CHALMERS
UNIVERSITY OF TECHNOLOGY

Enforcing Privacy Requirements on Oblivious Network Agents

A Software Solution of a Type System Generated from COPPA

Bachelor of Science Thesis in Computer Science

JOEL ARDARVE
ROBERT FINNE
ADI HRUSTIC
JOHAN SVENNUNGSSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2019

Bachelor of Science Thesis

Enforcing Privacy Requirements on Oblivious Network Agents

A Software Solution of a Type System Generated from COPPA

JOEL ARDARVE
ROBERT FINNE
ADI HRUSTIC
JOHAN SVENNUNGSSON

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2019

Enforcing Privacy Requirements on Oblivious Network Agents
A Software Solution of a Type System Generated from COPPA
JOEL ARDARVE
ROBERT FINNE
ADI HRUSTIC
JOHAN SVENNUNGSSON

© JOEL ARDARVE, ROBERT FINNE, ADI HRUSTIC,
JOHAN SVENNUNGSSON 2019.

Supervisor: Nils Anders Danielsson, Department of Computer Science and Engineering
Examiner: Wolfgang Ahrendt, Department of Computer Science and Engineering

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2019
Thesis template provided by David Frisk under the CC BY 4.0 license

Abstract

The last couple of decades have shown increases in worldwide privacy-oriented regulatory practices focused on online usage. According to the 2000 U.S. federal law Children's Online Privacy Protection Act (COPPA), a child is not allowed to send personal information to a website without consent from its parent. Since the General Data Protection Regulation (GDPR) came into force in 2018, companies in the European Union have a legal obligation to incorporate some privacy-oriented practices with regard to the processing of personal data. Robin Adams and Sibylle Schupp have created a method for designing a privacy-compliant architecture that satisfies some given privacy constraints without needing to know the source code or internal structure of any individual component. Using message passing over TCP/IP and a public-key cryptosystem, this project shows that Robin and Sibylle's theory can be translated into actual software by creating an open-source software implementation in C++, with COPPA under consideration.

Keywords: Computer communications, Privacy, GDPR, COPPA, C++, Type systems

Sammanfattning

Under de senaste decennierna har databehandling reglerats allt hårdare i världen. Enligt den amerikanska internetlagen Children's Online Privacy Protection Act (COPPA) får ett barns personliga information inte skickas till en hemsida utan förälderns samtycke. Inom den Europeiska Unionen finns den liknande regleringen General Data Protection Regulation (GDPR), som började gälla år 2018. Robin Adams och Sibylle Schupp har tagit fram en metod för att göra kommunicerande mjukvarukomponenter datasekretessförenliga med avseende på angivna sekretessvillkor utan att ha tillgång till komponenternas källkod eller vetskap om deras interna struktur. Genom att använda TCP/IP samt asymmetrisk kryptering visar detta projekt, med en implementation skriven i programmeringsspråket C++ och med COPPA som exempel, att Robin och Sibylles teori kan överföras till riktig mjukvara.

Nyckelord: Datorkommmunikation, Privacy, GDPR, COPPA, C++, Typsystem

Acknowledgements

We thank our supervisor Nils-Anders Danielsson for his patience and endless support. We would also like to express our gratitude towards Robin Adams for taking the time to meet us and putting us on the right path. His explanations and illustrations regarding his work were highly appreciated in a desperate time of need.

Contents

1 Introduction **1**
1.1 Background . . . 1
1.2 Previous work . . . 2
1.2.1 Architectures, messages and types . . . 2
1.2.2 Modeling privacy requirements . . . 2
1.2.3 Method application . . . 3
1.3 Problem . . . 4

2 Method **9**
2.1 Network topology . . . 9
2.2 Programming language . . . 10
2.2.1 Library . . . 10
2.3 JavaScript Object Notation (JSON) . . . 10
2.4 Public-key cryptosystem . . . 11
2.5 The setting . . . 12
2.6 Assumptions . . . 13

3 Result **15**
3.1 Software architecture . . . 15
3.2 Message structure . . . 17
3.2.1 Signing messages . . . 17
3.2.2 Nestling messages . . . 18
3.2.3 Serializing messages . . . 19
3.3 Trace of messages . . . 19
3.3.1 t_0: *Website* sends $C_{Parent}(POLICY)$ to *Parent* . . . 20
3.3.2 t_1: *Parent* sends $C_{Website}(CONSENT)$ to *Website* . . . 20
3.3.3 t_2: *Website* sends $P_{Website}(CONSENT)$ to *Child* . . . 21
3.3.4 t_3: *Child* sends $C_{Website}(PDATA)$ . . . 22
3.3.5 t_4: *Website* receives $C_{Website}(PDATA)$ . . . 22
3.4 Incomplete trace . . . 22

4 Discussion and Conclusion **25**
4.1 Time complexity of creating messages . . . 25
4.2 TCP vs UDP . . . 25
4.3 Limitations . . . 26
4.3.1 Withdrawal of consent . . . 26

4.3.2 Multiple children or parents 27
4.4 Security vulnerabilities 27
4.4.1 Data sniffing and message spoofing 28
4.4.2 Denial of service 28
4.4.3 Client malfunction 28
4.5 Final words 28

Bibliography **29**

1
Introduction

This project explores a novel method [1] to formally secure computer communications according to specified privacy requirements. This section starts with a brief survey of the need for and difficulties of verifying whether software systems satisfy privacy requirements imposed on them. Following this inspection of the broader problem, a part of the section is devoted to a review of the method. The section ends with a statement of the context in which the project work was carried out. This will be a specific example of some agents on a network whose communications will to be secured by utilizing the method, provided a practical implementation turns out to be possible.

1.1 Background

Since the General Data Protection Regulation (GDPR) came into force in the European Union, companies have a legal obligation to incorporate some privacy-oriented practices with regard to the processing of personal data [2]. To accomplish this, one could independently implement the technical requirements needed, but given the continual rise of dependency of electronic communications in the general public [3]–[5] this becomes increasingly harder to achieve. The other alternative is to hire experts in the field to make the necessary implementations. Although this can satisfy the needed requirements, other issues arise from having third-party implementations. One being the cost of the implementations itself and maintenance of software, the other being an issue of trust. Having a functional, freely licensed, open source solution would provide an inherently trustworthy and economically viable option.

In software systems where users send messages to each other, there are challenges regarding how to verify that any imposed privacy requirements are satisfied by all components of the system. By taking privacy into account during the design of software systems, we can gain some confidence that privacy-relevant properties hold during their operation. One example of such a practice, Privacy by design [6], treats privacy as a core principle during the software engineering process, rather than an afterthought. Following these principles can help in designing a privacy-safe system, but if we do not have access to the source code of the entire system we cannot formally verify that it is privacy-safe. This becomes an issue if the interacting components of the system are controlled by different parties, as is often the case in practice. Additionally, any formal verification that has taken place on a component

of the system becomes obsolete if the component is updated.

1.2 Previous work

Adams and Schupp [1] propose a different method of verifying, at design-time, that some specified privacy requirements cannot be violated by any agent* of a software system. The method shifts focus from the inner workings of the agents and instead imposes restrictions on the types of the messages that are passed between them. The authors show how an existing type system for messages can be extended according to a set of privacy requirements. In the paper, they prove that any message that is typeable under the new type system cannot violate the privacy requirements.

This subsection will give an overview of the method. What follows is not a detailed review, and the proofs given in the paper will not be reiterated. Rather, the goal is to convey an intuition of why and how the method works, and to establish some definitions used throughout the remainder of this report.

1.2.1 Architectures, messages and types

Adams and Schupp define an *architecture* as a set of *agents* who send *messages* to each other. Every message can be typed in a *type system* associated with the architecture, so that the set of possible messages the agents can send are determined by the type system. Accordingly, a message can be seen as *a piece of data* of some type, and for simplicity these terms will be used interchangeably.

A simple example of an architecture with the set of agents $\{\alpha, \beta\}$ and the type system defined by the set of types $\{A, B\}$ is given by Figure 1.1.

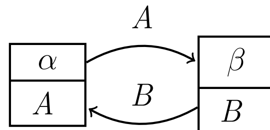


Figure 1.1: A simple architecture. Agent α can send messages of type A to β, who in turn can send messages of type B to α.

To improve the readability of this report, a sentence such as "α sends A to β" has the meaning "α sends a message of type A to β".

