.context = {
      bot_id: digression_bot,
      continuation: request.context
    }
  }
  return request
}

function resume(response) {
  if (response.context.resume) {
    response.context =
      response.context.continuation
  }
  return response
}
```

The only change required in the client application is to substitute the call to the conversational agent with a call to a wrapper function that mimics the conversation service and executes `digress` and `resume` before and after the conversational agent. Both processors are lightweight and only

```
1  C: Hello!
2     I can help you with the Computer Store.
3     What would you like to do?
4  H: Please list my past orders.
5  C: You must sign in to see your past orders.
6     Do you want to sign in?
7  H: Yes
8  C: Please enter your name and client id.
9  H: My name is Alice and my key is
10    [client id]
11 C: Thanks. You are now authenticated.
12    You previously ordered:
13    - a personal hp 15in
14    - a apple 13in for work
```

Figure 9. On-the-fly authentication example.

```
1  C: Hello!
2     I can help you with the Computer Store.
3     What would you like to do?
4  H: Order a 13-inch personal mac
5  C: What brand would you like?
6  H: Mac
7  C: What brand would you like?
8  H: Macintosh
9  C: What brand would you like?
10 H: I hate these stupid useless chatbots!
11 C: You sound upset, would you like to
12    continue this conversation?
13 H: Yes
14 C: Ok, glad to hear it.
15    Please restate your request calmly.
```

Figure 10. Sentiment-based escalation example.

reorganize the context of the request when a simple condition is satisfied. They impose negligible overhead and require no additional resources to store the continuation.

The `resume` processor is generic and does not depend on the digression (as long as all dialogs end with a context element `resume`). However, the programmer still needs to add one pre-processor for each digression. Section 5 presents an implementation that allows doing that in a modular way without modifying the client application or the conversational service.

We now show how the context digression mechanism can be used to implement two security features: on-the-fly authentication and sentiment-based escalation.

4.2 On-the-fly Authentication

Consider the dialog example of Figure 9 based on our computer store chatbot. The user wants to access the history of her past orders, which requires her to authenticate (Lines 5–11). The authentication is implemented as a digression and thus seamlessly integrated into the conversation without modifying the original computer store chatbot. This digression is an example of a subdialog that should be triggered whenever the conversation reaches a particular node of the dialog state machine (see Figure 3). The JavaScript code is the following:

```
function auth(request) {
  let node = request.context.node
  if (node == 'check auth' && not request.context.auth){
    request.context = {
      bot_id: authenticate_bot,
      continuation: request.context
    }
  }
  return request
}
```

When the conversation reaches the node '`check auth`', if the user is not yet authenticated (that is, if the context variable `context.auth` is `false`) we launch the chatbot `authenticate_bot`. When the digression returns, the conversation either follows the `$auth` transition or the $\overline{\texttt{\$auth}}$ transition in Figure 3 depending on whether or not the user successfully authenticated.

4.3 Sentiment-based Escalation

Sentiment-based escalation monitors the tone of the user utterances and escalates the conversation whenever the tone gets too angry or disgusted. Figure 10 shows an example where the chatbot does not understand the user answer and keeps asking the same question. The upset user finally gives up and starts complaining (Line 10), which triggers a digression to avoid further derailment of the chatbot (Lines 11–14).

To monitor the tone of user utterances we use a tone-analyzer service [IBM 2016c]. For each utterance, the tone analyzer returns a score, between 0 and 1, for five possible emotions: anger, disgust, fear, joy, and sadness.

If we detect that the level of anger or disgust exceeds a certain threshold ("a score greater than 0.75 indicates a high likelihood that the tone is perceived in the content" [IBM 2016c]), we trigger the escalation subdialog to ask if the user really wants to continue the conversation. The JavaScript code of the monitor function is the following:

```
function escalate(request) {
  let emotions = tone_analyzer(request.utterance)
  if (emotions.anger > 0.75 || emotions.disgust > 0.75) {
    request.context = {
      bot_id: escalate_bot,
      continuation: request.context
    }
  }
  return request
}
```

Compared to on-the-fly authentication, sentiment-based escalation is an example of digression triggered by a condition external to the dialog logic. In other words, this digression can be triggered at any point during the conversation and in any node of the state machine of Figure 3. It gets triggered like an exception, but when the digression returns, the conversation resumes where it left off.

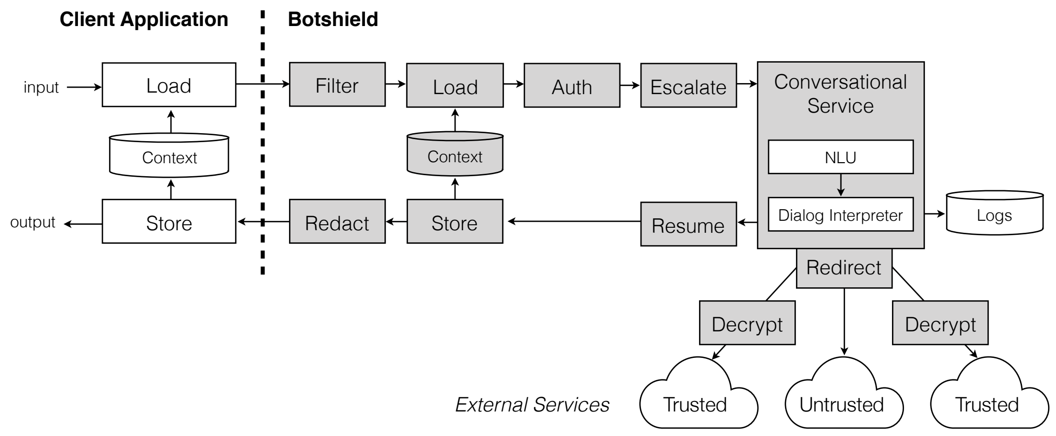

Figure 11. BotShield deployment of the protected computer store.

5 Non-Intrusive Implementation

This section presents the BotShield framework that can be used to deploy homomorphic redaction (Section 3) and context digressions (Section 4) to protect an existing chatbot. BotShield is designed to be non-intrusive in the sense that no manual changes to the original chatbot are necessary; instead, required modifications are transparently inserted based on a configuration.

We focus on chatbots built with a conversational service [Facebook 2011; IBM 2016b; Microsoft 2015; Patil et al. 2017] that is accessed, for instance, via a REST web API. BotShield provides an alternative web end-point to access the chatbot that channels all requests through a *pipeline* of pre- and post-processors to intercept requests and responses and secure their contents.

BotShield is implemented as a *serverless* application [Baldini et al. 2017]. Serverless computing, or *function-as-a-service*, allows programmers to run short-term stateless functions in the cloud triggered by events (in our case, new messages from the chatbot user). Major cloud providers now offer serverless platforms [Amazon 2014; IBM 2016a; Microsoft 2016]. The platform ensures timely, secure, fault-tolerant, and scalable execution of functions, leaving only the application design to the programmer. By implementing BotShield as a set of stateless functions in continuation passing style, we get these benefits of cloud functions for free.

Modifying the chatbot by hand to incorporate our protections would be tedious with the current commercially available chatbot programming models. At the same time, compared to a classic approach where the security mechanism would be directly implemented in the client application, BotShield is more modular, more scalable, and more secure. The presented mechanisms are generic solutions that can be used and combined for any chatbot. We can rely on the serverless platform to scale the resources required by BotShield with respect to the traffic going through the chatbot. The cloud implementation adds an extra level of security: BotShield processors remain active even if the client application is compromised.

5.1 Deployment and Configuration

Given a working chatbot, BotShield proceeds in two steps to deploy a protection: 1) Upload a set of processors on the serverless platform and define a *pipeline*, that is, one additional function that triggers in sequence the BotShield processors and the call to the chatbot. 2) If necessary, automatically update the chatbot based on simple configuration files. We then create a new web end-point that can be used to execute the pipeline. To switch to the protected version of the chatbot, the client application can use the newly-created pipeline end-point instead of the unprotected chatbot end-point provided by the conversational service.