1.2.2 Modeling privacy requirements

The privacy requirements that serve as input to the method are modeled in terms of constraints with the following form.

$$\alpha \ni A \Rightarrow \beta \ni B$$

This is to be read as "If the agent α has a piece of data of type A, then the agent β must have previously had a piece of data of type B".

*We will henceforth use the term agent to refer both to message-passing components in a (possibly networked) software system, and to any humans that control these components.

1.2.3 Method application

The method involves taking an architecture and a set of privacy requirements to generate an extension to the architecture. The extended architecture is said to be safe with respect to the privacy requirements, and will be referred to as the *safe architecture*. The type system of the safe architecture has been enhanced with additional types, determined by the privacy requirements. The method provides each agent with an *input* and an *output interface* through which all incoming and outgoing messages must pass. The interfaces adapt the agents to the new type system by transforming their existing messages into messages of the new types.

The method generates types of the form $C_\alpha(A)$. A piece of data of type $C_\alpha(A)$ is called a *certified term*, and agent α can extract a piece of data of type A from it, but no other agent can. It also generates types of the form $P_\alpha(A)$. A piece of data of type $P_\alpha(A)$ is to be thought of as a proof that agent α has a piece of data of type A. Data can only be sent between the agents if it is of one of these two types.

Finally, the extended, safe architecture will have *constructors* with function types of the form $A \to B$. From a constructor of type $A \to B$ and a piece of data of type A, a piece of data of type B can be created.

The constraints are enforced by adding constructors of appropriate types to the agents' input and output interfaces. Figure 1.2 shows the result of applying the method to the simple architecture of Figure 1.1 with the requirement that α cannot send a message before it has received a message from β.

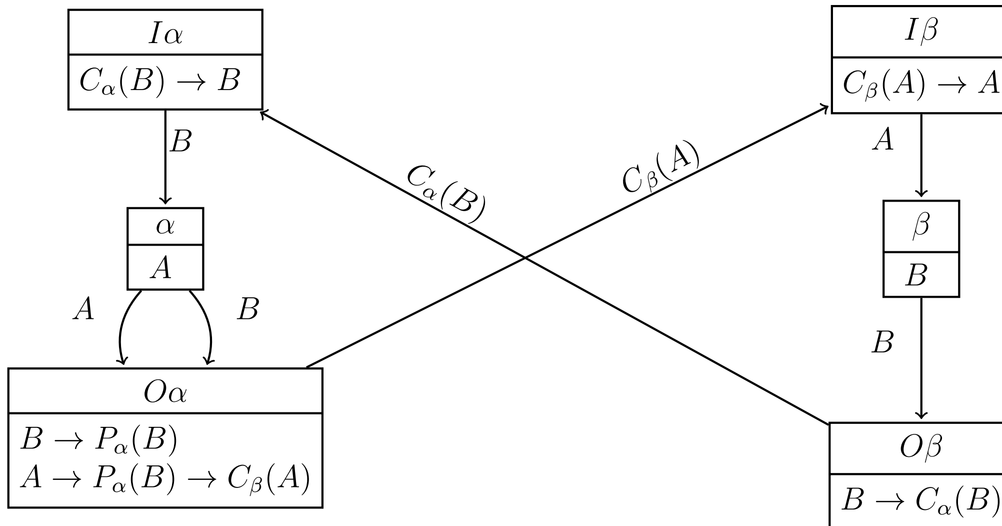


Figure 1.2: The safe architecture generated from the architecture of Figure 1.1 and the constraint $\beta \ni A \Rightarrow \alpha \ni B$. The initial, unsafe architecture has been extended with input and output interfaces for the agents, and with new types.

The input and output interfaces are labeled, respectively, I and O, followed by the name of the agent to which it belongs. Agent α may at any time send a message of type A to its output interface $O\alpha$, but this data will not travel any further unless a piece of data of type $C_\beta(A)$ is created. This can only happen if a piece of data of type $P_\alpha(B)$ is created, and this in turn is only possible if a piece of data of type

B reaches $O\alpha$. The intuition behind $B \to P_\alpha(B)$ is that if $O\alpha$ receives a message of type B from α, then $O\alpha$ may construct a proof that α has been in possession of such a message. This is exactly what the constraint requires.

Note that the agents still send and receive messages of only the initial types A and B. With the exception that α now has to relay messages of type B to its output interface, each agent is communicating with its interfaces as if it was communicating with the other agent. This is a defining feature of the method: an existing architecture can be made safe with respect to some specified privacy requirements with little need to change the internal structure of any agent.

1.3 Problem

The goal of this project is to explore Adams and Schupp's method, as outlined by their paper. This will include deciding whether a practical implementation is possible, and if any obstacles not considered by the authors need to be overcome in order to make the method work in actual code.

The authors use the U.S. internet law Children's Online Privacy Protection Act (COPPA) as a motivating example of where their theory could be applied in a real scenario. COPPA states that for a child to be allowed to send private data to a website, the website first needs consent from the parent. The parent, in turn, must have received and read the website's privacy policy in order to send its consent to the website [7]. This project work is restricted to the same scenario. An attempt is made to create a software simulation with a website, a child and the child's parent, where they obey COPPA as a consequence of the application of the method. Additionally, a requirement not stated by COPPA will be imposed on the software solution, namely that it has to be *possible* for the website to receive the child's private data.

For the simulation, the participants of the scenario are modeled as the three agents *Child*, *Parent* and *Website*. In the description of COPPA, there are three different kinds of messages involved; the child's private data, the website's policy and the parent's consent. These are modeled as the types *PDATA*, *POLICY* and *CONSENT*, respectively.

As was alluded to in the previous section, a property of the method is that it can enforce privacy requirements on an existing software system by simply providing it with an add-on. This is the approach taken in this project, and the first architecture to be considered is shown in Figure 1.3.

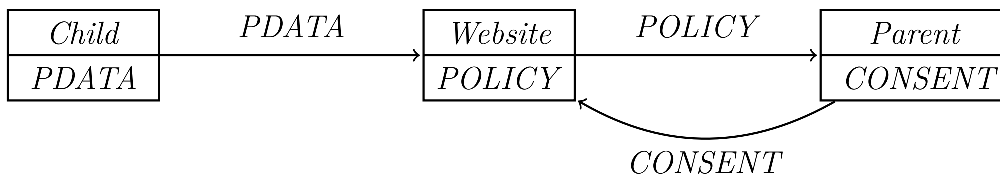


Figure 1.3: An unsafe architecture.

The architecture of 1.3 is unsafe with respect to COPPA, because *Child* may send *PDATA* to Website without either *POLICY* or *CONSENT* having been sent.

A representation of COPPA can be given by the following two constraints.

$$Website \ni PDATA \Rightarrow Website \ni CONSENT \tag{1.1}$$

$$Website \ni CONSENT \Rightarrow Parent \ni POLICY \tag{1.2}$$

Figure 1.4 shows the safe architecture generated from these constraints and the architecture of Figure 1.3