We designed BotShield to be as modular as possible. The basic BotShield pipeline, shown in Figure 11, contains predefined protections (e.g., PII protection), but can be extended with ad-hoc processors (e.g., on-the-fly authentication, or sentiment-based escalation). The deployment is incremental. Given a working BotShield pipeline, it is possible to upload a new processor in the serverless platform and add it to the pipeline without redeploying the other processors.

Figure 11 presents the architecture of the BotShield framework deployed for the computer store with PII protection (Section 3), context digressions for on-the-fly authentication, and sentiment-based escalation (Section 4). Grey boxes correspond to serverless functions. Compared to Figure 1 we substituted the direct call to the conversational service with a call to the BotShield pipeline. At each new user utterance, an execution step proceeds as follows:

1. filter the text to redact private information,
2. check if authentication is required,
3. analyze the tone of the utterance and decide whether or not to escalate,
4. call the conversation service (and other services),

5. possibly resume from a completed digression,
6. replace redacted private information with a human readable text.

PII protection. The PII protection based on homomorphic redaction (Section 3) relies on the four processors `Filter`, `Redirect`, `Decrypt`, and `Redact` (see Figure 11). `Filter`, the first processor of the BotShield pipeline, scans the user utterance and replaces private information with secured entities that can be understood by the chatbot and attaches encrypted versions of the private information to the request. This ensures that private information is protected as soon as possible. `Redact`, on the other end of the pipeline, replaces the secured entities with human readable text (e.g., `XXXX-XXXX-XXXX-7807` for a credit card).

External services called by the chatbot sometimes require access to the private information (e.g., an online payment service that needs access to credit card numbers). `Redirect` intercepts requests to external services and, based on a preconfigured whitelist of authorized services, instead of directly calling the service, triggers a pipeline that starts with `Decrypt` to decode the encrypted information. Furthermore, `Redirect` filters encrypted information such that even a trusted service can only see information it is allowed to see.

To deploy the PII protection, BotShield uploads the processors `Filter`, `Redirect`, `Decrypt`, and `Redact`. In addition, for each service `S` in the whitelist, BotShield automatically defines a new pipeline `P(S)` = `Decrypt` → `Call(S)`. During execution, `Redirect` substitutes the call to `S` with calls to `P(S)`. During the deployment we also automatically update the chatbot to add the secured entities to the set of entities of the chatbot NLU. This is the only modification to the original chatbot and is required to be able to recognize and exploit the secured entities.

Context digression. As discussed in Section 4, a digression is implemented with an ad-hoc processor that monitors an arbitrary condition and substitutes the context when the condition is satisfied. When the digression ends, a generic processor `Resume` substitutes the context with the content of the continuation to resume the conversation (Section 4).

To complete our computer store example, we only need to upload three processors: `Auth` that triggers the authentication (Section 4.2), `Escalate` that triggers the sentiment-based escalation (Section 4.3), and `Resume`. To add another digression, the chatbot designer only needs to upload the corresponding processor and add it to the main pipeline. Adding context digressions does not require modifying the original chatbot application code. Instead, the additional chatbot logic is implemented via separate modular chatbot scripts.

5.2 Context Protection

The handling of input/output is left to the client application. The client application typically also manages the conversation context (as in Figure 1), but BotShield provides an extra layer of security by maintaining its own version of the context (the additional `load` and `store` processors in Figure 11). Unfortunately, serverless functions are required to be stateless and cannot be used to store the context between conversation turns. We thus rely on an external database service. Two additional functions are added to the pipeline to store and load the context from the database before and after the call to the conversation service.

Table 2. Access right to the different context parts for the main components of chatbot.

	Context		
	public	private	continuation
Communications wires	✗	✗	✗
Database service	✓	✓	✓
Conversational service	✓	✓	✗
External services	✓	✗	✗
Client application	✓	✗	✗
User	✗	✗	✗

The context can be split into three parts: 1) the public context contains variables set by the client application that can be used to communicate information to the chatbot outside user utterances; 2) the private context contains all the information required to resume the conversation between two turns (e.g., the current node in the state machine of Figure 3); and 3) the possible continuation spawned when a digression is triggered.

The copy of the context maintained by BotShield allows the protection of the private context even if the application client is compromised. The private part of the context can thus be separately managed by BotShield and omitted from the response returned to the client application. In addition, the processor that calls the conversational service can omit the continuation from the context before making the call, and restore it immediately after. The conversational service thus only sees the context of the ongoing conversation and cannot corrupt the continuation. Table 2 articulates the security guarantees by summarizing access rights for the main components of a chatbot deployed with BotShield.

5.3 Experiments

In our experiments, we used the serverless platform IBM Cloud Functions [IBM 2016a] and designed the computer store chatbot with Watson Assistant [IBM 2016b]. The database service used to store the context is Cloudant [IBM 2008]. All the BotShield processors are written in TypeScript [Bierman et al. 2014] and compiled to JavaScript before the deployment. We used this application for all the chat transcripts presented in the previous sections.

Table 3 analyzes the latency of the computer store chatbot, as well as overhead introduced by BotShield. All server-side

Table 3. Latency in milliseconds (average execution time based on 200 activations).

Computer Store				
App	Load	Store	WAssistant	Order
54	448	331	276	93
BotShield Overhead				
Filter	Redirect	Decrypt	Subdialog	Escalate
55	13	26	3	238

components ran in the same cloud region. Store and load latencies are high, because we have not spent much effort on decreasing them, and we expect them to come down, for instance, via caching. The added overhead of BotShield is small (except for `escalate` due to the call to the tone-analyzer service).

6 Related Work

Related to our **homomorphic redaction**, Salesforce has an offering called Live Agent with a PII protection feature [Salesforce 2016]. Live Agent is a platform for online human-to-human chats between consumers and customer service representatives (in contrast to chatbots, which offer human-to-computer chats). Like BotShield, it lets users configure sensitive data rules for blocking PII from entering the chat. But unlike BotShield, it is not homomorphic, in that the data is completely blocked and cannot be classified by the natural-language understander or used by authorized actions.

The MrCrypt [Tetali et al. 2013] and JCrypt [Dong et al. 2016] projects use partially homomorphic encryption for cloud services. Like BotShield, they employ multiple homomorphic schemes to enable multiple kinds of computation, and these schemes sit in a lattice between clear and opaque. But unlike BotShield, MrCrypt and JCrypt are not designed for chatbots and depend on computationally expensive homomorphic schemes, whereas BotShield uses inexpensive schemes tailored for chatbots.

At a high level, homomorphic redaction is about protecting the chatbot from unwanted inputs. Besides PII, another form of unwanted inputs consists of poisoning a chatbot with bad training data, which unfortunately happened to the Tay chatbot [Wikipedia 2016]. Outside of chatbots, a common protection against unwanted inputs is taint analysis, which can be dynamic [Wall et al. 2000] or static [Guarnieri et al. 2011]. BotShield differs by focusing on PII in chatbots.

While BotShield is about protecting chatbots from PII, there are also projects where chatbots are part of the solution for PII protection [Dutta et al. 2017; Harkous et al. 2016].

Related to our **environment digression**, VoiceXML offers subdialogs [Lucas 2000; McGlashan et al. 2004]. VoiceXML is a W3C standard dialog markup language, i.e., a programming language for specifying chatbots. Going beyond VoiceXML, a recent catalog of chatbot patterns includes not just subdialogs but also digression [Hirzel et al. 2017]. BotShield differs from these standard chatbot features and patterns by making digressions non-intrusive and by isolating the environments.

Aspect-oriented languages like AspectJ offer non-intrusive digression [Kiczales et al. 2001]. One could characterize BotShield as aspects for chatbots, plus environment isolation.

BotShield uses continuations not only to implement environment digression but also to get the advantages of serverless computing including scaling and fault tolerance [Baldini et al. 2017]. This continuation-based style has been used to good effect in web programming even before the advent of serverless computing [Graunke et al. 2001], and the synergy between functional and distributed programming remains an active area of exploration [Haller et al. 2018].