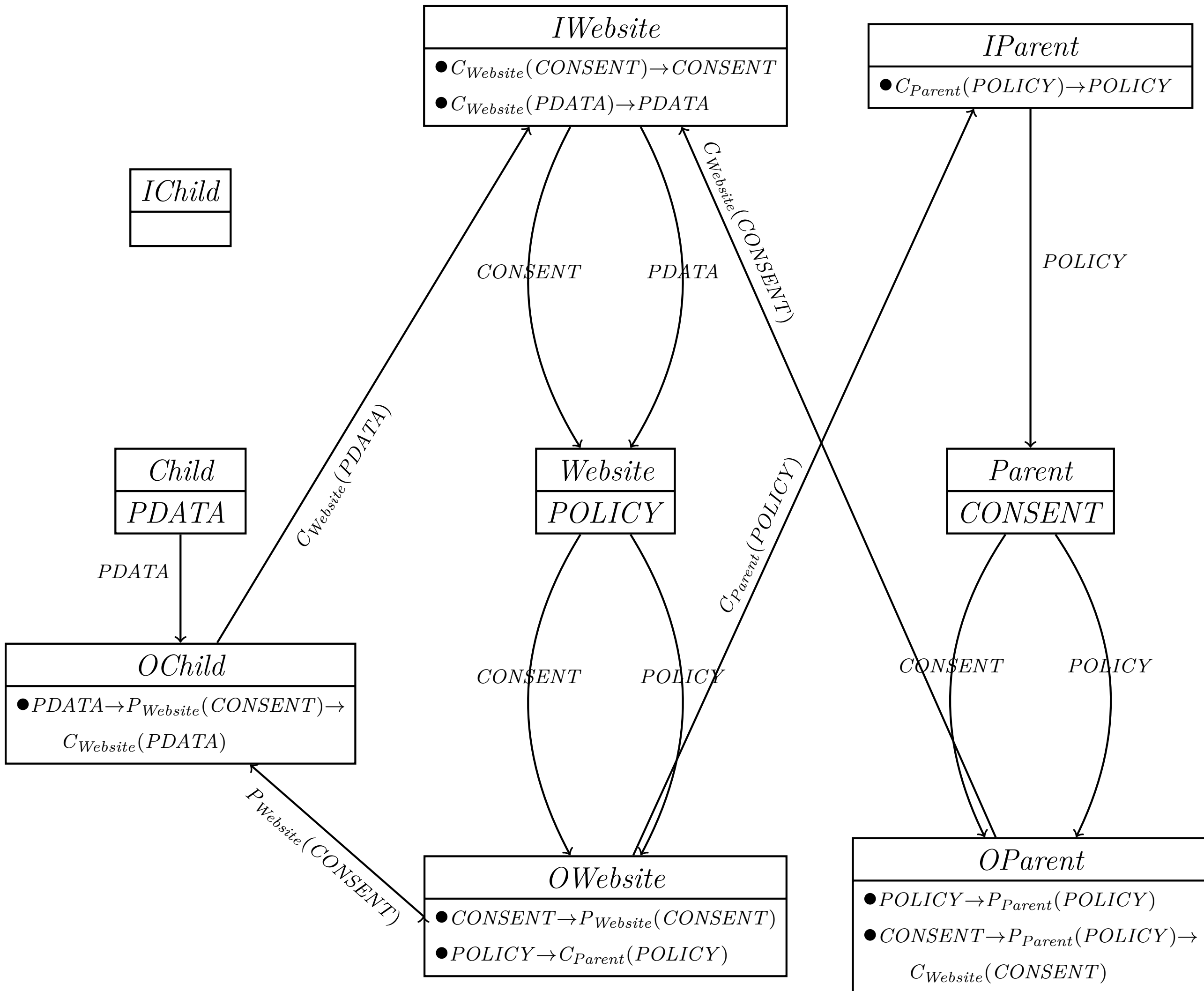


Figure 1.4: A safe architecture

As shown, the method gives each initial agent an input and an output interface. (*IChild* is not used because *Child* does not receive any messages.) *Child* cannot send its message directly to *Website* anymore. The message is instead passed to *OChild*, where a constructor is found with which it can generate data of type $C_{Website}(PDATA)$ from a piece of data of type *PDATA*, and a piece of data of type

$P_{Website}(CONSENT)$. What is needed for the message to be sent is a piece of data representing a proof that *Website* has received a piece of data of type *CONSENT*.

As stated, a requirement imposed on the software implementation is that it should be possible for *Website* to receive a message of type *PDATA*. In this conceptual model, it is, and it can be shown with a sequence of messages where this occurs. This is done below, and the constructors involved at each step is included for clarity.

Website sends *POLICY* to *OWebsite*, which has the following constructor.

$$OWebsite : POLICY \rightarrow C_{Parent}(POLICY) \tag{1.3}$$

From *POLICY*, *OWebsite* creates $C_{Parent}(POLICY)$ and sends it to *IParent*.

$$IParent : C_{Parent}(POLICY) \rightarrow POLICY \tag{1.4}$$

IParent extracts *POLICY* from $C_{Parent}(POLICY)$ and sends it to *Parent*. *Parent*, in turn, sends *POLICY* and *CONSENT* to *OParent*.

$$OParent : \begin{cases} POLICY & \rightarrow P_{Parent}(POLICY) \\ CONSENT & \rightarrow P_{Parent}(POLICY) \rightarrow C_{Website}(CONSENT) \end{cases} \tag{1.5}$$

From *POLICY*, *OParent* creates a proof that *Parent* possesses *POLICY*. This proof in conjunction with *CONSENT* is then used to create $C_{Website}(CONSENT)$, which is sent to *IWebsite*.

$$IWebsite : C_{Website}(CONSENT) \rightarrow CONSENT \tag{1.6}$$

IWebsite extracts *CONSENT* and sends it to *Website*, which passes it along to *OWebsite*.

$$OWebsite : CONSENT \rightarrow P_{Website}(CONSENT) \tag{1.7}$$

Website creates a proof that *Website* possesses *CONSENT* and sends it to *OChild*.

$$OChild : PDATA \rightarrow P_{Website}(CONSENT) \rightarrow C_{Website}(PDATA) \tag{1.8}$$

Now, *Child* sends *PDATA* to *OChild*. Having both *PDATA* and $P_{Website}(CONSENT)$, *OChild* is able to assemble $C_{Website}(PDATA)$ and send it to *IWebsite*.

$$IWebsite : C_{Website}(PDATA) \rightarrow PDATA \tag{1.9}$$

Finally, *IWebsite* can send *PDATA* to *Website*.

Compliance to a privacy requirement by all agents of a software system is normally difficult to verify, because constantly auditing the actions of all agents quickly grows

unwieldy as the number of agents increases. Using this method, it is enough to consider what the agents do locally – as long as all messages they send are typeable under the generated, safe type system, the privacy requirement is guaranteed to hold. In light of this, the software solution will include a passive third-party listener that intercepts all messages the agents send. Adams and Schupp expect that such a listener should be able to determine whether a message is typeable under the safe type system. The listener should also be able to confirm this property using only the data present in the current intercepted message. That is, it should not have to keep a history of what messages have been sent in the past. Critically, the content of the message should not be visible to the listener, only its type.

2

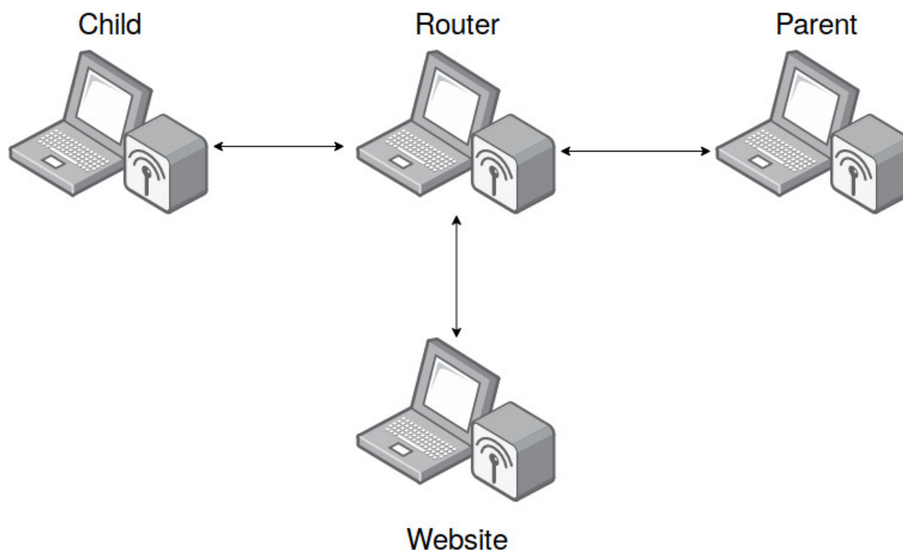
Method

Adams and Schupp define their method within the framework of mathematical logic. In particular, they use notation from type theory and set theory, and only briefly mention how concepts from their theoretical definition could be implemented in practice. The notion of certified terms seemed to lend itself naturally to encryption and decryption, but some significant design decisions and assumptions had to be made to initiate a software solution. This section discusses these design decisions and assumptions: the representation of types, the environment in which the agents were to act and the programming languages to use.

2.1 Network topology

In spite of the nature of the scenario under consideration, the method does not require that the agents of an architecture communicate over a network. However, to make the software simulation as realistic as possible, it was decided that the agents were to be acting on a real network. The software of each agent were run on its own machine, and they were connected to each other and communicating using TCP/IP over a Wi-Fi network. The connection-oriented solution of TCP was used to preserve reliability.

Figure 2.1: Network topology of the software solution



In Figure 2.1, the three devices *Child*, *Website* and *Parent* are software solutions of the agents presented in Section 1.3. *Router*, the fourth device, is the default gateway and is used for establishing a local network with the other three devices.

2.2 Programming language

The code used was mainly written in C++ with a few minor exceptions. Being a statically typed language, it was considered a suitable candidate for translating a theoretical type system into a software solution. Since potentially any type of device connected to the network could be considered an agent in the architecture, the translated interfaces should ideally be able to run independently from each other. Thus, C++ was deemed a suitable language for the interface as well.

2.2.1 Library

One major issue with latest version of C++ (C++17) is that the language only has low-level support for network-related functions such as stream sockets. Since a high-level, more object-oriented approach was preferred, and developing a fully functioning network library was deemed outside the scope of this project, an external library was used.

Portable Components (abbreviated POCO) is a C++ library used for developing network applications [8]. POCO provides a high-level approach to low-level network-functionality, but also comes with additional support for public key encryption, JSON parsing and encoding/decoding of binary data represented in base64 or hexadecimal format. This versatility of POCO made it a suitable tool for implementing the software solution while also eliminating the need for dependency on multiple libraries.

POCO was used to carry out network communication between the devices in Figure 2.1. It provided implementation of stream sockets in TCP; `Poco::Net::SocketStream` which is a bidirectional stream for reading from and writing to a socket. In order to create a `SocketStream`, a `Poco::Net::StreamSocket` is required. By using `StreamSocket` together with `Poco::Net::ServerSocket`, it can be used by a server to listen for incoming socket connections from a client. Or, the client can use `StreamSocket` together with `Poco::Net::SocketAddress` to connect to a server that is already listening for incoming socket connections. The difference between `SocketAddress` and `ServerSocket` is that `SocketAddress` takes an IP-address and port number as arguments while `ServerSocket` only require the port number the server should listen on as an argument.

2.3 JavaScript Object Notation (JSON)

From reading Adams and Schupp's paper, a property of a message was identified as a piece of data with a type and a value. For the extended architecture, it was concluded that the messages would somehow have to be composed of other messages. This was because the messages were to be constructed using constructors with function types, and a message of type B created with $A \to B$ would have to comprise the

parameterized message of type A. The messages were to be passed around and used in other constructors, some with an arity of 2. What was needed was a message structure where the size of the data would not become cumbersome. For the third-party listener, the structure would at the same time have to provide easy and effective examination of the messages. It was concluded that no specific rule-set on how to structure the message data was necessary, nor was this presented in the paper.

However, the messages needed to contain enough information to identify their type, inconsequential how this was achieved. Anyone intercepting the messages would also have to be able to verify their type by deriving every chain of the message trace all the way from the beginning of the creation of the first message. It was decided that the messages were to be nested, to allow for later recursive look-up.

JavaScript Object Notation (JSON) is a file format commonly used for storing data in text-format and is widely used in several programming languages [9]. A JSON object contains at least one key-value pair, where the key given returns its attached value. This not only results in easy look-up, but also for a clean and easy way of nestling messages into themselves. Since JSON uses human-readable text, it simplifies development and debugging. With these properties, the JSON format was deemed a viable solution for message structure.

A JSON-object is a key-value structure that begins with `{` and ends with `}`. A *key* is denoted by `"<key>": <value>`. If multiple keys exist within the JSON, they are separated by a comma "`,`" after `<value>`. *value* can have several different types, but the only ones needed for understanding the message structure presented in Section 3.2 is *string* and *object*. Strings as a `<value>` are expressed by beginning with `"` and ending with `"`. If `<value>` is a JSON object, it also begin with `{` and end with `}` but require a key. E.g.

```
1    "<key>": {
2        "<key>": <value>
3    }
```

2.4 Public-key cryptosystem

Adams and Schupp's theory suggested an architecture were some of the sent messages cannot be read by anyone but their intended recipients. Clearly, encrypting messages would suffice. It was also decided that some means of signing and verification of messages was needed. This was for two reasons, the first of which had to do with the proof type. The paper suggested that a piece of data of type $P_\alpha(A)$ should be a zero-knowledge proof which guarantees that α has a piece of data of type A, without revealing its value. For the intents and purposes of this project, it was determined that nothing more complicated than a signature by α on a piece of data of type A would be sufficient to represent a piece of data of type $P_\alpha(A)$.

For the second reason, the interfaces were to assemble new messages from previously sent ones according to their type, and each agent were sending a message of a different type. In effect, this would mean that the continuation of the message chain would be strictly depended on previous events. *Parent*, for example, would

not be allowed to send *CONSENT* without first having received *POLICY* from *Website*. The continuation is therefore dependent on the previous creation of a message (*Website* sending *POLICY* to *Parent*), but it was also deemed appropriate that the recipient could be able to verify that the message created indeed came from the right agent (*Website*, in the mentioned example).

A public-key cryptosystem was used to meet all these criteria. Before *POLICY* was to be sent to *Parent*, a message of type $C_{Parent}(POLICY)$ would be created using *Parent*'s public key, and it would be decrypted again using *Parent*'s private key before arriving at *Parent*. And if a message of type *POLICY* was signed with *Parent*'s private key, anyone knowing *Parent*'s public key could draw the conclusion that *Parent* had been in possession of the message. Finally, if any sender signs its message with its private key, the recipient (or anyone else) could verify the message against the sender's public key. If the message's content could be verified, one could conclude that the message indeed came from the stated sender.

The RSA (Rivest–Shamir–Adleman) cryptosystem was chosen for this purpose, where each agent would have have its own key-pair; a public key and a private key [10]. The key-pair was created using a key size of 2048 bits (RSA-2048). 2048 bits is the recommended size to use until year 2030. If the key-pair needs to be secure after year 2030, a key size of at least 3072 bits is recommended.[11, p. 53-56] RSA-768 – a key size of 768 bits – was broken in 2010 and is today considered unsafe to use [12].

Plain, textbook RSA is deterministic, meaning that for a given plaintext message and a given public key, the message is always encrypted to the same ciphertext. Likewise, for a given message and a given private key, the signing function of the cipher always produces the same signature. POCO's RSA implementation, however, is probabilistic, so a given plaintext can be encrypted into many different ciphertexts. This is achieved by means of introducing randomized elements to the messages at the time of their encryption. Naturally, the ciphertexts of the software solution will be longer than their corresponding plaintexts.

2.5 The setting

Adams and Schupp's method was devised in such a way that it can enforce privacy requirements on agents of an architecture without the agents having to change their behaviour in a considerable way. Accordingly, the unsafe architecture of Figure 1.3 was implemented to serve as a starting point when trying to create the software solution of the method. In other words, the solution was to take the form of an add-on. *Child* and *Website* were the initiators of their connections to *Website*, establishing their connections to designated port numbers at the *Website*'s IP address. The agents were sending messages of their initial types *POLICY*, *CONSENT* and *PDATA* when commanded to, or optionally after random delays.

Listing 2.1: The JSON object created by *Website* and sent to *Parent*

```
{
    "Type": "POLICY",
    "Value": "This is some policy"
}
```

Listing 2.2: The JSON object created by *Child* and sent to *Website*

```
{
    "Type": "PDATA",
    "Value": "This is some private data"
}
```

Listing 2.3: The JSON object created by *Parent* and sent to *Website*

```
{
    "Type": "CONSENT",
    "Value": ""
}
```

(*CONSENT* was interpreted as a type with only one possible value, and this value was represented by the empty string.) Of course, nothing prevented *Child* from sending its message before *Website* and *Parent* had sent theirs. Chapter 3 presents an attempt to secure these agents according to COPPA, while at the same time making it possible for *Website* to receive *Child*'s message. This was to be done with as small changes as possible to the agents' software.

2.6 Assumptions

During the development, a set of assumptions were operative. The first assumption was that our encryption method [10] was secure. Adams and Schupp [1] do not propose a way to exchange keys between the agents. Therefore, the second assumption made was that the key exchange had already been completed and that each agent had access to the relevant keys. The final assumption was that all agents were non-malicious, meaning that each agent did not intent to harm, break or abuse the system.