Outside of chatbots, the need to protect the environment during subroutines is particularly acute when those subroutines operate at a different level of security, such as when Java calls C code [Lee et al. 2009; Li and Srisa-an 2011; Siefers et al. 2010]. Another case where isolating environments is beneficial is in multi-threaded programs [Berger et al. 2009] and in unsafe languages [Necula et al. 2002; Nethercote and Seward 2007]. BotShield also isolates environments, but unlike these projects, BotShield focuses on chatbots.

Differential privacy is about preventing private information from leaking *out of* a system, which is dual to homomorphic redaction in BotShield preventing private information from leaking *in to* a system. We are not aware of any work applying differential privacy directly to chatbots. There is work on stealing artificial-intelligence models [Tramèr et al. 2016] or data [Fredrikson et al. 2014; Krishnamurthy and Willis 2009]. In the programming language literature, one can use type systems for differential privacy [Reed and Pierce 2010], which can help prevent PII from leaking out of a database via aggregate information.

The issue of preventing private information from leaking out of a system is not new, as exemplified by two approaches from the previous century. In medical databases, it is common practice to scrub the entire database to create a copy that can then be shared for other analytics [Herting Jr and Barnes 1998]. Another commonly used approach is data masking, which scrambles data for confidentiality [Johnson et al. 1994]. Homomorphic redaction in BotShield is complementary to these approaches, since it prevents private information from entering the system in the first place and preserves the ability of the system to perform limited computation over it.

7 Conclusion

This paper introduces BotShield, a framework for protecting chatbots. BotShield provides security mechanisms that weave code addressing security concerns into the hand-authored portion of the chatbot, while making sure not to impede the machine-learned portion. More specifically, BotShield comprises two security mechanisms (homomorphic redaction and context digression), and implements them non-intrusively (requiring no manual modification of the base chatbot). As artificial intelligence becomes more pervasive, we need clear security guarantees so we can trust it. BotShield takes a step in that direction via specific techniques to secure chatbots. We believe that there is additional scope for securing artificial-intelligence applications by drawing even more deeply on established disciplines such as programming languages.

Future work Of course, we do not claim that BotShield is the last word on chatbot security. Rather, there are several avenues of interesting future work.

- The current implementation of BotShield resides entirely on the server, but it may be useful to shift some of it onto the client. For example, if the client would avoid sending private information to the server in the first place, that would shrink the trusted computing base. One could explore shipping part of BotShield to the client in the form of JavaScript code generated for that purpose.
- The current implementation of BotShield requires no modifications to the conversational service, but it could also be baked into the conversational service. Keeping it separate kept it general and reduced the barrier to adoption. But having it built-in may reduce latency and may make future features easier to implement. We anticipate that security features will become common-place in chatbot platforms.
- One protection BotShield does not yet afford but could in the future is bot detection. Companies intend their chatbots to be used by humans (customers and employees). But malicious entities might script bots to talk to them, for denial-of-service attacks or worse. BotShield could monitor incoming traffic to detect whether it is generated by a bot, and mitigate it, for instance by deliberately slowing down its responses.
- Another protection BotShield does not yet afford but could in the future is differential privacy. Section 6 compares and contrasts the features currently implemented by BotShield with differential privacy. BotShield could monitor outgoing traffic to detect whether it reveals much cumulative information, and mitigate it, for instance by overt or possibly covert redaction.
- BotShield can already be used for on-the-fly authentication, but does not yet attempt to make guarantees about authorization. In other words, it is currently up to the chatbot designer to guard portions of the scripted dialog to ensure the user has signed in. Automating that process would probably require information-flow analysis on the chatbot code itself.

Acknowledgment

We thank Alan Braz and Stephen Fink for their help and early feedback. We are also grateful to the shepherd and the reviewers for their valuable remarks.

References

Amazon. 2014. Lambda. (2014). https://aws.amazon.com/lambda/ (Retrieved June 2018).

Ioana Baldini, Perry Cheng, Stephen J. Fink, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Philippe Suter, and Olivier Tardieu. 2017. The Serverless Trilemma: Function Composition for Serverless Computing. In *Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*. 89–103.

Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. 2009. Grace: Safe Multithreaded Programming for C/C++. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 81–96.

Gavin Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *European Conference for Object-Oriented Programming (ECOOP)*. 257–281.

Daniel G. Bobrow, Ronald M. Kaplan, Martin Kay, Donald A. Norman, Henry Thompson, and Terry Winograd. 1977. GUS, a Frame-Driven Dialog System. *Artificial Intelligence* 8, 2 (1977), 155–173.

Herbert H. Clark and Susan E. Brennan. 1991. Grounding in Communication. *Perspectives on Socially Shared Cognition* 13 (1991), 127–149.

C. Cowan, F. Waggle, and Calton Pu. 2000. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. In *DARPA Information Survivability Conference and Exposition (DISCEX)*. 119–129.

Yao Dong, Ana Milanova, and Julian Dolby. 2016. JCrypt: Towards Computation over Encrypted Data. In *Conference on Principles and Practices of Programming in Java (PPPJ)*. 8:1–8:12.

Saurabh Dutta, Ger Joyce, and Jay Brewer. 2017. Utilizing Chatbots to Increase the Efficacy of Information Security Practitioners. In *Conference on Advances in Human Factors in Cybersecurity (AHFE)*. 237–243.

Facebook. 2011. Messenger Platform. (2011). https://developers.facebook.com/docs/messenger-platform/ (Retrieved June 2018).

Matthew Fredrikson, Eric Lantz, Somesh Jha, Simon Lin, David Page, and Thomas Ristenpart. 2014. Privacy in Pharmacogenetics: An End-to-End Case Study of Personalized Warfarin Dosing. In *USENIX Security Symposium*. 17–32.

Paul T. Graunke, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen. 2001. Automatically Restructuring Programs for the Web. In *Conference on Automated Software Engineering (ASE)*. 211–222.

Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. 2011. Saving the World Wide Web from Vulnerable JavaScript. In *International Symposium on Software Testing and Analysis (ISSTA)*. 177–187.

Philipp Haller, Heather Miller, and Normen Müller. 2018. A Programming Model and Foundation for Lineage-Based Distributed Computation. *Journal on Functional Programming (JFP)* 28, e7 (2018).

Hamza Harkous, Kassem Fawaz, Kang G. Shin, and Karl Aberer. 2016. PriBots: Conversational Privacy with Chatbots. In *Workshop on the Future of Privacy Indicators (WSF@SOUPS)*.

Robert L. Herting Jr and Michael R. Barnes. 1998. Large Scale Database Scrubbing Using Object Oriented Software Components. In *American Medical Informatics Association Annual Symposium (AMIA)*. 508–512.

Martin Hirzel, Louis Mandel, Avraham Shinnar, Jérôme Siméon, and Mandana Vaziri. 2017. I Can Parse You: Grammars for Dialogs. In *Summit oN Advances in Programming Languages (SNAPL)*. 6:1–6:15.

IBM. 2008. Cloudant NoSQL Database Service. (2008). https://www.ibm.com/cloud/cloudant (Retrieved June 2018).

IBM. 2016a. Cloud Functions. (2016). https://www.ibm.com/cloud/functions (Retrieved June 2018).

IBM. 2016b. Watson Assistant. (2016). https://www.ibm.com/watson/services/conversation/ (Retrieved June 2018).

IBM. 2016c. Watson Tone Analyzer Service. (2016). https://www.ibm.com/watson/services/tone-analyzer/ (Retrieved June 2018).

Donald Byron Johnson, Stephen M. Matyas, An V. Le, and John D. Wilkins. 1994. The Commercial Data Masking Facility (CDMF) Data Privacy Algorithm. *IBM Journal of Research and Development* 38, 2 (1994), 217–226.

Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. 2001. An Overview of AspectJ. In *European Conference for Object-Oriented Programming (ECOOP)*. 327–354.

Balachander Krishnamurthy and Craig E. Willis. 2009. On the Leakage of Personally Identifiable Information via Online Social Networks. In *Workshop on Online Social Networks (WOSN)*.