3
Result

3.1 Software architecture

An obstacle encountered immediately pertained to the fact that the agents in the safe architecture were to receive data from their input interfaces, and write data to their output interfaces. *Parent*, for instance, would receive *POLICY* from *IParent*, and send *CONSENT* to *OParent*. If the method was to take the form of a pure add-on, *Parent* would have to send and receive messages on the same connection, oblivious to the fact that it was not communicating directly with *Website* anymore. In a software solution that did not approach the problem with an add-on in mind, this would not be an issue because *Parent* could simply have two connections instead of one. However, as stated in Section 1.2, the path chosen for the software solution is based on applying an add-on to an existing unsafe architecture, which meant the messages would somehow have to be relayed. Thus, for each agent α an additional interface $R\alpha$ had to be introduced.

It was decided that each set of interfaces $\{I\alpha, O\alpha, R\alpha\}$ would be run on the same device as its associated agent α. This was done to reduce the complexity of the software solution, by making it localized as opposed to distributed – the solution would still involve network communication between three physical devices.

The architecture of the software solution is shown in Figure 3.1. Ignoring the R interfaces, the architecture strongly resembles the proposed safe architecture of Figure 1.4. Aside from each agent α, a set of three interfaces $\{I\alpha, O\alpha, R\alpha\}$ are run on the same device as agent α, and the set $\{\alpha, I\alpha, O\alpha, R\alpha\}$ can be seen as a safe agent. E.g., *Website*'s corresponding set of interfaces $\{IWebsite, RWebsite, OWebsite\}$ makes *Website* safe. In this particular scenario, *Child* never receives any data, which meant that the interface *IChild* would be unnecessary. Hence, it was not implemented.

The dashed and solid lines of 1.4 are stream socket connections, both local and remote. *Child* and *Parent* connected directly to their relay interfaces, on the same port numbers that *Website* expected them to connect to. However, it was *RWebsite* that initiated the actual connections to *Website*, on port numbers which *Website* expected the agents to connect to. This meant that the initial, unsafe agents could be connected to the privacy-safe extension by having *Child* and *Parent* connect to `localhost` instead of *Website*'s IP address.

Looking at the conceptual model of Figure 1.4, when *OParent* receives *POLICY* from *Parent* it is able to draw the conclusion that *Parent* has had *POLICY*. Sim-

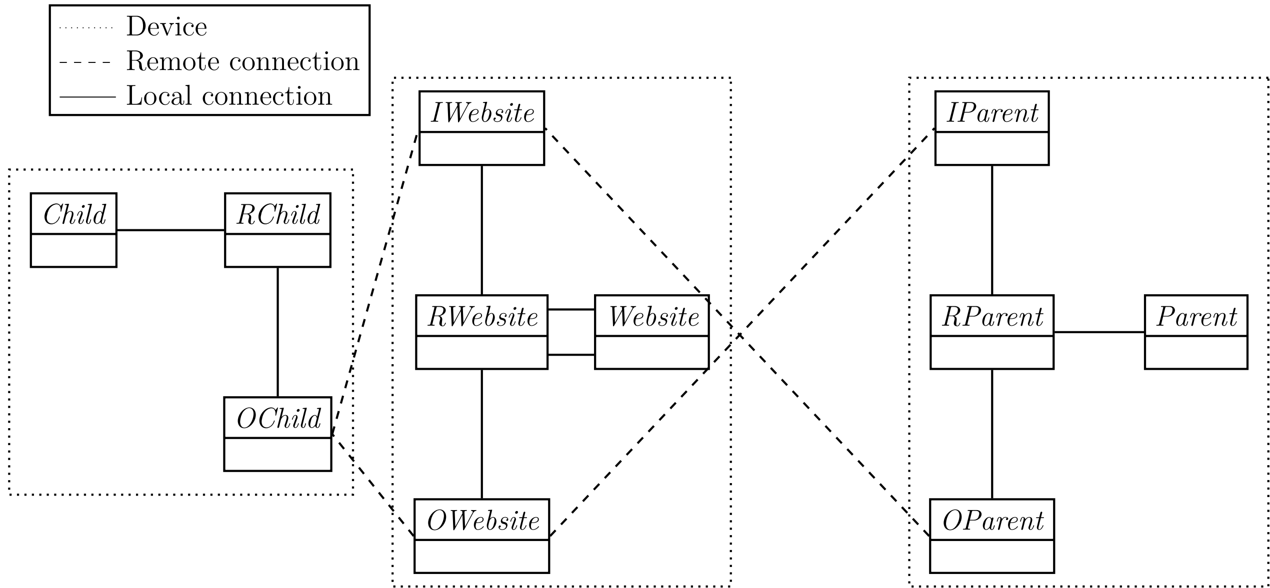


Figure 3.1: Devices and connections of the safe software architecture.

ilarly, when *OWebsite* receives *CONSENT* from *Website* it is able to send a proof to *OChild* that *Website* has had it. Because of the relay interfaces of the software solution, any data arriving at an output interface could, in theory, also have come from the input interface without ever reaching the agent. It was interpreted that the output interfaces would not be able to construct their proofs according to the exact same logic presented by Adams and Schupp.

On the other hand, it was reasoned that while implementers of this method would not have any control over the agents' behaviour or internal structures, nor over any network links between them, the one aspect they *could* control was the interfaces. It was therefore decided that it was reasonable for an output interface to trust its adjacent relay interface when it stated that a message it relayed originated from the agent.

After deciding on this path to trust the relay interfaces, another step was taken. When $RParent$ receives $POLICY$ from $IParent$, it relays this to $Parent$ as suggested by the generated safe architecture. But it *also* relays it to $OParent$ directly, without waiting for it to bounce back from $Parent$. $RWebsite$ has the corresponding behavior with $CONSENT$ at $Website$'s device. In this way, no change whatsoever needed to be made to the agents' behavior, but whether this approach could formally be shown to preserve the privacy properties according to the constraints was not investigated, and a proof not attempted.

As for the public-key cryptosystem used to create the data of types $C_\alpha(A)$ and $P_\alpha(A)$ for some agent α and some type A, the keys are used only by the input and output interfaces. For example, when $POLICY$ reaches $OWebsite$ from $IWebsite$ (via $RWebsite$), a message of the type $C_{Parent}(POLICY)$ is created by encrypting $POLICY$ with $Parent$'s public key. And when a message of type $CONSENT$ reaches $OWebsite$, a message of type $P_Website(CONSENT)$ is created by signing it with $Website$'s private key.

3.2 Message structure

The core problem of translating Adams and Schupp's theory into practice was creating a set of rules for message structure. These rules should apply from the creation of the initial type to the end of the message trace cycle. It became clear early on in the process that the way and order messages were exchanged over the network, was more crucial for the upholding of the type system than their actual content. If a generic, modular, solution was to be created, the interfaces handling the messages passed between them need to process the data in a similar way. Having every agent create their own message type was also deemed unnecessary since the only attribute separating the types from each other was the arbitrary name given to them. Instead, a JSON object created by an agent would contain a key of name `Type`, where the agent would have their type attached to this key as a string value. This object would, in direct comparison to Adams and Schupp's theory, be read as $\alpha \ni A$. Proving ownership of these messages is further discussed in Section 3.2.2. Naturally, the actual message sent by an agent would follow similar logic. A JSON key of name `Value` would be created with the message attached to this key as a string value.

Once the JSON object had its simplest structure (Listing 3.1), it needed to fulfill the requirement of being unreadable to anyone other than the intended recipient. The JSON object was therefore encrypted with the receiver's public key. Again, a direct comparison to Adams and Schupp's theory would be the method discussed in 1.2 with the creation of the certified term of type $C_\alpha(A)$. Having the sender, agent β, encrypt the JSON object with the public key belonging to the receiver, agent α, ensures that agent α, and only agent α, may extract (decrypt) the piece of data of type A. The type in the same example would be known by reading the previous mentioned `Type` key. The other piece of data generated by the same method, $P_\alpha(A)$, is discussed in Section 3.2.1.

Listing 3.1: The simplest JSON object created by an agent

```
1 {
2     "Type": "TYPE",
3     "Value": "STRING"
4 }
```

3.2.1 Signing messages

A security issue that arose during the development was an agent's ability to verify the identity of the sender. The certified term of type $C_\alpha(A)$ created by Adams and Schupp's method does not certify that the sender is the actual creator of the type, but rather certifies that only the receiver may extract the contents. In other words, the possibility existed for a malicious agent with the public key of the receiver to fake its identity as a trusted agent and send incorrect messages. This would render the whole message trace invalid and break the type system. The solution found was having the sender sign its finished JSON object with its own private key, and send the JSON along with created signature to the receiver (explained in Section 3.2.3). The receiver could then verify the JSON and the signature against the intended sender. If the verification passed, the agent would claim ownership of the correct

intended message, i.e. $P_\alpha(A)$. If the verification failed however, the receiving agent would simply dismiss the whole message.

In the software solution, the two interfaces *IWebsite* and *IParent* have the purpose of receiving serialized dot separated hex messages, deserializing them and then verifying the sender's signature.

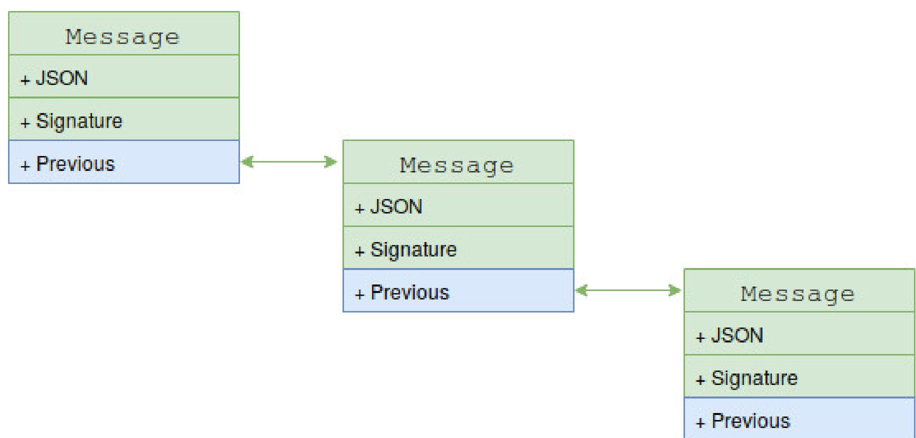
3.2.2 Nestling messages

As discussed in Section 2.3: verification of the type system must be possible at any point in time during the message chain and every type created must be verifiable all the way from the beginning of the chains creation. An elegant solution to this was already an attribute of JSON objects. One could easily store JSON objects in other JSON objects. To reuse the nested objects, one simply accesses the key (or keys) holding nested JSON objects. Since the data sent between agents was separated into a hexadecimal signature and a JSON object, it was deemed rational to create a `Previous` key containing two other keys named `JSON` and `Signature` (Listing 3.2). To verify an appended message, one just accesses the value in these two keys and makes the verification. Every single nested JSON object could with this be verified, with the original creators public key, as long as the appropriate signature was added (Figure 3.2).

Listing 3.2: An example of the JSON sent by interface *OParent* to interface *IWebsite*

```
1  {
2      "Previous": {
3          "JSON": {
4              "Type": "POLICY",
5              "Value": "*ENCRYPTED*"
6          },
7          "Signature": "*HEX SIGNATURE*"
8      },
9      "Type": "CONSENT",
10     "Value": "*ENCRYPTED*"
11 }
```

Figure 3.2: Simplified visualization of nested messages



3.2.3 Serializing messages

In order to send messages between different agents over a stream socket in a network-topology, it was not only necessary to establish how data would be serialized by the sender, but also how data would be deserialized by the receiver. Adhering to the importance of message exchange over the network as mentioned in Section 3.2, it was clear that messages passed between agents had to arrive in a predictable, ordered format. Also, since agents sign their messages before sending, the signature and the JSON object are initially two separate data parts. The method for signing a JSON object is a one-way function that produces a sequence of bytes that cannot be used to derive the original object. Due to this inability, the unsigned, original part of the message had to be separated from the signed part.

To avoid potential synchronization issues when the sender had received a signature and had yet to receive the unsigned JSON object (or vice versa), it was decided the two separate parts ought to be concatenated into a single message. Given that the signature hash can be encoded to hexadecimal format, it seemed natural to encode the JSON object into hexadecimal as well, and then separate the two parts of the message using dot (`.`) separation. Any other potential message added would also be hexadecimal encoded and appended to the original message with dot separation. A few simple rules for the encoded message were decided on too. The first two hexadecimal values represented a JSON object and its signature respectively. Anything appended represented extracted key values from the JSON. An encoded message could look like: `json_hex.sign_hex.keyval_hex`. The reason for this will become more clear in Section 3.3.

Listing 3.3: Example of a serialized message: `json_hex.sign_hex.keyval_hex`

```
68656c6c6f.776f726c64.6765206f73732066656d6d61207461636b
```

where `68656c6c6f` corresponds to `json_hex` etc.

3.3 Trace of messages

In Section 1.3, it was shown that it is possible for *Website* to receive *Child*ś message of type *PDATA* in the presented conceptual model. This section will in detail explain the trace from when the initial message is created, to when the final message that ends the trace is received. That is, from when *Website* requests consent to communicate with *Child* to actually receiving it and establishing the communication stream.

3.3.1 t_0: *Website* sends $C_{Parent}(POLICY)$ to *Parent*

Listing 3.4: $C_{Parent}(POLICY)$ represented as JSON

```
1  {
2      "Type": "CparentPOLICY",
3      "Value": "LyGLsar0uqQFsw1N..."
4  }
```

The initial action of the message trace happens when *Website* sends its privacy policy of type *POLICY* to *Parent*, as described in Equation 1.3. *Website* creates a JSON object in accordance to Listing 3.1 and sends it over the network. The message is then caught by *RWebsite* on the same port that *Website* originally connected to the network. From the agent's point of view, it is still communicating to the other agents as if the extra layer did not exist. By removing this dependency to the underlying network, Adams and Schupp's criterion of *"an approach that can be used to design a privacy-compliant architecture without needing to know the source code or internal structure of any individual component"* was fulfilled [1].

Meanwhile, *OWebsite* constructs a JSON object and adds the correct type `CparentPOLICY` as a key value for `Type` upon its initialization. After receiving the *POLICY* message from *RWebsite*, it encrypts this data with the intended receivers (*Parent*) public key and attaches it to the `Value` field. Finally the completed JSON object (Listing 3.4) is signed with *Website*'s private key, encoded into hex as presented in Section 3.2.3 and sent over the network to *IParent*.

3.3.2 t_1: *Parent* sends $C_{Website}(CONSENT)$ to *Website*

Listing 3.5: $C_{Website}(CONSENT)$ represented as JSON

```
1  {
2      "Previous": {
3          "JSON": {
4              "Type": "CparentPOLICY",
5              "Value": "LyGLsar0uqQFsw1N..."
6          },
7          "Signature": "2fc7d06edb27fe83..."
8      },
9      "Type": "CwebsiteCONSENT",
10     "Value": "p1jaNqPBrQm46XMV..."
11 }
```

IParent, like every other input interface, is constantly listening to its in-port for certified terms. Upon arrival, it will decode and deconstruct the hex message in order to verify the decoded message against the attached signature. If the signature is invalid the message will be dropped immediately and *IParent* will continue listening for the next incoming message. If, however, the signature is valid, it will extract the encrypted JSON object of type *POLICY* attached in the `Value` key, decrypt it with the *Parent*'s private key, encode it back to hex, and append it to the original message received. This new message will be sent directly to *RParent*. Since *RParent* is expecting a message from *IParent* to be of the exact structure

`json_hex.sign_hex.keyval_hex`, it knows to break it up at the second dot separator and relay the first half to *OParent* and the second half to *Parent*.

Two simultaneous processes may happen while *IParent* is awaiting the certified term from *OWebsite*: First, the *Parent* agent may consent to *Website*'s policy, even without having received it. This does not go against the type system since no message of type *CONSENT* is allowed be *sent back* to *Website* before *POLICY* arrives. This is further explained in section 3.4. *RParent* relays the consent message (of structure Listing 3.1) from *Parent* to *OParent* immediately.

Second, *OParent* may start constructing an unfinished JSON object of type $C_{Website}(CONSENT)$ as shown in Equation 1.5. It then awaits the *Parent*'s consent message from *RParent* and the previous message from *IParent*, also via *RParent*. When the encoded consent message arrives, it decodes the hex message, encrypts the JSON object with *Website*'s public key and attaches it to the `Value` field. Similarly, when the JSON object from *IParent* arrives, it will split the concatenated hex message, decode the necessary content and attach it to the `Previous` field as shown in Listing 3.5.