Byeongcheol Lee, Martin Hirzel, Robert Grimm, and Kathryn McKinley. 2009. Debug All Your Code: Portable Mixed-Environment Debugging. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 207–225.

Du Li and Witawas Srisa-an. 2011. Quarantine: A Framework to Mitigate Memory Errors for JNI Applications. In *Conference on Principles and Practice of Programming in Java (PPPJ)*. 1–10.

Bruce Lucas. 2000. VoiceXML for Web-based Distributed Conversational Applications. *Communications of the ACM (CACM)* 43, 9 (2000), 53–57.

Erika McCallister, Timothy Grance, and Karen A. Scarfone. 2010. Guide to Protecting the Confidentiality of Personally Identifiable Information (PII). *National Institute of Standards and Technology Special Publication (NIST-SP) 800-122* (2010).

Scott McGlashan, Daniel C. Burnett, Jerry Carter, Peter Danielsen, Jim Ferrans, Andrew Hunt, Bruce Lucas, Brad Porter, Ken Rehor, and Steph Tryphonas. 2004. Voice Extensible Markup Language (VoiceXML) Version 2.0. (2004). https://www.w3.org/TR/voicexml20/ (Retrieved June 2018).

Michael F. McTear. 2002. Spoken Dialogue Technology: Enabling the Conversational Interface. *ACM Computing Surveys (CSUR)* 34, 1 (2002), 90–169.

Microsoft. 2015. Bot Framework Documentation. (2015). https://azure.microsoft.com/en-us/services/bot-service/ (Retrieved June 2018).

Microsoft. 2016. Azure Functions. (2016). https://functions.azure.com/ (Retrieved June 2018).

George C. Necula, Scott McPeak, and Westley Weimer. 2002. CCured: Type-Safe Retrofitting of Legacy Code. In *Symposium on Principles of Programming Languages (POPL)*. 128–139.

Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Conference on Programming Language Design and Implementation (PLDI)*. 89–100.

Sam Newman. 2015. *Building Microservices: Designing Fine Grained Systems*. O'Reilly.

Amit Patil, K. Marimuthu, and R. Niranchana. 2017. Comparative Study of Cloud Platforms to Develop a Chatbot. *International Journal of Engineering & Technology* 6, 3 (2017), 57–61.

Jason Reed and Benjamin C. Pierce. 2010. Distance Makes the Types Grow Stronger: A Calculus for Differential Privacy. In *International Conference on Functional Programming (ICFP)*. 157–168.

Salesforce. 2016. Block Sensitive Data in Chats. (2016). https://releasenotes.docs.salesforce.com/en-us/winter16/release-notes/rn_live_agent_block_sensitive_data.htm (Retrieved June 2018).

Joseph Siefers, Gang Tan, and Greg Morrisett. 2010. Robusta: Taming the Native Beast of the JVM. In *Conference on Computer and Communication Security (CCS)*. 201–211.

Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. 1999. N Degrees of Separation: Multi-dimensional Separation of Concerns. In *International Conference on Software Engineering (ICSE)*. 107–119.

Sai Deep Tetali, Mohsen Lesani, Rupak Majumdar, and Todd Millstein. 2013. MrCrypt: Static Analysis for Secure Cloud Computations. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 271–286.

Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2016. Stealing Machine Learning Models via Prediction APIs. In *USENIX Security Symposium*. 601–618.

Mandana Vaziri, Louis Mandel, Avraham Shinnar, Jérôme Siméon, and Martin Hirzel. 2017. Generating Chat Bots from Web API Specifications. In *Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*. 44–57.

Larry Wall, Tom Christiansen, and Jon Orwant. 2000. *Programming Perl* (third ed.). O'Reilly.

Wikipedia. 2016. Tay (bot). (2016). https://en.wikipedia.org/wiki/Tay_(bot) (Retrieved June 2018).

Saman T. Zargar, James Joshi, and David Tipper. 2013. A Survey of Defense Mechanisms Against Distributed Denial of Service (DDoS) Flooding Attacks. *IEEE Communications Surveys and Tutorials* 15, 4 (2013), 2046–2069.