3.3.3 t_2: *Website* sends $P_{Website}(CONSENT)$ to *Child*

Listing 3.6: $P_{Website}(CONSENT)$ represented as JSON

```
1  {
2      "Previous": {
3          "JSON": {
4              "Previous": {
5                  "JSON": {
6                      "Type": "CparentPOLICY",
7                      "Value": "LyGLsar0uqQFsw1N..."
8                  },
9                  "Signature": "2fc7d06edb27fe83..."
10             },
11             "Type": "CwebsiteCONSENT",
12             "Value": "p1jaNqPBrQm46XMV..."
13         },
14         "Signature": "c689a5902da76324..."
15     },
16     "Type": "PwebsiteCONSENT"
17 }
```

IWebsite processes the incoming message in the same manner similarly to *IParent* in t_2, and reroutes it internally to the expected receivers via *RWebsite*. The noticeable difference is that no new type (JSON object) is created this time by *Website*. When *Parent* concedes to *Website*'s policy, it allows *Website* to establish an encrypted communication route with *Child* (Equation 1.6). However, in order for *Website* to prove to the *Child* agent that it indeed is in ownership of the consent, it will generate a new JSON object in *OWebsite* adding only the new type name and attaching all other previous messages received to it. Effectively representing this message as a message of type $P_{Website}(CONSENT)$ (Equation 1.7). This is the reason why no `Value` key is generated in Listing 3.6.

3.3.4 t_3: *Child* sends $C_{Website}(PDATA)$

Listing 3.7: $C_{Website}(PDATA)$ represented as JSON


```
{
    "Previous": {
        "JSON": {
            "Previous": {
                "JSON": {
                    "Previous": {
                        "JSON": {
                            "Type": "CparentPOLICY",
                            "Value": "LyGLsar0uqQFsw1N..."
                        },
                        "Signature": "2fc7d06edb27fe83..."
                    },
                    "Type": "CwebsiteCONSENT",
                    "Value": "p1jaNqPBrQm46XMV..."
                },
                "Signature": "c689a5902da76324..."
            },
            "Type": "PwebsiteCONSENT"
        },
        "Signature": "84f90e55a59cfbc5..."
    },
    "Type": "CwebsitePDATA",
    "Value": "pxPzLZ3Cg8Nx/Wxu..."
}
```


Unlike the rest of the agents, *Child* does not have an input interface according to Adams and Schupp's. Instead the processes usually undergone by an input interface (verification, decoding, extraction etc), is done directly in *OChild*. Similar to t_0, *Child* does not know whether messages are received by *Website* simply by trying to send them. *OChild*, while waiting for the consent, will create the *Child*'s JSON object and constantly update the last intended message with the latest one sent by *Child*. Once the proof (Equation 1.7) is received, *OWebsite*will execute the final steps for creating the message of type $C_{Website}(PDATA)$ and send it to *IWebsite*.

3.3.5 t_4: *Website* receives $C_{Website}(PDATA)$

Finally, *IWebsite* verifies the expected certified term received from *Child*, ensuring that nothing has been tampered with so far. Lastly it extracts the protected data *PDATA* using *Website*'s private key and relays it to *Website* via *RWebsite*, completing the trace.

3.4 Incomplete trace

While the previous section shows the case where *PDATA* arrives at *Website* after the latter has exchanged messages with *Parent*, this section will consider the case where *Child* attempts to send *PDATA* without *Parent* sending *CONSENT*. In order for *OChild* to send $C_{Website}(PDATA)$ to *IWebsite*, the two terms *PDATA*and $P_{Website}(CONSENT)$ are required for the message of type $C_{Website}(PDATA)$ to be

created.

The constructors in the software solution are not of actual function types. Rather, their implementation is based on an observation that e.g. $PDATA \rightarrow P_{Website}(CONSENT) \rightarrow C_{Website}(PDATA)$ can be seen as conditional existence of one the two first terms. In other words, if both $PDATA$ *and* $P_{Website}(CONSENT)$ exist, *OChild* is allowed to create $C_{Website}(PDATA)$. In a situation where *Child* wants to send its $PDATA$ and *OChild* has not received $P_{Website}(CONSENT)$ from $OWebsite$, this conditional existence is enforced by letting a threaded function for creating $C_{Website}(PDATA)$ wait indefinitely. The function waits until a conditional variable (a `Poco::Event`) has been signaled by an incoming message over the socket connection from either $OWebsite$ or *RChild*. After an event is triggered, one of the two terms required for creating $C_{Website}(PDATA)$ exists. When events has been triggered for both terms of type $PDATA$ and $P_{Website}(CONSENT)$, *OChild* is allowed to create $C_{Website}(PDATA)$. If the event for one of the two terms is never triggered, $C_{Website}(PDATA)$ is never sent.

A similar code comparison can be done for Equation 1.5, when $OParent$ needs $POLICY$ and $P_{Parent}(POLICY)$ in order to create $C_{Website}(CONSENT)$. The main difference is that both these required terms are created by the same, safe agent, while *OChild* depends on a term created by the remote interface $OWebsite$. However, $IParent$ still needs to receive $POLICY$ from $OWebsite$ in order for the trace to progress any further.

4

Discussion and Conclusion

4.1 Time complexity of creating messages

A message created at t_0 with no previous messages appended to it is assumed to have a length of at most n. Signing is linear, hence signing the message is also linear, $\mathcal{O}(n)$. The message in t_1, also with a length of at most n, has the message from t_0 appended to its final encoded message. Signing now costs $\mathcal{O}(n) + \mathcal{O}(n) \in \mathcal{O}(n)$. Following this logic, a message at t_i would have a signing cost of $i \cdot \mathcal{O}(n)$. If i is substituted for n, the complexity is $\mathcal{O}(n^2)$, which is the cost of completing a whole message trace. A similar complexity argument can be made for verifying the whole trace, beginning at any point in the trace. Since verification is linear, recursively verifying each message created in the proof above would yield a complexity $\mathcal{O}(n^2)$.

The reason for a quadratic complexity stems from solving the problem of recursive verification (Section 1.2) with every agent appending the previous received message to its own before signing its own message. A possibility for combating this was discussed but never explored. The idea was to have each agent only sign its own created message, and then have it appended to the received one. This would change the complexity of completing the trace to $\mathcal{O}(n)$, since signing would be bound by the number of agents m in the trace rather than the accumulated length of the signed message n in the trace, where $m \geq n$.

$$\mathcal{O}(n) + \mathcal{O}(n) + \mathcal{O}(n) + \ldots \in \mathcal{O}(n)$$

Similarly, verification would also be linear since only the latest message created in the trace would need to be verified. The possible security impacts of this idea were not discussed.

4.2 TCP vs UDP

One of the main reasons for choosing TCP over UDP was due to the properties of the protocol; reliability and connection-orientedness. Except for `Type`, all key-values in each JSON-message contain long key-values in either base64 or hexadecimal format. Long messages have to be segmented into multiple packets. If a packet is lost during transmission, TCP will try again. UDP, on the other hand, will simply warn the receiver a packet has been lost. Both TCP and UDP segment long messages if the data does not fit into a single packet, but only TCP is defined by its property of

reliably delivering complete messages.

In an optimal local network where no packets are lost, it might have been beneficial to use UDP over TCP. UDP, being connectionless, would reduce the need for establishing socket connections with each agent within the architecture. In a more realistic scenario where data is transmitted over the internet, *Website* might have a couple of thousand users and require *CONSENT* from hundreds of parents. Establishing thousands of socket connections with each agent might become a scalability problem. However, without the reliability, error control and order TCP provides, it would have been harder to i system at the application layer to implement some of the features TCP has but UDP lacks.

4.3 Limitations

The method uses a language of constraints with expressions of the form $\alpha \ni A \Rightarrow \beta \ni B$. These can be used to express that some data can exist only if some other data has existed, and it is also possible to express the locations of the different data. As shown in this project, it is straightforward to express a requirement that data collectors disclose what they do with data collected from users. However, there are some real-world privacy requirements that cannot be enforced using this method, such as obligation to delete data after some specified time.

4.3.1 Withdrawal of consent

The type system generated by Adams and Schupp's method [1] for this example does not provide a way for *Parent* to withdraw an existing consent. A law-abiding *Website* will have no problem removing *CONSENT* upon request from *Parent*. *Parent* just needs to contact *Website* in order for *CONSENT* to be withdrawn. A simple solution to this issue could be to add an expiry date of n days from the message's time of creation as a key-value to the message of Section 3.3.2. However, this might imply a higher degree of trust between *Parent* and *Website* which is based on the belief *Website* is in fact law-abiding and will automatically remove *CONSENT* once the expiry date has passed.

Under the assumption that *Website* 's intentions are good (meaning it has no malicious intent of using a retracted consent), but does not comply with the deletion of expired consents, another solution could be extending the type system with a new type. The reasoning is that *CONSENT* potentially can be reused by *Website* once it has been extracted and stored. Essentially, this means *Website* can keep receiving protected data from *Child* even though *Parent* explicitly has withdrawn its consent.

But what if *CONSENT* is never actually stored? By introducing a new type D, $D_\alpha(A)$ would be read as *"proof that agent α has possessed A but deleted it"*. When *Website* receives *CONSENT*, the expiry date is extracted and once the *CONSENT* is deleted, a new term of type $D_{Website}(CONSENT)$ is created together with the expiry date previously mentioned. By replacing *Child*'s requirement of $P_{Website}(CONSENT)$ with $D_{Website}(CONSENT)$, two different agents would be allowed to verify if the term of type $D_{Website}(CONSENT)$ has expired or not. Dis-

regarded here is any legal rhetoric that a proof of previous possession of consent cannot be used interchangeably with actual consent.

Two potential issues with this solution might arise. First, a dishonest *Website* could edit and extend the expiry date before sending the proof to *Child*. There is no way to prove *CONSENT* actually existed after its deletion. The first one is solvable if *Child* has previous knowledge about how long the default expiration time is, meaning *Child* can verify whether the expiry date has been tampered with.

4.3.2 Multiple children or parents

Another limitation is the support for a scenario involving multiple *Parent* agents and/or multiple *Child* agents simultaneously trying to complete a trace of messages. Adams and Schupp only state that such a scenario would require a large number of types; a new type for each new agent and each initial type. In such a scenario, it would be required that an agent α should not attempt to extract data from $C_\beta(A)$.

The latter part is not solved in the software solution, and could potentially occur if *Website* accidentally sends a term of type $C_{Parent_1}(POLICY)$ over the wrong socket connection and agent $Parent_2$ receives it instead. If $Parent_2$ expects to receive a term of type $C(POLICY)$ from *Website*, it means the verification of the signature will pass and $Parent_2$ will attempt to decrypt the message intended for $Parent_1$. However, the attempt will be unsuccessful and only garbage data will be extracted. One way to prevent any attempts at decrypting data intended for another agent could be by adding an unique identifier as a key in the message, e.g. `Id` with the receiver's MAC address as a value. The receiver could then access the value of key and compare it to its own MAC address before any attempts at decrypting data are performed.

A solution to any scenario involving multiple parents or children was not attempted. Implementing such scenarios is left as future work.

4.4 Security vulnerabilities

A common practice in information security is the CIA triad. This consist of three important pillars of security in a system, confidentiality, integrity and availability. Confidentiality consists of blocking unauthorized personnel from viewing confidential information. This is commonly verified using a password, biometric scanner or a token. Integrity consists of blocking unauthorized personnel from changing or deleting information, and is commonly solved using access control. Availability, the last pillar in the CIA triad, consists of having access to the data if needed. A way of breaking this would be only allowing users to log in to their email during the day and not the night [13].

In this section, possible security vulnerabilities are mentioned and how the software solution presented in this report hold up against them. Even though it can resist a few mentioned potential security threats, numerous different threats might exist. Therefore, to keep this project in its intended scope, this section is left open to amendments.

4.4.1 Data sniffing and message spoofing

If a malicious party gained access to the network and was able to view the content of the messages, the confidentiality of the messages would be broken. The attacker would not be able to view anything of value since all the private information passed between the agents is encrypted using RSA-2048, which was considered secure in section 2.4.

If a malicious party attempted to break the integrity of a message by changing it before passing it to the intended recipient, the changed message would be noticed by the system. Since each message has a signature created using the private key of the sender and the content of the message, the receiver's input interface would notice that the message has been changed, and discard it.

4.4.2 Denial of service

A possible way for a malicious party to break the availability of the system is to block the communication across the network. The availability and consistency of the network is out of scope for this implementation, and no countermeasure against it is provided. This will not allow the child to send private information, but instead prevent the parent from sending consent and would result in a denial of service on the child part.

4.4.3 Client malfunction

If the child's client acts malicious and attempts to send private information without the consent of a parent, the output interface will notice the absence of a consent and block the outgoing message. Another case of malfunctioning is if the parent attempts to send consent without the website's policy, and even in this case, the message will be blocked by the parents interface.

4.5 Final words

The goal of this project was to construct a software solution of Adams and Schupp's method for generating a type system, based on COPPA. By using software representations of unsafe agents and making them follow a safe architecture just by a simple add-on to the base system, it was shown that with only small modifications, the method could be made to work in practice.

Bibliography

[1] R. Adams and S. Schupp, "Constructing independently verifiable privacy-compliant type systems for message passing between black-box components", in *Lecture Notes in Computer Science*, Springer International Publishing, 2018, pp. 196–214. DOI: 10.1007/978-3-030-03592-1_11.

[2] European Parliament and Council. (May 2016). Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation), [Online]. Available: http://eur-lex.europa.eu/legal-content/EN/TXT/?uri=OJ:L:2016:119:TOC.

[3] Sveriges Riksbank. (Sep. 2017). The Riksbank's e-krona project, Report 1, [Online]. Available: https://www.riksbank.se/globalassets/media/rapporter/e-krona/2017/rapport_ekrona_uppdaterad_170920_eng.pdf.

[4] ——, (Oct. 2018). The Riksbank's e-krona project, Report 2, [Online]. Available: https://www.riksbank.se/globalassets/media/rapporter/e-krona/2018/the-riksbanks-e-krona-project-report-2.pdf.

[5] Internetstiftelsen. (2018). Svenskarna och internet, [Online]. Available: https://internetstiftelsen.se/docs/Svenskarna_och_internet_2018.pdf.

[6] A. Cavoukian. (2011). Privacy by Design The 7 Foundational Principles, [Online]. Available: https://www.ipc.on.ca/wp-content/uploads/resources/7foundationalprinciples.pdf (visited on 02/05/2019).

[7] Office of the Federal Register (United States). (1998). Children's Online Privacy Protection Act, [Online]. Available: https://www.ecfr.gov/cgi-bin/text-idx?SID=4939e77c77a1a1a08c1cbf905fc4b409&node=16%5C%3A1.0.1.3.36&rgn=div5#se16.1.312_15 (visited on 02/14/2019).

[8] *POCO C++ Libraries - Simplifiy C++ Development*. [Online]. Available: https://pocoproject.org/ (visited on 04/03/2019).

[9] *Introducing JSON*. [Online]. Available: https://json.org/ (visited on 04/09/2019).

[10] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems", *Communications of the ACM*, vol. 26, no. 1, pp. 96–99, Jan. 1983. DOI: 10.1145/357980.358017.

[11] E. Barker, "Recommendation for key management part 1: General", Tech. Rep., Jan. 2016. DOI: 10.6028/nist.sp.800-57pt1r4.

[12] T. Kleinjung, K. Aoki, J. Franke, A. K. Lenstra, E. Thomé, J. W. Bos, P. Gaudry, A. Kruppa, P. L. Montgomery, D. A. Osvik, H. te Riele, A. Timofeev, and P. Zimmermann, "Factorization of a 768-bit RSA modulus", in *Advances*

in Cryptology – CRYPTO 2010, Springer Berlin Heidelberg, 2010, pp. 333–350. DOI: 10.1007/978-3-642-14623-7_18.

[13] J. Andress, *The Basics of Information Security: Understanding the Fundamentals of InfoSec in Theory and Practice*, ser. Syngress basics series. Elsevier Science, 2014, ISBN: 9780128008126. [Online]. Available: https://books.google.se/books?id=9NI0AwAAQBAJ.